01 Jan 1988

A Practical C Language Compiler/Optimizer for Real-Time Implementations on a Family of Floating Point DSPs

J. Hartung

Steven L. Grant
Missouri University of Science and Technology, sgrant@mst.edu

S. G. Haigh

D8.2

A PRACTICAL C LANGUAGE COMPILER/OPTIMIZER FOR REAL-TIME IMPLEMENTATIONS ON A FAMILY OF FLOATING POINT DSPs

John Hartung, Steven L. Gay, Stephen G. Haigh

AT&T Bell Laboratories
Holmdel, N. J.

ABSTRACT

Digital Signal Processors (DSPs) have traditionally been used in realtime applications with very high data throughput. For this reason, system designers have been reluctant to accept the degradation in performance inherent in machine code compiled from high-level languages such as "C". The problem is compounded by the fact that DSPs use pipelined architectures to achieve their high data throughput, resulting in hazards and latencies between instructions. Simple compiler implementation cannot take advantage of latent instructions, resulting in a conservative and inefficient executable program. This problem has been addressed in the C compiler package for the AT&T *WE* ® DSP32 family by the addition of a post optimizer and an extensive application library.

This paper gives an overview of the DSP32 family architecture, describes the operation of the basic compiler and optimization strategies, and provides an example of the use of the compiler.

1. *INTRODUCTION*

Digital signal processor performance has improved by more than an order of magnitude in less than one decade. Early fixed-point DSPs could implement simple signal processing algorithms, such as a DTMF receiver, on a single channel, in real time, when coded in assembly language. Current devices such as the AT&T *WE* ® DSP32C Digital Signal Processor have sufficient address space and performance to implement more complex applications, such as image and graphics processing. These tasks require considerably more code as well as a much larger data space. Although large gains have been made in DSP hardware, it is only recently that high level language compilers have been introduced for these devices. It is more difficult to write an efficient compiler for a pipelined processor such as a DSP with a specialized instruction set than for a general purpose processor. To provide an adequate level of efficiency, it is essential that the special assembly language instructions provided in a DSP be used by the compiler, and that code optimization be performed to make optimum use of the pipeline.

This paper describes a C compiler for the AT&T family of floating point DSPs (DSP32 and DSP32C). First, the architecture of the DSP32/C is described. In the following sections the operation of the basic compiler, and optimization strategies are described. Finally, an example application is given with emphasis on the requirements needed for the source code to produce efficient executable code.

2. *DSP32/C ARCHITECTURE*

The DSP32 is a single chip, NMOS, programmable digital signal processor with 56 kbytes of addressable memory, including 2 kbytes ROM and 4 kbytes RAM internal to the chip [1]. The DSP32C is a CMOS version of the NMOS chip with faster clock speed, enhanced instruction set and 16 Mbytes address space [2].

High throughput is achieved by an architecture which includes two processing units. The control arithmetic unit (CAU) has 21 integer registers and a microprocessor-like instruction set for operations such as basic ALU functions, address generation, and program control. The data arithmetic unit (DAU) is based on a high speed floating point multiplier/accumulator and has 4, 40-bit floating point accumulators and a pipelined floating point arithmetic instruction set designed for DSP applications. Also, to speed throughput, on chip serial and parallel DMA and input/output conversion instructions are provided.

The DSP32 has a 160 nS instruction cycle time, and can execute 12.5 million floating point operations per second (MFLOPS). The DSP32C is twice as fast as the DSP32.

The DSP32 C compiler is implemented for both devices, but some new features of the DSP32C give additional benefits to systems developers using the compiler. A number of additional instructions have been added to the device, including a zero overhead loop control instruction, an IEEE standard floating point format conversion instruction, and a bit reversed addressing mode for the FFT algorithm. The wider 24 bit address bus on the DSP32C enables much larger memory resident applications programs than are currently available on the NMOS device.

