Matlab Extensions for the Development, Testing and Verification of Real-Time DSP Software

David P. Magee, Ph.D.
Texas Instruments
12500 TI Boulevard, MS 8649
Dallas, TX 75243
(214) 480-1236
magee@ti.com

ABSTRACT

The purpose of this paper is to present the required tools for the development, testing and verification of DSP software in Matlab. The paper motivates a DSP Simulator concept that can be combined with the MATLAB executable interface to develop, evaluate and test DSP software within a single environment. Programming guidelines and optimization results are also provided to demonstrate the effectiveness of the intrinsics software development approach.

Categories and Subject Descriptors

D.2.6 [**Programming Languages**]: Programming Environments – *Integrated Environments.*

General Terms

Algorithms, Design, Experimentation, Verification.

Keywords

C Intrinsics, Matlab, DSP Software, Optimization, Verification

1. INTRODUCTION

Over the past ten years, Matlab has become the preferred algorithm development, verification and simulation tool for many engineers. With its suite of application specific toolboxes, an engineer can quickly integrate and test newly developed algorithms with existing toolbox modules to create an algorithm simulation for a particular application. With this simulation, the performance of the individual modules as well as the entire algorithm chain can be evaluated. For system modeling and simulation requirements, Simulink has become a popular choice for many engineers. For simulating and generating embedded DSP software for a system, the Real-Time Workshop is available within Simulink. The tool contains a blockset of algorithms for a variety of applications so that the user can co-simulate the bit true performance of the DSP software with other algorithms and components in the system. For each algorithm in the blockset, there is a corresponding library function that has been optimized for the target DSP. Once the system performance has been validated in simulation, target software can be generated by linking the appropriate modules from the library in the preferred software development environment.

This bit true simulation and software development flow from MathWorks works well for algorithms that exist in the Real-Time Workshop blockset because there is an optimized version of the software for the target DSP in the library. However, the Real-Time Workshop tool currently cannot generate optimized DSP software from a general m-file in Matlab or a general Simulink block diagram. As a result, DSP software engineers are forced to translate Matlab files to DSP software manually.

However, steps are underway to automate this process so that DSP software can be automatically generated from Matlab and Simulink. This paper will discuss a design flow for DSP software development and testing within Matlab to ensure that efficient and optimized DSP instruction level code is generated by the DSP compiler. The concepts and tools discussed in this paper can then be leveraged to develop an automated tool.

2. DESIGN FLOW

Matlab provides a quick and easy mechanism for engineers to conduct algorithm exploration and experimentation. This work often occurs long before a product or a prototype has even been considered. Once a design proceeds from the initial stages of development, a system architecture must be developed and hardware/software partitioning decisions are made. From a verification perspective, the fixed-point implementation of each algorithm must be compared to its floating-point equivalent. Many times, the original Matlab code is used as the floating point reference. The issue becomes how to represent the hardware or software implementation in Matlab.

For software, the representation can be performed using a DSP instruction set simulator written in C [1,2]. Using Matlab's executable (MEX) file interface, this simulator can be compiled with DSP software written in C/C++ to create an object file that can be executed directly from the Matlab prompt. Figure 1 shows a block diagram of the individual components required for this design flow. Currently, the C/C++ code is generated manually, i.e. by the DSP software engineer. However, automation of this step is being investigated by several companies [3].

The process begins with the DSP software engineer defining the MEX interface between Matlab and the C/C++ code. This process includes defining the input arguments, the output arguments, the argument data types and the argument data sizes. Matlab has many pre-defined functions for handling input data, output data, temporary data and dynamic memory allocation. For a complete list of functions and a description of the MEX interface, see [4].

The next step is to interface the MEX function to the DSP software functions. By properly mapping the MEX function input and output data to the input and output data for the DSP software function, the DSP software function prototypes can remain the same in the Matlab simulation and the real DSP hardware. Note that special care must be taken when passing complex valued arrays of data from the Matlab environment to the DSP software function. Matlab stores a complex data array in memory as two arrays: the first array contains all of the real parts and the second array contains all of the imaginary parts. However in DSP memory, a complex data array should be stored as real and imaginary pairs, which is equivalent to interleaving the Matlab data.

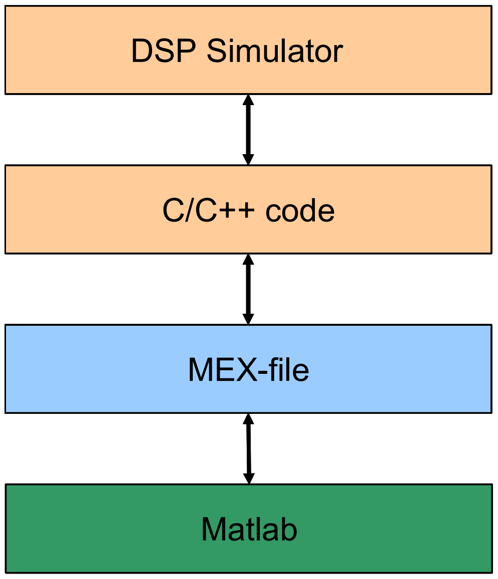


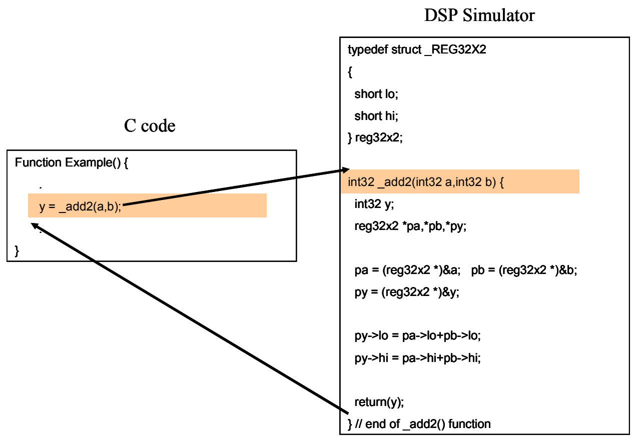
Figure 1. DSP Software Development in Matlab

The last step is for the DSP software engineer to write the relevant DSP software for the particular algorithm. This software can be strictly C/C++ code or can more tightly coupled to the target processor using intrinsics and the DSP Simulator. The use of the DSP Simulator will be discussed in the next section. Once all of the C/C++ software is written, it is then compiled in the Matlab host environment and then executed like any other function from the Matlab prompt.

The importance of a single design, development and verification environment should not be underestimated. The time savings for software development, the number of engineers required to support the floating-point and fixed-point development, the ability for more thorough testing and the ability to compare floating-point vs. fixed-point performance are some of the tangible benefits.

3. DSP SIMULATOR

With the DSPs and associated compiler features from Texas Instruments (TI), the DSP Simulator can be designed to numerically represent the instruction set of the processor so that the C/C++ software can make direct calls to assembly instructions. This mapping can occur because the TI compiler recognizes special C instructions called intrinsics. Intrinsics are special function calls in C that the compiler recognizes and understands so that it can map them directly to the corresponding assembly instruction. The DSP Simulator is simply a collection of C functions that represent the mathematical behavior for each intrinsic. Figure 2 shows the _add2() intrinsic as an example.



```
Function Example() {
     .
    y = _add2(a,b);
     .
}
```

```
typedef struct _REG32X2
{
  short lo;
  short hi;
} reg32x2;

int32 _add2(int32 a,int32 b) {
  int32 y;
  reg32x2 *pa,*pb,*py;

  pa = (reg32x2 *)&a;   pb = (reg32x2 *)&b;
  py = (reg32x2 *)&y;

  py->lo = pa->lo+pb->lo;
  py->hi = pa->hi+pb->hi;

  return(y);
} // end of _add2() function
```

Figure 2. Intrinsic Example: _add2()

To further demonstrate how a DSP software engineer might write a particular algorithm using intrinsics, the radix-2 portion of an FFT algorithm is shown in Figure 3.

```
// inside the Radix-2 stage
for(k=Nover2;k>0;k--)
  {
     .
     // compute the real part
     // (x0.real-x1.real)*w1.real
     reg2 = _mpyhir(w1,reg1real);
     // (x0.imag-x1.imag)*w1.imag
     reg3 = _mpylir(w1,reg1imag);
     reg2 -= reg3;
     // compute the imag part
     // (x0.imag-x1.imag)*w1.real
     reg4 = _mpyhir(w1,reg1imag);
     // (x0.real-x1.real)*w1.imag
     reg5 = _mpylir(w1,reg1real);
     reg4 += reg5;
     .
}
```

Figure 3. Radix-2 FFT Stage

Notice that the _mpyhir() and _mpylir() intrinsics are used to twiddle the real and imaginary parts of the radix-2 stage. Since the real and imaginary parts of the complex data are stored as the upper and lower 16 bit parts of a 32 memory location, these intrinsics, which perform special dot products, can compute the complex multiplication in fewer cycles than the common implementation using multiplier and adders.

The benefit of this programming style is that the DSP software engineer can write "assembly" level programs in a C/C++ programming environment and avoid the associated hassles of assembly programming (pipeline scheduling, register allocation, unit allocation, stack manipulation, parallel instruction debugging, etc.). These burdens are placed on the compiler, which is better designed to handle such issues. Furthermore, since the same fixed-point C/C++ software can simulated in Matlab and compiled to run on the DSP, the software engineer only has to maintain one version of C/C++ software for a particular algorithm. This benefit can greatly impact the support requirements for a system, particularly when the design becomes a full product.

4. PROGRAMMING GUIDELINES

To get the best optimization results from the TI compiler, the C/C++ software must be structured so that it compiler friendly. This process can be broken down into several steps. First, the data path for the software must be carefully designed. The endianness of the input and output data for the algorithm must be prescribed as well as the storage format for complex data. Data alignment (8/16/32 bit boundaries) of input, output and internal data is also an important design consideration for the data path. As part of this consideration, the incrementing of the input and output data should also be considered. Finally, the load and store capabilities of the target processor should be well understood because most software optimizations are limited by data bus bandwidth.

Once the data path for the algorithms has been determined, the instruction set of the processor must be studied. The particular set of intrinsics supported by the compiler for the target processor must be understood. This information along with the DSP processor architecture, will determine the number of multiplications and additions that can occur per clock cycle. It is also important to understand the level of intrinsics support for manipulating real and complex data types. For example, many new processors can perform several complex multiples per cycle if the data is stored appropriately. Finally, the register sizes and the associated instructions must be understood when using the corresponding intrinsic.

After considering these guidelines, the DSP software engineer must map the algorithm to target DSP instruction set. For TI DSPs, the registers are often 32 bits. As a result, it is convenient to compress four 8 bit real values, two 16 bit real values or one 32 bit real value into a 32 bit memory location. Similarly, it is convenient to map two 16 bit complex values (8 bit real/8 bit imag) or one 32 bit complex value (16 bit real/16 bit imag) into a 32 bit memory location. For 64 bit complex values (32 bit real/32 bit imag), two 32 bit registers are required. Once the data has been compressed in memory, the mathematical operations required by the algorithm must be mapped to the appropriate set of intrinsics. The easiest and most straightforward mapping is one-to-one. However, most algorithms are not initially designed with a DSP instruction set in mind. Thus, the DSP software engineer should consider and evaluate different mappings to achieve the most efficient software in terms of DSP cycles required for execution while maintaining a mathematically equivalent implementation.

Once the algorithms have been mapped to DSP software with intrinsic function calls, additional compiler directives can be added to the software to improve compilation of the software. For example, pragmas such as MUST_ITERATE, PROB_ITERATE and UNROLL can be used to tell the TI compiler how to unwrap software loops. The DATA_ALIGN pragma tells the compiler to align data arrays on specific memory boundaries. The _amemX_const() intrinsic can then be used in functions to tell the compiler how data is aligned so that the appropriate load and store operations can be performed.

Compiler keywords such as const, interrupt, restrict and volatile are also important for getting highly optimized DSP software:

- The const keyword should be added to scalars and arrays to denote constant data values.
- The interrupt keyword should be added to interrupt service routines (ISRs) to make them maskable interrupt functions and to ensure proper context switching.
- The restrict keyword is added to function pointer arguments to denote that only this pointer will access the particular memory in question.
- The volatile keyword is added to pointers to denote that the data value in the corresponding memory address location is temporary and that another process can change the value. This keyword is particularly useful when reading a register to determine if a process has completed.

For a complete list of programming guidelines for TI DSPs, see the relevant programmer's guide [5,6,7].

5. SOFTWARE OPTIMIZATION

The previous section discussed many programming guidelines for developing DSP software using intrinsics. However, the problem is how you determine when the software is fully optimized. Unfortunately, there is usually no clear cut answer. For a particular mapping, the DSP software engineer can determine the limiting operation for a particular algorithm and determine how close the compiled software is to an ideal number. For example, many algorithms are multiply intensive. Once the total number of multiplies required for an algorithm is divided by the number of multiplies that the target process can perform per cycle, the ideal limit can be computed. The engineer must then decide if the overhead for the particular function (ratio of actual cycles to the ideal cycles) is acceptable. If not, then further optimization is necessary. This additional optimization may include a new mapping of the algorithm which will require a new ideal limit.

Table 1. Optimization Results

Algorithm	C Intrinsics (cycles)	Code Size (bytes)	Assembly (cycles)	Code Size (bytes)
autocorr	97	800	66	384
bit_unpack	124	192	108	192
bk_massey	302	416	262	320
ch_search	485	1056	321	864
dotprod	32	160	29	160
fir_cplx	1227	832	985	448
forney	195	864	156	864
maxval	38	128	34	128
rs_encoder	463	512	402	288
syndrome	510	1216	478	1152
vecsum	45	96	40	96

For target processors that process multiple instructions in parallel, the DSP software engineer can also evaluate the resulting assembly code to determine of all of the units are being utilized during each clock cycle of the function. If not, then additional optimization may be required. Loop unrolling can often help in this area. The pragmas described in the previous section can help the compiler perform this operation automatically.

To demonstrate the effectiveness of the TI compiler to handle C/C++ software written with intrinsics, Table 1 shows TMS320C64x cycle and code size comparisons between intrinsics software and assembly code for several algorithms related to communications systems. For most of the algorithms, the cycle counts for the intrinsics software are within 10% of the cycle counts for the assembly code. What is not shown here is the development time required for each algorithm, where assembly programming can take from 4 to 5 times longer than the intrinsics approach.

6. LIMITATIONS

While the design flow discussed in the previous sections has many benefits, it is not without limitation. Currently, the DSP software engineer must perform the algorithm mapping from floating-point to fixed-point manually. Furthermore, the resulting DSP software architecture is limited to the creativity of the engineer. To reduce the amount of time required to generate fixed-point DSP software from floating-point Matlab code, the mapping process could be automated. The same programming guidelines presented in this paper can be incorporated into an automated software generation tool to reduce the amount of time required to produce compiler friendly and cycle efficient DSP software. By analyzing data types (i.e. real, complex) and data value ranges in Matlab, the tool could easily map the data path and mathematical operations to the appropriate intrinsic functions. Once the DSP software has been generated, the floating-point versus fixed-point performance can be evaluated in Matlab. Once an acceptable level of performance has been achieved, the same DSP software can be ported to the embedded processor in the system.

7. CONCLUSIONS

This paper has presented a design flow for the development, verification and simulation of DSP software within Matlab. The combination of C/C++ software with intrinsic function calls and a DSP simulator allow the DSP software engineer to evaluate the performance of a fixed-point implementation of an algorithm without hardware. This flexibility permits the engineer to evaluate many algorithm mappings more efficiently as well as to evaluate the mapping of an algorithm to different target processors.

The importance of a single design environment for development, verification and simulation should not be underestimated. The time savings for software development, the number of engineers required to support the floating-point and fixed-point development, the ability for more thorough testing and the ability to compare floating-point vs. fixed-point performance are some of the tangible benefits. Furthermore, the need to only maintain one version of DSP software can greatly minimize code maintenance issues in production software systems.

8. REFERENCES

[1] Magee, David P., "Intrinsic Modeling within Matlab to Accelerate DSP Software Development", Texas Instruments Developer's Conference, Houston, TX, August 2002.

[2] Magee, David P., "Programming the C6x Family of DSPs: A C Intrinsics Tutorial", Texas Instruments Developer's Conference, Houston, TX, February 2005.

[3] Catalytic, Inc. Web Page, http:// www.catalyticinc.com/, April 2005.

[4] *Matlab External Interfaces Reference*, Version 6, July 2002.

[5] *TMS320C28x Optimizing C/C++ Compiler User's Guide*, SPRU514, August 2001.

[6] *TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide*, SPRU025e, August 1999.

[7] *TMS320C6000 Programmer's Guide*, SPRU198D, March 2004.