

## Rounding Algorithms 101

---

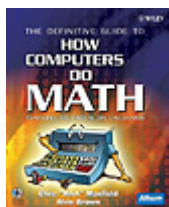
We all remember being taught the concept of rounding in our younger years at school. Common problems involved monetary values, such as rounding some amount like \$5.19 to the nearest dollar (which would be \$5 in the case of this example). However, although this may seem simple at a first glance, there's a lot more to rounding than might at first meet the eye ...

### View Topics

- [Introduction](#)
- [Round-Toward-Nearest](#)
- [Round-Half-Up \(Arithmetic Rounding\)](#)
- [Round-Half-Down](#)
- [Round-Half-Even \(Banker's Rounding\)](#)
- [Round-Half-Odd](#)
- [Round-Ceiling \(Positive Infinity\)](#)
- [Round-Floor \(Negative Infinity\)](#)
- [Round-Toward-Zero](#)
- [Round-Away From-Zero](#)
- [Round-Up](#)
- [Round-Down](#)
- [Truncation \(Chopping\)](#)
- [Round-Alternate](#)
- [Round-Random \(Stochastic Rounding\)](#)
- [Sign-Magnitude Binary Values](#)
- [Rounding Signed Binary Values](#)
- [Summary and Further Reading](#)

---

### Introduction



Before we start, we'd just like to point out that this paper on rounding is just one of the many resources to be found on our website at [www.DIYCalculator.com](http://www.DIYCalculator.com), which supports our recently published book *How Computers Do Math* (ISBN: 0471732788). Festooned with nuggets of knowledge and tidbits of trivia, this book, which features the virtual DIY Calculator, provides an incredibly fun and interesting introduction to the way in which computers perform their magic in general and how they do math in particular. But we digress ...

One key fact associated with rounding is that it involves transforming some quantity from a greater precision to a lesser precision. For example, suppose that we average out a range of prices and end up with a dollar value of \$5.19286. In this case, rounding the more precise value of \$5.19286 to the nearest cent would result in \$5.19, which is less precise.

This means that if we have to perform rounding, we would prefer to use an algorithm that minimizes the effects of this loss of precision. (The term "algorithm", which is named after the legendary Persian astrologer, astronomer, mathematician, scientist, and author Al-Khawarizmi [circa 800-840], refers to a detailed sequence of actions that are used to accomplish or perform a specific task.)

We especially wish to minimize the loss of precision if we are performing some procedure that involves cycling round a loop performing a series of operations (including rounding) on the same data over and over again. If we fail, the result will be so-called "creeping errors," which refers to errors that increase over time as we iterate around the loop.

So how hard can this be? Well, things can become quite interesting, because there is a plethora of different rounding algorithms that we might use. These include *round-toward-nearest* (which itself encompasses *round-half-up* and *round-half-down*), *round-up*, *round-down*, *round-toward-zero*, *round-away-from-zero*, *round-ceiling*, *round-floor*, *truncation* (chopping), ... and the list goes on.

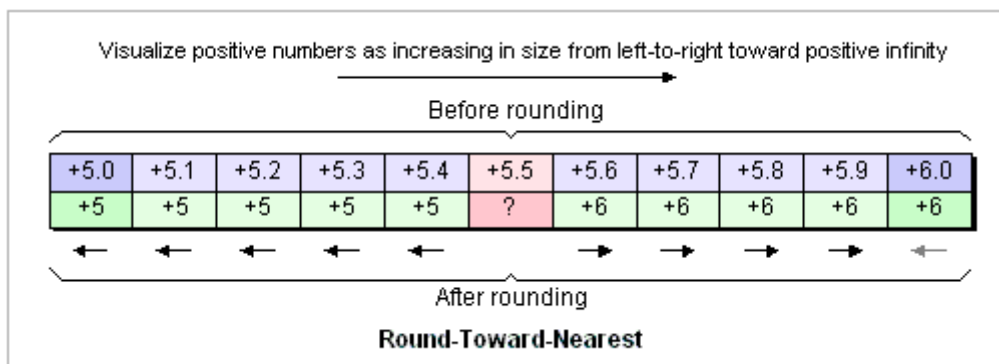
Just to increase the fun and frivolity, some of these terms can sometimes refer to the same thing, while at other times they may differ (this can depend on who you are talking to, the particular computer program or hardware implementation you are using, and so forth). Furthermore, the effect of a particular algorithm may vary depending on the form of number representation to which it is being applied, such as unsigned values, sign-magnitude values, and signed (complement) values.

In order to introduce the fundamental concepts, we'll initially assume that we are working with standard (sign-magnitude) decimal values, such as +3.142 and -3.142. (For the purposes of these discussions, any number without an explicit sign is assumed to be positive.) Also, we'll assume that we wish to round to integer values, which means that +3.142 will round to +3 (at least, it will if we are using the *round-toward-nearest* algorithm as discussed below). Once we have the basics out of the way, we'll consider some of the implications associated with applying rounding to different numerical representations, such as signed binary numbers.

[Top](#)

## Round-Toward-Nearest

As its name suggests, this algorithm rounds towards the nearest significant value (this would be the nearest whole number, or integer, in these case of these particular examples). In many ways, this is the most intuitive of the various rounding algorithms, because values such as 5.1, 5.2, 5.3, and 5.4 will round down to 5, while values of 5.6, 5.7, 5.8, and 5.9 will round up to 6:

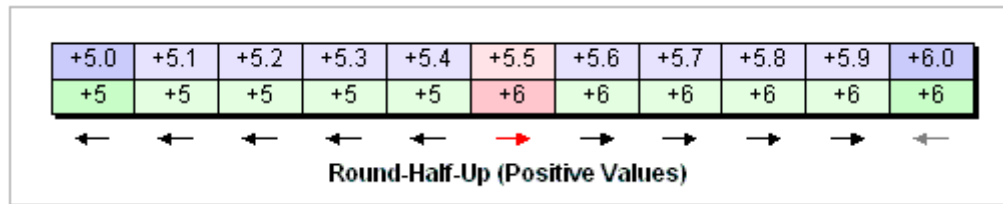


But what should we do in the case of a "half-way" value such as 5.5. Well, there are two obvious options: we could round it up to 6 or we could round it down to 5; these schemes, which are introduced below, are known as **Round-Half-Up** and **Round-Half-Down**, respectively.