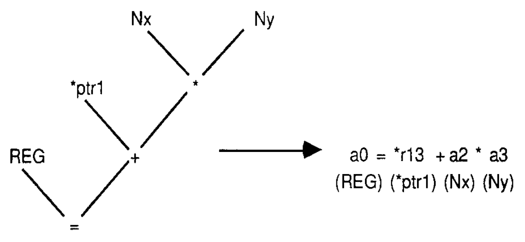
3. *C COMPILER*

The DSP32 C compiler is a complete implementation of the C language. It supports all integer data types and single precision floating point. The compiler is an implementation of the UNIX portable C compiler, which guarantees portability of C programs from UNIX System V machines to the DSP32. It also facilitates upgrades to new C standards such as ANSI C, and language extensions, such as C++.

In addition to handling generic operations, like +, -, *, /, %, &, | and ~, the DSP32 C compiler is capable of taking full advantage of the DSP32's rich instruction set for digital signal processing applications. For example, many complex operations such as the multiply accumulate instruction: a = b + c * d; or the tap update instruction a = b + (c=d) * e; are compiled into a single DSP32 instruction.

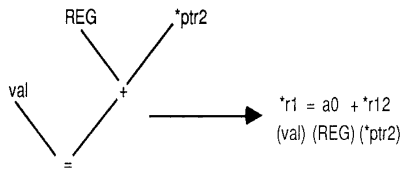
Figure 1 shows how a complex C expression is compiled into DSP32 code. The compiler front-end is common to any UNIX portable C compiler. Its job is to parse the incoming C program and generate an internal representation of binary C operators, called a C tree. During code generation, the back-end matches portions of the tree to a table of templates. These templates describe how basic C operations can be mapped to available DSP32 instructions. Code generation selects the best matching template available for a given C expression and generates assembly code based on information supplied with the template. The template matching strategy is called "maximal munch" because it selects one of many ways to generate code for an operation by biting off the biggest part of the tree it can match.

In figure 1, the whole expression cannot be implemented by a single DSP32 instruction. The code generator matching algorithm trys to match as large a part of the C tree as it can in order to generate the most efficient code. In this case the multiply and accumulate is matched first as shown below.



Because this operation will not result in the final expression value, an intermediate value is generated, which is saved in a scratch register, REG, which is assigned to a0. r13 was assigned to ptr1, a2 to Nx, and a3 to Ny.

After this operation, the remaining tree looks like this:



The already matched part was replaced by the scratch register specified by REG. This subtree is then matched to a single DSP32 instruction and the generated code is:

```
a0 = *r13 + a2 * a3
*r1 = a0 + *r12
```

where r1 is used to point to val, and r12 is assigned to ptr2.

4. *OPTIMIZATION*

Since DSP32's are typically used for high speed, real time applications, it is important for the compiler to be very efficient. The DSP32 C compiler has two methods of generating highly efficient code:

1. The compiler is able to match the best available DSP32 instruction(s) to a C expression. This includes special DSP instructions like the ones shown above and special, low overhead loop control instructions.

2. A code optimizer follows the compilation phase to improve the execution speed and program space of the compiled program. The optimizer can treat a program in a more global way than the compiler alone.

Unlike a compiler, which has a specific well defined task to perform, i.e. conversion of a C source program into assembler code, the process of optimization is more heuristic. Some commonly used general purpose microprocessor optimizations and some pipeline optimizations are described below.

Generic optimizations are largely aimed at making better use of the DSP32's instruction set and resources, and improving control flow. In general, integer operations in RISC chips, such as the DSP32, take place in machine registers. Global register allocation and register overloading modify the compiler generated program to optimally use machine registers to hold operands. Control flow is improved by using efficient loop control instructions and expanding loops and functions in-line.

Pipeline optimization analyzes program data flow and calculates data dependencies due to pipelining. Then, in order to satisfy data dependencies, pipeline optimization reorganizes the instruction sequence if possible or uses nops to flush the pipeline. Some data dependencies cannot be resolved with the information passed down from a basic C program. Extra user control of the optimization process is provided at the C source level for such cases.

