Fixed-Point C Language for Digital Signal Processing

Wonyong Sung and Jiyang Kang
Department of Control and Instrumentation Engineering
and Inter-university Semiconductor Research Center
Seoul National University
Shinlim-Dong, Gwanak-Gu, Seoul 151-742 KOREA
E-mail: wysung@hdtv.snu.ac.kr and jiyang@hdtv.snu.ac.kr

Abstract

Fixed-point C language is proposed for convenient and efficient programming of fixed-point digital signal processors. This language has a 'fix' data type that can have an individual integer word-length according to the range of a variable. It can add or subtract two data having different integer word-lengths by automatically inserting shift operations. The accuracy of fixed-point multiply operation is significantly increased by storing the upper part of the multiplied double-precision result instead of keeping the lower part as conducted in the integer multiplication. The quantization noise resulting from fixed-point arithmetic is significantly reduced when compared with conventional integer programs. The execution speed is much, nearly 20 times, faster than a floating-point C program in fixed-point digital signal processors.

1 Introduction

Although C language is not very ideal for describing digital signal processing algorithms, it is widely used for simulation and algorithm verification. Especially, C compilers for floating-point digital signal processors are gaining acceptance because of the shortened development time and the improved compiler efficiency. However, C compilers for fixed-point digital signal processors have met with little acceptance [1] especially because of the overhead in executing floating-point operations using fixed-point data path.

One may use the '**int**' data type in C language not to use floating-point operations, but it results in a severe loss of accuracy, especially for performing multiply operations even after careful scaling of the program. In the integer multiplication of two 16 bit operands, the lower 16 bit part is stored as the result among 31 bit of the product as illustrated in Fig. 1. Thus, we have to severely scale down the input of the multiplier in order to prevent overflows. Obviously, it is also tedious to develop integer programs.

In this paper, we introduce a new data type,'**fix**,' and define corresponding arithmetic operations to solve these problems. A fixed-point C compiler for

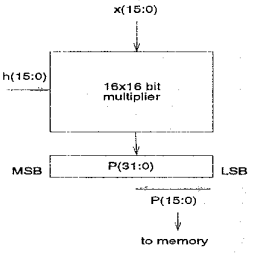

Figure 1: Integer multiplication

TMS320C50, Texas Instruments' fixed-point digital signal processor, is prototyped by modifying a retargetable C compiler [2]. The execution speed and the SQNR of the floating-point, the integer, and the fixed-point implementations are compared by using a biquad IIR digital filter program.

2 Fixed-point data representation and arithmetic rules

An integer variable or constant in C language consists of, usually, 16 bits, and the LSB(Least Significant Bit) has the weight of one for the conversion to or from the floating-point data type. This can bring overflows or unacceptable quantization errors when a floating-point digital signal processing program is converted to an integer version. Therefore, it is necessary to assign a different weight to the LSB of a variable or a constant [3] [4]. In order to assign a different weight, we employed the fixed-point data type that can have an individual integer word-length as follows:

```
fix(integer_wordlength) variable_name;
```

Figure 2 shows a 16 bit fixed-point data format with the integer word-length of W_I and the fractional word-length of W_F. Note that the range (R) that a

1058-6393/96 $5.00 © 1996 IEEE
Proceedings of ASILOMAR-29

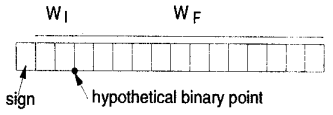

Figure 2: A 16 bit two's complement fixed-point data format with the W_I of 2

variable can represent and the quantization step (Q_s) are dependent on the integer word-length as follows.

$$-2^{W_I} \leq R < 2^{W_I} \quad (1)$$

$$Q_s = 2^{-W_F} = 2^{-(15-W_I)} \quad (2)$$

For example, with a variable of '`fix(2) x`' that has the integer word-length of 2, the fixed-point signal `x` can represent a data between -4 and 4 with the quantization step of 2^{-13}. Note that we can derive the `unsigned`, `long`, and `short` formats based on the same idea.

The arithmetic rules employing this fixed-point data representation and a hardware data-path having a 16 bit by 16 bit two's complement multiplier, a 32 bit ALU, and a barrel shifter can be derived as follows.

