**Input:**

A signed m-bit integer. (a fixed point number is just a shifted integer and unsigned m-bit integer inputs can also be treated as a signed bit integers but with the sign bit set to constant ).

**Output:**

*𝑝* where are generic constants and is some value where is a large enough signed number to represent all possible outputs given the input (so this will be some function of and the rounding mode, so can be found at design time). Note that any rational value (a set denoted by the symbol ) can be represented by these constraints on .  
The rounding mode can be set (as a generic or another input (say )) to any of RTU (round to nearest, tie round up), RTNI (round to negative infinity – same as floor rounding) or RTPI (round to positive infinity – same as ceiling rounding). To someone skilled in the art, the below can be altered to fit any rounding mode, where the rounding direction is independent of what two representable values the exact answer is located between.

**Algorithm/Implementation:**

1. If a and and are not ‘coprime’ (they contain common factors), these can be removed and the values of and can be redefined. If , then trivially for all inputs and ‘sensible’ rounding choices and , which requires no hardware logic.  
   **Therefore, from here we can assume that are coprime and .**
2. This next calculation will show that the binary expansion of always takes the form:  
     
     
     
   For some and (left padded with 0’s to be width (also derived below)) and the binary point can be located somewhere along the infinite sequence (including above B).  
     
   First check for the ‘eveness’ of – how many times can it be divided by 2 until you get an odd number.  
   So there will exist a such that where is odd or, equivalently, coprime to 2, which will be important later.  
   Therefore, we have where and , so .   
   Since is odd, there’s a well-known result that says that there will exist such that is an integer multiple of : , so .   
   (For , this is trivial since = 0, so can just be set to 0, and (for neatness with the proofs to follow below, assume in this case)). However for , this comes from the fact that 2 is coprime to so is a member of the multiplication group modulo . This coprime set can be shown to be a finite group, so 2 will have a finite order (number of multiplications by itself) in this group, such that = 1 (1 is the multiplicative group identity element). (The value is typically chosen to be the smallest positive integer greater than 0 where this equality occurs – it’s a factor of the totient function ( where is the prime factorisation of ) which is the size of the multiplicative group. However, can be used if below, if finding the smallest factor requires too much effort).   
   This implies there exists integer such that as required).  
     
   Note that is an bit unsigned integer since is a purely fractional number, so forms the infinitely recurring blocks of length in the above binary expansion form for .  
   **Hence**

B

A

A

A

...

………

1. The aim now is to convert this infinite binary expansion into a canonical signed digit (CSD) form. The reason for this is to reduce the number of rows in the final multiplication array. This is akin to how a multiplication of would be performed in an addition array. The binary form of 15 is , so multiplying by involves an addition array with 4 rows, but instead expanding 15 in CSD form: (where here stands for -1) and using this in a CSD array only involves 2 additions, where one row requires negating, but this can be done fairly cheaply via , logically negating (very cheap in hardware) followed by an increment, which can be merged very cheaply into the whole array addition.  
     
   In binary expansions there are only 2 states {0,1} (an S can be present if the binary value is a signed number, but only in the msb) and in CSD there are 3 {S,0,1} where I use the s to stand for -1 to neaten the notation (rather than having –‘s everywhere).   
   Converting a finite binary expansion into a finite CSD expansion involves working up from the least significant bit (lsb) in an iterative process to the left, until only 0s remain above the most significant bit (msb). Keep travelling along the value from the lsb until you hit a consecutive group of 1s or Ss (a group must have size greater than 1).If a set of 1s, this can be changed to an S followed by 0s followed by a 1. If the overflow 1 overlaps an S, then these annihilate each other. For a groups of Ss, the same is true except swapping the Ss and 1s over. If no group is found, we must have The algorithm then continues form the position of the overflow bit of the group found.  
   Note that CSD forms always have at least 1 0 on either side of a 1 or S, making them sparser expansions than binary expansions and ideal for building shallow (area/time saving) constant multiplication arrays. Their value can also very easily be negated by performing the swap 1 and they can be shown to be unique for any finite (terminating) binary expansion.   
   All this should be well known but message me if you want a proof.  
     
   e.g. (the red CSD values show the progression of the above rules from lsb to msb).
2. Unfortunately   
     
     
     
   (with the binary point located places above the red circled boundary between the and values – this can include entirely above the value ), is an infinite fixed point value, so there’s no lsb to start the binary to CSD algorithm at, so a different approach is needed to convert a binary expanded rational number into CSD (assuming this is even possible!).  
     
