C/C++ COMPILER SUPPORT FOR SIEMENS TRICORE DSP INSTRUCTION SET

Hao Shi, Roger Arnold and Karl Westerholz

Siemens Microelectronics, Inc
2480 North First Street
San Jose, California 95131, USA

ABSTRACT

How to make compilers more useful for developing DSP applications and reduce reliance on assembly coding has long been a topic of interest in the DSP community. This paper presents Siemens solutions for supporting its TriCore DSP/microcontroller architecture, including SIMD instructions, at the C/C++ level. Two solutions based on either extending C/C++ language with the new built-in DSP data types or developing an external DSP class library are investigated. First cut implementations of both methods have achieved 80% coverage of the TriCore instruction set, which is 30 percent higher than the coverage before DSP support was added.

1. INTRODUCTION

A DSP processor distinguishes itself from a general-purpose processor by its cycle-efficient instruction set [1]. Single-cycle multiply-accumulate (MAC) instructions, saturated overflow behavior and the left-justified fraction arithmetic operations are the norm in a DSP instruction set. In addition, dedicated addressing modes such as bit-reverse and circular buffer addressing, together with load/store instructions that can be issued in parallel with the arithmetic operations, make the Fast Fourier Transform (FFT) and the digital filter algorithms very efficient. A zero-overhead loop instruction is essential for speeding-up the tight loops of vector manipulation. With the fast growth of multimedia applications in recent years, SIMD (Single Instruction Multiple Data) instructions are also becoming popular.

Supporting these DSP specific instructions in C/C++ language has long been a topic of interest [6][7][8], since the ability to program DSP applications in these languages would help to reduce system cost and time-to-market.

A primitive way to do so is to wrap every DSP instruction with a C intrinsic or library function [2]. The advantage of this approach is that the applications, built upon these basic DSP functions, can be ported without extra effort if all the basic DSP functions are ported. The explicit DSP function layer, however, is not transparent to both C language rules and the C programmer. This leads to less efficient code and poor code readability

Beyond the basic functions that wrap DSP instructions for C programmer, TriCore's development tools introduce a second wrapper that wraps the basic DSP functions with new data types or classes and the corresponding operations. With this second wrapper, the layer of the basic functions is hidden. One does not have to remember several hundred basic functions in order to write DSP C/C++ code that is efficient and easy to read.

The rest of this paper is organized as follows. In Section 2, we discuss the definition of the DSP extensions or classes: what are the data types that we need and why. Some detailed rules of the data type operations will also be discussed. In Section 3, we then compare two different implementation approaches and evaluate the effectiveness of the language construction in supporting DSP. Finally, we summarize major findings and outline future work.

2. DSP SUPPORT IN C/C++

2.1 Four Levels of Support

As shown in Fig. 1, there are four levels of support for DSP C/C++ code development.

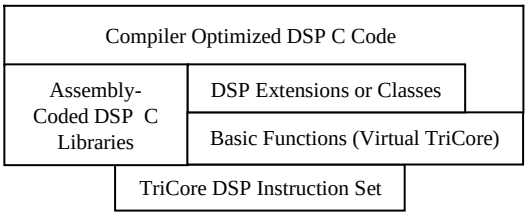

Figure 1. Four Levels of DSP C Language Support

First, there are assembly-coded DSP C libraries that take full advantage of the TriCore DSP instructions, including SIMD instructions. These can be called directly from C programs. At a parallel level, we have a layer of the basic functions that wrap the individual DSP instructions of the architecture and make them accessible from C code. From a C programmer's point of view, this layer of basic functions is a "virtual TriCore." Above that level, an abstract layer of the data types of either C language extensions or C++ classes wraps the basic function layer to make it transparent to C/C++ programmer. Finally, at the top level, the TriCore compiler can automatically identify the proper cases to insert MAC, zero-overhead loop and other efficient DSP instructions based on the expression and the loop information. In this paper, we will concentrate on the abstract layer of data types. To simplify our discussion, we use the term *data type* to indicate both built-in C language DSP extensions and the dedicated DSP C++ classes when it does not cause confusion.

Now, in what sense do we use the term C extensions? Any programming language is characterized by a set of *data types*, an associated set of *operations* on those data types, and a *syntax*, or

set of rules for declaring objects and expressing operations on those objects. The set of *built-in* data types and operations of the C language is inadequate for use in DSP applications; additional types and operations are needed. Adding them to C introduces some new keywords, and requires new operators to be added to the *intermediate language* (IL) that the compiler uses to translate the C source code for the code generator. However, it does not require any change to the basic language syntax. C programs that use the new types and operations still look like C programs, both to the programmer and to the C compiler.

