Numbers, Representations, and Ranges

Ivor Page[1]

1.1 Unsigned Numbers

A k digit unsigned system in radix r has range $[0, r^k - 1]$.

For unsigned binary, the range is $[0, 2^k - 1]$. The result of every arithmetic operation on these numbers is within this range. So, for example, if we perform the sum: $22 - 37$ in a 6 bit unsigned binary system, the result is as follows:

$$\begin{array}{rr} & 010110 \\ - & 100101 \\ \hline & 110001 \end{array}$$

The result has value 49. The correct signed result is -15, but the representation does not support negative values. When doing the unsigned operation $\square$ on two unsigned vlaues X and Y represented in k bit unsigned notation, the result is $X \square Y mod 2^k$, where $\square \in \{+, -, \times\}$.

For any sequence of add and subtract operations in a k bit unsigned system in which the result is positive, the result will be correct in the k least significant digits of the result.

When viewed as bit-patterns, the results obtained from arithmetic operations on unsigned values are identical to those obtained when those operations are performed on the same values in the 2's complement system. Indeed the operation of the ALU is almost identical in both cases. The difference comes in how the overflow flag operates:

Unsigned operations do not set the overflow flag.

The programmer must understand this subtle difference and use these representations appropriately.

If a program performs only integer operations, there is no difference in the bit patterns of the results when we change the declarations from `int` (2's complement signed values) to `unsigned int` or vice-versa. Indeed, we can mix these types freely without changing the appearance of the results as binary bit patterns. (Note that Java does not support unsigned types.)

[1]University of Texas at Dallas

1.2 Sign and Magnitude Representation

In this system the leftmost bit of a binary representation is the sign, usually 1 for negative values. The magnitude occupies the remaining $k-1$ bits and is a simple positive binary integer. The range of this system is $[-2^{k-1}+1, 2^{k-1}-1]$. Here are some examples:

Value	Representation in a 16-bit Sign and Magnitude Representation
-32767	1111 1111 1111 1111
-32766	1111 1111 1111 1110
-1	1000 0000 0000 0001
0	1000 0000 0000 0000
0	0000 0000 0000 0000
1	0000 0000 0000 0001
32766	0111 1111 1111 1110
32767	0111 1111 1111 1111

There are two representations of zero and the arithmetic process is not independent of the sign bit. i.e. the logical steps to compute the sum $A + B$ depend on the signs of the two values. Symmetrical range is an advantage. Negation is trivial and cannot cause overflow, but the complications of the arithmetic process and the associated additional time delays are probably the reasons that this system is not used for integers. It is used, however, for storing the significands (binary fractions) in the IEEE floating point system. We will discuss it again in that module.

1.3 Biased Representation

In this system a value V is represented by $V + B$, where B is the *bias*.

Value	16-bit Biased Representation with bias $B = 2^{15}$
-32768	0000 0000 0000 0000
-32767	0000 0000 0000 0001
-1	0111 1111 1111 1111
0	1000 0000 0000 0000
1	1000 0000 0000 0001
32766	1111 1111 1111 1110
32767	1111 1111 1111 1111

The most significant bit is the sign bit, 1 implies zero or greater.
The range of this system is $[-B, 2^k - 1 - B]$.
Addition of two biased values X, Y, requires the subtraction of the bias: $X + Y - B$.
Subtraction requires addition of the bias: $X - Y + B$.
Negation requires the addition of twice the bias: $-X + 2B$.
Overflow can occur during negation.

If the bias is an integral power of 2, many of the apparent difficulties can be alleviated.

This system is used to represent the exponents of IEEE floating point values. We will consider it again there.

1.4 Signed Radix-Complement Systems

An Example from History.