5. *APPLICATION LIBRARY*

To aid program development and improve efficiency, a library of highly optimized generic and DSP functions are provided with the compiler. These functions include some of the standard UNIX C functions such as "printf", string manipulation etc. and mathematical functions such as "sin", "log" etc. Also included is a substantial set of DSP functions such as filters, fft's and matrix manipulation. All are directly callable from C programs. A list of the application library functions is shown below. Versions of each function are provided which are optimized for speed or code size.

- Arithmetic
  floating point format conversion, divide, square root, raise to power, generate random number, cosine, sine, tangent, arccosine, arcsine, arctangent, log base 2, e, 10, antilog base 2, e,10
- Matrix
  multiply, inverse
- Filter
  FIR, IIR, LMS (real), LMS (complex)
- FFT
  Window functions, FFTs
- Graphics/Imaging
  grey scale, histogram equalization

6. *REAL-TIME PROGRAMMING TECHNIQUES*

While the DSP32 C compiler is a full implementation of the C language, certain programs and programming techniques yield more efficient assembly code than others. The following set of rules are provided as guidelines for efficient C programming for the DSP32 compiler.

- Use pointers to array elements: Accessing data in arrays is much faster by pointer than array index, particularly if sequential elements are accessed.

- Use post modification of pointers: DSP32 pointer registers are post modified in hardware concurrently with data move operations.

- Avoid long data types: Because DSP32 integer registers and operations are 16 bits, it is more efficient to use those types, where possible.

- Avoid bit fields: There are no native DSP32 instructions to manipulate bits in a word, and the implementation of these functions is not efficient.

The following two rules need to be observed in the current release of the optimizer, however, register assignment and overloading techniques will make them unneccessary.

- Use register variables: Operations on register variables are considerably faster than fetching operands from memory. There are nine registers available to the user for integers and memory pointers and two registers available for float values.

- Declare register variables with limited scope within blocks: The compiler is able to more efficiently reuse registers for register variables when the scope of the variables is defined.

One technique that is not covered in this list is the use of in-line coding of loops. In-line coding reduces the impact of the overhead associated with loop control structures. In DSP32 a single instruction implements the decrement, test, and conditional branch associated with a loop control structure. The percentage of real time represented by this instruction is dependent on the number of other instructions executed within the loop. This overhead can be minimized by expanding the loop body, and reducing the number of iterations executed. The DSP32C improves loop execution speed with an instruction to execute n instructions m times without any looping overhead.

An important optimization strategy used to obtain efficient code is the reorganization of code to make full use of the arithmetic pipeline stages. Inefficient code is produced when nops have to be introduced to flush the pipeline when later instructions are dependent on results which are currently being calculated in the pipeline. In some cases, where pointers are used to access memory, data dependencies cannot be determined at compile time from the source code. The optimizer inserts nops in these cases to account for the worst case dependencies.In these cases, however, the programmer has additional knowledge of these data dependencies, and can provide this information to the compiler with the directives asm("@NO_MEM_DELAY"); and asm("@MEM_DELAY"); which suppress the insertion of nops to flush the pipeline. The example given below illustrates the use of these directives.

The compiler also allows the user to program directly in assembly language within the C source code. References to C variables, both in memory and registers, can be made in the assembly language, and are resolved by the compiler.

7. *EXAMPLE*

An algorithm can be implemented, as well as tested, using the DSP32 C compiler and application library. Figure 2 shows a block diagram of a test configuration for an LMS adaptive filter. In this example, the LMS filter is used as an echo canceler, and will estimate the impulse response of the echo path, which is an IIR filter. The LMS filter is implemented directly in C, while the application library is used for the IIR filter and noise source.

Figure 3 shows a C implementation of the application. The function gauss() produces a unity variant gaussian random number, x. The function iird(), from the application library, filters x with an IIR filter and places the result into y. The LMS adaptive filter produces an FIR estimate of the IIR impulse response, and filters x with that FIR filter to form an estimate of y. The two main sections of the LMS algorithm are the correlation which is used to estimate the impulse response, and the convolution which is used to form an estimate of y.