The C++ language differs from C, in that the language itself provides a built-in means to extend the data types and operations available to programmers. The data types and operations needed for DSP support can be expressed in C++ *class libraries*, without resort to language extensions.

There are subtle but significant differences between C language extensions vs. C++ class libraries as bases for supporting DSP applications. Before getting to those, however, we will examine some of the specific data types and operations required.

2.2 Fraction Data Types

Integers in C language are right-justified fixed-point data types. Left-justified fixed-point data types (pure fraction), however, are also very useful in many DSP applications. Fractions offer better dynamic range in both input and output side of a multiply-accumulator of fixed word length. Therefore, TriCore provides full support for the fraction data types [5].

To make these cycle-efficient fraction arithmetic instructions available to a C/C++ programmer, three fixed-point data types have been introduced. The formats of these new data types are illustrated in Fig. 2. Both the 32-bit fraction *_fract* and the 16-bit short fraction *_sfract* are defined in the range of [-1, 1). The 64-bit signed accumulator data type *_accum* is defined in the range of $[2^{-17}, 2^{17})$ with both whole number and fraction parts. It is introduced for the accumulation (or multiply-accumulation) of the fraction data types with 17 guard-bits.

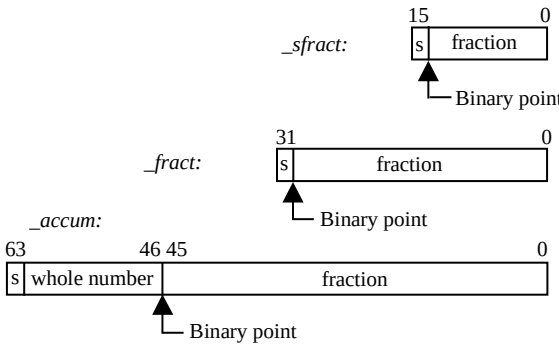

Figure 2. Format of the New Fixed-Point Data Types

Logic and arithmetic operations of the fraction operands are the same as those for the integers except multiplication and division. Dedicated fraction multiplication is implemented by a TriCore DSP instruction that does both multiplication and binary point adjustment in one cycle. The division operator "/" is defined as to return a fraction result, and thus assumes that the magnitude of the dividend is smaller than that of the divisor. A second fraction division function, which corresponds to integer division, is provided for the case where the magnitude of the dividend is larger than that of the divisor.

According to the C language rule, when mixed data types appear in the same arithmetic operation, the less advanced data type should be first promoted to the advanced data type before the evaluation is done. The promotion rules for fraction data types are as follows: *_sfract* is the least advanced data type, *_fract* the second least advanced, and *_accum* between *long* and *int*. There are two exceptions of these promotion rules in the mixed integer-fraction multiplication and division. Since they are common operations in DSP applications, special rules are defined for better accuracy and efficiency. A commutative multiplication operator "*" of an integer and a fraction will always give a fraction regardless of whether or not the magnitude of the product is less than one. An additional basic function that returns an integer result from of a mixed multiplication is provided. In mixed division, however, the data type of the quotient follows that of the dividend. Thus, *long/_fract* gives a *long*, while *_fract / long* gives a *_fract*.

Data type conversions are important not only for supporting mixed data type operations but also for constant fraction representations. A constant should be treated implicitly as a fraction if (1) its value is in between [-1, 1), and (2) it is assigned to a fraction variable or appears in a fraction expression. Similarly, an *_accum* constant can be defined as a constant with its value in the range of $[-2^{17}, 2^{17})$, and is assigned to an *_accum* variable, or appears in an *_accum* or a fraction expression. Since speed is most important for DSP applications, converting a fixed-point constant to its internal expression should be done at compile time.

2.3 Packed Data Types

SIMD instructions exploit the parallelism of processing multiple sub-word data fields within one instruction. Although it is theoretically possible for compiler to pack data implicitly by unrolling a loop and merging iterations, doing so is beyond the capability of available compilers. Instead, explicitly packed data types are defined. They offer the flexibility for C programmer to use the SIMD instructions consciously. Packed data types and their operations are especially useful for image processing.

Fig. 3 shows the format of the *_packb* and *_packhw* data types. We can also pack two *_sfract* data types to form a complex fraction data type. Special efforts were made to make TriCore manipulate complex numbers efficiently.