   For now, let’s just look at the value which is all the values above with all the green circled boundaries (appropriately shifted). Note this value can also be written as where is the logical negation of the n-bit unsigned value .   
   The following proves that at least one of the CSD expansions of or glued together indefinitely to the right will give a valid CSD form i.e. all 1s and s surrounded by at least one 0 on each side.

A

A

A

B

...

1. This proof unfortunately has many cases and I haven’t found a nice simple way of surmising them yet.  
     
   First case, the CSD of unsigned n-bit , , when glued together gives a valid CSD form (e.g. the final step of the CSD algorithm doesn’t make become an bit number and there are no touching 1s or Ss at the boundaries of the values (msb and lsb). If this is so, then we’re done.  
     
   Otherwise, must either have a 1 overflowing into the column, causing overlap when we glue these values together (call this case 2a) or there’s no overlap, but the lsb and msb of are both either 1 or S, so they aren’t isolated by 0s when the n-bit s are glued together (call this case 2b).  
     
   Note that ) where is an -bit number still since , so will also be a number (with a possible 1 overflow into the position.  
     
   For case 2a, we have that is an bit CSD value with top two bits 1 and 0 (since 1s need are isolated in CSD form). Therefore will also be an bit CSD value with top two bits 1 and 0. This is because but is still an bit binary number. It’s also guaranteed to be bits if is because the largest bit number in CSD form is ( odd) or 1010101..10 ( even) and the smallest bit number is 10S0S0S0S..0S0 ( odd) or 10S0S0S..S0S ( even). So is guaranteed to overflow because it’s the CSD of whose CSD doesn overflow to bits.  
   , so the top two bits of the CSD go from 10 to 00 since the S cancels the 1 in the position. Therefore, can be represented by less than CSD bits, so can be glued together in bit blocks and still retain a CSD form.  
     
   For case 2b, we have that is an bit CSD with 1 as an msb (it can’t be S because ) and an lsb which isn’t 0 (e.g. 1 or S), which implies is an odd number. Therefore is even, so will have a 0 lsb.  
   If is still remains an bit CSD with a 1 msb, the addition of will transform this 1 into an in the position of . This will be isolated below since is in CSD form and also above (when the bit blocks are glued together) since the the lsb is 0 since is even.  
   If happens to overflow to a bit CSD this will be exactly cancelled by and will be representable by less than CSD bits, so can be glued together in bit blocks and still retain a CSD form.  
     
   This completes the proof.  
   Depending on the value of , it can be easy to pick either (if ) or (if ) otherwise both need to be checked to see which gives the correct CSD form, which should typically give a final addition array with the least rows.  
     
   (NOTE: when , then and . This is the only case where both and will give valid infinite CSD form expansions   
   (I’m not sure if this is particularly well known – I can give a proof is needed. It comes from the fact that 0. akin to how in binary expansions ).  
   In these cases the designer can make a choice on which to choose. I think typically choosing will give a final truncated array with the same number of rows to reduce but the final 2-addition may not be as wide. This is only an intuition and both forms should be checked to be sure in each instance).
2. We now have that:

where where the are unsigned bit **binary** integers which represent the values of the 1s and Ss (negative 1s) in the CSD expansion **.** Similarly for .

There’s is a slight ‘gotcha’ here with the fact that we cannot expand the whole of as either or in CSD form glue/concatenate this to and then calculate the finite CSD form of in the usual way because then we cannot guarantee that the whole value is in CSD form. The intuition for this is that in the following block diagram below, the red boundary is different from all the green boundaries since even though all have an infinite copy of s on their right and a copy of on their left, the red boundary has a copy of on its left, so we must treat it differently. By stopping a green boundary early as we have in the expansions above, we can still use the above proof to show that taking the finite CSD of  or the finite CSD of and concatenating to the appropriate infinitely recurring or chain we will have a valid CSD expansion for the entirety of .