[Top](#)

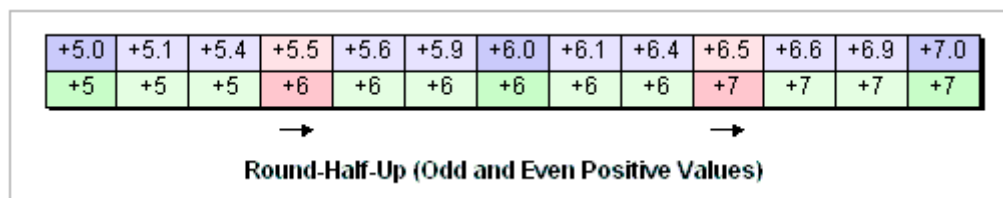
## Round-Half-Up (Arithmetic Rounding)

This algorithm, which may also be referred to as *arithmetic rounding*, is the one that we typically associate with the concept of rounding that we learned at school. In this case, a "half-way" value such as 5.5 will round up to 6:



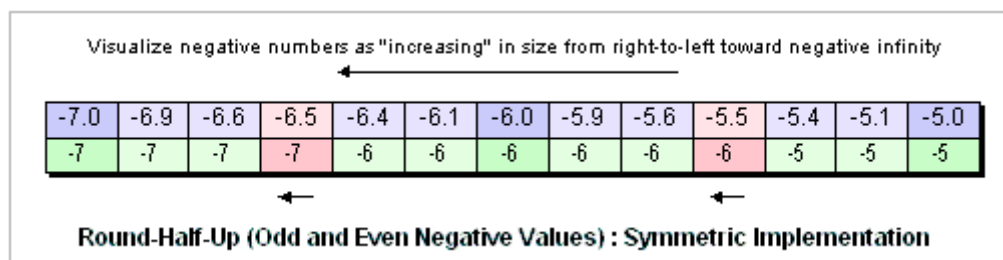
One way to view this is that (at this level of precision and for this particular example) we can consider there to be ten values that commence with a "5" in the most-significant place (5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, and 5.9). On this basis, it intuitively makes sense for five of the values to round down and for the other five to round up; that is, for the five values 5.0 through 5.4 to round down to 5, and for the remaining five values 5.5 through 5.9 to round up to 6.

Before we move on, it's worth taking a few moments to note that "half-way" values will always round up when using this algorithm, irrespective of whether the final result is odd or even:



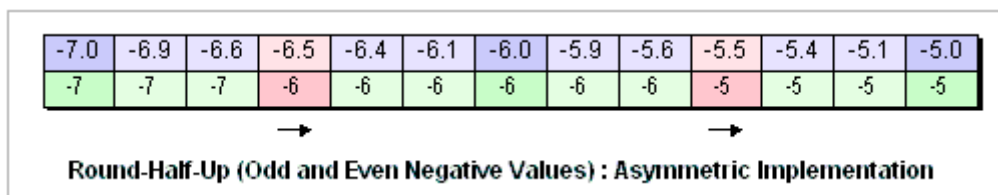
As we see from the above example, the 5.5 value rounds up to 6, which is an even number; and the 6.5 value rounds up to 7, which is an odd number. The reason we are emphasizing this point is that the **Round-Half-Even** and **Round-Half-Odd** algorithms – which are introduced later in this paper – would treat these two values differently. (Note that the only reason we've omitted the values 5.2, 5.3, 5.7, 5.8, 6.2, 6.3, 6.7, and 6.8 from the above illustration is to save space so that we can display both odd and even values.)

The tricky point with this *round-half-up* algorithm arrives when we come to consider negative numbers. There is no problem in the case of the values like -5.1, -5.2, -5.3, and -5.4, because these will all round to the nearest integer, which is -5. Similarly, there is no problem in the case of values like -5.6, -5.7, -5.8, and -5.9, because these will all round to -6. The problem arises in the case of "half-way" values like -5.5 and -6.5 and our definition as to what "up" means in the context of "round-half-up." Based on the fact that positive values like +5.5 and +6.5 round up to +6 and +7, respectively, most of us would intuitively expect their negative equivalents of -5.5 and -6.5 to round to -6 and -7, respectively. In this case, we would say that our algorithm was *symmetric* (with respect to zero) for positive and negative values:



Observe that the above illustration refers to negative numbers as "increasing" in size toward negative infinity. In this case, we are talking about the absolute size of the values (by "absolute", we mean the size of the number if we disregard it's sign and consider it to be a positive quantity; for example, +1 is bigger than -2 in real-world terms, but the absolute value of -2, which is written as  $|-2|$ , is +2, which is bigger than +1).

But we digress. The point is that some applications (and some mathematicians) would regard "up" as referring to positive infinity. Based on this, -5.5 and -6.5 would actually round to -5 and -6, respectively, in which case we would class this as being an *asymmetric* (with respect to zero) implementation of the *round-half-up* algorithm:



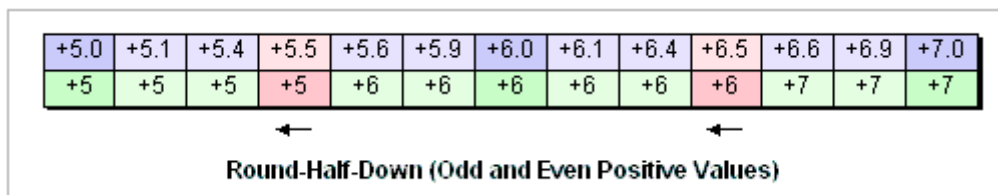
The problem is that different applications may treat things differently. For example, the *round* method of the Java Math Library provides an asymmetric implementation of the *round-half-up* algorithm, while the *round* function in the mathematical modeling, simulation, and visualization tool MATLAB® from **The MathWorks** provides a symmetric implementation. (Just for giggles and grins, the *round* function in Visual Basic for Applications 6.0 actually implements the **Round-Half-Even** [Banker's rounding] algorithm discussed later in this paper.)

As a point of interest, the symmetric versions of rounding algorithms are sometimes referred to as *Gaussian implementations*. This is because the theoretical frequency distribution known as a “Gaussian Distribution” – which is named after the German mathematician and astronomer Karl Friedrich Gauss (1777-1855) – is symmetrical about its mean value.

[Top](#)

## Round-Half-Down

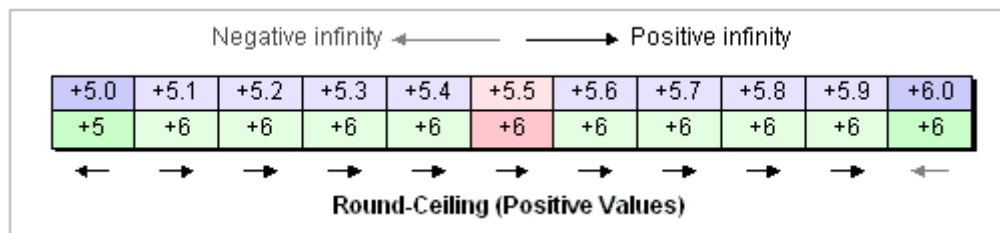
Perhaps not surprisingly, this incarnation of the **Round-Toward-Nearest** algorithm acts in the opposite manner to its **Round-Half-Up** counterpart discussed above. In this case, "half-way" values such as 5.5 and 6.5 will round down to 5 and 6, respectively:



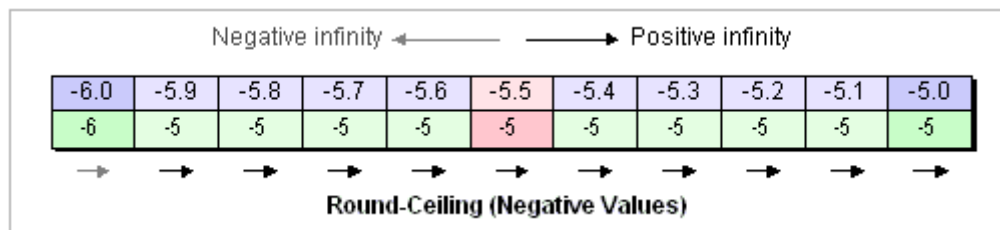
Once again, we run into a problem when we come to consider negative numbers, because what we do with "half-way" values depends on what we understand the term "down" to mean. On the basis that positive values of +5.5 and +6.5 round to +5 and +6, respectively, a symmetric implementation of the *round-half-down* algorithm will round values of -5.5 and -6.5 to -5 and -6, respectively:







By comparison, in the case of a negative number, the unwanted digits are simply discarded. For example, the values -5.0, -5.1, -5.2, -5.3, -5.4, -5.5, -5.6, -5.7, -5.8, and -5.9 will all be rounded to -5:



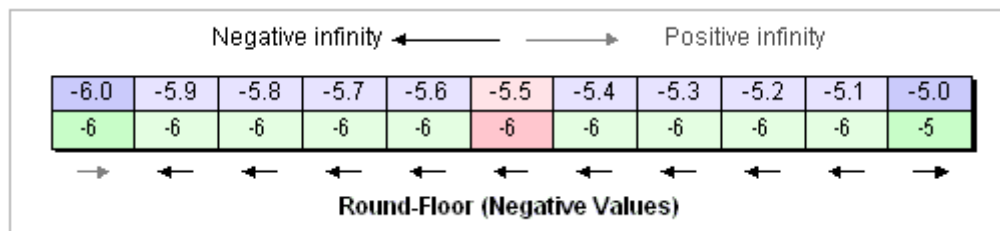
From the above illustrations, it's easy to see that the *round-ceiling* algorithm results in a cumulative positive bias. Thus, in the case of software implementations, or during analysis of a system using software simulation, this form of rounding is sometimes employed to determine the upper limit of the algorithm (that is, the upper limit of the results generated by the algorithm for a given data set) for use in diagnostic functions.

In the case of hardware (a physical realization in logic gates), this algorithm requires a significant amount of additional logic – especially when weighed against its lack of compelling advantages – and is therefore rarely used in hardware implementations.

[Top](#)

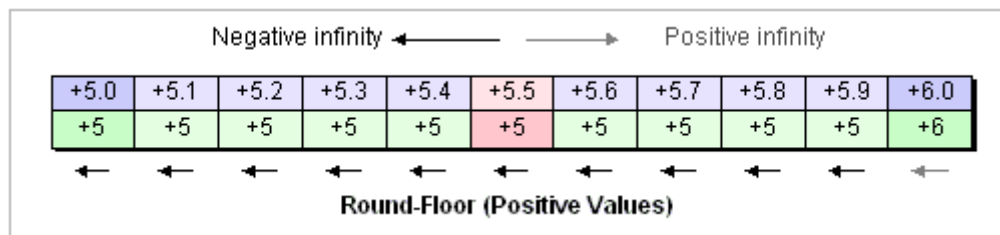
### Round-Floor (Toward Negative Infinity)

This is the counterpart to the **Round-Ceiling** algorithm introduced above; the difference being that in this case we round toward negative infinity. This means that, in the case of a negative value, the result will remain unchanged if the digits to be discarded are all zero; otherwise it will be rounded toward negative infinity. For example, -5.0 will be rounded to -5, but -5.1, -5.2, -5.3, -5.4, -5.5, -5.6, -5.7, -5.8, and -5.9 will all be rounded to -6 (similarly, -6.0 will be rounded to -6, while -6.1 through -6.9 will all be rounded to -7):



By comparison, in the case of a positive value, the unwanted digits are simply discarded. For example, the values +5.0, +5.1, +5.2, +5.3, +5.4, +5.5, +5.6, +5.7, +5.8, and +5.9 will all be rounded to +5





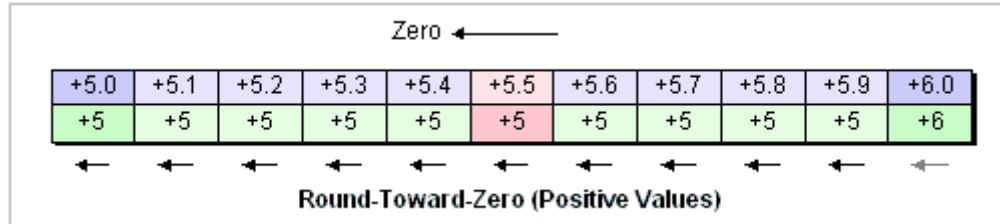
From the above illustrations, it's easy to see that the *round-floor* algorithm results in a cumulative negative bias. Thus, in the case of software implementations, or during analysis of a system using software simulation, this form of rounding is sometimes employed to determine the lower limit of the algorithm (that is, the lower limit of the results generated by the algorithm for a given data set) for use in diagnostic functions.

Furthermore, when working with signed binary numbers this approach is “cheap” in terms of hardware (a physical realization in logic gates) since it involves only a simple truncation (the reason why this should be so is discussed later in this paper); this technique is therefore very often used with regard to hardware implementations.

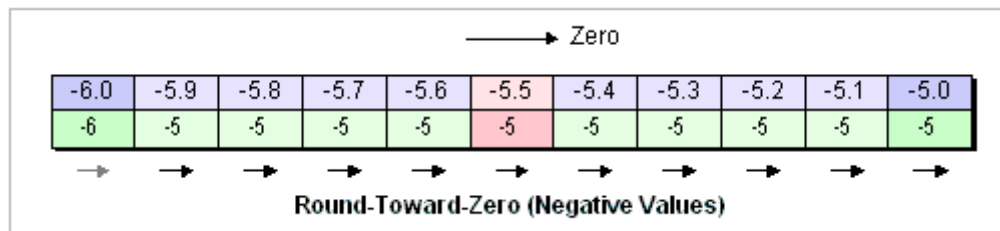
[Top](#)

## Round-Toward-Zero

As its name suggests, this refers to rounding in such a way that the result heads toward zero. For example, +5.0, +5.1, +5.2, +5.3, +5.4, +5.5, +5.6, +5.7, +5.8, and +5.9 will all be rounded to +5 (this works the same for odd and even values, so +6.0 through +6.9 will all be rounded to +6):



Similarly, -5.0, -5.1, -5.2, -5.3, -5.4, -5.5, -5.6, -5.7, -5.8, and -5.9 will all be rounded to -5 (again, this works the same for odd and even values, so -6.0 through -6.9 will all be rounded to -6):



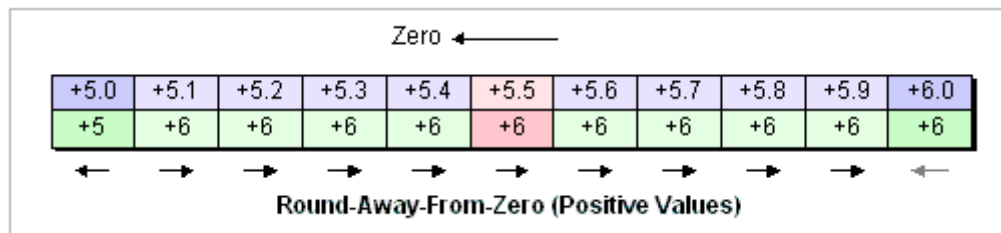
Another way to think about this form of rounding is that it acts in the same way as the **Round-Floor** algorithm for positive numbers and as the **Round-Ceiling** algorithm for negative numbers.