Early electro-mechanical calculators used decimal arithmetic. Punched cards and printers were the main input/output devices. Values were represented by the positions of gears. In mathematical terms these systems could represent decimal values of a certain number of digits, depending on the size of the counters (numbers of gear wheels) employed. Negative numbers were represented using the 10's complement system. To negate the 8 digit 10's complement value, 00,085,456, subtract it from the 9 digit value, 100,000,000, giving 99,914,544. By convention, any representation with a 9 in the left-most, or 10^7 position is a negative value.

If we add this representation of -85,456 to the representation of the positive value 115,145, we get 100,029,689. The most significant digit is beyond the range of the 8 digit number system. The answer that remains is 29,689, which is the correct answer. If, on the other hand, we add 71,103, the answer is 99,985,647. To see the magnitude of this negative value, we negate it by subtracting it from 100,000,000, giving 14,353. The value obtained represents -14,353. These are the correct answers.

There is an alternative way to negate a 10's complement number. Subtract each digit from 9 and write down the answer, digit at a time in any order, then add 1 to the result.

What is going on here?

Any negative value with magnitude V is represented by $10^8 - V$. When we add this negative value to the positive value W, we get $10^8 - V + W$. If $W > V$ the answer is the positive quantity $W - V$ plus 10^8. The value 10^8 is discarded since it is beyond the range of the 8 digit system. If $W < V$ the answer is the negative value obtained by subtracting the positive value $V - W$ from 10^8. Note here that V and W are magnitudes.

The system described above is known as a *radix-complement system.* A 10's complement system with N digits, including the *sign digit*, has range $[-10^{k-1}, +10^{k-1} - 1]$.

Here are some examples of the representation:

Value	Representation in 8 digit 10's complement
-10,000,000	90,000,000
-9,999,999	90,000,001
-1	99,999,999
0	00,000,000
1	00,000,001
9,999,999	09,999,999

1.5 2's Complement Representation

Here are some examples of a 16 bit 2's complement system.

Value in Decimal	Representation in 16 bit 2's complement
-32768	1000,0000,0000,0000
-32767	1000,0000,0000,0001
-1	1111,1111,1111,1111
0	0000,0000,0000,0000
1	0000,0000,0000,0001
32766	0111,1111,1111,1110
32767	0111,1111,1111,1111

The range, for an N digit 2's complement system is $[-2^{k-1}, +2^{k-1} - 1]$.

In any radix-complement system the most negative value is always one larger in magnitude than the most positive value.

1.5.1 Subtraction in 2's complement systems

The 2's complement system is the most widely used representation for signed numbers. Almost all modern computers use it.

Subtraction is easily performed by pre-negating the operand to be subtracted (the subtrahend) and adding it to the minuend. Negation requires inversion of all the bits of the representation, followed by the addition of *ulp* to the result. The term *ulp* means *unit in the least significant position*, or just 1×2^0 for integers. In general, *ulp* may mean 1×2^n, where the integer n is less than zero for fractional systems. An *ulp* is always, as its name suggest, the smallest value of the representation.

At first sight, it would seem that subtraction should take twice as long as addition because of the need to add *ulp* during the negation. This is not the case. We can easily accommodate the addition of 1 by setting the carry-in of the least significant digit of the adder to 1.

The subtraction is described mathematically as $A - B = A + B^{compl} + ulp$, where B^{compl} represents the *1's complement* of the bit-pattern of B. The subtraction operation proceeds exactly the same, independent of the sign of B.

1.5.2 2's Complement More Formally

It is important to distinguish the *value* of a number from its *representation*. We shall normally write the value of a number using decimal notation with a preceding plus or minus sign. The representation will be dependent on the number system we are discussing. In binary systems, the representation will normally be a bit vector.

A k bit 2's complement representation A has bits $a_{k-1}, a_{k-2}, \cdots, a_1, a_0$, where a_{k-1} is the sign bit and a_0 is the least significant bit.

The negative of A, as we have seen, is obtained by inverting all its bits and adding 1 to the result.

Negative of $A = A^{compl} + ulp$.