...

A

A

A

B

So:

where and .

1. To summarise, we have shown that there exists (not necessarily unique) a CSD expansion of any rational number (even when trivially (this was not assumed in the above proof however)).  
   It has the form where , the 1s and -1s are surrounded on each side by at least one 0 and the can all be derived from ( being the length of ). The CSD form looks like:

...

.

Binary point places to the left

Going back to the signed m-bit integer input , this can be written as

, where   
  
where can now be treated as an unsigned m-bit integer, so all bits in can be treated the same as unsigned bits () instead of there being a ‘differently interpreted’ sign bit in the msb.  
The equality can be seen by considering both cases when and and remembering is a signed m-bit value and is unsigned.  
  
(NOTE: if is instead an unsigned bit value, this previous step can be ignored and . Or if you can extend the unsigned bit value to a signed +1 bit value but with , and follow on with the proof below, except with and there will be more constant bits that need to be collected in the final truncated array. This should be clear to someone skilled in the art).

1. We now aim to make a multiplicative array (which will in general extend indefinitely to the right) whose sum is exactly   
     
      
     
     
   where the constant so far (which we will keep adding to in further steps below)   
     
   and we have as before and   
    are both unsigned bit integer values and multiplied by positive values, so every partial product bit in the array can be treated as 0 or a positive 1 (all the sign-bit extension/negative value complication has been shifted into the value of the constant ).
2. We now have an unsigned additive array containing shifted copies of and (e.g. bits of the form and ) with a location for the binary point, plus a constant whose sum is equal to  
   .  
     
   Some of the following observations can be used on arbitrary additive arrays if additional information is known about what exact value the sum can take for all required inputs. However, here we will concentrate on the constructed CSD array whose sum takes values: .  
     
   In order to make this array implementable in hardware, we will need to remove, in general, an infinite amount of bits from the right hand side, (perhaps correcting for this by the addition of a constant , which can in turn be merged into the value of ), in order to leave a finite number of partial product bits to sum. We need to show that this can be done in a way which doesn’t affect the accuracy of the ½ ulp accuracy rounding for .  
     
   Let’s first assume that we are using RTNI rounding to the nearest integer and that we have removed an infinite set of partial product bits (all bits of the array apart from the constant ), whose sum is equal to . This depends on because this sum is a function of the input bits (and their logical negations). since because all the bits are unsigned.  
     
   It’s easy to find inputs and such that and . All that needs to be done is to find the most significant bits of all the in this removed set and check whether they are or . There cannot be two bits of with the same index in the same column, due to the array being made up of copies of and by differing (distinct) amounts.  
   To find the value of , set , if the most significant bit of in the removed group is and if . If (in unusual cases) an bit of doesn’t belong to the removed group, has no effect on (so is effectively don’t care – for simplicity sake set to 0, to make unique).   
   The reason for this is that the most significant value of the of has a higher value than the sum of all those in less significant columns (in any CSD array shape) and that the can take independent values for different values of in , since we’re considering all inputs of signed .  
   To find , use the opposite rules for setting , so in actual fact we have , the logical negation of as an -bit string.  
     
   The value of can easily be calculated since it’s the difference between the exact answer (where is again treated as a signed -bit value) and the sum of finite values of the CSD array including the value of (which may well be infinite but well defined), which can be denoted as . So , and so no nasty infinite summing is required, since can be easily calculated as a multiple of exactly by hand (by calculating ), so given a choice of removed bits which only leave a finite CSD array (including constant **)** behind, it requires only two calculations to find & such that and the bounds are retained.   
     
   Can such a set of partial product bits of and be removed and replaced by a corrective constant , whilst maintain the accuracy demanded by the rounding for calculating , and if so, what is a good heuristic for doing this to make fast/small hardware?
3. For clarity I have split up the three rounding cases RTNI, RTPI and RTU below:  
   A rounding point of a rounding mode is defined to be the point, either side of which the infinitely precise solution will jump to a different consecutive representable number (in our case these are integers).  
     
