Variable Length Extended-Precision Fixed-Point Arithmetic Applied to FIR Filtering

Jeyhan Karaoğuz

Center for Communications and Signal Processing
Department of Electrical and Computer Engineering
North Carolina State University

TR-91/22
December 1991

VARIABLE LENGTH EXTENDED-PRECISION FIXED-POINT ARITHMETIC APPLIED TO FIR FILTERING

by

Jeyhan Karaoguz

1. INTRODUCTION

Todays digital signal processing products such as the Motorola 56000 and TMS320 family offer extended-precision arithmetic and long accumulator sizes. The meaning of extended-precision arithmetic in this context is to multiply or add two fixed-point integer numbers of length b^* bits and store the result in $2b$ bits. Since the result cannot be stored in the accumulator as $2b$ bits, it is rather kept in several registers depending on the type of algorithm used.

The need to simulate extended-precision fixed-point arithmetic is essential because simulation is the primary tool in designing DSP algorithms and the performance of those algorithms such as adaptive filtering depends heavily on the type of arithmetic used and the finite register lengths. However, the implementation of extended-precision fixed-point arithmetic in a computer presents problems, since the workstations do not have the luxury of having long accumulator sizes like DSP products do. The basic integer representation in common workstations varies from 16 to 32 bits depending on the type of integer, i.e., long or short. With one bit required for the sign, the range of an integer is between -2^{31} and $2^{31} - 1$ on a 32-bit machine.Therefore, in order to perform extended-precision fixed point arithmetic exceeding 32 bits, several registers should be used for the operands and the result. In addition,

---

* b is the accumulator size

the computer code should support variable word-length fixed-point arithmetic to be able to simulate a variety of DSP products with different accumulator sizes.

2. SOFTWARE IMPLEMENTATION

The implementation of variable-length fixed-point arithmetic is built upon two basic functions, namely, *mult_var.c* for multiplication and *add_var.c* for addition. There are also other supplementary functions called *part.c* and *round_var.c*. These functions are presented in Appendix A. The *part.c* is used to partition the operands into two equal length registers representing the most and least significant parts. The *round_var.c* is used to round the result of addition to the appropriate user defined register length.

2.1 Multiplication function *mult_var.c*

As mentioned above, the first step is to separate the most significant and least significant parts of the operand by the *part.c* function as illustrated in Figure 1. Next, the partitioned operand is passed to *mult_var.c* where the operand is further partitioned into three registers as in Figure 2. The reason for further partitioning is to take care of the sign bit and determine the magnitude of each operand. The extended-precision multiplication algorithm employed in *mult_var.c* is similar to the one used in the TMS320 [1] differing in the handling of right-shifts and sign-arithmetic. The multiplication of partitioned operands is as follows,

```
                X2 X1 X0
              x Y2 Y1 Y0
          ---------------
                   X0xY0
                X1xY0
                X0xY1
             X2xY0
             X1xY1
             X0xY2
          X2xY1
          X1xY2
       + X2xY2
    ---------------------
          W4 W3 W2 W1 W0
```

Finally, the five part result is recombined into two data words.

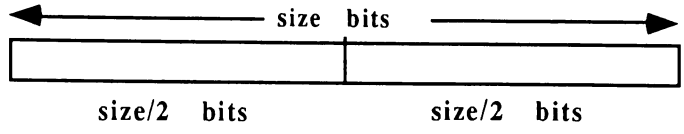

Figure 1. Most and Least Significant Parts

The function *mult_var.c* requires both operands to be the same size. The result is stored as twice the size of operands. Therefore, for sizes exceeding 32 bits, the result is passed to other functions in two registers representing the most and least significant parts. If the result is negative, then the two's complement value is passed to the other functions.

Due to the type of partitioning, the word-length cannot take odd numbers.

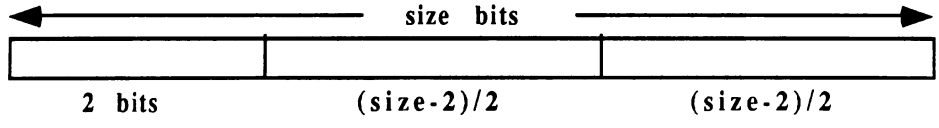

Figure 2. Partitioning in Multiplication Function

2.2 Addition function *add_var.c*

The addition function *add_var.c* also processes partitioned operands since the word-length can exceed 32 bits. It accepts its input operands in two equal length registers representing the most and least significant parts. These partitioned operands are further divided into total of four parts.

The addition operation is as follows,

$$\begin{array}{r} X3\ \ X2\ \ X1\ \ X0 \\ +Y3\ \ Y2\ \ Y1\ \ Y0 \\ \hline W3\ W2\ W1\ W0 \end{array}$$

The reason for further partitioning is to be able to handle the carry resulted from the addition of the least significant parts when the size is 32 bits. The function *add_var.c* is designed to be compatible with *mult_var.c* because usually addition follows multiplication in DSP applications such as FIR filtering. Therefore, if the common variable word-length is the same in both functions then *add_var.c* adds the outputs of several *mult_var.c* functions in extended-precision. In *add_var.c*, the variable *word-length* refers to each partitioned operand. Therefore the size of each operand is twice the variable *word-length* and so is the result. The negative operands are accepted in two's complement form because two's complement arithmetic is used in addition.