[Top](#)

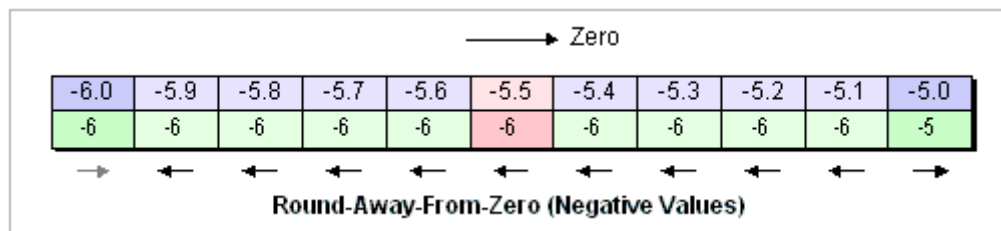
## Round-Away-From-Zero



This is the counterpart to the **Round-Toward-Zero** algorithm introduced above; the difference being that in this case we round away from zero. For example, +5.1, +5.2, +5.3, +5.4, +5.5, +5.6, +5.7, +5.8, and +5.9 will all be rounded to +6 (this works the same for odd and even values, so +6.1 through +6.9 will all be rounded to +7):



Similarly, -5.1, -5.2, -5.3, -5.4, -5.5, -5.6, -5.7, -5.8, and -5.9 will all be rounded to -6 (again, this works the same for odd and even values, so -6.1 through -6.9 will all be rounded to -7):



Another way to think about this form of rounding is that it acts in the same way as the **Round-Ceiling** algorithm for positive numbers and as the **Round-Floor** algorithm for negative numbers.

[Top](#)

## Round-Up

The actions of this rounding mode depend on what one means by "up". Some applications understand "up" to refer to heading towards positive infinity; in this case, *round-up* acts the same way as the **Round-Ceiling** algorithm. Alternatively, some applications regard "up" as referring to an absolute value heading away from zero; in this case, *round-up* acts in the same manner as the **Round-Away-From-Zero** algorithm.

[Top](#)

## Round-Down

This is the counterpart to the **Round-Up** algorithm introduced above. As you might expect by now, the actions of this mode depend on what one means by "down". Some applications understand "down" to refer to heading towards negative infinity; in this case, *round-down* acts the same way as the **Round-Floor** algorithm. Alternatively, some applications regard "down" as referring to an absolute value heading toward zero; in this case, *round-down* acts in the same manner as the **Round-Toward-Zero** algorithm.

[Top](#)

## Truncation (Chopping)

Also known as *chopping*, truncation simply means discarding any unwanted digits. This means that in the case of the standard (sign-magnitude) decimal values we've been considering thus far – and also when working with unsigned binary values – the actions of truncation are identical to those of the **Round-Toward-Zero** algorithm.

With regard to the previous point, as a rule of thumb, when working with software applications that are inputting, manipulating, and displaying decimal values, performing a truncation operation on -3.5 will return -3 when working with most programming languages.

But things are never simple; when working with signed binary values in hardware implementations of accelerator and digital signal processing (DSP) functions, the actions of truncation typically reflect those of the **Round-Floor** algorithm (that is, truncating -3.5 will result in -4, for example). See also the discussions on **Rounding Sign-Magnitude Binary Numbers** and **Rounding Signed Binary Numbers** later in this paper.

[Top](#)

---

### Round-Alternate

Also known as *alternate rounding*, this is similar in concept to the **Round-Half-Even** and **Round-Half-Odd** schemes discussed earlier, in that the purpose of this algorithm is to minimize the bias that can be caused by always rounding “half-way” values in the same direction.

In the case of the **Round-Half-Even** algorithm, for example, it would be possible for a bias to occur if the data being processed contained a disproportionate number of odd and even “half-way” values. One solution is to use the *round-alternate* approach, in which the first “half-way” value is rounded up (for example), the next is rounded down, the next up, the next down, and so on.

[Top](#)

---

### Round-Random (Stochastic Rounding)

This may also be referred to as *random rounding* or *stochastic rounding*, where the term “stochastic” comes from the Greek *stokhazesthai*, meaning “to guess at.” In this case, when the algorithm is presented with a “half-way” value, it effectively tosses a metaphorical coin in the air and randomly (or pseudo-randomly) rounds the value up or down.

Although this technique typically gives the best overall results over large numbers of calculations, it is only employed in very specialized applications, because the nature of this algorithm makes it difficult to implement and tricky to verify the results.

[Top](#)

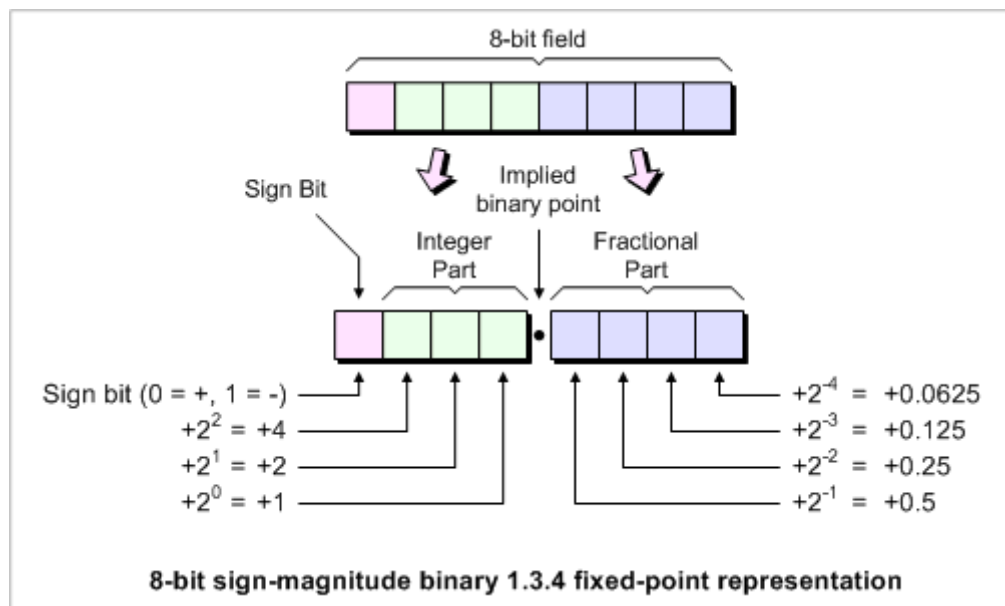
---

### Rounding Sign-Magnitude Binary Values

All of our previous discussions have been based on our working with standard sign-magnitude decimal values. Now let's consider what happens to our rounding algorithms in the case of binary representations. For the purposes of these discussions, we will focus on the *truncation* and *round-*

*half-up* algorithms, because these are the techniques commonly used in hardware implementations constructed out of physical logic gates (in turn, this is because these algorithms entail relatively little overhead in terms of additional logic).