   When removing bits from a CSD array and adding an additive constant , we are adding . When it then comes to adding the remaining finite array and truncating away the fractional bits (or just selecting the required integer bits for ) we require that neither the addition of or (the two extremes) can alter the intended integer value for .  
     
     
   **RTNI**  
     
   The rounding points for RTNI rounding are the integers.  
     
   Let be the smallest distance above or equal to an integer that a value of can take. This for our use is clearly 0, since implies that (however, for strange use cases/arbitrary arrays this may well be different).  
     
   Let be the smallest distance strictly below or equal to an integer that a value of can take. Typically for us this will be as there will typically exist a value of where , but this may not be the case if .  
     
   We require that:  
     
    and that . This fact can be used to show the two non-bold inequalities are in fact implied by the two **bold** equalities and we have:  
     
     
   so for a value of (the rationals are dense on the real number line) to exist, we must have:  substituting are expected values for and , so we have: and   
     
   **RTPI**  
     
   The rounding points for RTPI rounding are the integers.  
     
   Let be the smallest distance strictly above an integer that a value of can take. Typically for us this will be as there will typically exist a value of where , but this may not be the case if .  
     
   Let be the smallest distance below or equal to an integer that a value of can take. This for our use is clearly 0, since implies that (however, for strange use cases/arbitrary arrays this may well be different).  
     
   We require that:  
     
    and that . This fact can be used to show the two non-bold inequalities are in fact implied by the two **bold** equalities and we have:  
     
     
   so for a value of to exist, we must have:  substituting are expected values for and , so we have: and   
   **.**   
     
   **RTU**  
     
   The rounding points for RTNI rounding are the exact halfway points between the integers.  
     
   Let be the smallest distance above or equal to an integer halfway point that a value of can take.   
   For us this will typically be , since there will most likely be a value of such that , unless perhaps if (however, for strange use cases/arbitrary arrays this may well be different).  
     
   Let be the smallest distance strictly below an integer halfway point that a value of can take.   
   For us this will typically be , since there will most likely be a value of such that , unless perhaps if (however, for strange use cases/arbitrary arrays this may well be different).  
     
   We require that:  
     
    and that . This fact can be used to show the two non-bold inequalities are in fact implied by the two **bold** equalities and we have:  
     
     
   so for a value of to exist, we must have:  substituting are expected values for and , so we have: and   
     
     
   **NOTE:**   
   For arbitrary type of arrays we typically have to conservatively assume that , since we don’t know their values. This means we have , meaning must be a constant for all inputs, which typically means and no truncation can occur (this is because in an arbitrary array, we have no idea how far a carry can propagate to the right).  
   It’s the fact here that for functionality that allows truncation with constant correction to be possible, whilst maintain ½ ulp rounding accuracies (e.g. RTNI, RTPI, RTU).  
     
   **NOTE:**  
   The above was only sufficient and not necessary. Throughout it was assumed that corresponded to a specific, worse case, value of (and similarly with ). So there may remain extra freedom in the amount of truncation/choice of which still maintains the required functionality.
4. All 3 rounding modes we have concentrated on gave the same general relation:  
   (the difference being in the conditions on the correctional constant ).  
     
   A good heuristic is to try and remove a set of partial products bits such that the smallest finite amount remain which obey this equality, as this will make the smallest addition array, which should give the best hardware QoR (in terms of area and delay as the array reduction and final 2-addition are minimised).  
     
   The best way to truncate the CSD array, whilst minimising the strictly monotonic growth of , is by removing lsb columns first until there comes a point where . When this occurs, retain the most significant column just discarded (so now we have ) as required.  
   One can now go further and start removing individual partial product bits from this column. To minimise growth in , choose index bits of to remove which has a different logical negation to the most significant bit of in the removed group and then the value of should also be such that changing its value (whilst keeping all other bits of constant) causes the greatest difference in .   
   Keep repeating this process until, all the index values of in the retained row have the same logical negation as their most significant index members in the removed set of partial product bits – in this case the value of will grow by where is the weight of the retained column.  
     