2.2.1 Overflow management in *add_var.c*

There are two modes for addition in *add_var.c* which are determined by the variable *saturation_mode*. When enabled, any overflow in addition due to finite register length results in the accumulator contents being replaced with the largest positive value if the overflowed number is positive, or the largest negative value if negative. The largest positive and negative values are determined by the variable *word-length*. When *saturation_mode* is disabled, any overflow i..e. positive or negative, is ignored and addition is performed. This mode is useful in cases when the user knows the end result of the addition is bounded by the word-length. In such cases, although overflow might occur in intermediate additions, the result will come out correctly because of the two's complement arithmetic used in *add_var.c*.

2.3 Round-off function *round_var.c*

The function round_var.c rounds off an fixed-point integer variable by the number of bits specified [2]. If the round-off bits are equal to the number of bits which are used to represent the fractional part of the fixed-point number then the result is the rounded decimal equivalent of the fixed-point number. The variable size is important in determining whether or not any overflow occurs. If the rounded integer value cannot be placed in the accumulator of size bits then overflow occurs. In this case, the accumulator contents are replaced by the largest positive or negative number allowed for the specified accumulator size according to the type of overflow.

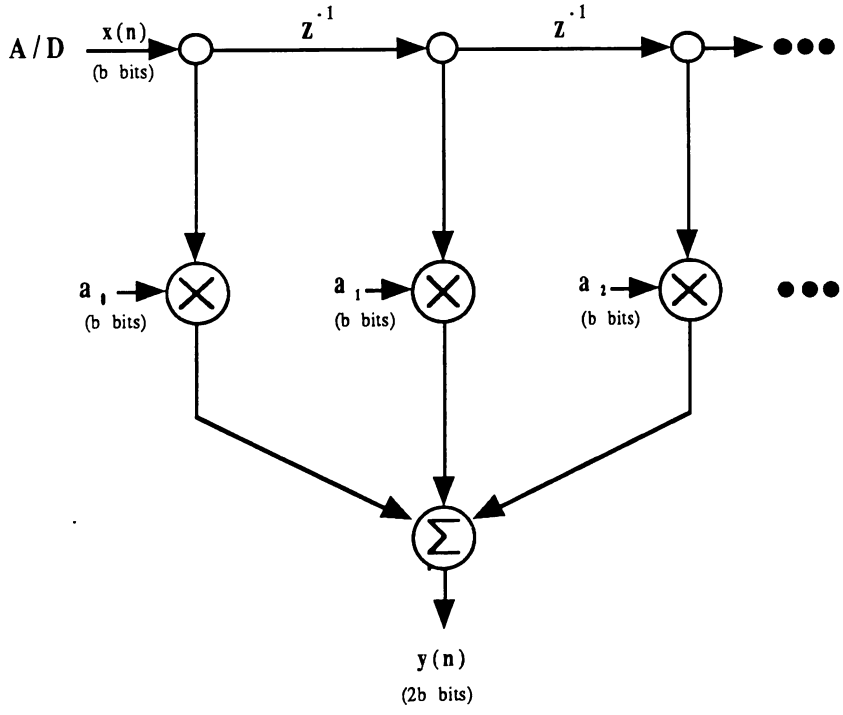

Figure 3. Tapped Delay Line

3. TYPICAL APPLICATION : FIR DIGITAL FILTER

The most typical DSP application where variable length extended-precision arithmetic can be employed is FIR filtering. As illustrated in figure 3, the tapped delay line FIR filter is used with an input coming out of an A/D convertor. The gain coefficients a's are b-bit fixed-point integers with q bits representing the fractional part. Also, the input x(n) is a b-bit fixed-point number but with n bits representing the fractional part. Let's assume that b is 32 bits. In this case, if we do not use extended-precision fixed-point arithmetic in the multiplication node, then we would have to round the result of the multiplication to 32-bits at that node. However, by means of extended-precision arithmetic this can be avoided and each multiplication node can carry out the multiplication with 2b bits. Also, the results coming from multiplication nodes, can be added at the addition node in extended-precision. The result y(n) can be be rounded to any user defined value by *round_var.c* function. The number of bits for rounding depends on the application such that if q+n bits are used in rounding then the result is going to be the decimal representation of y(n). If any arbitrary value k is used for rounding than the result will be the fixed-point representation of y(n) with q+n-k bits representing the fractional part.

As mentioned earlier, the arbitrary register length is supported by the extended-precision arithmetic software. Therefore, the arithmetic can be carried out not only in extended-precision but also in any precision as small as 4 bits accumulator size.

