Fixed-point Simulation Utility for C and C++ Based Digital Signal Processing Programs

Seehyun Kim and Wonyong Sung

Department of Control and Instrumentation Engineering
and Inter-University Semiconductor Research Center
Seoul National University
Shinlim-Dong, Gwanak-Gu, Seoul, KOREA 151-742
Email: shark@hdtv.snu.ac.kr and wysung@signal.snu.ac.kr

Abstract

This utility software automatically converts a floating-point digital signal processing program written in C or C++ language to a fixed-point program. The conversion is conducted by defining a new fixed-point data class and utilizing the operator overloading characteristic of the C++ language. A generalized fixed-point format which consists of the wordlength, integer wordlength, sign, overflow, and quatization mode is employed for specifying a fixed-point variable or a fixed-point constant. Thus, it is possible to simulate the finite wordlength and the scaling effects of digital signal processing programs very easily.

1 Introduction

Although most of the digital signal processing algorithms are developed using floating-point arithmetic, VLSI or fixed-point digital signal processor based implementation of them requires fixed-point arithmetic for the sake of hardware cost and speed. However, the fixed-point implementation can suffer from excessive finite wordlength effects due to overflows and quantization noise[1]. Therefore, it is necessary to simulate digital signal processing algorithms using fixed-point arithmetic before implementation. However, it has been considered a tedious process to convert a floating-point algorithm to a fixed-point one. In this paper, a fixed-point preprocessing utility software which automatically converts a floating-point digital signal processing program written in C or C++ to a fixed-point equivalent is developed. The wordlength, integer wordlength, sign, overflow handling scheme, and quantization mode of individual variables and constants can be assigned using comments in the floating-point program. Then, the developed fixed-point preprocessor modifies the floating-point data type to the fixed-point type, and attaches a header file defining the fixed point class. The operations associated with the fixed-point data type, such as '=', '+', '−', '×', and '/', are also defined at the class declaration. In the modified program, fixed-point arithmetic operations, instead of floating-point arithmetic, are conducted automatically due to the operator overloading capability of the C++ language[2]. Except for declaring the variables to fixed-point class type and adding the fixed-point header file, any other part of the original program is changed during the conversion process. Since digital signal processing programs can be written differently according to the implementation architectures, this utility software can also be used for comparing the fixed-point characteristics of different implementation architectures.

2 Generalized Fixed-point Data Format

For the representation of the fixed-point data, the generalized fixed-point format[3] using the attributes specified in the following format is employed:

$$< wordlength, integer\_wordlength, sign\_overflow\_quantization\_mode > \quad (1)$$

The wordlength is the total number of bits for representing a fixed-point signal. The accuracy of representation or the amount of quantization noise is determined by the wordlength. The integer wordlength

is the number of bits to the left of the (hypothetical) binary-point. The sign can be either unsigned('u') or two's complement('t'). The two's complement sign format requires one additional bit. For example, in the fractional format, the MSB (Most Significant Bit) is the sign bit, and the hypothetical binary-point is assumed to be located just next to the sign bit. Thus, any signal value in the fractional format cannot be larger than one, which is a serious limitation for the translation of a floating-point program to a fixed-point counterpart. The integer wordlength corresponds to 0 in the fractional format. The generalized fixed-point format allows any integer wordlength in order to overcome the limitation of the fractional format. The positive range of a signal value that can be represented equals to $2^{integer\ wordlength}$. For example, a signal having an integer wordlength of 2 can represent a signal between -4 to +4 without overflow. The overflow mode specifies whether no treatment ('o') or saturation ('s') scheme is used when overflow ocurrs, and the quantization mode denotes whether rounding ('r') or truncation ('t') is used for the quantization of lower significant bits. For example, a fixed-point format of '< 10, 2, "tsr" >' corresponds to the wordlength of 10 bits, integer wordlength of 2 bits, two's complement representation, saturation when overflow ocurrs, and rounding for wordlength reduction. A 10 bit binary data 0101000000 with the above format should be interpreted as 010.1000000, which corresponds to a real value of 2.5. In this generalized fixed-point format, two data can be added or subtracted only when their integer wordlengths are the same. The integer wordlength can be changed by using arithmetic shift operations. Arithmetic right-shift of one bit increases of the integer wordlenth by one. Thus, the number of shifts required can be obtained by comparing the integer wordlength of the two input data. The C++ header file which defines the generalized fixed-point class is shown in Fig. 1. As shown in this figure, the generalized fixed-point class, class `gFix`, has six private members, which are the mantissa(`long m`), the wordlength(`short wl`), the integer wordlength(`short iwl`), and attributes(`char represent, saturation, round`). The maximum wordlength allowed is 32 bits, but will be extended to a larger number at the next version.

