SOC-Based Signal Processing: Meeting Performance Goals With Tailored DSPs

Steven Leibson
Tensilica, Inc.
3255-6 Scott Blvd
Santa Clara, CA 95112
1-408-327-7335

sleibson@Tensilica.com

ABSTRACT

This paper describes a new approach to developing very high-performance SOC-based DSP systems using configurable and extensible microprocessors tailored to individual DSP algorithms.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems, Signal Processing Systems.

General Terms

Algorithms, Performance, Design, Experimentation, Verification.

Keywords

Algorithms, Architectures, Embedded H/W Design, SoC/IP Design, ASIC Design/IP, Embedded S/W Design, Performance, Design, Test and Verification, MPEG4, FFT, Viterbi.

1. INTRODUCTION

For more than 30 years, the fixed-ISA (instruction set architecture) microprocessor has largely defined how we develop electronic systems. Similarly, single-chip DSPs have come to dominate DSP system design since they were introduced 20 years ago. Few engineers now remember how application-tailored DSP systems based on bit-slice processors were developed in the 1970s to exactly fit algorithmic requirements. Even fewer people remember how to create such designs because we simply don't need to use this design approach any more. As they have come to dominate DSP system design, fixed-ISA DSPs have solved application performance problems in two ways. The first approach has been to increase clock rate, mirroring the same trend in PC processors. If power dissipation is not especially important, rapidly escalating clock rates can go a long way towards force-fitting a particular DSP to an application. The alternative approach for fixed-ISA DSPs is to provide more computational resources that operate in parallel so that the DSP can perform more work per clock. Designers of fixed-ISA DSPs attempt to develop architectures that are good at executing a wide range of algorithms, but not tailored to any specific application. This design approach can reduce clock-rate requirements, but it doesn't match the clock-rate or gate-usage efficiencies of a tailored solution. The design of portable and battery-powered products based on DSP technologies increasingly calls for taking a different approach, one that more closely matches the DSP resources to the application tasks so that very high clock rates (and correspondingly high power dissipations) are not required. Although bit-slice DSP design is no longer in vogue, ASIC and SOC technologies provide an ideal medium for tailoring processors to DSP applications, using newly available configurable and extensible processor technology.

2. DSP SOFTWARE DEVELOPMENT

First, let's look at the software development process currently used to develop applications for fixed-ISA DSPs. Figure 1 illustrates a typical flow for developing DSP application software [1][3]. Work starts not with the DSP but with the algorithm. Application developers generally start with high-level tools and languages such as C or C++, The MathWorks' Matlab, Mathsoft's MathCAD, or National Instruments' LabView. These high-level programming languages and development packages allow developers to create, test, and validate primary algorithmic ideas and smaller independent algorithms and sub-algorithms using tools that deal with the algorithm in a state that's divorced from a particular DSP or hardware implementation.

Next, the developers translate the main algorithm and sub-algorithms in C to create a portable and processor-independent application code base. C-level simulation, performed on a PC or workstation, then proves that the recoded algorithms perform as expected. After integrating the sub-algorithms and other application software modules into a coherent whole, the entire program (now written in C or C++) is recompiled for a target DSP and the resulting application code is tested and profiled on the target processor. However, C compilers are often unable to produce efficient assembly code for DSPs because DSPs have limited and heterogeneous register sets, not the large homogeneous register sets of general-purpose processors that normally serve as compiler targets [6][7].