4. CAPSIM SIMULATION

The variable-length extended-precision arithmetic can also be implemented in CAPSIM. The fixed-point arithmetic stars such as *fxgain.s* and *fxadd.s* use the *mult_var.c* and *add_var.c* functions. The stars *fxgain.s* and *fxadd.s* are given in Appendix B. Every time *fxgain.s* star is executed, this star multiplies its operands by calling *mult_var.c* function. Similarly, the *fxadd.s* star calls the *add_var.c* function each time an addition is carried out. The variables such

as *word-length, number of bits to represent fraction, round-off bits* etc., are provided as parameters to *fxgain.s* and *fxadd.s* stars.

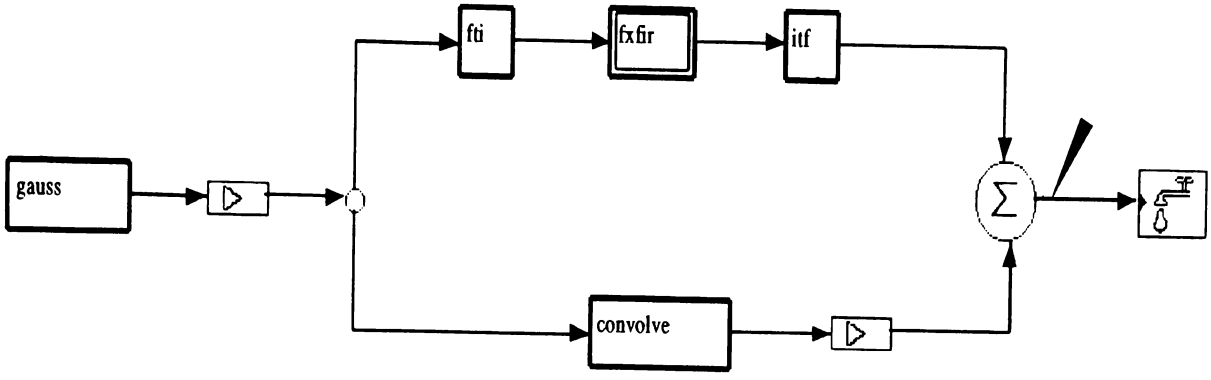

Figure 4. CAPSIM Topology

Figure 4 illustrates the CAPSIM topology which simulates a fixed-point FIR filter. The topology also compares the fixed-point arithmetic results with corresponding floating-point arithmetic results. For the comparison purposes, there are two branches in the topology. The input signal source star *gauss.s* is a Gaussian noise generator which gives out output samples with variance 1. It generates 128 samples. Next, the signal is amplified by the gain star *gain.s*. The gain is 1000000. The reason for having such a large gain is to see the round-off and finite precision effects clearly. After the amplification, the signal is split into two branches. On the top branch, it goes through the fixed-point FIR filter which is illustrated as *fxfir* galaxy. The *fti.s* and *itf.s* stars are for float to integer and integer to float conversions respectively. The tapped delay line *fxfir* galaxy is illustrated in Figure 5. The FIR filter has five coefficients which are 0.9, 0.8, 0.7, 0.6, 0.5. On the bottom branch, the signal is convolved with the impulse response having the same taps as the fixed-point fir filter. However, the arithmetic in the convolution is the conventional floating-point arithmetic of the workstation being used.

Finally, the results of the top and bottom branches are subtracted and the error variance is calculated by the *stats* probe connected at the output.

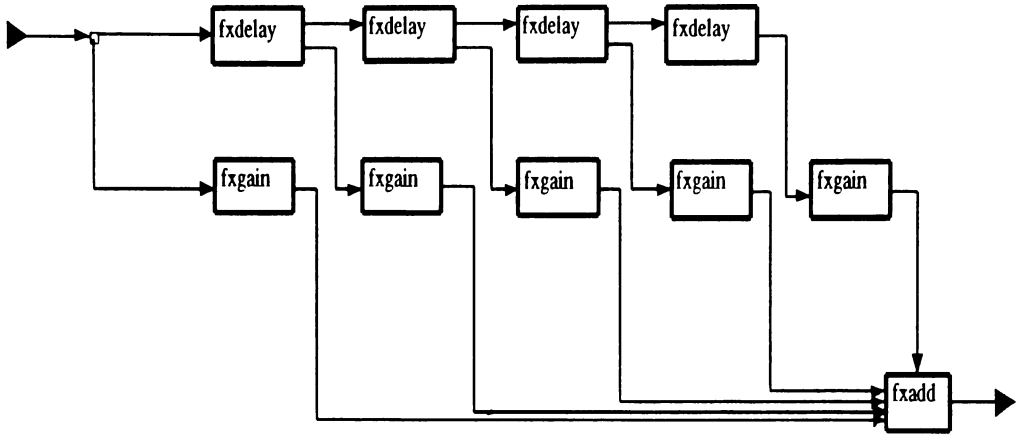

Figure 5. Tapped Delay Line in CAPSIM

Since the Gaussian signal source generates random samples with variance 1, the amplified signal at the input of the *fxfir* galaxy is also random and bounded by 22 bits. Each *fxgain.s* star represents its gain parameter as a 32 bits fixed-point integer. Therefore, each *fxgain.s* star multiplies its 32 bits gain variable with 22 bits input signal resulting in at most 54 bits integer numbers. The 54 bits outputs are added at the *fxadd.s* star and the result is rounded for comparison with the bottom branch. As mentioned earlier, the accumulator size for the stars *fxgain.s* and *fxadd.s* can take any arbitrary value. However, one should consider overflows if smaller accumulator size is chosen processing large integer values.

The following table presents the result of the CAPSIM simulation. The entry *number of bits* specifies the number of bits used in representing the fractional part of the FIR filter coefficients. The error variance is calculated with reference to the floating-point arithmetic of the computer.

NUMBER OF BITS	ERROR VARIANCE
6	111957280
8	6997175
10	437362
12	27323
14	1709
16	107
18	7.7
20	1.26
22	0.85
24	0.92
26	0.92
28	0.92
30	0.92

From the results, it is apparent that the number of bits needed to represent the fractional part of the coefficients cannot take values less than 18 because the error variance is large. The reason of having such a large error variance without any overflow is that there are 128 samples and each sample is in the order of milllion. Therefore, although the accumulator size is large enough for the result, the number of bits for representing the fractional part is not adequate. As the number of bits is increased, the error variance decreases significantly. After 22 bits, the error variance is bounded by the round-off process rather than finite-precision effects.

REFERENCES

[1] "Second-Generation TMS320 User's Guide", Texas Instruments, Section 5.

[2] "Fixed-point Roundoff Error Analysis of the Exponentially Windowed RLS Algorithm for Time-Varying Systems", S. H. Ardalan, S. T. Alexander, IEEE Transactions on ASSP, Vol-35, No. 6, June 1987.

APPENDIX A

```
/*
 * This function multiplies two fixed-point numbers in
 * variable precision. The variable size specifies the word
 * length of each number.The fixed-point numbers are accepted
 * in size/2 bits pairs each representing the most significant
 * and least significant size/2 bits. The result is stored in
 * a word length of twice the size. For example if 32-bit size
 * is used the result is stored in 64-bit format. The way of
 * storing 64-bit is in two 32 bit registers namely OW1_ptr,
 * OW0_ptr where OW1_ptr is the most significant size/2 bits.
 * The sign of the product is determined by the numbers and
 * less_flag's which are recieved from part.c function.
 *
 * Programmer : KARAOGUZ, Jeyhan.
 * Date       : 9/15/90
 */

multvar(less_flag1, less_flag2, size, X1, X0, Y1, Y0, OW1_ptr, OW0_ptr)

/*
 * less_flag1 is for X1, X0 and less_flag2 is for Y1, Y0.
 */

int less_flag1, less_flag2, size, X1, X0, Y1, Y0, *OW1_ptr, *OW0_ptr;
{

int acc, tmp;
int X2, Y2;
int W4, W3, W2, W1, W0;
int sign, partsize, carry;
int andvalue1, andvalue2;

/*
 * partsize is used to put the results in four registers of length
 * size/4, also it is used in the repartition of X1, X0, Y1, Y0.
 */

partsize = (size-2)/2;

andvalue1 = 1;
andvalue1 <<= (size-2)/2;
andvalue1 -= 1;

andvalue2 = 1;
andvalue2 <<= size/2 ;
andvalue2 -= 1;

/*
 * Determine the sign of the product.
 */

if (less_flag1 == 0 && less_flag2 == 0) {

/*
 * For this case, both numbers are either positive or negative but
 * if it is negative then it is greater than or equal to 1<<(size/2)
 * therefore determine the sign by EXORing the most significant
 * size/2 bits. The most significant bits (X1) and (Y1) are checked
 * because those numbers come as negative from part.c function
 * whereas Y0 and X0 come as absolute valued.
 */
        acc = X1^Y1;
        acc >>= (size-1);
        sign = acc & 1;
/*
 * Then, take the absolute value after setting the sign bit.
```

```
 */
        if (X1 < 0)
                X1 = (-X1);
        if (Y1 < 0)
                Y1 = (-Y1);
        }

else if (less_flag1 == 1 && less_flag2 == 0) {

/*
 * In this case, the number X is negative and less than 1<<(size/2)
 * and the number Y is either positive or negative therefore set the
 * sign bit after checking (Y1) the most significant bits.
 */
        if (Y1 < 0) {
                sign = 0;
                Y1 = (-Y1);
                }
        else
                sign = 1;
        }

else if (less_flag1 == 0 && less_flag2 == 1) {

/*
 * This case is just the opposite of previous one.
 */

        if (X1 < 0) {
                sign = 0;
                X1 = (-X1);
                }
        else sign = 1;
        }

/*
 * This case means both flags are 1 therefore both numbers are
 * negative and less than 1<<(size/2). The sign of the product
 * should be positive therefore the sign bit is set to 0. Since
 * the coming parts are absolute valued from part.c function
 * there is no need to take the absolute value again.
 */

else
        sign = 0;

/*
 * Do the repartition of X1, X0 and Y1, Y0. New values X1, X0
 * and Y1, Y0 are of length partsize. The remaining bits are
 * held in X2 and Y2. Note that each number is positive since
 * they are all absolute valued.
 */

acc = X1 << (size/2);
acc += X0;
tmp = acc << 1;
X1 = tmp >> (size/2);
X1 &= andvalue2;
X0 = acc & andvalue1;
acc = X1;
X2 = acc >> partsize;
X1 = acc & andvalue1;

acc = Y1 << (size/2);
acc += Y0;
tmp = acc << 1;
```

```
Y1 = tmp >> (size/2);
Y1 &= andvalue2;
Y0 = acc & andvalue1;
acc = Y1;
Y2 = acc >> partsize;
Y1 = acc & andvalue1;

/*
 * Multiply |X| and |Y| to produce the result |W|
 */

W4 = X2 & Y2;
acc = X0 * Y0;
tmp = acc << 1;
W1 = tmp>> (size/2);
W1 &= andvalue2;
W0 = acc & andvalue1;

acc = (X1 * Y0) + (X0 * Y1) + W1;
tmp = acc << 1;
W2 = tmp>> (size/2);
W2 &= andvalue2;
W1 = acc & andvalue1;

acc = (X2 * Y0) + (X1 * Y1) + (X0 * Y2) + W2;
tmp = acc << 1;
W3 = tmp>> (size/2);
W3 &= andvalue2;
W2 = acc & andvalue1;

tmp = W4 << partsize;
acc = tmp + (X2 * Y1) + (X1 * Y2) + W3;
tmp = acc << 1;
W4 = tmp>> (size/2);
W4 &= andvalue2;
W3 = acc & andvalue1;

/*
 * Recombine W and generate the result since at the moment
 * the result is in 5 different registers. What we want to
 * do is to put them in two registers eventually. This is
 * done in two steps. In this phase they are put in four
 * size/4 length registers.
 */

acc = W1 << partsize;
acc += W0;
W0 = andvalue2 & acc;
W1 = acc >> (size/2);
acc = W2 << (partsize-1);
acc += W1;
W1 = andvalue2 & acc;
W2 = acc >> (size/2);
acc = W3 << (partsize-2);
acc += W2;
W2 = andvalue2 & acc;
W3 = acc >> (size/2);
acc = W4 << (partsize-3);
acc += W3;
W3 = andvalue2 & acc;

/*
 * Take 2's complement if sign is 1
 */

if ( sign == 1) {
```

```
        W0 = ~W0;
        W0 &= andvalue2;
        W0 += 1;

        if ( W0 > andvalue2 )
                carry = 1;
        else
                carry = 0;

        W0 &= andvalue2;
        W1 = ~W1;
        W1 &= andvalue2;
        W1 += carry;

        if ( W1 > andvalue2 )
                carry = 1;
        else
                carry = 0;

        W1 &= andvalue2;
        W2 = ~W2;
        W2 &= andvalue2;
        W2 += carry;

        if ( W2 > andvalue2 )
                carry = 1;
        else
                carry = 0;

        W2 &= andvalue2;
        W3 = ~W3;
        W3 &= andvalue2;
        W3 += carry;
                        }
/*
 * Now, output the result in two size/2 length registers.
 */

acc = W3 << (size/2);
*OW1_ptr = acc + W2;
acc = W1 << (size/2);
*OW0_ptr = acc + W0;

}
```

```
/*
 * This function adds two fixed-point numbers in variable precision.
 * The addition is done in 2's complement arithmetic i.e., if the
 * operands are negative they are accepted in two's complement form.
 * Also the output is in 2's complement form if it is negative.
 * The reason of doing the arithmetic in 2's complement form instead
 * of sign arithmetic is because it is easier to implement.The input
 * numbers are accepted in two registers of length size bits. The
 * result is stored in a register of length twice the size. This
 * function is compatible with the mult_var.c function if the size
 * variables are the same. That is, you can add the outputs of
 * multiplication function with add_var.c. This is very convenient
 * for FIR filter applications.
 *
 * Programmer : KARAOGUZ, Jeyhan.
 * Date       : 9/17/90
 */

addvar(size,saturation_mode,X1,X0,Y1,Y0,OW1_ptr,OW0_ptr)

/*
 * Variable saturation_mode specifies whether the result should be
 * saturated or not in case of overflow.
 */

int size, saturation_mode, X1, X0, Y1, Y0, *OW1_ptr, *OW0_ptr;

{

int X3, X2, Y3, Y2, W3, W2, W1, W0;
int acc, max, carry=0, saturation_carry=0, check=0;
int pos_overflow=0, neg_overflow=0, pos_max1, pos_max0, zero_flag=0;
int neg_least1, neg_least0, both_pos_flag=0, both_neg_flag=0;


max = 1;
max <<= (size/2);
max -= 1;

/*
 * In case of saturation, calculate the saturation values
 */

pos_max1 = (1 << (size - 1)) - 1;
pos_max0 = (1 << size) - 1;

if (size == 32)
        pos_max0 = 0xffffffff;

neg_least1 = (1 << (size - 1));
neg_least0 = 0;

/*
 * Following three 'if' statements are used to determine the
 * signs of the input numbers. Sign of the numbers are needed
 * because the saturation modes are different for negative
 * and positive results sice we are using 2's complement
 * arithmetic. Note that if input numbers have opposite sign
 * there won't be any saturation therefore the saturation mode
 * is set to zero.
 */

if (((X1 >> (size-1)) == 0) && ((Y1 >> (size - 1)) == 0))
        both_pos_flag = 1;

if ((((X1 >> (size-1)) & 1) == 1) && (((Y1 >> (size - 1)) & 1) == 1))
```

```
        both_neg_flag = 1;

if (both_pos_flag != 1 && both_neg_flag != 1)
        saturation_mode = 0;

/*
 * Break the input numbers into four diffrent registers. This is
 * convenient in determining the carry's and saturation.
 */

X2 = X1 & max;
acc = X1 >> (size/2);
X3 = acc & max;
acc = X0 >> (size/2);
X1 = acc & max;
X0 = X0 & max;


Y2 = Y1 & max;
acc = Y1 >> (size/2);
Y3 = acc & max;
acc = Y0 >> (size/2);
Y1 = acc & max;
Y0 = Y0 & max;

/*
 * Add input numbers and calculate the result
 */

acc = X0 + Y0;

if (acc > max) {
        W0 = acc & max;
        carry = acc >> size/2;
        }
else
        W0 = acc;

acc = X1 + Y1 + carry;

if (acc > max) {
        W1 = acc & max;
        carry = acc >> size/2;
        }
else  {
        W1 = acc;
        carry = 0;
        }

acc = X2 + Y2 + carry;

if (acc > max) {
        W2 = acc & max;
        carry = acc >> size/2;
        }
else {
        W2 = acc;
        carry = 0;
        }

/*
 * Check if saturation mode is set, if it is then saturate the result in
 * in case of oveflow to the values determined at the beginning. If the
 * mode is not set to 1 then do your arithmetic ignoring any overflow.
 * This is convenient in the case where you might have overflow in
 * intermediate steps but your result is bounded by the word-length.
```

```
 * In that case 2's complement arithmetic gives  the correct result even
 * overflow might occur in intermediate additions.
 */

if (saturation_mode == 1) {

        acc = X3 + Y3 + carry;
        W3 = acc & max;
        saturation_carry = acc >> size/2;
        check = (W3 >> ((size/2)-1)) & 1;

        if (W3 == 0 && W2 == 0 && W1 == 0 && W0 == 0)
                zero_flag = 1;

        if (saturation_carry == 1 && check == 0 && zero_flag != 1) {
                printf("negative overflow in addition\n");
                neg_overflow = 1;
                }

        if (both_neg_flag == 1 && zero_flag == 1) {
                printf("negative overflow in addition\n");
                neg_overflow = 1;
                }

        if (check == 1 && both_pos_flag == 1) {
                printf("positive overflow in addition\n");
                pos_overflow = 1;
                }

        if (pos_overflow != 1 && neg_overflow != 1) {

                acc = W3 << (size/2);
                *OW1_ptr = acc + W2;
                acc = W1 << (size/2);
                *OW0_ptr = acc + W0;
                }
        else {
                if (pos_overflow == 1) {
                        *OW1_ptr = pos_max1;
                        *OW0_ptr = pos_max0;
                        }
                else {
                        *OW1_ptr = neg_least1;
                        *OW0_ptr = neg_least0;
                        }
                }
       }
else {

        acc = X3 + Y3 + carry;

        if( acc > max ) {
                W3 = acc & max;
                carry = acc >> size/2;
                }
        else
                W3 = acc;


        acc = W3 << (size/2);
        *OW1_ptr = acc + W2;
        acc = W1 << (size/2);
        *OW0_ptr = acc + W0;
        }

}
```

```
/*
 * This function performs rounding operation. The variable roundoff_bits
 * should have the same value as the number of bits to represent the
 * fractional  part of the input number to this function. Then,  the
 * output number is the decimal value of the fractional input number.
 * Decimal value is found after rounding operation. The size variable
 * is the same variable that is used in mult_var.c and add_var.c.
 *
 * Programmer : KARAOGUZ, Jeyhan
 * Date       : 10/6/90
 */

roundvar(size,output_size,roundoff_bits,X1,X0,OUT_ptr)
int size, output_size, roundoff_bits;

/*
 * X1 and X0 are received from add_var.c function.
 */

int X1, X0, *OUT_ptr;
{
int halfmask, halfmax, max, mask, sign_check, tmp;
int i, check, OUT, shift, point=0;

if (size != 32) {
        halfmax = 1;
        halfmax <<= size;
        halfmax -= 1;
        }
else
        halfmax = -1;

mask = 1;
mask <<= (2*size);
mask -= 1;

if (roundoff_bits <= 32) {
        halfmask = 1;
        halfmask <<= (32 - (roundoff_bits - 1));
        halfmask -= 1;
        }
else
        halfmask = 0;

max = 1;
max <<= (output_size - 1);
max -= 1;

/*
 * Determine the sign of the number.
 */

sign_check = X1 >> (size - 1);
sign_check &= 1;

/*
 * If the total size is less than 32 then combine X1 and X0 by first
 * shifting X1 to the left by size bits and then simply adding it to
 * X0.
 */

if (size < 16) {

        X1 <<= size;
        OUT = X1 + X0;
```

```
        if (sign_check == 1) {
                OUT = ~OUT;
                OUT &= mask;
                OUT += 1;
                }

        OUT >>= (roundoff_bits - 1);
        OUT += 1;
        OUT >>= 1;


        if (OUT > max) {
                printf("overflow\n");
                if (sign_check == 0)
                        OUT = max;
                else
                        OUT = (-max);
                }

        if (sign_check == 1)
                OUT = (-OUT);
        }
else {

        if (sign_check == 1) {
                X1 = ~X1;
                X1 &= halfmax;
                X0 = ~X0;
                X0 &= halfmax;
                X0 += 1;
                if (X0 > halfmax && halfmax != -1) {
                        X1 += 1;
                        X0 &= halfmax;
                        }
                if (size == 32 && X0 == 0)
                        X1 += 1;
                }


        tmp = X1;

        for (i = 0 ; i < 32 ; i++) {
                check = tmp & 1;
                if (check == 1)
                        point = i;
                tmp >>= 1;
                }

        if ( ((point - roundoff_bits + 1) + size ) < 31 || (X1 == 0)) {

                shift = size - roundoff_bits + 1;

                if (shift < 0)
                        X1 >>= (-shift);
                else
                        X1 <<= shift;

                X0 >>= (roundoff_bits - 1);
                X0 &= halfmask;
                OUT = X1 + X0;
                OUT += 1;
                OUT >>= 1;

                if (OUT > max) {
                        printf("overflow\n");
                        if (sign_check == 0)
```

```
                                OUT = max;
                        else
                                OUT = (-max);
                        }

                if (sign_check == 1)
                        OUT = (-OUT);
                }
        else {
                printf("use more roundoff bits\n");
                OUT = 0;
                }
        }

*OUT_ptr = OUT;

}
```

```
/*
 * This function accepts an integer input and breaks it into two
 * parts according to size variable. The output integer numbers are
 * to be used by mult_var.c function, therefore they are compatible
 * with that function. The reason for breaking the integer number
 * into two is because of the multiplication algorithm used in
 * mult_var.c function. The length of each part is size/2. The max
 * positive value that can be broken into two parts has the value
 * ((1<<(size-1))-1
 *
 * Programmmer : KARAOGUZ, Jeyhan
 * Date        : 9/23/90
 */

part(size,input,X1_ptr,X0_ptr,less_flag)

/*
 * The variable size specifies the length of the input integer.
 * X1_ptr and X2_ptr are output parts. The pointer less_flag is
 * set to 1 if the number is negative and less than 1<<(size/2)
 * This flag is used in mult_var.c function to determine the sign
 * of the product correctly.
 */

int size, input, *X1_ptr, *X0_ptr, *less_flag;

{

int sign = 0;
int val, X1, X0, tmp;

val = 1;
val <<= (size/2);
val -= 1;

*less_flag = 0;

if ( input<0 ) {
/*
 * Calculate the absolute value and set the sign bit to 1.
 */
        input = ~input;
        input += 1;
        sign = 1;
/*
 * Check if the number is less than 1<<(size/2).
 */
        if ( input <= val )
                *less_flag = 1;
        };
/*
 * Break the number into two parts
 */

X0 = input & val;
tmp = input >> (size/2);
X1 = tmp & val;

/*
 * if less_flag is not 1 and sign is 1 then take the two's
 * complement of only X1. This is done to convey the negative
 * number information to mult_var.c function.
 */

if (*less_flag != 1)
        if (sign == 1) {
```

```
                X1 = ~X1;
                X1 += 1;
                };
/*
 * Output the parts
 */

*X1_ptr = X1;
*X0_ptr = X0;

}
```

APPENDIX B

```
/* fxgain.s */
/************************************************************************

                            fxgain()

************************************************************************

        This star multiplies the incoming data stream by the
        parameter "Gain factor" in fixed-point arithmetic.The
        star is capable of doing extended precision arithmetic
        upto 64 bits result which is to be rounded to at least
        32 bits after the fxadd.s star.

        Parameters :
        1 - (float) factor : FIR tap coefficient
        2 - (int)   qbits  : Number of bits to represent the
                             fraction
        3 - (int)   size   : Total word length including the
                             integer part and the sign bit

        Programmer : KARAOGUZ, Jeyhan
        Date       : 9/26/90
*/

input_buffer

        int x;
end

parameters

        param_def = "Gain factor";
        float factor = 1.0;
        param_def = "Number of bits to represent fraction";
        int qbits = 8;
        param_def = "Word length";
        int size = 32;

end

states

        int obufs;
        int fxfactor;
        int fxfactor1;
        int fxfactor0;
        int less_flag1
        int less_flag2;
        int max;
        int* overflow;

end

declarations

        int i, samples, val;
        int input, input1, input0;
        int out1, out0;

end

initialization_code

        if ((obufs = no_output_buffers()) <= 0) {

                fprintf(stdout,"gain: no output buffers\n");
```

```
                return(2);
                }

        overflow = (int*)calloc(1,sizeof(int));

        if (size > 32) {

                fprintf(stdout,"size can not be greater than 32\n");
                return(4);
                }

        if ((size & 1) == 1) {

                fprintf(stdout,"Sorry, size can not be an odd number\n");
                return(4);
                }

        if (qbits > 30) {
        /*
         * Because 1<<31 becomes a negative number in this machine
         */
                fprintf(stdout,"At most 30 bits are allowed for fraction\n");
                return(4);
                }
        /*
         * Calculate the maximum number to be represented by size bits
         */
        max=1;
        max <<= (size-1);
        max -= 1;

        val=1;
        val <<= qbits;

        if (factor>0.0)

                fxfactor = (int)(factor * val + 0.5);
        else
                fxfactor = (int)(factor * val - 0.5);

        if (fxfactor > max || (-fxfactor) > max) {

                fprintf(stdout,"gain can not be represented by size bits\n");
                return(4);
                }

        part(size,fxfactor,&fxfactor1,&fxfactor0,&less_flag1);

end

main_code

for(samples = min_avail(); samples >0; --samples) {

        it_in(0);
        input = x(0);

        if (input > max || (-input) > max) {
                fprintf(stdout,"input can not be represented by size bits\n");
                return(0);
                }

        part(size,input,&input1,&input0,&less_flag2);

        multvar(less_flag1,less_flag2,size,fxfactor1,fxfactor0,
                                        input1,input0,&out1,&out0);
```

```
        for (i=0; i<obufs; i++) {

                it_out(i);
                outi(i,0) = out1;
                it_out(i);
                outi(i,0) = out0;

        }
}
return(0);

end
```

```
/* fxadd.s */
/************************************************************************

                        fxadd()

************************************************************************

        This star adds all of its input samples. The input is
        accepted in pairs coming from fxgain.s star. The output
        is rounded by the number of bits specified by the parameter
        roundoff_bits.

        Parameters :
        1 - (int) roudoff_bits
        2 - (int) size : The size of input number
        3 - (int) output_size : output register word-length

        Programmer : KARAOGUZ, Jeyhan
        Date       : 9/30/90
*/

parameter

        param_def = "roundoff bits";
        int roundoff_bits = 8;
        param_def = "Word length";
        int size = 32;
        param_def = "outputsize";
        int output_size = 32;
        param_def = "saturation mode";
        int saturation_mode = 1;


end

states

        int ibufs;
        int obufs;
        int* overflow;
end

declarations

        int i, j, samples;
        int sum1, sum0, input1, input0, out1, out0, out;
end

initialization_code

        overflow = (int*)calloc(1,sizeof(int));

        if (size > 32) {

                fprintf(stdout,"size can not be greater than 32\n");
                return(4);
                }

        if ((size & 1) == 1) {

                fprintf(stdout,"Sorry, size can not be an odd number\n");
                return(4);
                }

        /*
         * store as state the number of input/output buffers
```

```
         */

        if ((ibufs = no_input_buffers()) < 1) {

                fprintf(stderr,"fxadd: no input buffers\n");
                return(2);
                }

        if ((obufs = no_output_buffers()) < 1) {

                fprintf(stderr,"fxadd: no output buffers\n");
                return(3);
                }
end

main_code

        /*
         * read one sample from each input buffer and add them
         */

        for (samples = (min_avail() >> 1);samples > 0; --samples) {

                sum1 = 0;
                sum0 = 0;

                for (i=0; i<ibufs; ++i) {

                        it_in(i);
                        input1 = ini(i,0);
                        it_in(i);
                        input0 = ini(i,0);
                        addvar(size,saturation_mode,input1,input0,sum1,sum0,&out1,&
                        sum1 = out1;
                        sum0 = out0;
                }

                roundvar(size,output_size,roundoff_bits,sum1,sum0,&out);

                for (i=0; i<obufs; ++i) {

                        it_out(i);
                        outi(i,0) = out;
                }
        }

        return(0);      /* at least one input buffer empty */

end
```