3 Operators for the gFix Class

The `gFix` class supports most of the basic assignment and arithmetic operations supported in C or C++ languages. The list of the operations supported can be found at the operator list in Fig. 1. Brief explanations of them are as follows.

1. The assignment operator, '=', converts the input data according to the fixed-point format of the left side variable, and assigns the format converted data to this variable. The input data, which is the evaluated result at the right side, can have either a floating-point or a fixed-point data type. If the given format of the left side variable does not have enough precision for representing the input data, the data is modified according to the attributes of the left side variable, such as saturation, rounding, or trucation.
2. The operation of the fixed-point add operator, '+', is shown in Fig. 2. In order to prevent any loss of accuracy during the operation, it first computes the maximum integer and fractional wordlengths of two input data. For example, when the integer and fractional wordlengths for the first operand are 2 and 8, respectively, and those for the second operand are 4 and 6, respectively, the internal data has the integer wordlength of 5 and fractional wordlength of 8. After then, the input data are aligned by using shift operations, and added in fixed-point.
3. The fixed-point multiply operator, '*', is also described in Fig. 2. The wordlength of the product is the sum of the wordlengths of the two input data minus one in order to eliminate the superfluous sign. And, the integer wordlength becomes the sum of the two input integer wordlengths.
4. Arithmetic right shift operator, '>>', reduces the integer wordlength by one, and increases the fractional wordlength by one. The total number of bits is not changed.
5. Arithmetic left shift operator, '<<', increases the integer wordlength by one, and, instead, reduces the fractional wordlength by one.

As described in the above, there is no loss of accuracy during the fixed-point add, multiply, arithmetic right shift, and arithmetic left shift operations. The format conversion is occurred only at the assignment operator. Thus, two programs shown in Fig. 3-(a) and (b) can have the different fixed-point results. In Fig. 3-(b), the result of "$a + b$" is format converted to 10 bit data, and then added to the operand c, and then format converted again.

4 Examples

A first order IIR digital filter and an 8-point DCT program using Chen's algorithm[4] are implemented. Floating-point programs written in C++ are converted to fixed-point equivalents by the developed preprocessor.

4.1 IIR filter

A C++ language program for a first order recursive digital filter using floating-point arithmetic is shown in Fig. 4. Double precision floating-point format is used for declaring the varibles and the filter coefficient. The fixed-point format for each variable and constant is represented as comments. Thus, the program or algorithm can be verified throughout the floating-point simulation. Then, this program is automatically converted to the fixed-point program as shown in Fig. 5 using the proposed fixed-point preprocessor. Due to the operator overloading capability of the C++ language, the arithmetics using the variables or constants defined as `gFix` data type are conducted according to the class definition in the header file, gFix.h. The fixed-point program can also be simulated by an ordinary C++ compiler.

4.2 8-point DCT algorithm

A number of fast DCT algorithm have been reported to reduce the number of multiplications[5].But it is known that many of them have problems in accuracy when they are realized with finite wordlength[6]. By using the proposed preprocessor, the fixed-point characteristics of fast algorithms can also be compared. In this section, Chen's algorithm[4] is implemented as shown in Fig. 6.

5 Concluding Remarks

A proprocessing software that converts a floating-point digital signal processing program written in C or C++ language to a fixed-point one has been developed. Although the automatically generated C++ fixed-point program is not for a real time implementation, but it can model the fixed-point implementations using VLSI or fixed-point digital signal processors at the bit-level accuracy. Any digital signal processing programs, whether linear, non-linear, or time-varying, can be simulated in the fixed-point arithmetic using this software. Since programs using the proposed fixed-point data type can be easily converted to integer programs or VHDL codes, this work can be extended to efficient VLSI or general purpose digital signal processor based implementations. Also, an automatic wordlength optimization software for C or C++ based digital signal processing programs is being developed by combining this fixed-point simulation software and the wordlength optimizing algorithm[7].