There are two mathematical ways to obtain the value of a k digit 2's complement number A:

$$\text{Value of } A = \begin{cases} \sum_{i=0}^{k-1} a_i 2^i & : \quad a_{k-1} = 0 \\ -2^k + \sum_{i=0}^{k-1} a_i 2^i & : \quad a_{k-1} = 1 \end{cases}$$

Alternatively,

$$\text{Value of } A = -a_{k-1}2^{k-1} + \sum_{i=0}^{k-2} a_i 2^i$$

where a_{k-1} is the sign digit and a_i is the digit in the 2^i position.

These two forms are equivalent, independent of the sign of A.

1.5.3 The Term "2's Complement"

The term describes a **notation** for representing both positive and negative values. It is also used to describe the **process of negating** a value. We say that we *take the 2's complement* of a value in order to negate that value. Confusion arises because we can negate, or take the 2's complement, of any value, either positive or negative. A number *in 2's complement notation* may be either negative or positive. The sign bit tells us its sign.

1.6 Overflow

In any number system, the result of an arithmetic operation may not necessarily have a representation in that system. Here is a table showing when overflow *can* occur with signed values:

$A + B$

	Sign of A	
	+	−
Sign of B		
+	Yes	No
−	No	Yes

$A - B$

	Sign of A	
	+	−
Sign of B		
+	No	Yes
−	Yes	No

Consider the following example using a 4 bit 2's complement representation:

$$\begin{array}{rr} & 0101 \\ + & 0011 \\ \hline & 1000 \end{array}$$

Here we are adding +5 and +3, but the answer is -8. Note that the bit pattern of the answer is the same as that of the correct answer in its 4 least significant bits. The correct answer is obtained by extending the representation to five bits by concatenating a zero (in this example) to the left of the above answer. This zero would become the new sign bit. In practice we cannot simply extend the representation of a number system by concatenating extra bits, but see the module on multi-precision working.

In this example, it is easy to see that overflow has occurred because we have added two positive values and obtained a negative result. It would be equally simple if we added two negative values and got a positive result, or subtracted a negative value from a positive one and got a negative result.

A simple way has been developed for detecting overflow in the 2's complement representation. Notice what happens to the carry into and out of the sign bit during these operations. If the values of those carries differ, then overflow has occurred. A simple EX-OR gate is sufficient to detect overflow in 2's complement addition. For subtraction, overflow cannot occur while complementing each bit of the subtrahend. Overflow can occur during the addition step, but the EX-OR gate is sufficient to detect all such cases.

Theorem: *An overflow has occurred in the addition of two 2's complement values iff the carries into and out of the sign digit differ.*

Proof:

By cases. We only need consider addition since subtraction is performed by negation of the subtrahend followed by addition.

If we are adding two positive values, the two sign digits are zero, so the carry-out of the sign is always zero. The carry-into the sign digit is 1 iff the sum of the two values exceeds the range of the representation. Clearly this condition should generate an overflow. If the carry-into the sign digit is zero, then the sum of the two values is within the range of the representation and overflow is not indicated.

If we are adding two negative values, the two sign digits are 1 and therefore there is always a carry-out from the sign digit. The carry-in to the sign digit is zero iff the result is beyond the range of the representation. Overflow must then be generated.

Note also that the simple operation of negating a value can cause overflow in any radix-complement system, since the negative range is always one larger than the positive range. If we try to negate the most negative value, the answer will not fit into the representation. □

1.7 1's Complement Representation

The 1's complement system has rarely been used. It was used in the CDC 6600, but is not used in any modern computer systems. It is an example of a *Diminished Radix system.*

To negate a number represented in 1's complement, we simply invert each bit:

Negative of $A = A^{compl} = 2^k - 1 - A$.

As with 2's complement, the most significant bit is the *sign bit.* Its value is 1 for negative numbers and zero for positive numbers. The value of a number in this representation has 2 equivalent mathematical forms:

$$\text{Value of } A = \begin{cases} \sum_{i=0}^{k-1} a_i 2^i & : \quad a_{k-1} = 0 \\ -2^k + 1 + \sum_{i=0}^{k-1} a_i 2^i & : \quad a_{k-1} = 1 \end{cases}$$

Alternatively,

$$\text{Value of } A = -a_{k-1}[2^{k-1} - 1] + \sum_{i=0}^{k-2} a_i 2^i$$

The range of a k bit 1's complements system is symmetrical: $[-2^{k-1} + 1, +2^{k-1} - 1]$.

Here are some examples of 16 bit 1's complement numbers.

Value in Decimal	Representation in 16 bit 2's complement
-32767	1000,0000,0000,0000
-1	1111,1111,1111,1110
0	1111,1111,1111,1111
0	0000,0000,0000,0000
1	0000,0000,0000,0001
32766	0111,1111,1111,1110
32767	0111,1111,1111,1111

There are two representations of zero, often denoted 0_- and 0_+ (or -0 and $+0$). This anomaly creates a small problem for detection of zero. One solution is to pass the result of every arithmetic operation through a simple logic filter that changes any occurrence of 0_- to the all-zeros pattern of 0_+.

A further complication in the 1's complement scheme arises during arithmetic. Since subtraction is performed by pre-negating the subtrahend, and adding the result to the minuend, we need only consider addition.

The sum of two 1's complement positive values proceeds as for unsigned numbers. If there is a carry into the sign digit, overflow has occurred. Consider the example below in a 6 bit 1's complement system:

$$\begin{array}{cr} \multicolumn{2}{c}{5+9} \\ \hline & 000101 \\ + & 001001 \\ \hline & 001110 \end{array}$$

The result is correct and the representations and the steps performed are exactly as for unsigned or 2's complement numbers. Now consider an example where both arguments are negative. We shall add the 1's complements of the values above:

$$\begin{array}{cr} \multicolumn{2}{c}{-5 \ + \ -9} \\ \hline & 111010 \\ + & 110110 \\ \hline & 110000 \end{array}$$

There is a carry-out of 1 from the most significant digit, which will be discarded since it cannot fit into the 6 bit number system. The 6 bits of the answer represent -15, not the value -14 expected.

To understand this, start with two positive values, A and B.

Next, we form $-A$ and $-B$.

$-A$ is represented by $2^k - 1 - A$, and

$-B$ is represented by $2^k - 1 - B$.

The sum of these is represented by $(2^k - 1 - A) + (2^k - 1 - B) = 2^{k+1} - 2 - (A + B)$

The correct representation is $2^k - 1 - (A + B)$.

Since the value 2^{k+1} is outside the range of the system, it does not show up in the answer. The answer is an *ulp* smaller than it should be. The correction is simple.

Whenever there is a carry-out of 1 from the most significant position in 1's complement addition, an *ulp* is added to the result.

This scheme is known as *end-around carry.* The carry-out of the most-significant digit of the adder is connected to the carry-in of the least significant digit.

As with any representational system, there is more than one way to ascribe meaning to values and processes within the system. For example, if we think of the values in a k bit 1's complement system as representing positive numbers, their range would be $[0, 2^k - 2]$ (recall that there are 2 versions of zero, 000...0, and 111...1). The k bit 1's complement adder is then a *mod* $(2^k - 1)$ unsigned adder.

Consider the above examples again where the 1's complement bit patterns are interpreted as representing positive values. In the second example, (-5)+(-9), the sum would be interpreted as 58 + 54 and the bit pattern of the result, 110001, would represent 49, which is (58 + 54) mod 63.

A k bit 1's complement adder with end-around carry is a *mod* $(2^k - 1)$ adder of unsigned values that have range $[0, 2^k - 2]$.

Question: Since the end-around carry causes a cycle amongst the logic gates of a ripple-carry 1's complement adder, is there any potential for an add operation to continue for ever?