2.1 Addition or subtraction

Two data can be added or subtracted after equalizing the integer word-length for them. The integer word-length can be changed by arithmetic shift operations. An arithmetic right shift of 1 increases the integer word-length by 1. For example, the program shown in Fig. 3-(a) can be compiled as depicted in Fig. 3-(b). Figure 3-(c) also shows the compiled assembly codes for the TMS320C50 fixed-point digital signal processor.

```
fix(2) x; /* IWL of 2 */
fix(3) y; /* IWL of 3 */

y = x + y;
```

(a)

```
y = ADD (SFR(1)x, y);
```

(b)

```
LACC    _x, 15
ADD     _y, 16
SACH    _y, 0
```

(c)

Figure 3: '`fix`' data type add-operation

Note that `SFR(1)` represents an one-bit arithmetic right shift.

2.2 Multiplication

Two's complement multiplication of two 16 bit data, `x` and `y`, yields a 31 bit result in the `P` register, `P(30:0)`, as shown in Fig. 1. Note that an extra sign bit is eliminated in the two's complement multiplication process. In the fixed-point C, the upper 16 bit part, `P(30:15)`, is stored as the product as illustrated in Fig. 4. For example, the program shown in Fig. 5-(a) can be compiled as depicted in Fig. 5-(b). Figure 5-(c) also shows the compiled codes for the TMS320C50 using the fixed-point C compiler.

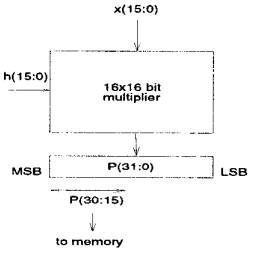

Figure 4: Fixed-point multiplication

```
fix(2) x; /* IWL of 2 */
fix(3) y; /* IWL of 3 */

y = x * y;
```

(a)

```
P_reg(31:0) = MUL (x, y);
y = SFL(2)P_reg(30:15);
```

(b)

```
LT      _x
MPY     _y
SPM     1       ; shift left the product
                ; by one bit
PAC
SACH    _y, 2
```

(c)

Figure 5: '`fix`' data type multiply-operation

Note that SACH(store high accumulator) instruction for storing the upper part of the product is used instead of the usual SACL(store low accumulator) instruction. Here, `SFL(2)` represents a two-bit arithmetic left-shift.

3 Determination of the number of shifts

The TMS320C50 has three barrel shifters at the data-path as shown in Fig. 6. Each of them will

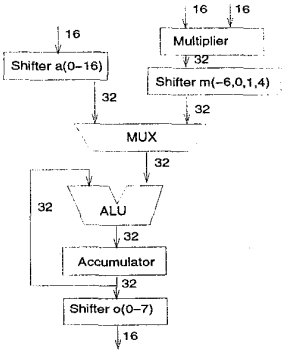

Figure 6: Simplified data path of the TMS320C50

be denoted as `shifter a`, `shifter m`, and `shifter o`. The `shifter a` is for scaling the direct input to the accumulator, and used in the instructions such as ADD(add), SUB(subtract), and LACC(load). The `shifter m` is for scaling the multiplier output, and affects the result of the instructions such as PAC(load with product) and APAC(accumulate with product). The `shifter o` scales the accumulator output when the data is moved into memory location by the SACL or SACH instructions. Each shifter is significantly different from each other in its shifting capability.

We know the integer wordlength of all the input and output operands to the data path, but do not have any information to the range of the accumulator which can only be obtained at the simulation time.

3.1 Determination of the number of shifts for `shifter m`

The number of shifts for `shifter m` should be determined first because it does not support continuous scaling. The number of shifts of -6 is used when the integer wordlength of the accumulator is larger than that of the multiplied results, such as the case of loop operation. The number of shifts of 1 is employed for eliminating the superfluous sign bit in two's complementary multiplication. Note that selecting the number of shifts of -6 can prevent the overflows in the accumulator, but increases the quantization noise. Since we do not know the integer wordlength of the accumulator at the compile time, we initially assume the number of shifts of -6 for the `shifter m`, which means a very conservative scaling approach. This initial setting should be modified when appropriate number of shifts cannot be found at the `shifter a` and `shifter o`. The number of shifts and the resulting integer wordlength of the accumulator can be described as follows.

$$sm = 1, 0, or -6 \quad (3)$$

$$W_I(ACC) = W_I(op1) + W_I(op2) + 1 - sm \quad (4)$$

3.2 Determination of the number of shifts for `shifter a`

A 16 bit data at the input of `shifter a` is aligned to the least significant 16 bit of the 32 bit ALU input. Thus, a zero bit shift at the `shifter a` corresponds to 16 bit right shifts. The number of shifts for the `shifter a` can be determined according to Eq.(5). Note that the integer wordlength of the accumulator has been determined in Eq.(4).

$$sa = W_I(op) + 16 - W_I(ACC) \quad (5)$$

When there is no multiplication, the integer wordlength of the accumulator is simply determined as the maximum integer wordlength of all the operands in the code tree.

