FIXED-POINT C COMPILER FOR TMS320C50 DIGITAL SIGNAL PROCESSOR

Jiyang Kang
Wonyong Sung
School of Electrical Engineering
Seoul National University, KOREA
jiyang@hdtv.snu.ac.kr and wysung@dsp.snu.ac.kr

ABSTRACT
A fixed-point C compiler is developed for convenient and efficient programming of TMS320C50 fixed-point digital signal processors. This compiler supports the '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. Several target specific code optimization techniques are employed to improve the compiler efficiency. The empirical results show that the execution speed of a fixed-point C program is much about an order of magnitude, faster than that of a floating-point C program in a fixed-point digital signal processor.

1. INTRODUCTION
The reduction of development time by employing a high-level language is very much needed for the programming of digital signal processors. 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] [2] especially because of the overhead in executing floating-point operations using a 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. Since the lower part of the 31 bit product is stored as the result in the integer multiplication, the input data to the multiplier have to be severely scaled down in order to prevent overflows. It is also tedious to develop integer programs.

In this paper, we propose a C compiler that supports the 'fix' data type and corresponding arithmetic operations to solve these problems. The execution speed and the finite word-length effects of the floating-point, integer, and fixed-point implementations are compared by using biquad IIR digital filter and ADM coder programs. Texas Instruments’ fixed-point digital signal processor, TMS 320C50, is employed [3], and the GNU compiler, gcc, is modified. Discussions on our prototype fixed-point C compiler based on the tcc retargetable C compiler [4] is shown in [5]. Relevant approaches for the Motorola 56000 DSP and automatic scaling at assembly level can be found in [6] and [7], respectively.

2. FIXED-POINT ARITHMETIC RULES
An integer variable or a 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 [7] [8]. For this purpose, we employed a fixed-point data type that can have an individual integer word-length as follows:

\[ \text{fix} (\text{integer wordlength}) \text{ variable name;} \]

Note that the range \( \{ R \} \) that a variable can represent and the quantization step \( Q_s \) are dependent on the integer word-length \( (IW L) \) as follows.

\[
-2^{IW L} \leq R < 2^{IW L}
\]

\[
Q_s = 2^{-15-(IW L)}
\]

The arithmetic rules based on this fixed-point data representation and a hardware data-path having a 16 bit by 16 bit two's complement multiplier, 16 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 modified by arithmetic shift operations. An arithmetic right shift of 1 increases the integer word-length by 1. For example, the program shown in Fig. 1-(a) can be compiled as depicted in Fig. 1-(b). Note that SFR(1) represents an one-bit arithmetic right shift.