   If there are equivalent choices in which bit to remove at each point, a sensible choice would be to remove the bit with the index of which is used the most in the retained set of bits in the finite array to reduce fan-out on the input . This typically means (assuming all bits have equal delay/treated as input from a register etc.) removing bits with higher index values (more significant) before lower ones.  
     
   Continue with the above process, as long as and you should achieve the truncation with the least partial products remaining with this inequality satisfied.  
   (This type of ‘ragged’ truncation is covered in another IMG patent for purely unsigned arrays (all bits have the same logical negation). I can provide more detail on why this truncation scheme should be optimal if necessary, but hopefully someone skilled in the art can also derive this from the fact that a CSD array always has adjacent rows shifted by 2 or more).
5. Instead of starting the search for the columns arbitrarily far to the left (and wasting computation) or just trying to guess and hoping to get lucky, we can construct a worse case bound which gives us a worse case column to start this above algorithm at.  
     
   Retaining columns below the position of the binary point, and removing all less significant columns, the value of  can be bounded below by 0 clearly since all bits are unsigned. The maximum value of  can be bounded below , since we cannot have more than one of in the same column (due to the CSD array) and there are of these. If all filled all columns below the column and they could all be 1 (both of which are clear overestimates), this sums to . Therefore we have that , so we just need to find positive integer large enough such that , so suffices to start the searching algorithm (to lower the value of and perform ragged truncation) in step 12.   
   Unfortunately, this algorithm still needs to be done step by step and for large arrays may become computationally expensive.
6. After the ragged truncation has been performed such that , a value of can be found depending on the rounding mode being used and this can be added to the value of to give – our constant to add to the now, finite, hardware implementable array. This may be an unsigned or signed fixed point value which can be sign-extended accordingly such that the correct value for is output ( can easily be calculated as a function of the generics and the signage of ).   
     
   The last snag is that maybe non-terminating and we have something that looks like the following:

………..

Finite array of shifted and

Notice, that the bits to the right of the red vertical line are all positive (lsbs of an unsigned or signed constant) so their value has no effect on the value of the output , so we can again truncate these bits of to give a finite constant value , which when added to the array, give the same correct value for . The array is now entirely accurate and finite, so can be implemented as a custom addition array in hardware (there are a variety of implementations to those skilled in the art).

………..

Finite array of shifted and

1. Assuming the least significant column of the finite array has weight this means that with the maximum amount of truncation we can guarantee that . If not, then we could simply remove another bit from the column and still have , which shows the maximum number of bits was not truncated, contradicting our assumption.   
   This lower bound on , implies a upper and lower bounds on (for each considered rounding mode), hence also on such that , consequently meaning that is either unique or can take the value of 2 consecutive multiples of .  
     
   In rare cases, a value of may have a number of trailing zeroes and a row of the finite array may then be ‘isolated’ and unable to generate any carries with or any other row of the array to affect. In this case, the bits of the array above the trailing zeroes can be removed since they don’t affect the output.   
     
   In other more usual cases, if there are 2 values of , choosing the larger e to add will allow one extra bit from the row to be removed (this can be seen by remembering that the smaller value of also gives an accurate value for for all inputs ). In unusual cases, if ends in a 0 and only one more bit remains in column , this can then also be removed, as it has no way of generating a carry to affect the output . (This is unusual, but can be seen in a worked example given below).  
     
   To summarise, typically, if is unique, no further bits can be removed but if it can take 2 values, taking value means that one more bit can be removed from the column (except in esoteric cases), making the final value for unique. It makes sense that no more freedom in the value of , corresponds to no further freedom in adding to the removed set of partial array bits.

Which case applies is dependent on generics as they dictate the shape and contents of the CSD array and this can only be seen (by me at least) at this late point in the algorithm. This doesn’t contradict our bounds above, as these were based on sufficient, worse case scenarios – the array formed isn’t necessarily minimal (as this extra optimisation (if present) easily shows).  
  