3.3 Determination of the number of shifts for `shifter o`

The number of shifts for the `shifter o` can be determined as shown in Eq.(6).

$$so = W_I(ACC) - W_I(output) \quad (6)$$

When the determined number of shifts for the `shifter o` is larger than 7, the integer wordlength for the accumulator should be decreased which can be conducted by selecting the number of shifts of zero, instead of -6, for the `shifter m`. The overall scaling procedure for the current code tree should be repeated in this case.

4 Overall compiler structure

A proposed fixed-point C compiler for TMS320C50 is implemented by modifying the *lcc*, a retargetable C compiler made by C. Fraser and D. Hanson [5] [6]. The overall development procedures using the fixed-point C compiler and its internal structures are shown in Fig. 7. Not only the back-ends of the lcc, which conducts the target specific code generation, but also the middle part were modified to implement a few unique features in this fixed-point C compiler, such as scaling and optimization. Although a new data type is introduced, the front-ends of the compiler are not modified by substituting the '`float`' by the '`fix`' data type.

The integer wordlength of each variable can be given either by manually or automatically using the the *Fixed-Point Optimization Utility* [4]. The *Fixed-Point Optimization Utility* converts the float data type variables into corresponding C++ range estimation classes, and estimates their integer word-lengths by simulation.

The overall compilation process is as follows. In the first step, the compiler preprocesses the source programs, conducts lexical analysis and parsing, and

builds the abstract syntax trees from the source programs. In the abstract syntax trees, each node represents one basic operation, and all type conversions implicit in the source codes are made explicit. In the second step, the compiler builds the directed acyclic graphs(dags) from the abstract syntax trees. The dags represents the intermediate codes. In the third step, with the dags and the integer word-length informations, the compiler determines the number of shifts to avoid overflows and utilize all the bits. In the fourth step, after appropriate optimization, the code generator produces the target assembly code by annotating the dags.

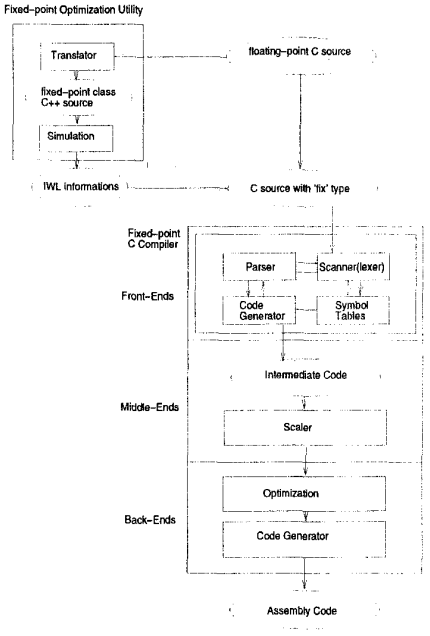

Figure 7: Development procedures using the fixed-point C compiler

5 Implementation example and comparison results

A biquad digital filtering program shown in Eq.(7) and (8) is implemented using floating-point, integer, and fixed-point data type.

$$u[n] = 1.683u[n-1] - 0.7843u[n-2] + x[n] \quad (7)$$
$$y[n] = u[n] - 0.669\,u[n-1] + u[n-2] \quad (8)$$

The integer program can be implemented using the integer version of equation shown in Eq.(9) and (10).

$$u[n] = Q(1.683 \times 2^s)\,Q(u[n-1] \div 2^s) \quad (9)$$

$$y[n] = \begin{array}{l} -Q(0.7843 \times 2^s)\,Q(u[n-2] \div 2^s) + x[n] \\ = u[n] + u[n-2] \\ -Q(0.669 \times 2^s)\,Q(u[n-1] \div 2^s) \end{array} \quad (10)$$

In the above equation, $Q()$ means the rounding to an integer value. If we assign s bits to the coefficients, the data needs to be shifted down by the same amount because the multiplied result of them should be in the lower 16 bit range to prevent overflows in the integer multiplication process. The optimum value for s is found to be 7 in this example. The SQNR when compared with the floating-point simulation result is just 21 dB because of the limited word-length for the coefficients and the data.

In the fixed-point implementation, the coefficients and the data can be represented in 16 bit full precision because there is no overflow in the multiplication. Thus, it is possible to obtain the SQNR of about 58 dB as shown in Table 1. The fixed-point C programs and its compiled codes are shown in Fig. 8 and 9.