We'll start by considering an 8-bit binary sign-magnitude fixed-point representation comprising a sign bit, three integer bits, and four fractional bits:

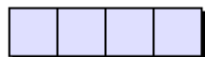


Note that the differences between unsigned, sign-magnitude, and signed binary numbers are introduced in our book *How Computers Do Math* (ISBN: 0471732788). For our purposes here, we need only note that – in the case of a sign-magnitude representation – the sign bit is used only to represent the sign of the value (0 = positive, 1 = negative). In the case of this particular example, we have three integer bits that can be used to represent an integer in the range 0 to 7. Thus, the combination of the sign bit and the three integer bits allows us to represent positive and negative integers in the range -7 through +7:





	Sign Bit	Integer Bits	
			Implied binary point
+0	0	0 0 0	0
+1	0	0 0 0	1
+2	0	0 0 1	0
+3	0	0 0 1	1
+4	0	0 1 0	0
+5	0	0 1 0	1
+6	0	0 1 1	0
+7	0	0 1 1	1
-0	1	0 0 0	0
-1	1	0 0 0	1
-2	1	0 0 1	0
-3	1	0 0 1	1
-4	1	0 1 0	0
-5	1	0 1 0	1
-6	1	0 1 1	0
-7	1	0 1 1	1
Decimal		Binary	

As we know (at least, we know if we've read the book {hint, hint}), one of the problems with this format is that we end up with both positive and negative versions of zero, but that need not concern us here.

When we come to the fractional portion of our 8-bit field, the first fractional bit is used to represent  $1/2 = 0.5$ ; the second fractional bit is used to represent  $0.5/2 = 0.25$ ; the third fractional bit is used to represent  $0.25/2 = 0.125$ , and the fourth fractional bit is used to represent  $0.125/2 = 0.0625$ . Thus, our four bit field allows us to represent fractional values in the range 0.0 through 0.9375 as shown below (of course, more fractional bits would allow us to represent more precise fractional values, but four bits will serve the purposes of our discussions):

Fractional Part										
Implied binary point										
	0	0	0	0	=	0.0	+ 0.00	+ 0.000	+ 0.0000	= +0.0000
	0	0	0	1	=	0.0	+ 0.00	+ 0.000	+ 0.0625	= +0.0625
	0	0	1	0	=	0.0	+ 0.00	+ 0.125	+ 0.0000	= +0.1250
	0	0	1	1	=	0.0	+ 0.00	+ 0.125	+ 0.0625	= +0.1875
	0	1	0	0	=	0.0	+ 0.25	+ 0.000	+ 0.0000	= +0.2500
	0	1	0	1	=	0.0	+ 0.25	+ 0.000	+ 0.0625	= +0.3125
	0	1	1	0	=	0.0	+ 0.25	+ 0.125	+ 0.0000	= +0.3750
	0	1	1	1	=	0.0	+ 0.25	+ 0.125	+ 0.0625	= +0.4375
	1	0	0	0	=	0.5	+ 0.00	+ 0.000	+ 0.0000	= +0.5000
	1	0	0	1	=	0.5	+ 0.00	+ 0.000	+ 0.0625	= +0.5625
	1	0	1	0	=	0.5	+ 0.00	+ 0.125	+ 0.0000	= +0.6250
	1	0	1	1	=	0.5	+ 0.00	+ 0.125	+ 0.0625	= +0.6875
	1	1	0	0	=	0.5	+ 0.25	+ 0.000	+ 0.0000	= +0.7500
	1	1	0	1	=	0.5	+ 0.25	+ 0.000	+ 0.0625	= +0.8125
	1	1	1	0	=	0.5	+ 0.25	+ 0.125	+ 0.0000	= +0.8750
	1	1	1	1	=	0.5	+ 0.25	+ 0.125	+ 0.0625	= +0.9375
Binary				Decimal						

**Truncation:** So, now let's suppose that we have a sign-magnitude binary value of 0101.0100, which equates to +5.25 in decimal. If we simply truncate this by removing the fractional field, we end up with an integer value of 0101 in binary, or +5 in decimal, which is what we would expect. Similarly, if we started with a value of 1101.0100, which equates to -5.25 in decimal, then truncating the fractional part leaves us with an integer value of 1101 in binary, or -5 in decimal, which – again – is what we expect. Let's try this with some other values as follows:

	Initial Binary value		Truncate
Positive Values	0 1 0 1 . 0 0 0 0 = +5.00		0 1 0 1 = +5
	0 1 0 1 . 0 1 0 0 = +5.25		0 1 0 1 = +5
	0 1 0 1 . 1 0 0 0 = +5.50		0 1 0 1 = +5
	0 1 0 1 . 1 1 0 0 = +5.75		0 1 0 1 = +5
	0 1 1 0 . 0 0 0 0 = +6.00		0 1 1 0 = +6
	0 1 1 0 . 0 1 0 0 = +6.25		0 1 1 0 = +6
	0 1 1 0 . 1 0 0 0 = +6.50		0 1 1 0 = +6
	0 1 1 0 . 1 1 0 0 = +6.75		0 1 1 0 = +6
Negative Values	1 1 0 1 . 0 0 0 0 = -5.00		1 1 0 1 = -5
	1 1 0 1 . 0 1 0 0 = -5.25		1 1 0 1 = -5
	1 1 0 1 . 1 0 0 0 = -5.50		1 1 0 1 = -5
	1 1 0 1 . 1 1 0 0 = -5.75		1 1 0 1 = -5
	1 1 1 0 . 0 0 0 0 = -6.00		1 1 1 0 = -6
	1 1 1 0 . 0 1 0 0 = -6.25		1 1 1 0 = -6
	1 1 1 0 . 1 0 0 0 = -6.50		1 1 1 0 = -6
	1 1 1 0 . 1 1 0 0 = -6.75		1 1 1 0 = -6

The end result is that, as we previously noted, in the case of sign-magnitude binary numbers, using truncation (simply discarding the fractional bits) is the same as performing the **Round-Toward-Zero** algorithm as discussed earlier in this paper. Also, as we see, this works exactly the same way for both positive and negative (and odd and even) values. (Compare these results to those obtained when performing this operation on signed binary values as discussed in the next section.)

**Round-Half-Up:** In addition to truncation, the other common rounding algorithm used in hardware implementations is that of **Round-Half-Up**. The reason this is so popular (as compared to a **Round-Half-Even** approach, for example) is that it doesn't require us to perform any form of comparison; all we have to do is to add 0.5 to our original value and then truncate the result. For example, let's suppose that we have a binary value of 0101.1100, which equates to +5.75 in decimal. If we now add 0000.1000 (which equates to 0.5 in decimal) we end up with 0110.0100, which equates to +6.25 in decimal. And if we then truncate this value, we end up with 0110, or +6 in decimal, which is exactly what we would expect from a **Round-Half-Up** algorithm. Let's try this with some other values as follows:

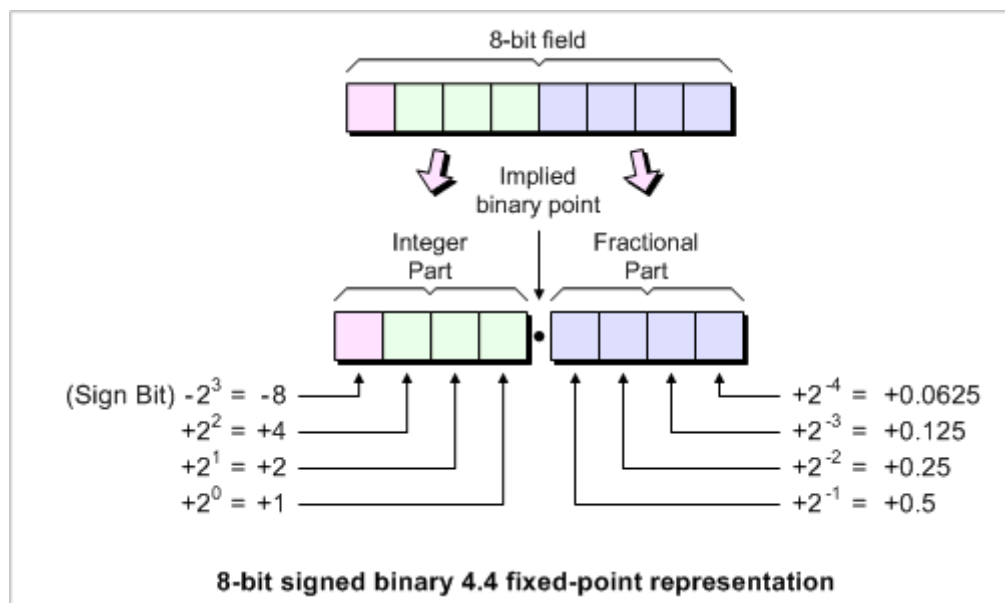
Initial Binary value		Add 0000.1000 (0.5 in Decimal)		Truncate
0 1 0 1 . 0 0 0 0 = +5.00		0 1 0 1 . 1 0 0 0 = +5.50		0 1 0 1 = +5
0 1 0 1 . 0 1 0 0 = +5.25	→	0 1 0 1 . 1 1 0 0 = +5.75	→	0 1 0 1 = +5
0 1 0 1 . 1 0 0 0 = +5.50		0 1 1 0 . 0 0 0 0 = +6.00		0 1 1 0 = +6
0 1 0 1 . 1 1 0 0 = +5.75		0 1 1 0 . 0 1 0 0 = +6.25		0 1 1 0 = +6
0 1 1 0 . 0 0 0 0 = +6.00		0 1 1 0 . 1 0 0 0 = +6.50		0 1 1 0 = +6
0 1 1 0 . 0 1 0 0 = +6.25	→	0 1 1 0 . 1 1 0 0 = +6.75	→	0 1 1 0 = +6
0 1 1 0 . 1 0 0 0 = +6.50		0 1 1 1 . 0 0 0 0 = +7.00		0 1 1 1 = +7
0 1 1 0 . 1 1 0 0 = +6.75		0 1 1 1 . 0 1 0 0 = +7.25		0 1 1 1 = +7
1 1 0 1 . 0 0 0 0 = -5.00		1 1 0 1 . 1 0 0 0 = -5.50		1 1 0 1 = -5
1 1 0 1 . 0 1 0 0 = -5.25	→	1 1 0 1 . 1 1 0 0 = -5.75	→	1 1 0 1 = -5
1 1 0 1 . 1 0 0 0 = -5.50		1 1 1 0 . 0 0 0 0 = -6.00		1 1 1 0 = -6
1 1 0 1 . 1 1 0 0 = -5.75		1 1 1 0 . 0 1 0 0 = -6.25		1 1 1 0 = -6
1 1 1 0 . 0 0 0 0 = -6.00		1 1 1 0 . 1 0 0 0 = -6.50		1 1 1 0 = -6
1 1 1 0 . 0 1 0 0 = -6.25	→	1 1 1 0 . 1 1 0 0 = -6.75	→	1 1 1 0 = -6
1 1 1 0 . 1 0 0 0 = -6.50		1 1 1 1 . 0 0 0 0 = -7.00		1 1 1 1 = -7
1 1 1 0 . 1 1 0 0 = -6.75		1 1 1 1 . 0 1 0 0 = -7.25		1 1 1 1 = -7

Thus, the end result is that, in the case of sign-magnitude binary numbers, adding a value of 0.5 and truncating the product gives exactly the same results as performing a *symmetrical* version of the **Round-Half-Up** algorithm as discussed earlier in this paper. As we would expect from the symmetrical version of this algorithm, this works exactly the same way for both positive and negative (and odd and even) values. (Compare these results to those obtained when performing this operation on signed binary values as discussed in the next section.)

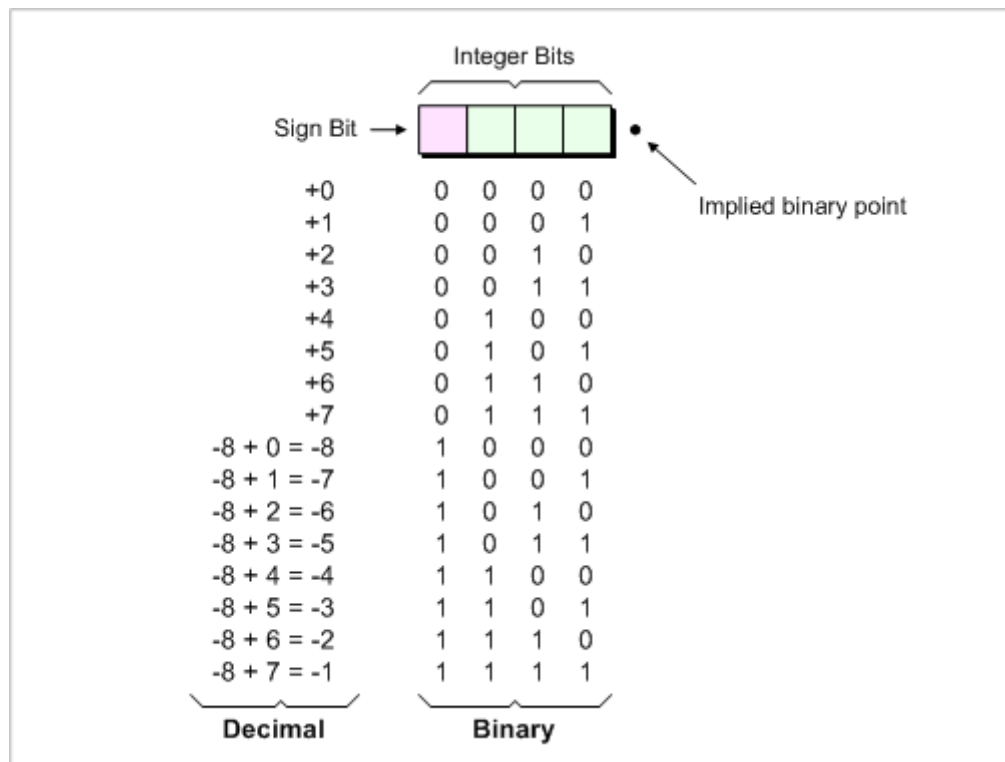
[Top](#)

## Rounding Signed Binary Values

For this portion of our discussions, we will base our examples on an 8-bit signed binary fixed-point representation comprising a four integer bits (the most-significant of which also acts as a sign bit) and four fractional bits:



Once again, the differences between unsigned, sign-magnitude, and signed binary numbers are introduced in our book *How Computers Do Math* (ISBN: 0471732788). For our purposes here, we need only note that – in the case of a signed binary representation – the sign bit is used to signify a negative *quantity* (not just the sign), while the remaining bits continue to represent positive values. Thus, in the case of our 4-bit integer field, a "1" in the sign bit reflects a value of  $-2^3 = -8$ , which therefore allows us to use this 4-bit field to represent integers in the range -8 through +7:



The fractional portion of our 8-bit field behaves in exactly the same manner as for the sign-magnitude representations we discussed in the previous section; the first fractional bit is used to represent  $1/2 = 0.5$ ; the second fractional bit is used to represent  $0.5/2 = 0.25$ ; the third fractional bit is used to represent  $0.25/2 = 0.125$ , and the fourth fractional bit is used to represent  $0.125/2 = 0.0625$ . Thus, our four bit field allows us to represent fractional values in the range 0.0 through 0.9375 as shown below (once again, more fractional bits would allow us to represent more precise fractional values, but four bits will serve the purposes of our discussions):



Fractional Part							
Implied binary point							
	0	0	0	0	=	0.0	+ 0.00 + 0.000 + 0.0000 = +0.0000
	0	0	0	1	=	0.0	+ 0.00 + 0.000 + 0.0625 = +0.0625
	0	0	1	0	=	0.0	+ 0.00 + 0.125 + 0.0000 = +0.1250
	0	0	1	1	=	0.0	+ 0.00 + 0.125 + 0.0625 = +0.1875
	0	1	0	0	=	0.0	+ 0.25 + 0.000 + 0.0000 = +0.2500
	0	1	0	1	=	0.0	+ 0.25 + 0.000 + 0.0625 = +0.3125
	0	1	1	0	=	0.0	+ 0.25 + 0.125 + 0.0000 = +0.3750
	0	1	1	1	=	0.0	+ 0.25 + 0.125 + 0.0625 = +0.4375
	1	0	0	0	=	0.5	+ 0.00 + 0.000 + 0.0000 = +0.5000
	1	0	0	1	=	0.5	+ 0.00 + 0.000 + 0.0625 = +0.5625
	1	0	1	0	=	0.5	+ 0.00 + 0.125 + 0.0000 = +0.6250
	1	0	1	1	=	0.5	+ 0.00 + 0.125 + 0.0625 = +0.6875
	1	1	0	0	=	0.5	+ 0.25 + 0.000 + 0.0000 = +0.7500
	1	1	0	1	=	0.5	+ 0.25 + 0.000 + 0.0625 = +0.8125
	1	1	1	0	=	0.5	+ 0.25 + 0.125 + 0.0000 = +0.8750
	1	1	1	1	=	0.5	+ 0.25 + 0.125 + 0.0625 = +0.9375
Binary				Decimal			

**Truncation:** So, now let's suppose that we have a signed binary value of 0101.1000, which equates to +5.5 in decimal. If we simply truncate this by removing the fractional field, we end up with an integer value of 0101 in binary, or +5 in decimal, which is what we would expect.

However, now consider what happens if we wish to perform the same operation on the equivalent negative value of -5.5. In this case, our binary value will be 1010.1000, which equates to  $-8 + 2 + 0.5 = -5.5$  in decimal. Thus, truncating the fractional part leaves us with an integer value of 1010 in binary, or -6 (as opposed to the -5 we received in the case of the sign-magnitude representations in the previous section). Let's try this with some other values as follows:

Initial Binary value				Truncate	
Positive Values	0 1 0 1 . 0 0 0 0	=	+5.00	→	0 1 0 1 = +5
	0 1 0 1 . 0 1 0 0	=	+5.25		0 1 0 1 = +5
	0 1 0 1 . 1 0 0 0	=	+5.50		0 1 0 1 = +5
	0 1 0 1 . 1 1 0 0	=	+5.75		0 1 0 1 = +5
	0 1 1 0 . 0 0 0 0	=	+6.00	→	0 1 1 0 = +6
	0 1 1 0 . 0 1 0 0	=	+6.25		0 1 1 0 = +6
	0 1 1 0 . 1 0 0 0	=	+6.50		0 1 1 0 = +6
	0 1 1 0 . 1 1 0 0	=	+6.75		0 1 1 0 = +6
Negative Values	1 0 1 1 . 0 0 0 0	= $-8 + 3 + 0.00$	= -5.00	→	1 0 1 1 = -5
	1 0 1 0 . 1 1 0 0	= $-8 + 2 + 0.75$	= -5.25		1 0 1 0 = -6
	1 0 1 0 . 1 0 0 0	= $-8 + 2 + 0.50$	= -5.50		1 0 1 0 = -6
	1 0 1 0 . 0 1 0 0	= $-8 + 2 + 0.25$	= -5.75		1 0 1 0 = -6
	1 0 1 0 . 0 0 0 0	= $-8 + 2 + 0.00$	= -6.00	→	1 0 1 0 = -6
	1 0 0 1 . 1 1 0 0	= $-8 + 1 + 0.75$	= -6.25		1 0 0 1 = -7
	1 0 0 1 . 1 0 0 0	= $-8 + 1 + 0.50$	= -6.50		1 0 0 1 = -7
	1 0 0 1 . 0 1 0 0	= $-8 + 1 + 0.25$	= -6.75		1 0 0 1 = -7

Observe the values shown in red/bold and compare these values to those obtained when performing this operation on sign-magnitude binary values as discussed in the previous section. The end result is that, as we previously noted, in the case of signed binary numbers, using truncation (simply discarding the fractional bits) is the same as performing the **Round-Floor** algorithm as discussed earlier in this paper.

**Round-Half-Up:** As we mentioned earlier, the other common rounding algorithm used in hardware implementations is that of **Round-Half-Up**. Once again, the reason this algorithm is so popular (as compared to a **Round-Half-Even** approach, for example) is that it doesn't require us to perform any form of comparison; all we have to do is to add 0.5 to our original value and then truncate the result.

As you will doubtless recall, in the case of positive values, we expect this algorithm to round to the next integer for fractional values of 0.5 and higher. For example, +5.5 should round to +6. Let's see if this works. If we have an initial signed binary value of 0101.1000 (which equates to +5.5 in decimal) and we add 0000.1000 (which equates to 0.5 in decimal) we end up with 0110.0000, which equates to +6.0 in decimal. And if we now truncate this value, we end up with 0110 or +6 in decimal, which is what we would expect.