Acknowledgments

This work was partly supported by the post-doctoral research program of the KOSEF (Korea Science and Engineering Foundation) to Wonyong Sung in 1993, and by the research funding of ISRC (Inter-university Semiconductor Research Center) 91-E-0019.

References

[1] L. B. Jackson, "On the interaction of roundoff noise and dynamic range in digital filters," *Bell System Technical Journal*, vol. 49, pp. 159–184, Feb. 1970.

[2] Bjarne Stroustrup, *The C++ Programming Language, 2nd Ed.*, Addison Wesley, 1993.

[3] Seehyun Kim and Wonyong Sung, "A floating-point to fixed-point assembly program translator for the TMS320C25," *IEEE Trans. Circuits and Systems*, Oct. 1994, to be published.

[4] W. H. Chen, C. H. Smith, and S. C. Fralick, "A fast computational algorithm for the discrete cosine transform," *IEEE Trans. Communication*, vol. COM-25, no. 9, pp. 1004–1009, Sep. 1977.

[5] B. G. Lee, "A new algorithm to compute the discrete cosine transform," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 32, pp. 1234–1245, Dec. 1984.

[6] Il Dong Yun and Sang Uk Lee, "On the fixed-point-error analysis of several fast DCT algorithms," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 3, no. 1, pp. 27–41, Feb. 1993.

[7] Wonyong Sung and Ki-Il Kum, "Wordlength determination and scaling software for a signal flow block diagram," in *Proc. International Conference on Acoustics, Speech and Signal Processing*, vol. 2, pp. 457–460, Apr. 1994.

```
class gFix
{
  long  m;          // mantissa
  short iwl;        // integer word-length
  short wl;         // total word-length
  char  represent;  // 't' or, 'u'
  char  saturation; // 's' or, 'o'
  char  round;      // 'r' or, 't'

public:

  // constructors
  gFix();
  gFix(short wlen, short iwlen, char *fmt);
  gFix(double d, short wlen, short iwlen, char *fmt);
  ......

  // assignment operators
  gFix& operator = (gFix& x);
  gFix& operator = (double d);

  // basic operators
  friend gFix operator + (gFix& x, gFix& y);
  friend gFix operator - (gFix& x, gFix& y);
  friend gFix operator * (gFix& x, gFix& y);
  friend gFix operator / (gFix& x, gFix& y);
  friend gFix operator << (gFix& x, short b);
  friend gFix operator >> (gFix& x, short b);
  ......

  // assignment based operators
  gFix& operator += (gFix& x);
  gFix& operator -= (gFix& x);
  gFix& operator *= (gFix&);
  ......

  // relational operators
  friend short operator == (gFix& x, gFix& y);
  friend short operator != (gFix& x, gFix& y);
  friend short operator >= (gFix& x, gFix& y);
  ......

  // miscellaneous operators
  friend istream& operator >> (istream& s, gFix& x);
  friend ostream& operator << (ostream& s, gFix& x);
  ......

};
```

Figure 1: Header file for the generalized fixed-point class declaration.

```
gFix operator + (gFix& x, gFix& y)
// assume that
//   result.wl = max( result.iwl +
//                max(x.fwl, y.fwl) + 1, MAXWL )
//   result.iwl = max(x.iwl, y.iwl) + 1
// ,where fwl means fraction word-length.
{
  short shift;
  short xflen, yflen;  // fraction length
  short maxflen;
  gFix  z;

  xflen = x.wl - x.iwl - 1;
  yflen = y.wl - y.iwl - 1;
  maxflen = ( xflen > yflen) ? xflen : yflen;
  z.iwl = (x.iwl > y.iwl ) ? x.iwl+1 : y.iwl+1;
  z.wl = z.iwl + maxflen + 1;
  if( z.wl > MAXWL ) z.wl = MAXWL;

  shift = x.iwl - y.iwl;
  if( shift >= 0 ) {
    if( shift < 31 )
      z.m = (x.m>>1) + (y.m>>(shift+1));
    else z.m = (x.m>>1);
  }
  else {
    if( shift > -31 )
      z.m = (x.m>>(-shift+1)) + (y.m>>1);
    else z.m = (y.m>>1);
  }
  return z;
}

gFix operator * (gFix& x, gFix& y)
// assume that
//   result.wl = x.wl + y.wl - 1
//   result.iwl = x.iwl + y.iwl
{
  gFix  z;

  z.wl = x.wl + y.wl - 1;
  if( z.wl > MAXWL ) z.wl = MAXWL;
  z.iwl = x.iwl + y.iwl;
  z.m = ((x.m>>16) * (y.m>>16)) << 1;

  return z;
}
```

Figure 2: Operators of the gFix class.

```
double a, b, c; /* <12, 0, "tsr"> */
double d;       /* <10, 1, "tsr"> */

d = a + b + c;
```

(a)

```
double a, b, c; /* <12, 0, "tsr"> */
double d;       /* <10, 1, "tsr"> */
double tmp;     /* <10, 1, "tsr"> */

tmp = a + b;
d = tmp + c;
```

(b)

Figure 3: Three operand addition using different architectures.

```
double Xin;    /* <12,0,"tsr"> */
double Yout;   /* <16,3,"tsr"> */
double Ydly;   /* <16,3,"tsr"> */
double Acoeff; /* <10,0,"tsr"> */
short i;

Acoeff = 0.979999;
Ydly = 0.;
for( i = 0; i < 1000; i++ ) {
  infile >> Xin ;
  Yout = Acoeff * Ydly + Xin ;
  Ydly = Yout ;
  outfile << Yout << '\n';
}
```

Figure 4: A floating-point C++ program for IIR digital filtering.

```
#include "gFix.h"

gFix Xin(12,0,"tsr");     /* <12,0,"tsr"> */
gFix Yout(16,3,"tsr");    /* <12,0,"tsr"> */
gFix Ydly(16,3,"tsr");    /* <12,0,"tsr"> */
gFix Acoeff(10,0,"tsr"); /* <12,0,"tsr"> */
short i;

Acoeff = 0.979999;
Ydly = 0.;
for( i = 0; i < 1000; i++ ) {
  infile >> Xin ;
  Yout = Acoeff * Ydly + Xin ;
  Ydly = Yout ;
  outfile << Yout << '\n';
}
```

Figure 5: The translated fixed-point simulation program for IIR digital filtering.

```
double X[8];      /* <12,8,"tsr"> */
double t[8];      /* <13,9,"tsr"> */
double Y[8];      /* <16,9,"tsr"> */
double C[7];      /* <14,0,"tsr"> */

C[0] = cos(PI/4.);
C[1] = cos(PI/8.);
C[2] = sin(PI/8.);
C[3] = cos(PI/16.);
C[4] = cos(3*PI/16.);
C[5] = sin(3*PI/16.);
C[6] = sin(PI/16.);

t[0] = X[0] + X[7];
t[1] = X[1] + X[6];
t[2] = X[2] + X[5];
t[3] = X[3] + X[4];
t[4] = X[0] - X[7];
t[5] = X[1] - X[6];
t[6] = X[2] - X[5];
t[7] = X[3] - X[4];

Y[0]=(C[0]*t[0]+C[0]*t[1]+C[0]*t[2]+C[0]*t[3])/2.;
Y[1]=(C[1]*t[0]+C[2]*t[1]-C[2]*t[2]-C[1]*t[3])/2.;
Y[2]=(C[0]*t[0]-C[0]*t[1]-C[0]*t[2]+C[0]*t[3])/2.;
Y[3]=(C[2]*t[0]-C[1]*t[1]+C[1]*t[2]-C[2]*t[3])/2.;
Y[4]=(C[3]*t[0]+C[4]*t[1]+C[5]*t[2]+C[6]*t[3])/2.;
Y[5]=(C[4]*t[0]-C[6]*t[1]-C[3]*t[2]-C[5]*t[3])/2.;
Y[6]=(C[5]*t[0]-C[3]*t[1]+C[6]*t[2]+C[4]*t[3])/2.;
Y[7]=(C[6]*t[0]-C[5]*t[1]+C[4]*t[2]-C[3]*t[3])/2.;
```

Figure 6: A C++ 8-point DCT program by Chen's algorithm.