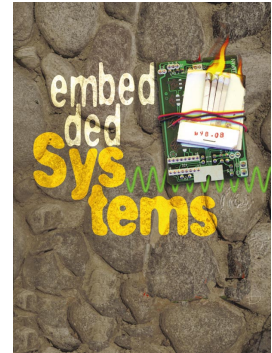


# DSP Processors Hit the Mainstream



Increasingly affordable digital signal processing extends the functionality of embedded systems and so will play a larger role in new consumer products. This tutorial explains what DSP processors are and what they do. It also offers a guide to evaluating them for use in a product or application.

*Jennifer Eyre*

*Jeff Bier*

Berkeley  
Design  
Technology  
Inc. (BDTI)

**E**ngineering terminology has a way of creeping into the public tongue, often initially by way of product marketing. For example, the time is long gone when only a few people were familiar with the unit “megahertz.” Although people are perhaps not entirely certain what a megahertz is, they are perfectly comfortable discussing and comparing the megahertz ratings of their computers.

In a similar way, many people became familiar with the word “digital” when companies introduced CD players in the 1980s.

These days, the once obscure engineering term “DSP” (short for digital signal processing) is also working its way into common use. It has begun to crop up on the labels of an ever wider range of products, from home audio components to answering machines. This is not merely a reflection of a new marketing strategy, however; there truly is more digital signal processing inside today’s products than ever before.

Consider this: Maxtor Corp. recently reported receiving its 10-millionth DSP processor from Texas Instruments for use in its disk drives. As further evidence, Forward Concepts, a DSP market research firm, reports that the 1997 market for DSP processors was approximately 3 *billion* dollars. But why is the market for DSP processors booming?

The answer is somewhat circular: As microprocessor fabrication processes have become more sophisticated, the cost of a microprocessor capable of performing DSP tasks has dropped significantly to the point where such a processor can be used in consumer products and other cost-sensitive systems. As a result, more and more products have begun using DSP processors, fueling demand for faster, smaller, cheaper, more energy-efficient chips. These smaller, cheaper, more efficient chips open the door for a new wave of products to implement signal-processing capabilities. It’s like a positive feedback loop.

There has always been much potential benefit to adding signal processing capabilities to products, but until recently, it’s simply been too expensive to be practical in most cases.

## WHAT MAKES IT A DSP PROCESSOR?

Although fundamentally related, DSP processors are significantly different from general-purpose processors (GPPs) like the Intel Pentium or IBM/Motorola PowerPC. To understand why, you need to know what is involved in *signal processing*. What is it about signal processing computations that spurred the development of a different type of microprocessor?

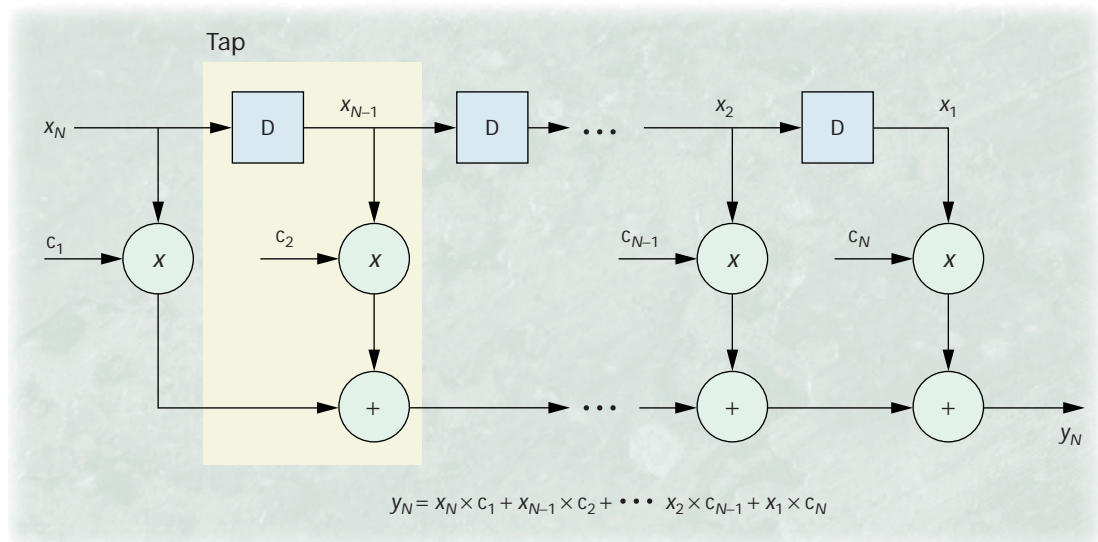
### Signal filtering

As a case study, we’ll consider one of the most common functions performed in the digital domain, *signal filtering*, which is simply manipulating a signal to improve signal characteristics. For example, filtering can remove noise or static from a signal, thereby improving its signal-to-noise ratio.

It may not be obvious why it is desirable to filter signals using a microprocessor rather than analog components, but consider the advantages:

- Analog filters (and analog circuitry in general) are subject to behavior variation depending on environmental factors, such as temperature. Digital filters are essentially immune to such environmental effects.
- Digital filters are easily duplicated to within very tight tolerances, since their behavior does not depend on a combination of components, each of which deviates to some degree from its nominal behavior.
- Once it is manufactured, the defining characteristics of an analog filter (such as its *pass-band frequency range*) are not easily changed. By implementing a filter digitally using a micro-

Figure 1. The finite impulse response (FIR) filters is a typical DSP algorithm. FIR filters are useful for filtering noise, and other important functions.



processor, you can change filter characteristics simply by reprogramming the device.

There are several kinds of digital filters; one commonly used type is called a *finite impulse response* (FIR) filter, illustrated in Figure 1. The mechanics of the basic FIR filter algorithm are straightforward. The blocks labeled D in Figure 1 are unit delay operators; their output is a copy of the input sample, delayed by one sample period. A series of storage elements (usually memory locations) are used to implement a series of these delay elements (this series is called a *delay line*).<sup>1</sup>

At any given time,  $N-1$  of the most recently received input samples reside in the delay line, where  $N$  is the total number of input samples used in the computation of each output sample. Input samples are designated  $x_N$ ; the first input sample is  $x_1$ , the next is  $x_2$ , and so on.

Each time a new input sample arrives, the FIR filter operation shifts previously stored samples one place to the right along the delay line. It then computes a new output sample by multiplying the newly arrived sample and each of the previously stored input samples by the corresponding coefficient. In the figure, coefficients are represented as  $c_k$ , where  $k$  is the coefficient number. The summation of the multiplication products forms the new output sample,  $y_N$ .

We call the combination of a single delay element, the associated multiplication operation, and the associated addition operation a *tap*. The number of taps and the values chosen for the coefficients define the filter characteristics. For example, if the values of the coefficients are all equal to the reciprocal of the number of taps,  $1/N$ , the filter performs an averaging operation, one form of a *low-pass filter*.

More commonly, developers use filter design methods to determine coefficients that yield a desired frequency response for the filter. In mathematical terms, FIR filters perform a series of dot products: They take an input vector and a vector of coefficients, perform pointwise multiplication between the coefficients and a sliding window of input samples, and accumulate

the results of all the multiplications to form an output sample.

This brings us to the most popular operation in DSP: the *multiply-accumulate* (MAC).

### Handling MACs

To implement a MAC efficiently, a processor must efficiently perform multiplications. GPPs were not originally designed for multiplication-intensive tasks—even some modern GPPs require multiple instruction cycles to complete a multiplication because they don't have dedicated hardware for single-cycle multiplication. The first major architectural modification that distinguished DSP processors from the early GPPs was the addition of specialized hardware that enabled single-cycle multiplication.<sup>2</sup>

DSP processor architects also added *accumulator registers* to hold the summation of several multiplication products. Accumulator registers are typically wider than other registers, often providing extra bits, called *guard bits*, to avoid overflow.

To take advantage of this specialized multiply-accumulate hardware, DSP processor instruction sets nearly always include an explicit MAC instruction. This combination of MAC hardware and a specialized MAC instruction were two key differentiators between early DSP processors and GPPs.

### Memory architectures

Another highly visible difference between DSP processors and GPPs lies in their memory structure.

**von Neumann architecture.** Traditionally, GPPs have used a von Neumann memory architecture,<sup>3</sup> illustrated by Figure 2a. In the von Neumann architecture, there is one memory space connected to the processor core by one bus set (an address bus and a data bus). This works perfectly well for many computing applications; the memory bandwidth is sufficient to keep the processor fed with instructions and data.

The von Neumann architecture is not a good design for DSP, however, because typical DSP algorithms require more memory bandwidth than the von

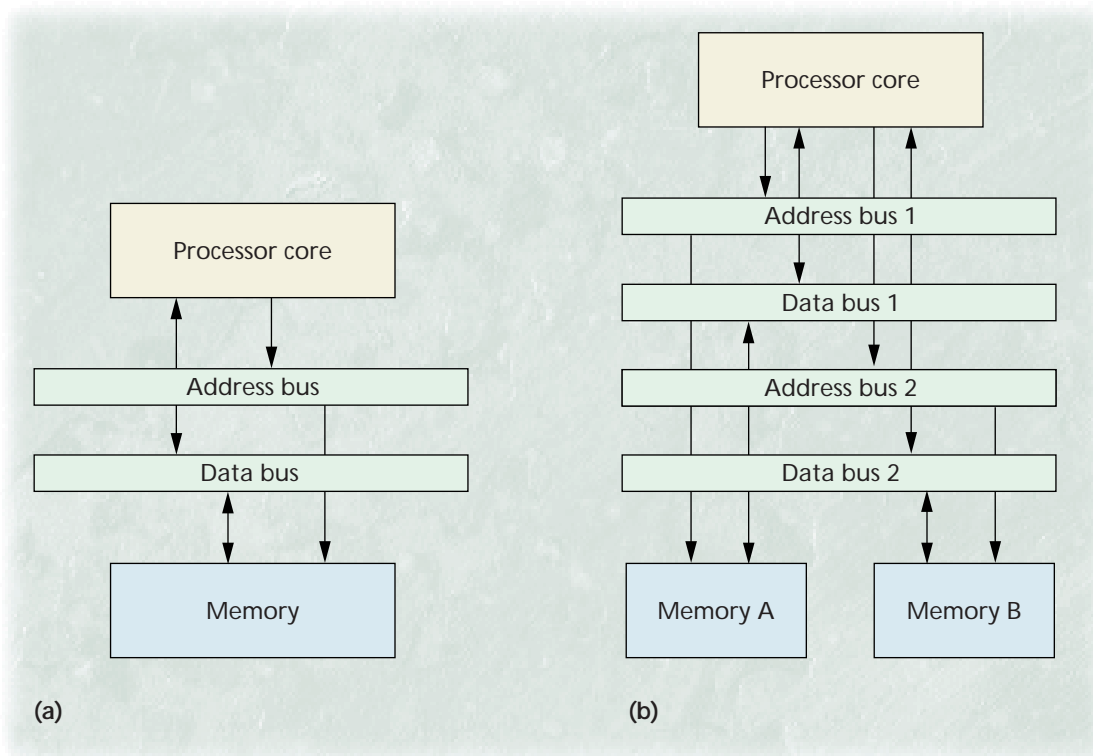


Figure 2. Many general-purpose processors use an (a) von Neumann memory architecture, which permits only one access to memory at a time. While adequate for general-purpose applications, the von Neumann architecture's memory bandwidth is insufficient for many DSP applications. DSP processors typically use a (b) Harvard memory architecture, which permits multiple simultaneous memory accesses.

Neumann architecture can provide. For example, to sustain a throughput of one FIR filter tap per instruction cycle (the hallmark of DSP processor performance), the processor must complete one MAC and make several accesses to memory within one instruction cycle. Specifically, in the straightforward case, the processor must

- fetch the MAC instruction,
- read the appropriate sample value from the delay line,
- read the appropriate coefficient value, and
- write the sample value to the next location in the delay line, in order to shift data through the delay line.

Thus, the processor must make a total of four accesses to memory in one instruction cycle. (In practice, most processors use various techniques to reduce the actual number of memory accesses needed to three or even two per tap. Nevertheless, virtually all processors require multiple memory accesses within one instruction cycle to compute an FIR filter at a sustained rate of one tap per instruction cycle.)

In a von Neumann memory architecture, four memory accesses would consume a minimum of four instruction cycles. Although most DSP processors include the arithmetic hardware necessary to perform single-cycle MACs, they wouldn't be able to realize the goal of one tap per cycle using a von Neumann memory structure; the processor simply wouldn't be able to retrieve samples and coefficients fast enough. For this reason, instead of a von Neumann architecture, most DSP processors use some form of Harvard architecture, illustrated by Figure 2b.

**Harvard architecture.** In a Harvard memory architecture, there are two memory spaces, typically partitioned as program memory and data memory (though there are modified versions that allow some crossover between the two). The processor core connects to these memory spaces by two bus sets, allowing two simultaneous accesses to memory. This arrangement doubles the processor's memory bandwidth, and it is crucial to keeping the processor core fed with data and instructions. The Harvard architecture is sometimes further extended with additional memory spaces and/or bus sets to achieve even higher memory bandwidth. However, the trade-off is that extra bus sets require extra power and chip space, so most DSP processors stick with two.

The Harvard memory architecture used in DSP processors is not unlike the memory structures used in modern high-performance GPPs such as the Pentium and PowerPC. Like DSPs, high-performance GPPs often need to make multiple memory accesses per instruction cycle (because of superscalar execution and/or because of instructions' data requirements). In addition, high-performance GPPs face another problem: With clock rates often in excess of 200 MHz, it can be extremely expensive (and sometimes impossible) to obtain a memory chip capable of keeping pace with the processor. Thus, high-performance GPPs often cannot access off-chip memory at their full clock speed. Using on-chip cache memory is one way that GPPs address both of these issues.

High-performance GPPs typically contain two on-chip memory caches—one for data and one for instructions—which are directly connected to the processor core. Assuming that the necessary information resides in cache, this arrangement lets the

One common characteristic of DSP algorithms is that most of the processing time is spent executing instructions contained within relatively small loops.

processor retrieve instruction and data words at full speed without accessing relatively slow, off-chip memory. It also enables the processor to retrieve multiple instructions or data words per instruction cycle.

Physically, this combination of dual on-chip memories and bus connections is nearly identical to a Harvard memory architecture. Logically, however, there are some important differences in the way DSP processors and GPPs with caches use their on-chip memory structures.

**Difference from GPPs.** In a DSP processor, the programmer explicitly controls which data and instructions are stored in the on-chip memory banks. Programmers must write programs so that the processor can efficiently use its dual bus

sets. In contrast, GPPs use control logic to determine which data and instruction words reside in the on-chip cache, a process that is typically invisible to the programmer. The GPP programmer typically does not specify (and may not know) which instructions and data will reside in the caches at any given time. From the GPP programmer's perspective, there is generally only one memory space rather than the two memory spaces of the Harvard architecture.

Most DSP processors don't have any cache; as described earlier, they use multiple banks of on-chip memory and multiple bus sets to enable several memory accesses per instruction cycle. However, some DSP processors do include a very small, specialized, on-chip instruction cache, separate from the on-chip memory banks and inside the core itself. This cache is used for storing instructions used in small inner loops so that the processor doesn't have to use its on-chip bus sets to retrieve instruction words. By fetching instructions from the cache, the DSP processor frees both on-chip bus sets to retrieve data words.

Unlike GPPs, DSP processors almost never incorporate a data cache. This is because DSP data is typically *streaming*: That is, the DSP processor performs computations with each data sample and then discards the sample, with little reuse.

### Zero-overhead looping

It may not be obvious why a small, specialized instruction cache would be particularly useful, until you realize that one common characteristic of DSP algorithms is that most of the processing time is spent executing instructions contained within relatively small loops.

In an FIR filter, for example, the vast majority of processing takes place within a very small inner loop that multiplies the input samples by their corresponding coefficients and adds the results. This is why the small on-chip instruction cache can significantly improve the processor's performance on DSP algo-

rithms. It is also why most DSP processors include specialized hardware for *zero-overhead looping*. The term zero-overhead looping means that the processor can execute loops without consuming cycles to test the value of the loop counter, perform a conditional branch to the top of the loop, and decrement the loop counter.

In contrast, most GPPs don't support zero-overhead hardware looping. Instead, they implement looping in software. Some high-performance GPPs achieve nearly the same effect as hardware-supported zero-overhead looping by using branch prediction hardware. This method has drawbacks in the context of DSP programming, however, as discussed later.

### Fixed-point computation

Most DSP processors use fixed-point arithmetic rather than floating-point. This may seem counterintuitive, given the fact that DSP applications must pay careful attention to numeric fidelity—which is much easier to do with a floating-point data path. DSP processors, however, have an additional imperative: They must be inexpensive. Fixed-point machines tend to be cheaper (and faster) than comparable floating-point machines.

To maintain numeric accuracy without the benefit of a floating-point data path, DSP processors usually include, in both the instruction set and underlying hardware, good support for saturation arithmetic, rounding, and shifting.

### Specialized addressing

DSP processors often support specialized addressing modes that are useful for common signal-processing operations and algorithms. Examples include modulo (circular) addressing (which is useful for implementing digital-filter delay lines) and bit-reversed addressing (which is useful for performing a commonly used DSP algorithm, the fast Fourier transform). These highly specialized addressing modes are not often found on GPPs, which must instead rely on software to implement the same functionality.

### EXECUTION TIME PREDICTABILITY

Aside from differences in the specific types of processing performed by DSPs and GPPs, there are also differences in their performance requirements. In most non-DSP applications, performance requirements are typically given as a maximum average response time. That is, the performance requirements do not apply to every transaction, but only to the overall performance.

### Hard real-time constraints

In contrast, the most popular DSP applications (such as cell phones and modems) are *hard* real-time

applications—all processing must take place within some specified amount of time in every instance. This performance constraint requires programmers to determine exactly how much processing time each sample will require; or at the very least, how much time will be consumed in the worst-case scenario. At first blush, this may not seem like a particularly important point, but it becomes critical if you attempt to use a high-performance GPP to perform real-time signal processing.

Execution time predictability probably won't be an issue if you plan to use a low-cost GPP for real-time DSP tasks, because low-cost GPPs (like DSP processors) have relatively straightforward architectures and easy-to-predict execution times. However, most real-time DSP applications require more horsepower than low-cost GPPs can provide, so the developer must choose either a DSP processor or a high-performance GPP. But which is the best choice? In this context, execution-time predictability plays an important role.

### Problems with some GPPs

For example, some high-performance GPPs incorporate complicated algorithms that use branching history to predict whether a branch is likely to be taken. The processor then executes instructions speculatively, on the basis of that prediction. This means that the same section of code may consume a different number of instruction cycles, depending on events that take place beforehand.

When a processor design layers many different dynamic features—such as branch prediction and caching—on top of each other, it becomes nearly impossible to predict how long even a short section of code will take to execute. Although it may be possible for programmers to determine the worst-case execution time, this may be an order of magnitude greater than the actual execution time. Assuming worst-case behavior can force programmers to be extremely conservative in implementing real-time DSP applications on high-performance GPPs. A lack of execution time predictability also adversely affects the programmer's (or compiler's) ability to optimize code, as we will discuss later.

### Easy for DSP processors

Let's compare this to the effort required to predict execution times on a DSP processor. Some DSP processors use caches, but the programmer (not the processor) decides which instructions go in them, so it's easy to tell whether instructions will be fetched from cache or from memory. DSP processors don't generally use dynamic features such as branch prediction and speculative execution. Hence, predicting the amount of time required by a given section of code is fairly straightforward on a DSP. This execution-time predictability allows pro-

grammers to confidently push the chip's performance limits.

### FIXED-POINT DSP INSTRUCTION SETS

Fixed-point DSP processor instruction sets are designed with two goals in mind. They must

- enable the processor to perform multiple operations per instruction cycle, thus increasing per-cycle computational efficiency (a goal also supported by endowing the processor with multiple execution units capable of parallel operation), and
- minimize the amount of memory space required to store DSP programs (critical in cost-sensitive DSP applications because memory contributes substantially to the overall system cost).

To accomplish these goals, DSP processor instruction sets generally allow programmers to specify several parallel operations in a single instruction. However, to keep word size small, the instructions only permit the use of certain registers for certain operations and do not allow arbitrary combinations of operations. The net result is that DSP processors tend to have highly specialized, complicated, and irregular instruction sets.

To keep the processor fed with data without bloating program size, DSP processors almost always allow the programmer to specify one or two parallel data moves (along with address pointer updates) in parallel with certain other operations, like MACs.

**Typical instruction.** As an illustration of a typical DSP instruction, consider the following Motorola DSP56300 instruction (X and Y denote the two memory spaces of the Harvard architecture):

```
MAC X0, Y0, A X: (R0)+, X0 Y: (R4)+N4, Y0
```

This instruction directs the DSP56300 to

- multiply the contents of registers X0 and Y0,
- add the result to a running total kept in accumulator A,
- load register X0 from the X memory location pointed to by register R0,
- load register Y0 from the Y memory location pointed to by R4,
- postincrement R0 by one, and
- postincrement R4 by the contents of register N4.