But what about negative values? If we have an initial value of 1010.1000 (which equates to  $-8 + 2 + 0.5 = -5.5$  in decimal) and we add 0000.1000 (which equates to 0.5 in decimal) we end up with 1011.0000, which equates to  $-8 + 3 = -5$  in decimal. If we then truncate this value, we end up with 1011 or -5 in decimal, as opposed to the value of -6 that we might have hoped for. Just to make the point a little more strongly, let's try this with some other values as follows:

Initial Binary value		Add 0000.1000 (0.5 in Decimal)		Truncate
0 1 0 1 . 0 0 0 0 = +5.00		0 1 0 1 . 1 0 0 0 = +5.50		0 1 0 1 = +5
0 1 0 1 . 0 1 0 0 = +5.25	→	0 1 0 1 . 1 1 0 0 = +5.75	→	0 1 0 1 = +5
0 1 0 1 . 1 0 0 0 = +5.50		0 1 1 0 . 0 0 0 0 = +6.00		0 1 1 0 = +6
0 1 0 1 . 1 1 0 0 = +5.75		0 1 1 0 . 0 1 0 0 = +6.25		0 1 1 0 = +6
0 1 1 0 . 0 0 0 0 = +6.00		0 1 1 0 . 1 0 0 0 = +6.50		0 1 1 0 = +6
0 1 1 0 . 0 1 0 0 = +6.25	→	0 1 1 0 . 1 1 0 0 = +6.75	→	0 1 1 0 = +6
0 1 1 0 . 1 0 0 0 = +6.50		0 1 1 1 . 0 0 0 0 = +7.00		0 1 1 1 = +7
0 1 1 0 . 1 1 0 0 = +6.75		0 1 1 1 . 0 1 0 0 = +7.25		0 1 1 1 = +7
1 0 1 1 . 0 0 0 0 = -5.00		1 0 1 1 . 1 0 0 0 = -4.50		1 0 1 1 = -5
1 0 1 0 . 1 1 0 0 = -5.25	→	1 0 1 1 . 0 1 0 0 = -4.75	→	1 0 1 1 = -5
1 0 1 0 . 1 0 0 0 = -5.50		1 0 1 1 . 0 0 0 0 = -5.00		1 0 1 1 = <b>-5</b>
1 0 1 0 . 0 1 0 0 = -5.75		1 0 1 0 . 1 1 0 0 = -5.25		1 0 1 0 = -6
1 0 1 0 . 0 0 0 0 = -6.00		1 0 1 0 . 1 0 0 0 = -5.50		1 0 1 0 = -6
1 0 0 1 . 1 1 0 0 = -6.25	→	1 0 1 0 . 0 1 0 0 = -5.75	→	1 0 1 0 = -6
1 0 0 1 . 1 0 0 0 = -6.50		1 0 1 0 . 0 0 0 0 = -6.00		1 0 1 0 = <b>-6</b>
1 0 0 1 . 0 1 0 0 = -6.75		1 0 0 1 . 1 1 0 0 = -6.25		1 0 0 1 = -7

As we see, in the case of signed binary numbers, adding a value of 0.5 and truncating the product gives exactly the same results as performing an *asymmetrical* version of the **Round-Half-Up** algorithm as discussed earlier in this paper. As we would expect from the asymmetrical version of this algorithm, this works differently for positive and negative values that fall exactly on the "half-way" (0.5) boundary. (Compare these results to those obtained when performing this operation on signed binary values as discussed in the previous section.)

## Summary and Further Reading

The following table reflects a summary of the main rounding modes discussed above as applied to standard (sign-magnitude) decimal values:

	← Negative Infinity								Zero	← Zero				Positive Infinity →							
Mode	-2.0	-1.7	-1.5	-1.3	-1.0	-0.7	-0.5	-0.3	0.0	0.3	0.5	0.7	1.0	1.3	1.5	1.7	2.0				
R-H-U (s)	-2	-2	-2	-1	-1	-1	-1	-0	0	0	1	1	1	1	2	2	2				
R-H-U (a)	-2	-2	-1	-1	-1	-1	-0	-0	0	0	1	1	1	1	2	2	2				
R-H-D (s)	-2	-2	-1	-1	-1	-1	-0	-0	0	0	0	1	1	1	1	2	2				
R-H-D (a)	-2	-2	-2	-1	-1	-1	-1	-0	0	0	0	1	1	1	1	2	2				
R-H-E	-2	-2	-2	-1	-1	-1	-0	-0	0	0	0	1	1	1	2	2	2				
R-H-O	-2	-2	-1	-1	-1	-1	-1	-0	0	0	1	1	1	1	1	2	2				
R-C	-2	-1	-1	-1	-1	-0	-0	-0	0	1	1	1	1	2	2	2	2				
R-F	-2	-2	-2	-2	-1	-1	-1	-1	0	0	0	0	1	1	1	1	2				
R-T-Z	-2	-1	-1	-1	-1	-0	-0	-0	0	0	0	0	1	1	1	1	2				
R-AF-Z	-2	-2	-2	-2	-1	-1	-1	-1	0	1	1	1	1	2	2	2	2				

R-H-U = Round-Half-Up  
R-H-E = Round-Half-Even  
R-C = Round-Ceiling  
R-T-Z = Round-Toward-Zero

R-H-D = Round-Half-Down  
R-H-O = Round-Half-Odd  
R-F = Round-Floor  
R-AF-Z = Round-Away-From-Zero

(s) = Symmetric  
(a) = Asymmetric

With regard to the above illustration, it's important to remember the following points (all of which are covered in more detail in our earlier discussions):

- The generic concept of **Round-Toward-Nearest** encompasses both the **Round-Half-Up** and **Round-Half-Down** modes.
- The term "arithmetic rounding" is another name for the **Round-Half-Up** algorithm (of which there can be both symmetric and asymmetric versions).
- Depending on the application/implementation, the actions of the **Round-Up** algorithm may either equate to those of the **Round-Ceiling** mode or to those of the **Round-Away-From-Zero** mode.
- Depending on the application/implementation, the actions of the **Round-Down** algorithm may either equate to those of the **Round-Floor** mode or to those of the **Round-Toward-Zero** mode.
- In the case of sign-magnitude or unsigned binary numbers, the actions of **Truncation** are identical to those of the **Round-Toward-Zero** mode. By comparison, in the case of signed binary numbers, the actions of **Truncation** are identical to those of the **Round-Floor** mode.

There are many additional resources available to anyone who is interested in knowing more about this &ndash; and related – topics. The following should provide some "food for thought," and please feel free to let us know of any other items you feel should be included in this list:

- 1) Mike Cowlshaw, an IBM fellow, has created an incredibly useful site that introduces a fantastic range of topics – including rounding – pertaining to the use of **Decimal Arithmetic** in computers. Also included are links to related resources.
- 2) David Goldberg has created a tremendously useful site entitled *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, which, you may not be surprised to learn, provides a great introduction to **Floating-Point** representations and operations (including rounding algorithms, of course).
- 3) Employed by many computers, microprocessors, and hardware accelerators, IEEE 754 is the most widely-used standard for floating-point arithmetic. Published by the Institute of Electrical and Electronic Engineers (**IEEE**), this standard covers all aspects of floating-point computation, including a number of rounding and truncation schemes. The related IEEE 854 standard generalizes the original IEEE 754 to cover decimal arithmetic as well as binary.
- 4) With regard to the previous point, a useful overview to the IEEE 754 floating-point representation is provided by **Steve Hollasch**. Also, a useful tool for performing IEEE 754 conversions from decimal floating-point to their 32-bit and 64-bit hexadecimal counterparts is available from **Queens College** in New York.
- 5) Regarded by many as being a definitive reference, volume 2 of the classic *Art of Computer Programming* by Donald Knuth focuses on **Seminumerical Algorithms** and covers a wide range of topics from random number generators to floating point operations and rounding techniques.
- 6) Last but certainly not least, this entire paper was spawned from just a single topic in our book *How Computers Do Math* (ISBN: 0471732788). If you've found this paper to be useful and interesting, just imagine how much fun you'd have reading the book itself!

[Top](#)