The code shown in Figure 3 follows the rules for efficient code generation. Register data types are used for the most often accessed integers, floats, and pointers; arrays are accessed using pointers rather than indices; and in-line coding is used to minimize loop overhead. Also, the FIR filter and the coefficient update sections have no data dependencies between their consecutive float instructions, therefore, each of these sections are bracketed with the optimizer directives asm("@NO_MEM_DELAY") and asm("@MEM_DELAY"). The resulting assembly code produced for the convolution section, shown in bold type in Figure 3, is shown in Figure 4. Note that the C instruction for the convolution and tap update, e+=(*lmssvp2--=*lmssvp--)**lmscoefp--; compiles into a single DSP32 assembly language instruction. The five time replication of this instruction represents the main body of the loop. The three instructions at the beginning of the loop and the two at label _L27 represent the loop initialization and the decrement, test, and conditional branch executed at each iteration of the loop. The assembly code generated can implement 277 taps for the LMS algorithm at a sample rate of 8 kHz and a DSP32 instruction cycle time of 160 ns.

When good programming practices are ignored, the compiler will give strikingly different results. An alternate implementation of the convolution and tap update function is e+=(lmssv[i+1]=lmssv[i])*lmscoef[i];. Here, arrays are accessed using indices rather than pointers. If, in addition, register data types are not used, 32 assembly language instructions will be required to implement this single C instruction. The assembly language generated from this source code, with the same assumptions as above, can only implement 9 LMS taps in real time.

This example demonstrates that the efficiency of the compiler is directly linked to the efficiency of the C source code and that the application of a few simple rules in the generation of the source code greatly improves the real-time capability of the resulting assembly code.

8. *CONCLUSION*

This paper has described the C compiler/optimizer for the DSP32/C Digital Signal Processor family. A simple set of rules for generating efficient assembly language code has been presented, along with an example signal processing application. The compiler has been fully verified with the AT&T standard compiler regression tests, and work is continuing on refining optimization strategies to make the most efficient use of the DSP32/C instruction set. As demonstrated by the example, the compiler and application library already provide effective tools for generating real time DSP32/C applications programs. The complete set of development tools includes the compiler/optimizer, assembler, linker, archiver, simulator/debugger, and libraries, including a subset of the standard C library, the C math library, and a C callable, assembly language DSP32/C application library.

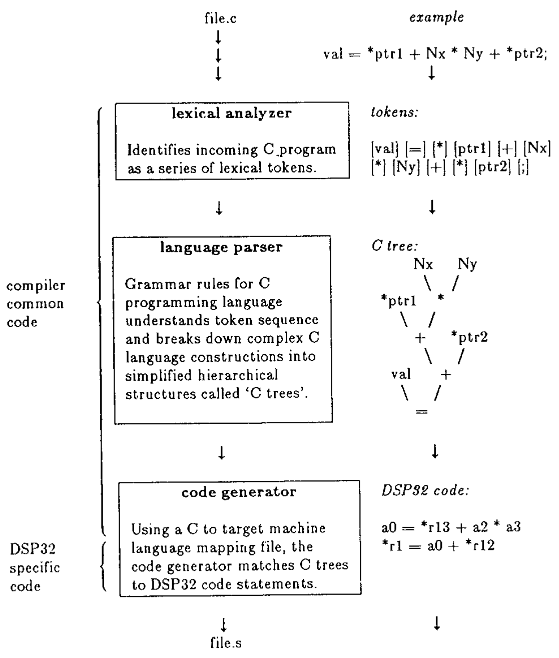
REFERENCES

1. AT&T, "*WE* ® DSP32 Digital Signal Processor Information Manual," December, 1986.

2. AT&T," *WE*® DSP32C Digital Signal Processor Advance Information Data Sheet," August, 1987.

3. Kristol, David M., " Four Generations of Portable C Compiler," in Proceedings of the Usenix Summer 1986 Conference.

4. Aho, A. V., Sethi, R., and Ullman, J. D., " Compilers, Principles, Techniques and Tools," Addison-Wesley, 1985.



Identifies incoming C program as a series of lexical tokens.