Table 1: Performance comparison according to the implementation methods

Method	Manual assembly[1]	Integer C[2]	Fixed-point C[3]	Floating-point C[4]
SQNR (dB)	64.09	21.27	57.69	-
# of machine cycle	16	28	43	928

[1] Assembly program translated by the *Autoscaler*[3].
[2] Texas Instruments' C compiler for TMS320C50, uses the '`int`' data type[8].
[3] Our fixed-point C compiler prototype, employs the '`fix`' data type.
[4] Texas Instruments' C compiler for TMS320C50, uses the '`float`' data type.

```
#define A0  1.6832491279784571603
#define A1  (-0.78434859552783486869)
#define B1  (-0.66910090629842178256)

fix(14) u[3];
fix(15) y;
fix(10) x;
...
/* delay */
u[2] = u[1], u[1] = u[0];

u[0] = A0*u[1] + A1*u[2] + xin;
y = u[0] + B1*u[1] + u[2];
...
```

Figure 8: A Fixed-point C program for the 2nd order IIR filter

```
LACC    _u+1, 16
SACH    _u+2, 0
LACC    _u, 16
SACH    _u+1, 0
LAR     AR2,#1
LT      _u+1
MPY     #6BBAh ; 1.6832491 with IWL 1
SPM     1
PAC
MAR     *,AR2
MAR     *0+
SACH    *, 0
LAR     AR2,#2
LT      _u+2
MPY     #9B9Ah ; -0.78434861 with IWL 0
SPM     1
PAC
MAR     *0+
SACH    *, 0
LAR     AR2,#1
MAR     *0+
LACC    *, 16
LAR     AR2,#2
MAR     *0+
ADD     *, 15
ADD     _xin, 11
SACH    _u, 1
LAR     AR2,#3
LT      _u+1
MPY     #0AA5Bh ; -0.66910088 with IWL 0
SPM     1
PAC
MAR     *0+
SACH    *, 0
LACC    _u, 15
LAR     AR2,#3
MAR     *0+
ADD     *, 15
ADD     _u+2, 15
SACH    _y, 0
...
```

Figure 9: The compiled version of the above fixed-point C program

The comparison results of the manual assembly, integer C, and fixed-point C programs are shown in Table 1. Texas Instruments C compiler for the TMS 320C50 are used for compiling the integer and the floating-point C programs, while our prototype compiler is used for the fixed-point C program. The comparison results show that the SQNR of the integer C program is hardly acceptable. But, the SQNR of the fixed-point C program is fairly close to that of the assembly program. Note that the fixed-point C compiler employs a conservative scaling strategy, which results in a slight increase of the quantization noise when compared with the manually assembled program.

The execution speed of the fixed-point C program is nearly 20 times faster than that of the floating-point version. Although the execution speed of the fixed-point C program is about 2.7 times slower than that of the manually coded assembly program, the gap can be narrowed down by employing several compiler optimization techniques[7]. Note that the execution speed difference between the integer C and the fixed-point C programs is mainly due to the code optimization capability of the compilers, not to the data types.

6 Concluding remarks

A fixed-point C language is defined, and a prototype compiler is designed to evaluate the usefulness of the language for digital signal processing applications. The comparison results show that the new language and its compiler can provide an acceptable compromise to the users of the fixed-point digital signal processors in terms of the SQNR, execution speed, and the development effort. The fixed- point C program can be prepared very easily by estimating the range of each variable using the *Fixed-Point Optimization Utility* and modifying the data type of original floating-point C application programs. This modification procedure does not need any change of original algorithms, such as coefficients and data scaling into the integer domain as required in the design of integer programs.

References

[1] *Buyer's Guide to DSP Processors*, Berkeley Design Technology, Inc.

[2] *TMS320C5x User's Guide*, Houston, Texas Instruments Inc., 1993.

[3] Seehyun Kim and Wonyong Sung, "A floating-point to fixed-point assembly program translator for the TMS320C25," *IEEE Trans. Circuits and Systems*, Vol. 41, no. 11, pp. 730–739, Nov. 1994.

[4] Seehyun Kim, Ki-Il Kum, and Wonyong Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs," in *Proc. 1995 IEEE Workshop on VLSI Signal Processing*, pp. 197-206, Oct. 1995.

[5] C. Fraser and D. Hanson, "A retargetable compiler for ANSI C," *SIGPLAN Notices*, Vol. 26, no. 10, 1995.

[6] C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin/Cummings, 1995.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison Wesley, 1986.

[8] *TMS320C2x/C2xx/C5x Optimizing C Compiler*, Houston, Texas Instruments Inc., 1995.