Answer: It is easy to show that the worst case time for addition is the same as for the 2's complement system if a ripple-carry adder is employed.

Negation of any value in the 1's complement system is faster than in 2's complement since the operation only requires inversion of all the bits of the number. Overflow cannot occur during negation.

1.8 Fixed Point

In a fixed point system, the radix point is assumed to be in a fixed location within the computer's integer representation. For example, 64 bit integers could be used to represent numbers with 20 fractional binary bits. One *ulp* would represent $2^{-20} \approx 0.000000954_{10}$. The computer's method for dealing with signed numbers would be independent of the programmer's assumption about the position of the binary point. Fixed point numbers in this system could be added and subtracted by just using the computer's corresponding integer operations.

Multiplication of two fixed point values in this system would promote the position of the point, doubling the number of fractional bits. Ideally a 128 bit result would be produced from the product of two 64 bit integers, but this is not usually the case without additional effort by the programmer to extend the results of multiplication to double-precision (see the module on multi-length working). Right shifting both values by ten places before multiplication is one simple solution, but it carries the obvious loss of precision.

Division of two fixed point numbers also causes some complications. For example. if 3 bits are used for the fractional part, then $8.375_{10} \div 6.825_{10}$ becomes $1000.011_2 \div 110.111_2$ in the programmer's mind. The computer sees the division using integers as 67/55 and gives the integer answer 1.

How can the result of division of two fixed point values with k assumed binary fractional digits be computed to k bits accuracy using only integer operations? To answer this we also need to consider the rounding method required.

Division by powers of two (or by constants) can easily be programmed.

1.8.1 Scaled Representation

If monetary amounts were to be processed, conversion of all values to cents would be an obvious and desirable strategy. This technique is used effectively whenever all values to be processed have a fixed number of fractional digits. For example, if Olympic race times were all recorded in thousands of a second or weights of some kind were all recored in hundredths of a gram, scaling all the values on input would be desirable.

Many systems of weights and measures still use mixed radices. For example, weights are commonly recorded in tons, pounds and ounces. In England, as well as lbs and oz, there are stones (one stone is 14lbs), quarters (one quarter is 28 lbs) and hundredweights (one hundredweight is 112 lbs). We weigh our body weight in stones and pounds and we weigh coal and potatoes in hundredweights.

Before decimal standardization on "Decimal Day," 15^{th} February 1971, English currency used pounds, shillings and pence. One shilling equalled 12 pence and 20 shillings equalled one pound. Coins and paper currency denominations (in pence) were the farthing, halfpenny, penny, tuppence thruppence, sixpence, shillings, florins, half-crowns, ten-shillings, pounds, etc. worth 1/4, 1/2, 1, 2, 3, 6, 12, 24, 30, 120, 240 pence respectively. It was a mess!

Horses were bought and sold in guineas. The guinea was the first gold coin struck, originally worth one pound, but the rising value of gold caused its value to rise to 21 shillings.

Ideally all values would be converted to the lowest valued units in input.

1.8.2 Conversion of Values From Decimal to Fixed Point Binary

Input of values with fractional parts requires some care. For example, to read a decimal value a.b, where a is the integer part and b is the fractional part, we must first understand the following.

- All integers in one radix convert exactly to integers in another radix (so long as both radices are integers).
- Conversion of fractional parts between two integer radices may lead to a repeating fractional part.

To read the decimal value $a_{k-1}a_{k-2}\cdots a_1a_0.a_{-1}a_{-2}\cdots a_{-m}$, $a_i \in [0,9]$, we could ignore the decimal point and read the digits of the number as an integer. Using Horner's rule we compute:

$$I = (...(a_{k-1} \times 10 + a_{k-2}) \times 10 + a_{k-3}) \times 10 + a_{k-4}) \cdots + a_0) \times 10 + a_{-1}) \times 10 + \cdots + a_{-m}$$

.