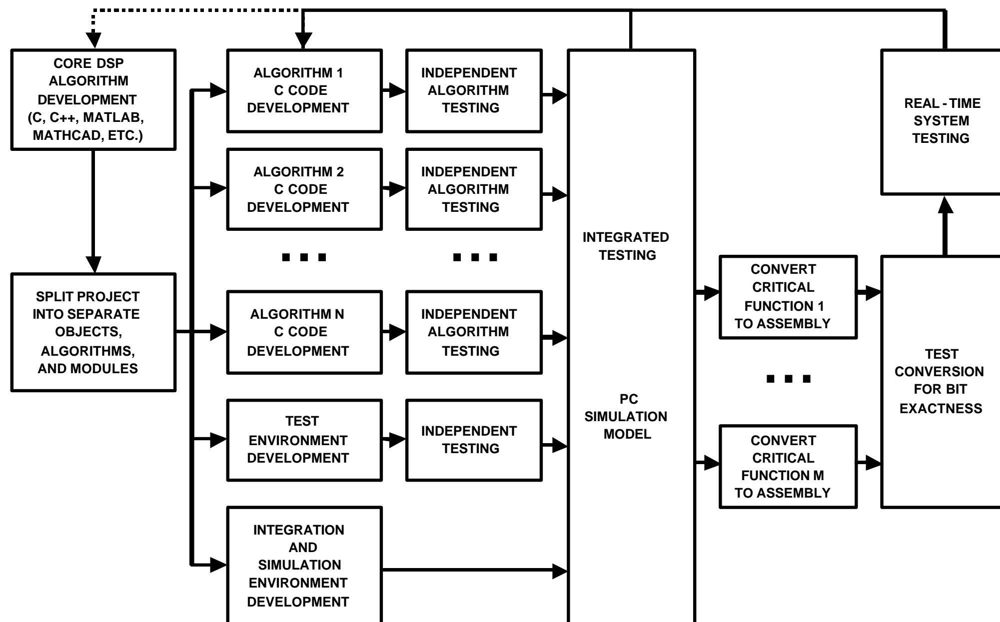

Figure 1. The Typical DSP Software Development Process

If the development team is extremely fortunate, the compiled algorithm code executes with the desired speed. However, fickle fortune rarely smiles on DSP developers. So to meet project performance goals, application software teams must almost universally convert critical sections of code into hand-tuned assembly code once a fixed-ISA DSP is selected. The software development team must generally try to closely map the assembly code to the DSP by hand. Otherwise, the DSP selected will probably end up being too expensive, too fast, or too power hungry for the intended application. As Rob Oshana of Texas Instruments has written, fitting a DSP algorithm to a fixed-ISA DSP is often "like trying to fit 10 pounds of algorithms into a five-pound sack." [5].

Hand-tuned assembly code maximizes use of a fixed-ISA DSP's internal resources so that as much computation as possible is performed in parallel and so that the DSP's memory busses are kept as busy as possible. Fixed-ISA DSPs have one or more multiplier/accumulators and may have other specialized resources, such as a barrel shifter, that are useful for commonly executed DSP algorithms. Hand tuning of code maximizes use of a DSP's computational and storage resources and maximizes throughput. Parallel execution units can significantly boost algorithmic performance if the algorithm developer can engage most or all of the execution units simultaneously.

The DSP application developer must also streamline data movement to keep all of the execution units fed with data. Wide pipes into the DSP help. DSPs typically employ Harvard architectures, separating external instruction and data memory to further increase bandwidth. High-performance DSPs typically have multiple, on-chip memory caches to support high-bandwidth flow of instructions and data to the DSP's computational elements. However, the fastest DSP memories are the processor's registers and a substantial portion of the code-optimization process is usually devoted to managing those registers so as to minimize access to slow external memory.

Assembly code developers must carefully dovetail their variables into the available registers because there's no way to add more registers to a fixed-ISA DSP if the existing register set proves inadequate [2]. This last item points out the limitation of fixed-ISA DSPs: designers of fixed-ISA DSPs determine the available resources (registers, execution units, and data buses) in the processor architecture before the silicon is cast. Consequently, fixed-ISA DSP architectures are better at executing some algorithms than others. That's just the nature of general-purpose engineering design; one size can't fit all.

A number of factors determine how rapidly a DSP executes an algorithm. Processor speed, measured in millions of instructions per second (MIPS), is obviously important. So is the use of fast multiplier units. Still greater speed comes from the use of special instructions that execute several operations in a single cycle. Finally, there is the use of pipelined architectures, which permit the simultaneous execution of different stages of multiple instructions [4]. These characteristics are shared by both modern DSPs and configurable and extensible microprocessors.

3. FIT THE DSP TO THE ALGORITHM

Configurable and extensible processors allow DSP developers to create processors specifically tailored to the target algorithms—producing a much better fit between processor and algorithm. As was possible in the days of bit-slice processor design, designers using configurable and extensible processors can add special-purpose, variable-width registers; specialized execution units; and wide data buses to reach an optimum processor configuration for

specific algorithms. These features allow developers to mold the processor's characteristics to the algorithm instead of trying to force-fit the 10-pound algorithm into the resources available in a 5-pound, fixed-ISA DSP. Consequently, application developers can more rapidly develop systems that meet all performance specifications using configurable and extensible processors than by using off-the-shelf, fixed-ISA DSPs.

As with hand-tuned assembly language, optimization points for a configurable and extensible processor implementation become apparent through code profiling. Optimization targets typically reside within the innermost software loops that execute thousands or millions of times per second. Reducing the instruction count of the object code inside of these loops will produce a huge and positive impact on system performance. The following three examples illustrate the sort of performance improvements algorithm developers can expect when using configurable and extensible processors. (All of the following examples are based on Tensilica's Xtensa microprocessor, a configurable and extensible scalar processor core designed for use in SOCs and FPGAs.)

4. ACCELERATING THE FFT

The heart of the decimation-in-frequency FFT algorithm is an operation called the "butterfly," which resides at the innermost loop of the FFT. Each butterfly operation requires six additions and four multiplications to compute the real and imaginary components of a radix-2 butterfly result. Using the TIE (Tensilica Instruction Extension) language, it's possible for a design team to augment the Xtensa processor's pipeline with four adders and two multipliers so that half of an FFT butterfly can be computed in one cycle (assuming that the silicon is fast enough). Xtensa's configurable data-bus interface can be defined to be as wide as 128 bits so that all four real and imaginary integer input components of each butterfly can be loaded into special-purpose FFT input registers in one cycle and all four computed output components can be stored to memory on one cycle as well. Because the load and store operations for each FFT butterfly require a cycle each, the most cost-efficient approach to the FFT computation is to stretch each FFT half-butterfly computation across two cycles, to occur in parallel with a load operation for a subsequent butterfly and a store operation for a prior buterfly. This approach saves hardware and matches the computational and data-transfer resources.

Practically speaking, it's very hard to create single-cycle, synthesizable multipliers for SOCs that operate at clock rates of several hundred Megahertz. Although it's possible to create hard-macro IP multipliers that operate in one clock cycle, SOC designers prefer to use synthesizable IP components whenever possible because such components allow maximum freedom in selecting semiconductor manufacturing processes and vendors. Consequently, it's much better to stretch the multiplication across two cycles so that the multiplier is not the critical timing element on the SOC. The additional multiplier latency does not affect throughput in this example and longer latencies can be accommodated through additional state storage in the butterfly execution unit.

This approach to computing the FFT butterfly adds a SIMD (single-instruction, multiple data) butterfly computation unit to the processor (using fewer than 35,000 gates including the two 24x24-bit multipliers) that operates as shown in Figure 2, while Figure 3 shows the detailed operational steps for computing the SIMD FFT butterfly using the 14 new FFT instructions created with TIE. The performance improvements achieved by using this approach over straight C code, and C code augmented with just the addition of a hardware multiplier (like a traditional DSP), appear in Table 1. The table also shows the code size of the FFT programs with and without the TIE extensions.

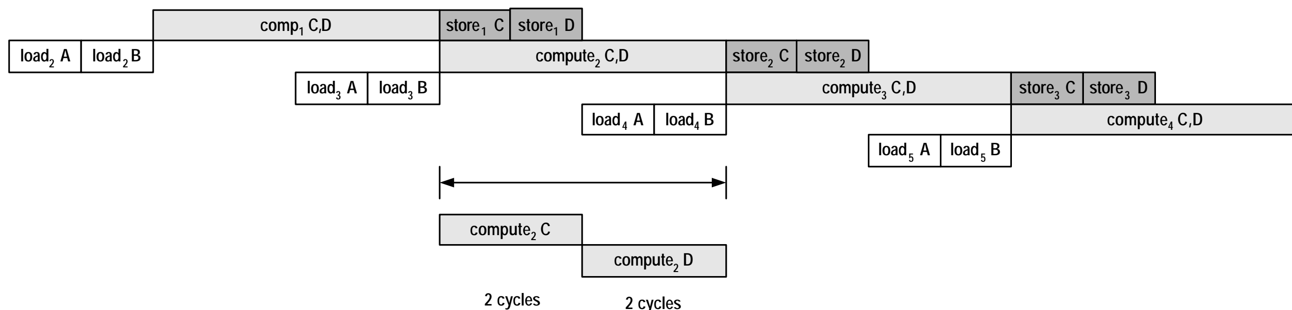

Figure 2. Overlapping loads, stores, and FFT computations

Table 1. Acceleration results from processor augmentation with FFT instructions

		C (with software multiplication)	C (with hardware multiplier)	C (with FFT butterfly TIE instructions)	Performance Improvement
Code Size (bytes)		430 + Libraries	430	158	
	FFT Length				
Performance (cycles)	128-Point	763548	169739	2269	337
	256-Point	1787645	386498	4711	379
	512-Point	3975245	867133	9841	404
	1024-Point	9241893	1922644	20603	449

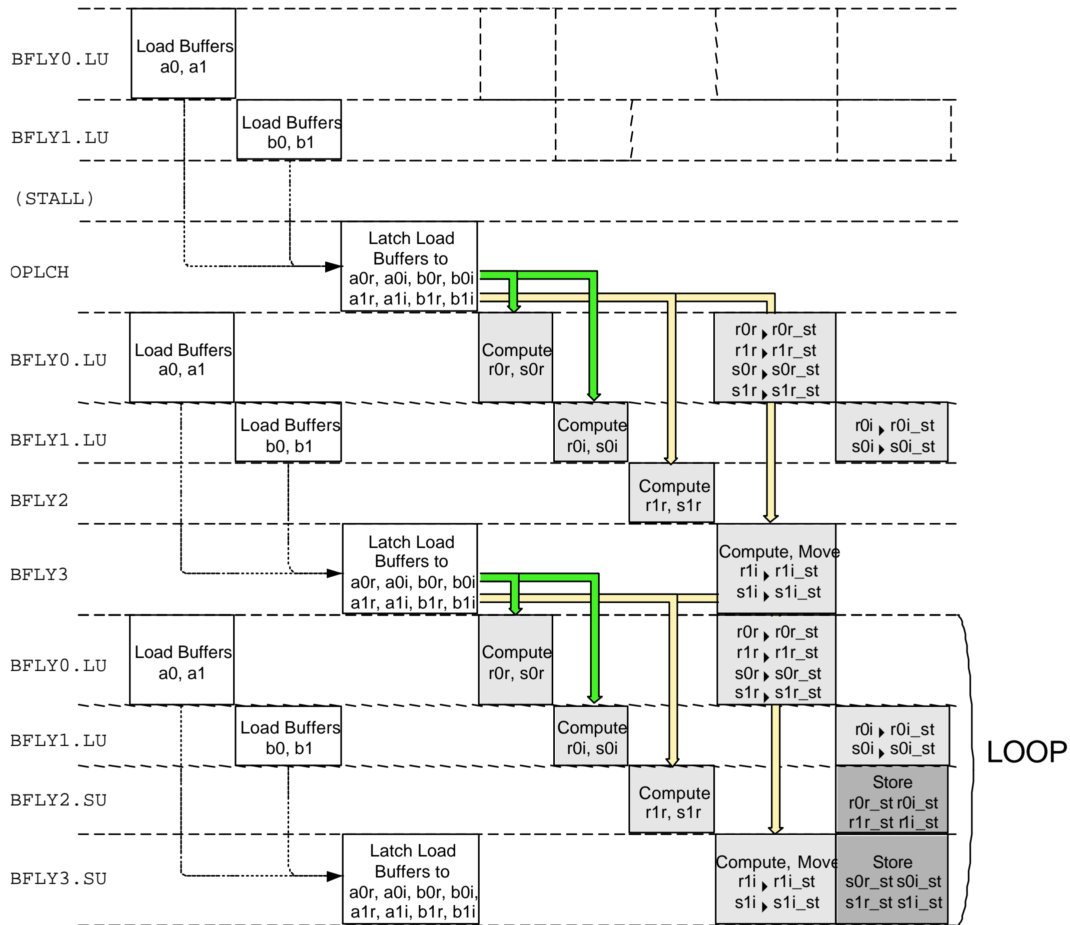

Figure 3. Detail of FFT instruction augmentation operations

5. ACCELERATING VITERBI DECODE

A different signal-processing example, Viterbi decoding, comes from GSM cellular telephony. GSM employs Viterbi decoding to pull information symbols out of a noisy communication channel. This decoding scheme employs "Viterbi butterfly" operations consisting of 8 logical operations (4 additions, 2 comparisons, and 2 selections) and performs 8 Viterbi butterfly operations to decode each symbol in the received digital information stream.

Typically, RISC processors need 50 to 80 instruction cycles to execute one Viterbi butterfly. A high-end VLIW DSP (TI's 320C64xx) requires only 1.75 cycles to compute each Viterbi butterfly. The TIE (Tensilica Instruction Extension) language allows a designer to add a Viterbi butterfly instruction to the Xtensa processor's ISA. This design uses the processor's configurable 128-bit I/O bus to load 8 symbols at a time, adds the pipeline hardware shown in Figure 4, and results in an average butterfly execution time of 0.16 cycles per butterfly. An unaugmented Xtensa processor executes Viterbi butterflies in 42 cycles, so the butterfly execution hardware (approximately 11,000 added gates) achieves a 250x speed improvement over the out-of-the-box Xtensa processor.

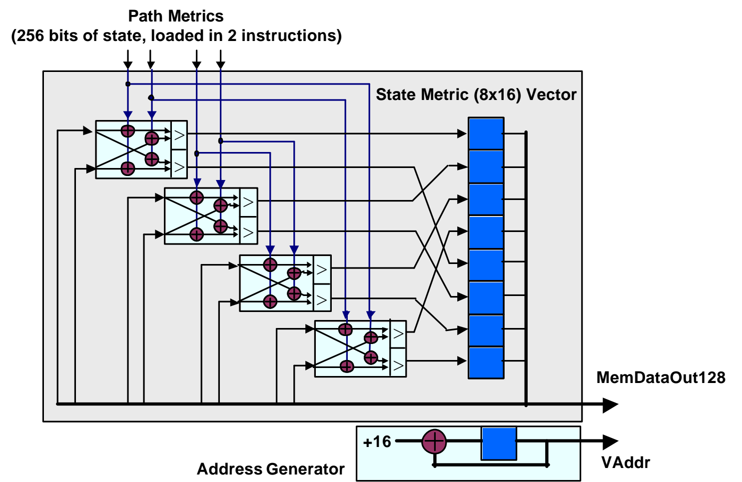

Figure 4. Detail of Viterbi butterfly augmentation

6. FAST MPEG4 MOTION ESTIMATION

MPEG4, the third example of achieving performance through instruction extension and parallel operation execution, is from the video world. One of the most difficult parts of encoding MPEG4 video data is motion estimation, which requires the ability to search adjacent video frames for similar pixel blocks. The search algorithm's inner loop contains a SAD (sum of absolute differences) operation consisting of a subtraction, an absolute value, and the addition of the resulting value with the previously computed value.

For a QCIF (quarter common image format) image frame, a 15 frames/s image rate, and an exhaustive-search motion-estimation scheme, SAD operations require slightly more than 641 million operations/s. As shown in Figure 5, it's possible to add SIMD (single instruction, multiple data) SAD hardware capable of executing 16 pixel-wide SAD instructions per cycle using TIE. (Note: Using the Xtensa processor's 128-bit maximum bus width, it's possible to load 16 pixels worth of data in one instruction.) The combination of executing all three SAD component operations in one cycle and the SIMD operation that computes the values for all 16 pixels in one clock cycle reduces the 641 million operations/s requirement to 14 million instructions/s, a substantial reduction. This MPEG4 motion-estimation accelerator is part of an entire MPEG4 decoder demonstration vehicle developed by Tensilica using Xtensa technology. The MPEG4 decoder adds approximately 92,000 to 112,000 gates to the base Xtensa processor and performs implements a 2-way QCIF video codec operating at 15 frames/s or QCIF MPEG4 decoding at 30 frames/s using approximately 30 MIPS for either operational mode.

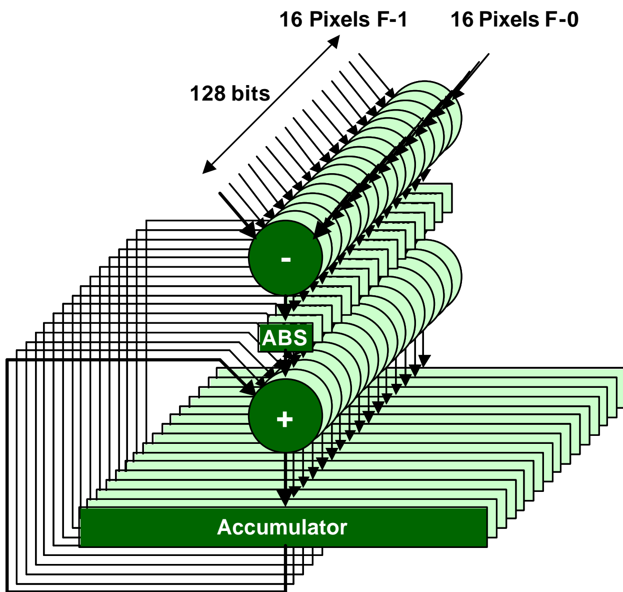

Figure 5. MPEG4 SIMD SAD (sum of absolute differences) instruction execution hardware

7. NEW SOC DESIGN METHOD FOR DSP

It's possible to outperform a fixed-ISA DSP using RTL-based DSP hardware design. In most RTL-based DSP designs, the datapath consumes the vast majority of the gates in a DSP block (see Figure 6). A typical datapath may be as narrow as 16 bits or hundreds of bits wide. The datapath will typically contain many data registers, representing intermediate computational states, and will often have significant blocks of RAM or interfaces to RAM that are shared with other RTL blocks. These basic datapath structures reflect the nature of the data and are largely independent of the finer details of the specific algorithm operating on that data.

By contrast, the RTL logic block's finite state machine contains nothing but control details. All the nuances of the sequencing of data through the datapath, all the exception and error conditions, and all the handshakes with other blocks are captured in this subsystem of the RTL block. This state machine may consume only a few percent of the block's gate count, but it embodies most of the design and verification risk due to its complexity. If a late design change is made in an RTL block, the change is much more likely to affect the state machine than the structure of the datapath.

This situation heightens the design risk. Configurable and extensible processors provide a way of reducing the risk of state-machine design by replacing hard-to-design, hard-to-verify state-machine logic blocks with pre-designed, pre-verified processor cores and application firmware, thus creating application specific DSPs.

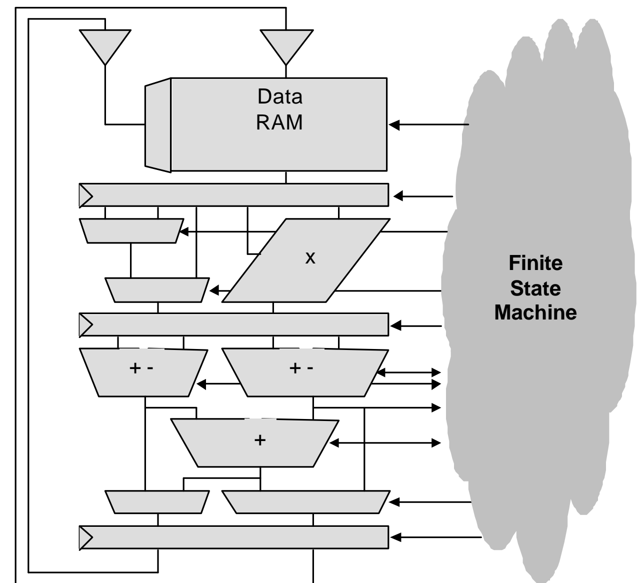

Figure 6. DSP using hardwired RTL: datapath + state machine

An application-specific DSP can implement datapath operations that closely match those of RTL-based DSP functions. The equivalent of the RTL datapath is implemented using the integer pipeline of the base processor, plus additional execution units, registers, and other functions added by the designers for a specific application.

Extended processors routinely use the same structures as traditional datapath-intensive RTL blocks: deep pipelines, parallel execution units, problem-specific state registers, and wide data paths to local and global memories. These extended processors sustain the same high throughput and support the same low-level data interfaces as typical RTL-based designs.