To further reduce the array may be possible but will require formal verification, exhaustive simulation or a superior algorithm I’m not aware of.  
This method of doing rational multiplication should work best for with a long recurring length and large values of (length of input ) compared to using the multiply-add approach to generate arrays.

**Extension:**

The ideas above could be extended by someone skilled in the art to calculate:  
 where , and the are independent inputs of differing sizes and signages and ROUND is RTNI, RTPI, RTU etc.

**Worked Example: for unsigned integer input**

1. By inspection (or use a highest common factor calculating algorithm, like Euler’s algorithm), it can be seen that where 7 & 9 are co-prime.  
   Using the multiply-add algorithm, gives us that (there was freedom in the additive constant , but the ‘cheapest’ in hardware was to add the acceptable value 0). In binary, the CSD form of 101945 is ‘10S00100S00100S001’, giving that   
   This is an array with maximum height 8 (including constant) array with total width 33 (and partial product bits). This is the array we’ll use as a benchmark to compare the new array formed by using a truncated CSD array.  
   (NOTE: this comparison isn’t entirely fair as I haven’t used the freedom in the additive constant to remove any lsbs, however in general, this new method will always give an array with equal to or fewer partial product array bits. Using the freedom in the additive constant and the new truncation techniques above, this array could be reduced to width 22 with 86 partial product bits – however, this was unknown at the time of this previous multiply-add patent).
2. , so group theory guarantees that , which is true since (the value is also in this case the smallest factor of {1,2,3,6} which this holds), so (binary expansion – not in CSD form since the two msb 1’s are touching).
3. which is a 7-bit value, so there’s has been an overflow (indeed ) so this means that doesn’t overflow and can furthermore be glued together in groups of 6 to give in an infinite CSD form. We now have that:
4. The column below the binary point to start the truncation search from has bits of the value where . Every bit less significant than this column can be safely removed in the knowledge that .  
   This is what we expect in this case because , so there exist plenty of values of inputs such that = 8 (e.g. every time ) and (e.g. every time ), thus and and our condition for the corrective constant is: .
5. By looking it at all the columns in the removed set (those with weight and below) we see that when and when . So as expected.
6. Removing another column (the weight column) we get when and when . So .
7. Removing the next most significant column (the weight column), we still find that , so we move onto trying to remove the whole of the column of the finite array.  
   We get when and when . So , therefore we require to keep some bits in the column in order to ensure .
8. It turns out that removing any bit from the column has the same effect (due to the recurring symmetry of the particular CSD array) as increasing the value of by so only 3 can be removed to make . Removing an additional bit would make .  
   It’s probably most sensible to remove and since these are the bits of which are the most significant and are likely to have the most fan-out in the finite truncated CSD array.  
   Doing so gives when and when (so we recover that ).
9. This gives the condition on the corrective constant that:

and adding to to give gives us:  
  
 to just fractional bits (to match the width of the finite CSD array), in this case, we get 2 possible values for :  
  
 and .

1. Choosing , the larger of the two values of , we can remove one of the additional bit from the column or Sticking with our earlier heuristic, we choose as that has the higher index number.  
   This is can be done since by removing it, for the inputs where , it’s as though we are using as our value of on the finite array (before removing ) and for the inputs when , it’s as if we’re effectively using , and we know that using both and will give accurate answers for for all inputs for from the analysis above.
2. Since is an even multiple of (it has a 0 in the column), we have that the is now ‘isolated’ in this column – it’s the only bit that can take a non-zero value and hence can’t generate any carries to the left to affect the value of , so in this unusual case, due to the particular values of and the input being a number, we can also remove this bit, meaning we have removed the entire column, without affecting the output.  
     
   We can now redefine the new, now unique, value of as .
3. This leaves us with our final array with maximum height 7 (one row being the constant), width 21 and with a partial product count of 83, which is fewer (or equal) than that of the multiply-add array (even with the additional truncation due to the freedom in the choice of constant), so we have managed to make a smaller additive array with using these new techniques in this particular case.
4. The final piece of pseudo-code is: