The best way to start developing DSP-assembly applications is to run your C source code and look at the resulting intermediate assembly code. But beware: You still have to contend with rounding and saturation arithmetic in the eventual application. Because standard ANSI C does not directly support the fixed-point fractional data type that most DSP applications require, you have no choice but to develop your demanding algorithms in assembler.

A common metric in DSP applications is that control code occupies 80% of the code space but consumes only 10% of the MIPS. DSP kernel code occupies only 20% of the code space but consumes 90% of the MIPS. You can use the compiler to quickly generate the assembly for the control code and focus your effort on the kernel code.

**WHY ASSEMBLY?**

Assembly language is a must in DSP applications for several reasons. One is that high-level languages (HLLs) don’t address timing.

“Some control applications require small sections of code to run in an exact number of cycles—not too many or too few,” says Andy Fritsch, C6000 fixed-point program manager at Texas Instruments. “Today, HLLs don’t have a way to express this feature, and the user is often required to write the code by hand.”

Coding in assembly also makes sense from an efficiency standpoint. According to Steve Kafka at Analog Devices, some more intensive computations, such as butterflies and convolutions, always sug-
DSP architectures influence code

Detailed knowledge of the target architecture is a must for developing efficient application code, especially knowing which “dirty tricks” you can use to take advantage of the architecture’s strengths. Programming DSPs becomes easy when the architecture and programming model of a DSP are well-structured. For example, a DSP that has a very orthogonal instruction set (that is, all commands work on all registers) is easier to program and optimize than a DSP whose commands work only on specific ALU registers. For example, Analog Devices’ ADSP-21062 Super Harvard Architecture Computer (SHARC) DSP features three independent parallel computation units: the ALU, the multiplier, and the shifter. Single multifunction instructions execute parallel ALU and multiplier operations.

The DSP features an enhanced Harvard architecture in which the data-memory bus transfers data and the program-memory bus transfers both instructions and data. A general-purpose, 10-port, 32-bit data-register file, combined with the ADSP-21000 Harvard architecture, allows unrestricted data flow between computation units and internal memory. The ADSP-21062 includes an on-chip instruction cache that enables three-bus operation for fetching an instruction and two data values. The cache selectively caches only those instructions whose fetch operations conflict with program-memory-bus data accesses.

This scheme allows fast execution of core, looped operations, such as multiply-accumulate (MAC) instructions and FFT butterfly processing. The device’s 48-bit instruction word accommodates a variety of parallel operations for concise coding. For example, the ADSP-21062 can execute a multiply, an add, a subtract, and a branch all in a single instruction.

ZSP Corp’s ZSP16401 is a 16-bit fixed-point DSP based on a superscalar architecture (register-based with multiple execution units) that employs a five-stage pipeline. The DSP features two ALUs and two MAC units, which use a 16-bit register-operand file for their source and destination. This register-rich architecture avoids accumulator bottlenecks and helps reduce immediate memory accesses. Further, you can use data memory as a general-purpose, scratchpad resource while reading and writing to device registers. There are no restrictions on reading and writing to data memory, unlike the segregated read and read/write memory scheme found in traditional DSPs.

Code development for the device has the look and feel of development for a μC assembly code (Figure 1). For example, you can write assembly-language routines as a linear sequence of instructions. You can also combine instructions that specify one or two operations with other instructions for parallel execution, but the code contains no scheduling information. Thanks to the device’s pipeline-control unit, which schedules as many as four instructions for execution within the same clock cycle, you need not worry about scheduling parallel execution. This feature is a major benefit, because it allows you to concentrate on the algorithm and its implementation rather than on the DSP’s architecture.

The inherent nature of DSP applications means that certain programming skills are unique to DSP-assembly programming. Coding DSP algorithms is mathematically intensive. Mapping those models onto a DSP’s architecture is a skill unique to DSP assembly programmers.

“You often trade off when coding an algorithm that requires 41 MIPS onto an architecture that can deliver only 40 MIPS,” says Analog Devices’ Kafka. “In such situations, a thorough understanding of the math involved in the algorithm will help the assembly programmer modify the algorithm so that it requires fewer MIPS while still delivering the desired results.”

According to Kafka, DSP-assembly programming also requires that you understand instruction stalls. Instruction stalling is important because it helps you optimize assembly. “Stalling” means that a previous instruction delays the next instruction because...
that instruction needs the previous result to execute. For example, in the following code fragment, the Motorola DSP56300 introduces two clock cycles after the first instruction but before executing the second instruction to ensure that the address register “inputPtr” contains valid data. You can optimize the above software by inserting another instruction between the following two instructions:

MOVE #InputBufferBase, inputPtr
MOVE Input_Mem:(inputPtr)+, inputBuf

Kafka believes that DSP-assembly programmers should also understand:

- various code optimizations, for example, software pipelining (execution of operations from different iterations of the loops in parallel);
- finite-arithmetic effects, such as overflow, saturation and cumulative effects;
- memory architectures, such as modified-Harvard and banked, and how to exploit them;
- special architectural features, such as circular buffers and bit-reversed addressing, and how to exploit them;
- required accuracy and precision, which allows you to use look-up tables or seed values for iterative approximations;
- math “tricks,” which can reduce explicit operations; for example, divide by 2 is a simple shift; and
- how to represent scalar fractional data.

**ALGEBRAIC OR MNEMONICS?**

Two methods exist for developing assembler code: One is by traditional mnemonic-assemblers; the other is by algebraic assembler. Analog Devices asserts that its algebraic assembly syntax helps you develop a clear understanding of DSP algorithms and their underlying mathematical models. Kafka says that intuitive (algebraic) syntax is important in developing DSP applications. He asserts that the code is easier to write, easier to maintain, and easier to read than a mnemonics-based syntax. See the listing below for a quick comparison:

Algebraic: ADSP-21060 instructions to find the maximum of two operands:

\[ R7 = \text{MAX} (R5, R6) \]

Mnemonics: TMS20C40 instructions to find the maximum of two operands:

\[ \text{LDF } R6, R7 \]
\[ \text{CMPF } R5, R7 \]
\[ \text{LDFLT } R5, R7 \]

For more source code, comparing Analog Devices algebraic syntax for the SHARC with the mnemonic syntax for Texas Instruments’ TMS20C40, go to www.analog.com/techsupt/application_notes/AN-403.pdf. According to Reid Tatge, a distinguished member of the technical staff in the Software Development Systems Group at Texas Instruments, using algebraic-assemble code versus using mnemonics-assemble code is mostly a matter of personal preference. Tatge sees no inherent advantages of either approach, either in the implementation of the algorithm in the assembler or in the programmer’s use of or ability to read a program.

Tatge argues that, although most of the instructions in RISC-like architectures map nicely to a familiar “algebraic” form, algebraic assembly gets complicated when you are handling instructions that affect status bits. Also, algebraic assemblers often use esoteric operations by inventing syntax for common types of DSP operations, such as saturated math, bit-reversed math, circular addressing, and add with carry. Algebraic assemblers typically revert to a mnemonics or functional form for these special instructions:

\[ r0 = \text{add}_{\text{saturate}} (r1,r2) \]

With the TMS320C54x DSPs, you can generate code with either an algebraic or a mnemonics/assembler. Tatge says there is a 1-to-1 translation between mnemonics and algebraic assembly on the C54x, so users are not penalized in cycle count or code size, regardless of which format they prefer.

**DEVELOPMENT TOOLS FOR DSP ASSEMBLY**

Debugging tools for assembly-language development are getting more sophisticated. The ZSP164xx DSP provides you with an integrated development environment (IDE) that allows you to rapidly and smoothly complete debugging. ZSP’s Profiler (also known as Simulator) gives valuable information on how the instructions are grouped so that an assembly programmer can optimize the code to the fullest extent and achieve maximum performance (Figure 1).

Analog Devices’ VisualDSP (Figure 2) introduces an easy-to-use project management environment, comprising an IDE and a debugger. VisualDSP lets you manage projects from start to finish from within one integrated interface. Because the project-development and debugging environments are integrated, you can easily move between editing, building, and debugging activities.

The IDE includes access to the SHARC DSP C compiler, C runtime library, assembler, linker, loader, simulator, and splitter. VisualDSP’s debugger has many features that greatly reduce debugging time. You can view C source interspersed with the resulting assembly code, profile execution of a range of instructions in a program, set simulated watch points on hardware and software registers, program and data memory, and trace instruction execution and memory accesses.

The C compiler generates efficient code that is optimized for both code density and execution time. The compiler allows you to include assembly-language statements inline. This feature means that you can program in C and still use assembly for time-critical loops. You can also use pretested math, DSP, and C runtime-library routines to shorten your time to market. A cy-
cle-accurate, instruction-level simulator lets you simulate your application in real time. Also, the VisualDSP tool features a well-defined application programming interface (API). Third-party products, runtime operating systems, emulators, HLL compilers, and multiprocessor hardware can interface seamlessly with VisualDSP and thereby simplify tool integration. VisualDSP follows the COM API format. Two API tools, Target Wizard and API Tester, are also available. You can find links to third parties that support Analog’s DSP product lines at www.analog.com/support/3rd_party/third_party.html.

Texas Instruments TMS320C6x DSP features two datapaths—each with 16 32-bit registers—shared by eight functional units. The C6x is a very-long-instruction-word (VLIW) processor; a composite instruction (line), formed by concatenating n short instructions, controls its functional units. In the C6x, the short instructions are 32 bits wide, and the line consists of a maximum of eight 32-bit instructions. Therefore, the C6x packs instructions to avoid inefficient use of program memory.

Instruction packing may result in alignment restrictions between lines and the physical-memory organization. In the C6x, the program is organized as eight instruction lines, or fetch packets. You may divide each fetch packet into groups of instructions or “execute packets,” which are issued in parallel. The C6x is restricted because the execute packet must be within an eight-word boundary. Every fetch packet must start with an execute packet. This stipulation implies that you must pad fetch packets that do not end with an execute packet with no-operation instructions. Furthermore, the need to track various functional units in a VLIW architecture can rapidly overwhelm a DSP programmer.

To help application developers, TI has introduced the concept of “linear” assembly language. Linear assembly language allows you to write assembly language without regard to registers and pipeline behavior, significantly easing the task of writing low-level code. An Assembly Optimizer, which performs many compilerlike optimizations, processes linear assembly by assigning registers and scheduling the instructions, effectively handling all the “hard” problems associated with writing code for a pipelined processor (Figure 3).

TI claims that the C6000 compiler offers an average improvement in execution speed; it is three times faster than state-of-the-art fixed-point DSP C compilers. The C compiler accepts ANSI C source code and produces efficient C6000 assembly-language source code, performing a variety of optimizations to improve the efficiency of the compiled code. The compiler incorporates four levels of state-of-the-art generic and target-specific optimizations. The level of optimizations is selectable.

Optimizations include those features specific to the TMS320C6201 processor, such as software pipelining, if conversion/predicated execution, memory-address cloning, memory-address-dependence elimination, branch optimizations/control-flow simplification, alias disambiguation, copy propagation, common subexpression elimination, redundant assignment elimination, loop-induction-variable optimizations/strength reduction, loop rotation, loop invariant code motion, inline expansion of function calls, register tracking/targeting, and cost-based register allocation. For more information, visit TI’s DSP third-party Web site (www.ti.com/sc/docs/dspsp/develop/3party.htm).

Even with all these development tools,
you need a good design methodology to develop consistent, top-quality assembly code. ZSP’s Stone suggests the following algorithm as a general method for generating consistent, top-quality assembly code:

**Design for correctness:**
1. Clearly define and document problem. (Anticipate problems beforehand.)
2. Start with a fixed-point C model. Document code and anticipate areas that you may need to write in assembly.
3. Compile C code; generate assembly.
4. Verify assembly in system.

**Possibility 1:**
5. Optimize C code by adding intrinsics, library functions, or assembly routines with C wrappers.
6. Verify C+ASM.
7. Go to 5.

**Possibility 2:**
5. Write ASM.
6. Verify ASM.
7. Optimize ASM.
8. Go to 6.

For developing high-quality assembly code, Raymond Horn, CEO of DSPecialists, a German design house specializing in DSP5600 assembly-tool development, recommends the following procedures:

- Optimize on the “planning level” when structuring an application in different modules and defining the interfaces.
- Optimize data position in memory.
- Use parallel data moves as often as possible.
- Carefully plan which registers to use for certain operands.
- Reuse existing code wherever possible when programming new software modules.
- Use uniform rules and naming conventions.
- Use macros for complex operations.
- Use the DMA controller as often as possible.
- Avoid double calculation of the same data by using intermediate results as input for different calculations.

Stone maintains that for applications that are based on a standard, such as a speech coder, it’s easier to justify development in assembly to maximize performance, because the standard will not change. However, Stone also says that for large applications with millions of lines of code and applications without a fixed standard, such as 3G Cellular Systems, it is better to adopt C.

You can reach NS Manju Nath at +852-2965-1534, fax +852-2976-0706, nsmanjunath@cahners.com.hk.

**References**
2. Considerations for selecting a DSP processor, ADSP-2115 Vs. TMS 320C5x, AN-393, Analog Devices.