Control of the extended-processor datapaths is very different from conventional RTL-based state-machine designs however and more resembles the use of fixed-ISA DSPs. Cycle-by-cycle control of the processor's datapaths is not fixed in hardwired state transitions but is implemented in software executed by the processor (shown in Figure 7). Control-flow decisions are made explicitly in branches; memory references are explicit in load and store operations; sequences of computations are explicit sequences of both general-purpose and application-specific computational operations.

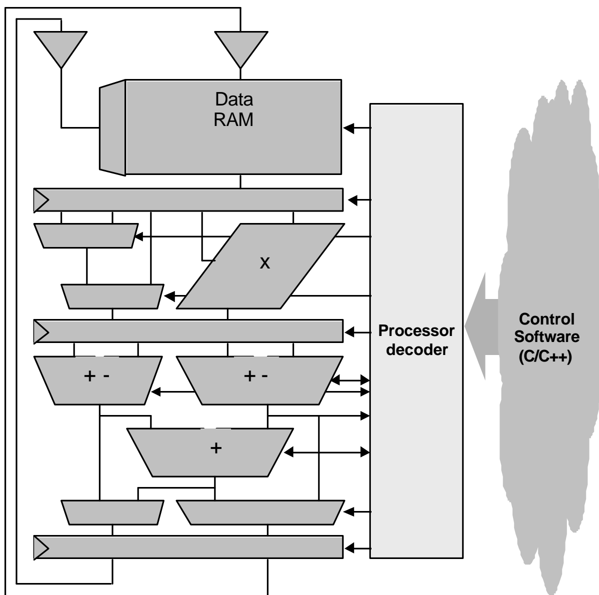

Figure 7. DSP using programmable hardware: datapath + processor + software

8. CONCLUSION

As shown by the three examples in this paper, it's possible to accelerate the performance of DSP algorithms using configurable and extensible microprocessor cores to create processors that are tailored to the specific DSP algorithm instead of resorting to assembly language coding. The advantage of using extensible processors is that designers can add precisely the resources (special-purpose registers, execution units, and wide data buses) required to achieve the desired algorithmic performance instead of shoehorning the algorithm into the fixed computational assets of a fixed-ISA DSP.

This design approach does not require that the members of the design team become processor designers. It only requires that the design team be able to profile existing algorithm code, to find the critical inner loops in that profiled code, and then define new processor instructions that will accelerate these critical loops. Only the last task differs from the existing software development process currently employed by many DSP system developers. The result of this new approach, the use of configurable and extensible microprocessor cores to implement DSP algorithms, is to greatly accelerate algorithm performance, often far beyond the abilities of today's most advanced DSPs.

9. REFERENCES

[1] Abkairov, Nikolay and Nazarov, Alexey. Tools of the trade: successful DSP software development and testing. EDN, February 21, 2002, 71-74, www.e-insite.net/ednmag.

[2] Arora, Manish, et al. Assembly Code Optimization Techniques for Real Time DSP Implementation of Speech Codecs… in proceedings of ICAASP, 2002 (Orlando, Florida, May 13-17, 2002), www.manisharora.com/files/NCC-2002.pdf.

[3] Berkeley Design Technology, Inc. DSP Algorithm Development Tools. DSP & Multimedia Technology, November 1993, reprint available at www.bdti.com.

[4] Nebeker, Frederik. Signal Processing: The Emergence of a Discipline 1948 to 1998. IEEE History Center, New Brunswick, NJ, 1998. ISBN 0-7803-9910-2.

[5] Oshana, Rob. Optimizing performance of DSPs. Electronic Engineering Times, December 9, 2002, 70-76, www.eetimes.com/story/OEG20021204S0040.

[6] Pegatoquet, Alain and Gresset, Emmanuel. Rapid Development of Optimized DSP Code From a High Level Description Through Software Estimations… in proceedings of DAC99 (New Orleans, Louisiana, June 20-24, 1999), 823-826, www.icspat.com/papers/593mfi.pdf.

[7] Zavojnovic, Vojin, et al. DSP Processor/Compiler Co-Design: A Quantitative Approach… in proceedings of ICSPAT (Boston, MA, October 7-10, 1996), 679-683, www.icspat.com/papers/593mfi.