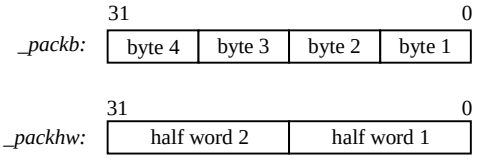

Figure 3. Format of the Packed Data Types

The way the packed data types are manipulated is determined by their dual characteristics. On one hand, data of the packed type

is moved around, operated upon, and passed to or returned from C functions as one unit. On the other hand, packed data types are structures with private members of the same basic types. Therefore, basic functions to initialize, access, and mutate these members have to be provided. Since a constant of a packed data type is defined as a 32-bit integer, a 32-bit integer can be used to initialize a packed variable.

We have implemented two sets of accessors and mutators. The first set, for example,

char _getbytei(_packb)*
void _setbytei(_packb, char)*

uses TriCore's sub-word load/store instructions *ld.b* and *st.b*. The second set, for example,

char _extractbytei(_packb)
_packb _insertbytei(_packb, char)

uses TriCore's insert/extract instructions. Choosing the right set will help generate the most efficient DSP code.

Arithmetic operations on the packed data types include addition, subtraction, multiplication, negation, absolute value, comparison, and count leading sign. Due to the inherent array characteristics of packed data types, no simple division is defined for them. Logic operations for packed data types include shift and bit-wise logic operations.

2.4 Circular Buffer Pointer

Many DSP applications use circular buffers for real-time data. Maintaining a circular buffer involves updating the buffer index modulo the buffer length. TriCore provides a specific addressing mode and a data structure for this kind of application. By introducing *_circ* qualifier or template, we made this special addressing mode available to the C/C++ programmer. Fig. 4 shows the format of this 64-bit data type.

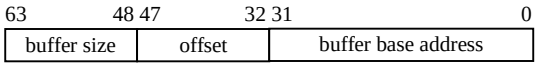

Figure 4. Format of the Circular Buffer Pointer Data Type

Similar to the packed data types, a circular buffer pointer also has dual characteristics. It can be passed to or returned from C functions as one entity, and has to be initialized with a dedicated basic function: *void _circ *_initcirc(void*, size_t, int)*. The first parameter points to the base address of the circular buffer that has been dynamically or statically allocated. The second parameter is the size in bytes of the circular buffer. The third one is the offset in bytes of an element in the buffer.

A circular buffer pointer can be converted into a normal pointer of any type. The converted address is equal to the sum of the base address and the offset. The concept of conversion is essential for circular buffer pointer operations, because comparison of two pointers, judgement of a null pointer and passing a circular buffer pointer to a pointer parameter declared as *void** are all based on the conversion.

Other circular buffer pointer operations include increment, decrement, subtraction or addition of a constant, and subtraction of two circular pointers. Subtraction of two circular buffer pointers to the same data type is legal even if the base addresses of them are different.

2.5 Saturating Data Types

Reflecting analog circuit behavior, DSP algorithms often expect saturation on overflow. TriCore provides a full set of arithmetic instructions to deal with both wrapping and saturating overflow. Although saturation behavior can be implemented as special operators, we chose to bind the overflow behavior with the new data types to maintain the second level abstraction of C/C++ language.

In C language extensions, a new qualifier, *_sat*, is introduced to follow the basic data types *int*, *long*, *_sfract*, and *_fract*. Qualifier *_sat* signals that all arithmetic operations related to the data type are performed with saturated instructions. Although conversions between *_sat* and non-*_sat* data types are not necessary, one can use type-cast to enable/disable saturated operations. The following code shows how type casting works in this context.

```
_fract _sat a;
_fract b, c;
…
/* Sum and product evaluated with saturation */
a = a * b +  c;.
a = (_fract _sat)b + c;
…
/* Sum and product evaluated without saturation */
b = (_fract)a + b * c;
```

In C++ implementation, a saturated class was developed for each of the type *int*, *long*, *_sfract* and *_fract*.

3. IMPLEMENTATION

3.1 Class Library vs. Language Extensions

Both the class library and the extension approaches have been investigated via direct implementation. Although there are pros and cons for each approach, both can be used to implement the features discussed in Section 2. In this sub section we compare the two approaches.

Class Library

The idea of the class library approach is to make the DSP support independent of the compiler so as to achieve better language portability and compatibility. This also means, however, we have to construct many overloaded functions to handle cases that, for built-in types, are handled automatically by the compiler. For example, any commutative binary operators between mixed data types have to be implemented twice, according to which operand is first. Also, overloaded functions in C++ are resolved on the basis of input argument types alone; the desired result type cannot be considered. That often leads to conversions that could be avoided if intelligence about the types were built-in to the compiler. To compensate, knowledge about the types must be added to the code generator and optimizer.

Another limitation of the class library approach is in the handling of literal constants. C++ does not provide any means to associate

compile-time processing of a literal string with a user-defined class type. The only way to construct a constant array of *_fract* data values—e.g., the coefficients of an FIR filter—is to overload the array with an array of *short* values (16-bit integers), initialized with integer values whose binary representations are equivalent to the desired *_fract* values.

Because features implemented with C++ *template* mechansism are not available to pure C language programs, it is impossible to support exactly the same level 3 programming interface with the two approaches. Nevertheless, the ability to develop software on a host system to which the DSP class library has been ported makes the class library approach attractive..

Language Extensions

The idea of using language extensions is to achieve the best code optimization and programming flexibility by introducing native data types. For example, it is easier in the language extension approach to implement the fraction operators inline, and consequently save the overhead of function calls often imposed by class library approach. The down side is the lack of high-level portability and the requirement for compiler front-end changes. Note, however, that the front-end changes are not fundamental; they are mostly just a matter of applying common C rules to an extended set of data types.

3.2 Virtual TriCore and Porting

As explained earlier, "virtual TriCore" is the term given to the layer of basic functions used to implement operations on the DSP data types defined in the DSP extensions or class libraries. Table 1 presents the measure of TriCore in supporting these functions. It shows that about 62% of the basic DSP functions are implemented with only one TriCore instruction.

Table 1. Measure of TriCore's High Level Language Support

Number of instructions in DSP intrinsic function	Percentage of intrinsic functions of specified number of instructions
1	62%
2	9%
3	10%
≥4	19%

From the TriCore designer's point of view, on the other hand, the more a compiler covers the TriCore instruction set, the better it supports TriCore. Statistics show that the compiler with the new data types covers 80% of TriCore instruction set, while compiler without the new data types covers only 50% of TriCore instruction set.

The above indicates that the virtual TriCore and TriCore match very well in supporting DSP C programming. Virtual TriCore also provides a means that makes porting of C code above it much easier.

The new DSP data types present the virtual TriCore in an intuitive way to C programmers. To port the C code above the virtual TriCore, a C preprocessor can be used to develop this hidden layer. It is only at this level, both C and C++ DSP codes become portable. Here, porting the code means porting the virtual TriCore to the target platform. When porting C extensions, one has to define the DSP data types as standard C types or structures with equivalent memory size, and interpret the variables of these data types correctly.

4. SUMMARY

This paper presented some Siemens solutions for supporting its TriCore DSP instruction set, including SIMD instructions, at C/C++ level. A solution that extends C/C++ language with new built-in DSP data types and a solution based on external DSP classes were investigated. The first cut implementation shows that 80% of the TriCore instruction set coverage can be achieved with either method. The major advantage of the class library approach is its high-level portability. The DSP extensions, on the other hand, deliver better code. The "virtual TriCore" concept was introduced as a C level language interface of TriCore DSP instructions and as the bridge for porting.

The next logical step is to optimize both TriCore and virtual TriCore to further increase one-instruction basic function rate. Optimization on the fourth level in Fig. 1 is of great interest. Wrapping the level one DSP library to achieve an even higher level abstract language construction is also under investigation.

5. ACKNOWLEDEMENT

We want to thank Craig Franklin and Green Hills compiler team, and Dick Streefland and Tasking compiler team for the hard development work on which this paper is based. Also thanks to Dr. Steve Zhang for his reviewing the paper and his good suggestions. Finally, thanks to Rod Fleck and Dieter Stengl of Siemens who give constant support to this effort.

6. REFERENCES

[1] H. Shi and R. E Owen, "Evaluating DSP Instruction Set Extensions in High-performance Processors." *Class Notes of DSP World Spring Design Conference*, April 1998.

[2] *Visual Instruction Set User's Guide*, Sun Microelectronics, May 1995.

[3] *ARM Signal Processing Architecture Reference Manual*, Advanced RISC Machines Ltd., March 1997.

[4] M. Levy. "DSP-Architecture Directory". *EDN*, May 1998.

[5] *TriCore Architecture Manual*, Siemens AG, Sept. 1997.

[6] K. Baldwin, R. Piedra et al, "Guildlines for Efficient C Code Generation in Accumulator-based DSPs." *ICSPAT Proceeding*, Sept. 1997.

[7] J. Mulder, T. Grotker et al, "C++ Based Implementation of Mixed Control/Data Flow Systems." *ICSPAT Proceeding*, Sept. 1997.

[8] R. J. Ridder, "Trends in Application Programming for DSP" *ICSPAT Proceeding*, Sept. 1997.