The result is too large by a factor 10^m.

Say, for example, we wanted to read 123.456 into a fixed point system with 10 fractional bits. Reading 123456 as an integer gives:

$$000 \cdots 00011110001001000000$$

Integer division by 10^3 will yield:

$$000 \cdots 01111011 = 123_{10}$$

We just lost the fractional part because integer division truncates to an integer result.

Instead, let's scale the integer representation by left shifting as far to the left as possible before dividing by 10^m. This should preserve as much precision as possible in a 32 bit integer.

In this example, there are 17 significant bits so we will left shift $p = 32 - 1 - 17 = 14$ places, preserving the sign bit:

$$0111100010010000 \cdots 0000$$

Division by 10^3 gives:

$$00000000000111101101110100101111$$

Finally, we right shift $14 - 10 = 4$ places to get to the fixed point system desired:

$$0000000000001111011.0111010010$$

The binary point is included for clarity.

1.8.3 A Brief Introduction to Rounding - Truncation

In the final right-shift operation above we lost 4 bits of the 32 bit pre-shift value. But the decimal fraction 0.456 has no exact representation as a binary fraction.

The error in our fixed point representation 0f 123.456_{10} is

$$0.456_{10} - 0.011101001_2 = 0.456 - 0.45507813 = 0.000921875_{10}$$

We accumulated this error both because fractions do not convert exactly between radices and because of the fixed length fractional part. Let's just consider the error caused by the right shift operation. It changed the representation from

$$00000000000111101101110100101111$$

to

$$00000000000000011110110111010010$$

The rounding method used in the shift operation is called **truncation**.

Here is the transfer graph just for positive values:

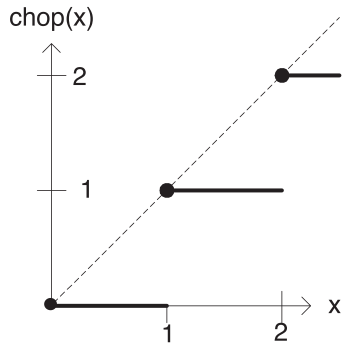

Figure 1: Truncation, Round Toward Zero

Along the x axis is the continuous variable given to infinite precision representing the value to be rounded. One unit along this axis corresponds to one ulp.

The y axis shows the values chosen by truncation. The large dots are the values chosen for the special cases when x is an exact integer on this scale.

For example, consider the last few bits of the number above before truncation: $10|1111$. The vertical bar shows the *truncation point*, the partition from one ulp to fractions of an *ulp*. On the transfer graph, this value would be 15/16 of an *ulp*, just to the left of 1 ulp. The corresponding value after the shift is zero, as indicated by the solid line segment from 0 to 1 on the x axis. The worst case error due to truncation is then one *ulp*.

We are also concerned with bias in the rounding method.

We only need consider the 2 bits to the immediate right of the truncation point. If all four possible settings of these 2 bits are equally likely then we get the following table:

$$\begin{aligned}
x_{-1}x_{-2} &= 00 \text{ Rounding error} = 0.00 - 0.00 = 0 \; ulp \\
x_{-1}x_{-2} &= 01 \text{ Rounding error} = 0.01 - 0.00 = 1/4 \; ulp \\
x_{-1}x_{-2} &= 10 \text{ Rounding error} = 0.10 - 0.00 = 1/2 \; ulp \\
x_{-1}x_{-2} &= 11 \text{ Rounding error} = 0.11 - 0.00 = 3/4 \; ulp
\end{aligned}$$

where x_{-1} and x_{-2} are the two bits to the immediate right of the truncation point.

The average error is 3/8 *ulp*. An unbiased system would give an average error of zero.