Grammar rules for C programming language understands token sequence and breaks down complex C language constructions into simplified hierarchical structures called 'C trees'.

Using a C to target machine language mapping file, the code generator matches C trees to DSP32 code statements.

Figure 1. Overview of compiler.

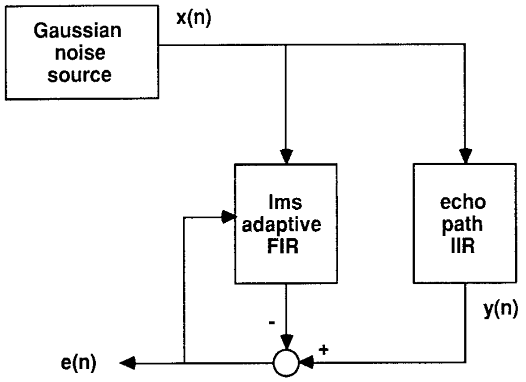


Figure 2. Adaptive Filter Example

```
#define NO_OF_TAPS 275
#define MU .001
#define NO_OF_SOS 2
float gauss(),iird();
main()
{
  register float *lmscoefp,*lmssvp,*lmssvp2,e,mu;
  register int i,j;
  static float lmscoef[NO_OF_TAPS],lmssv[NO_OF_TAPS+1];
  static float iircoef[5*NO_OF_SOS],iirsv[2*NO_OF_SOS];
  float x,y;

  initialize_lms();

  for(i=0;i<=10000;i++){

  x=gauss();
  y=iird(NO_OF_SOS,x,iircoef,iirsv);

/*  convolution of LMS filter */

     lmssvp=&lmssv[NO_OF_TAPS];
     lmssvp2=lmssvp+1;
     lmscoefp=&lmscoef[NO_OF_TAPS];
     lmssv[1]=x;
     e=0;

     asm("@NO_MEM_DELAY");
     for(j=(NO_OF_TAPS/5)-1;j-->=0;){
        e+=(*lmssvp2--=*lmssvp--)**lmscoefp--;
        e+=(*lmssvp2--=*lmssvp--)**lmscoefp--;
        e+=(*lmssvp2--=*lmssvp--)**lmscoefp--;
        e+=(*lmssvp2--=*lmssvp--)**lmscoefp--;
        e+=(*lmssvp2--=*lmssvp--)**lmscoefp--;
        }

     e-=y;
     asm("@MEM_DELAY");


/*  correlation of LMS filter */

     mu=2.0*MU*e;
     lmssvp=&lmssv[NO_OF_TAPS+1];
     lmscoefp=&lmscoef[NO_OF_TAPS];

     asm("@NO_MEM_DELAY);
     for(j=(NO_OF_TAPS/5)-1;j-->=0;){
        *lmscoefp-- -=mu**lmssvp--;
        *lmscoefp-- -=mu**lmssvp--;
        *lmscoefp-- -=mu**lmssvp--;
        *lmscoefp-- -=mu**lmssvp--;
        *lmscoefp-- -=mu**lmssvp--;
        }
     asm("@MEM_DELAY");
  }
}
```

Figure 3. Example Source Code

```
        r7 = r8 - 1
        goto  _L27
        nop
_L28:
        a3 = a3 + (*r11-- = *r12--) * *r13--
        a3 = a3 + (*r11-- = *r12--) * *r13--
        a3 = a3 + (*r11-- = *r12--) * *r13--
        a3 = a3 + (*r11-- = *r12--) * *r13--
        a3 = a3 + (*r11-- = *r12--) * *r13--

_L27:
        if(r7-- >= 0) goto _L28
        nop
```

Figure 4. Example Assembly Code