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

y = x + y;
```

(a)

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

(b)

Figure 1. Fixed-point add-operation
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. 2. Note that an extra sign bit is eliminated in the two’s complement multiplication process. In the multiplication of the ‘fix’ type data, the upper 16 bit part, P(30:15), is stored as the product, while the lower 16 bit part is stored in the integer multiplication (Fig. 2). For example, the program shown in Fig. 3-(a) can be compiled as depicted in Fig. 3-(b) by the fixed-point C compiler. Note that SFL(2) represents a two-bit left shift.

\[
\begin{align*}
\text{fix}(2)\ x; & \quad /\ast \text{IWL of 2 }\ast/ \\
\text{fix}(3)\ y; & \quad /\ast \text{IWL of 3 }\ast/ \\
y = x \ast y; & \quad (a) \\
P(31:0) = \text{MUL} \ (x, y); & \\
y = \text{SFL}(2)\ P(30:15); & \quad (b)
\end{align*}
\]

Figure 3. Fixed-point multiply-operation

3. COMPILER STRUCTURE

The proposed fixed-point C compiler for TMS320C50 is based on the GNU C compiler, gcc, from the Free Software Foundation [9]. The overall compiler structure is shown in Fig. 4. Not only the back-ends of the gcc, 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 code optimization.

The overall compilation process is as follows. In the first step, the compiler preprocesses source program, and obtains the integer word-length information of the variables used in the source program. The integer word-length can be given manually or determined automatically using the Fixed-Point Optimization Utility [8]. The Fixed-Point Optimization Utility converts the floating-point variables into a corresponding C++ range estimation class, and estimating their integer word-lengths by simulation. Note that the integer word-length of a constant is determined automatically by the compiler. Source programs are converted into syntax trees by the parser. These syntax trees are then transformed into intermediate representations known as register transfer language (RTL) expressions by the RTL generator.

In the second step, the compiler conducts several machine-independent optimizations that are supported by gcc, such as jump optimization, common subexpression elimination, and allocates registers.

In the next step, after appropriate peephole optimizations, the code generator produces the intermediate assembly code by annotating the RTL representations based on the instruction patterns defined in the target description file. At the same time, the compiler divides the source program into smaller blocks. The integer word-length of the accumulator in each block is uniquely determined on the basis of the integer word-length informations of the operands involved in the arithmetic operations. The integer word-lengths of temporary variables which are assigned automatically by the compiler are also determined in this step.

Finally, with the unscaled intermediate assembly code and various integer word-length informations, the scaler performs the scaling phase by applying the fixed-point arithmetic rules discussed before. The number of shifts remained undetermined in the intermediate assembly code are determined according to the word-lengths of relevant variables and the accumulator. The compilation steps focused on scaling procedures are shown in Fig. 5.
4. CODE OPTIMIZATIONS

In this compiler, several post optimization techniques are implemented. In TMS320C50, certain instructions are controlled by mode variables, such as the product-shift mode (PM), the data page pointer (DP), and the sign-extension mode (SXM). In order to use the direct addressing mode to access global variables efficiently, it is especially important to find and remove any redundant instructions to load the data page pointer (i.e., LDP). Whenever a redundant LDP instruction is identified and eliminated, two machine cycles can be saved. With the informations from the reaching definition analysis [10], we can eliminate redundant mode setting instructions.

In TMS320C50, another optimization point is to improve the allocation of local variables. Since address change more than one requires additional instructions (i.e., ADRK or SBRK), minimizing the number of such instructions by optimal allocation of local variables in order to utilize the post-increment and post-decrement addressing modes can improve the overall performance [11] [12]. In addition, removing redundant offset-updating instructions also greatly contributes to the performance of the indirect addressing mode. We employed an optimization algorithm which uses a table of variables to track the values of auxiliary registers and the auxiliary register pointer similar to the algorithm shown in [13].

Besides these optimizations, other peephole optimizations such as instruction compaction and loop optimization are also performed to fully exploit the architecture specific features of the TMS320C50. The instruction compaction is to combine multiple operations into a parallelized instruction and the loop optimization is to improve the performance of the test-and-branch sequence at the end of loop body.

5. EXPERIMENTAL RESULTS

A biquad IIR filter and an adaptive delta modulator were taken for the experimental examples. We used the TMS320C2x/5x optimizing C compiler from Texas Instruments (version 6.00) [14] to compile the floating-point C versions of the example programs, while our fixed-point C compiler was used for the fixed-point C versions. We set the optimization levels of both compilers to be the level 0 in all of our experiments.

5.1. Biquad Filter

A biquad digital filtering program shown in Eq. (3) and (4) is implemented using floating-point and fixed-point data types.

\[
\begin{align*}
    u[n] &= 1.683u[n-1] - 0.7843u[n-2] + x[n] \\
    y[n] &= u[n] - 0.669u[n-1] + u[n-2]
\end{align*}
\]

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 60 dB as shown in Table 1. In terms of the execution speed, it is shown that the fixed-point C program is about 23 times faster than the floating-point C program and 2.5 times slower than the hand-coded assembly program. Note that an optimally scaled integer C program only yields the SQNR of 21.27 dB [5].

<table>
<thead>
<tr>
<th>Method</th>
<th>Manual Assembly</th>
<th>Fixed-Point C</th>
<th>Floating-Point C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQNR (dB)</td>
<td>64.00</td>
<td>57.69</td>
<td>-</td>
</tr>
<tr>
<td>Machine Cycles</td>
<td>16</td>
<td>40</td>
<td>9.38</td>
</tr>
</tbody>
</table>

5.2. Adaptive Delta Modulator

An adaptive delta modulator program shown in Fig. 6 is implemented using floating-point and fixed-point data types. The compiled codes are shown in Fig. 7. Since this algorithm contains several conditional branches, the upper and the lower bounds of the number of machine cycles are presented here. In this example, the performance gap between the manual assembly program and the compiled fixed-point C program is narrowed. The fixed-point C program is 12 to 13 times faster than the floating-point C program, but it is still slower than the hand-coded assembly program by a factor of 1.42-1.58.

<table>
<thead>
<tr>
<th>Method</th>
<th>Manual Assembly</th>
<th>Fixed-Point C</th>
<th>Floating-Point C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQNR (dB)</td>
<td>64.00</td>
<td>57.69</td>
<td>-</td>
</tr>
<tr>
<td>Machine Cycles</td>
<td>16</td>
<td>40</td>
<td>9.38</td>
</tr>
</tbody>
</table>

6. DISCUSSIONS

In this paper, a fixed-point C compiler which introduces a new data type to support fixed-point arithmetic rules for TMS320C50 digital signal processor is presented. The comparison results show that the fixed-point C compiler can provide an acceptable compromise to the users of the fixed-point digital signal processor in terms of SQNR, execution speed, and the development efforts. Although the execution time of the compiled fixed-point code is 1.5 to 2.5 times longer than that of the hand-coded assembly program, the compiler can be improved by applying better target specific optimization techniques in the future. At this moment, the developed compiler does not support local fixed-point variables and fixed-point structures, but is useful for fixed-point prototyping of real-time digital signal processing applications.

ACKNOWLEDGMENTS

The research described in this paper was supported by the LG Electronics Research Center, KOREA.

REFERENCES

Figure 6. ADM example: fixed-point C program

```c
fix(12) sr; // reconstructed signal */
fix(16) step; // step size */
fix(12) se; // estimated signal */
fix(13) d; // difference */

/* delay line */
sgn2 = sgn1;
sgn1 = sgn0;
/* beginning of adm */
se = ACOEF * sr;
d = in - se;
sgn0 = SIG0(d);
code = sgn0;
if((sgn0&sgn1&sgn2)||!(sgn0&!sgn1&!sgn2))
    step = ALPHA * step + BETAf;
else
    step = ALPHA * step;
return (sr = (sgn0&(se+step*BIAS))
: (se=step*BIAS));
```

Figure 7. ADM example: compiled fixed-point C program

```c
LT _sr
MPYK 7AE1h
PAC
SACH _se, 1
LABP AB6
SBRK 3
LAC *16
SUB _se, 15
ADRR 5 ;
SACH *0
SACH d, O
...
LDPK _step
LT _step
MPYK 7C29h
PAC
ADDRK 7FFFH, 12
SACH _step, 1
B L6

L3:
LDPK _step
LT _step
MPYK 7C29h
PAC
SACH _step, 1

L6:
LDPK _sgn0
LAC _sgn0
BZ L7
LDPK _se
LAC _se, 16
ADD _step, 14
ADDRK 7FFFH, 11
B L10

L7:
LDPK _se
LAC _se, 16
SUB _step, 14
SUBK 7FFFH, 11

L10:
LABP AB6
ADRR 2
SACH *0
LAC *16
LDPK _sr
SACH _sr, 0
LAC *16
SUBK 2 ;
```