For negative values, the table becomes:

$$\begin{aligned}
x_{-1}x_{-2} &= 00 \text{ Rounding error} = -0.00 - (-0.00) = 0 \; ulp \\
x_{-1}x_{-2} &= 01 \text{ Rounding error} = -0.01 - (-0.00) = 1/4 \; ulp \\
x_{-1}x_{-2} &= 10 \text{ Rounding error} = -0.10 - (-0.00) = 1/2 \; ulp \\
x_{-1}x_{-2} &= 11 \text{ Rounding error} = -0.11 - (-0.00) = 3/4 \; ulp
\end{aligned}$$

The rounding method actually **Rounds towards minus infinity**. Again a bias of 3/8 *ulp* is present.

Here is the full transfer graph:

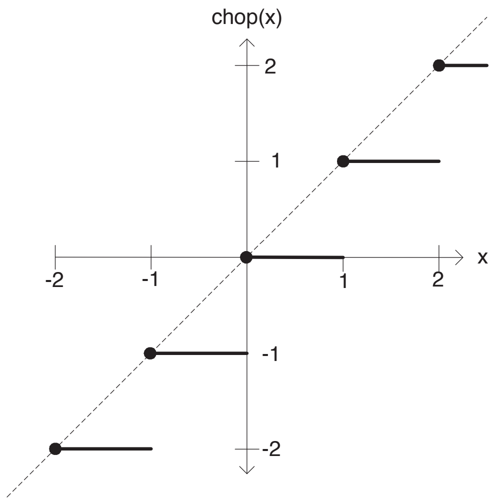

Figure 2: Truncation, Round Towards $-\infty$

1.8.4 Round to Nearest

A commonly used rounding technique for decimal fractions is called **round to the nearest**. $[0.0, 0.5)$ rounds to 0.0, $[0.5, 1.0)$ rounds to 1. We can implement this system by examining the bit to the immediate right of the truncation point. If it is 1, add *ulp* to the truncated fraction. Our value of 123.456_{10} then becomes

000000000001111011.0111010011

where the binary point has been added for clarity. The following table applies:

$$\begin{aligned}
x_{-1}x_{-2} &= 00 \text{ Rounding error} = 0.00 - 0.00 = 0 \textit{ ulp} \\
x_{-1}x_{-2} &= 01 \text{ Rounding error} = 0.01 - 0,00 = 1/4 \textit{ ulp} \\
x_{-1}x_{-2} &= 10 \text{ Rounding error} = 0.10 - 1.00 = -1/2 \textit{ ulp} \\
x_{-1}x_{-2} &= 11 \text{ Rounding error} = 0.11 - 1.00 = -1/4 \textit{ ulp}
\end{aligned}$$

The average error is -1/8 *ulp*. The system is biased.

Here is the transfer graph:

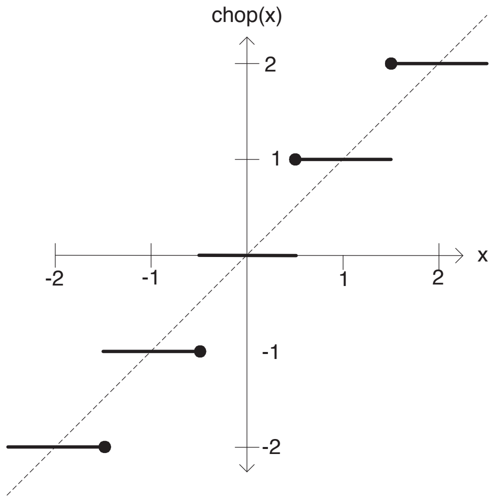

Figure 3: Round to Nearest

The effect of bias in a fixed point system can accumulate. If, for example, 1000 decimal values were read into a program using the above fixed point system with the *round to nearest* rounding method and these values were added together, the expected error would be 1000/8 *ulp*. That's 7 bits or error in a system with only 10 bits allocated for the fractional part. These considerations and others motivate floating point systems, which we will study in depth later in the course. Rounding methods will be considered again in that module.