This single instruction line includes all the operations needed to calculate an FIR filter tap. It is clearly a highly specialized instruction designed specifically for DSP applications. The price for this efficiency is

Most DSP applications depend on processing taking place within some specified amount of time in every instance. This execution time predictability is difficult to provide on high-performance GPPs.

A compiler can take C source code and generate functional assembly code for a DSP, but to get efficient code, programmers invariably optimize the program's critical sections by hand.

an instruction set that is neither intuitive nor easy to use (in comparison to typical GPP instruction sets).

**Difference from GPPs.** GPP programmers typically don't care about a processor instruction set's ease of use because they generally develop programs in a high-level language, such as C or C++. Life isn't quite so simple for DSP programmers, because mainstream DSP applications are written (or at least have portions optimized) in assembly language. This turns out to have important implications in comparing processor performance, but we'll get to that later.

There are two main reasons why DSP processors usually aren't programmed in high-level languages. First, most widely used high-level languages, such as C, are not well suited for describing typical DSP algorithms. Second, the complexity of DSP architectures—with their multiple memory spaces, multiple buses, irregular instruction sets, and highly specialized hardware—make it difficult to write efficient compilers.

It is certainly true that a compiler can take C source code and generate functional assembly code for a DSP, but to get efficient code, programmers invariably optimize the program's critical sections by hand. DSP applications typically have very high computational demands coupled with strict cost constraints, making program optimization essential. For this reason, programmers often consider the “palatability” (or lack thereof) of a DSP processor's instruction set as a key component in its overall desirability.

Because DSP applications require highly optimized code, most DSP vendors provide a range of development tools to assist DSP processor programmers in the optimization process. For example, most DSP processor vendors provide processor simulation tools that accurately model the processor's activity during every instruction cycle. This is a valuable tool both for ensuring real-time operation and for code optimization.

GPP vendors, on the other hand, don't usually provide this type of development tool, mainly because GPP programmers typically don't need this level of detailed information. The lack of a cycle-accurate simulator for a GPP can be a real problem for DSP application programmers. Without one, it can be nearly impossible to predict the number of cycles a high-performance GPP will require for a given task. Think about it: If you can't tell how many cycles are required, how can you tell if the changes you make are actually improving code performance?

### TODAY'S DSP LANDSCAPE

Like GPPs, the performance and price of DSP processors vary widely.<sup>4</sup>

### Low-cost workhorses

In the low-cost, low-performance range are the industry workhorses. Included in this group are Analog Devices' ADSP-21xx, Texas Instruments' TMS320C2xx, and Motorola's DSP560xx families. These processors generally operate at around 20 to 50 native MIPS (that is, a million instructions per second, not Dhrystone MIPS) and provide good DSP performance while maintaining very modest power consumption and memory usage. They are typically used in consumer products that have modest DSP performance requirements and stringent energy consumption and cost constraints, like disk drives and digital answering machines.

### Low-power midrange

Midrange DSP processors achieve higher performance through a combination of increased clock speed and more sophisticated hardware. DSP processors like the Lucent Technologies DSP16xx and Texas Instruments TMS320C54x operate at 100 to 120 native MIPS and often include additional features, such as a barrel shifter or instruction cache, to improve performance on common DSP algorithms.

Processors in this class also tend to have more sophisticated (and deeper) pipelines than their lower performance cousins. These processors can have substantially better performance while still keeping energy and power consumption low. Processors in this performance range are typically used in wireless telecommunications applications and high-speed modems, which have relatively high computational demands but often require low power consumption.

### Diversified high-end

Now we come to the high-end DSP processors. It is in this group that DSP architectures really start to branch out and diversify, propelled by the demand for ultrafast processing. DSP processor architects who want to improve performance beyond the gains afforded by faster clock speeds must get more useful DSP work out of every instruction cycle. Of course, architects designing high-performance GPPs are motivated by the same goal, but the additional goals of maintaining execution time predictability, minimizing program size, and limiting energy consumption typically do not constrain their design decisions. There are several ways DSP processor architects increase the amount of work accomplished in each cycle; we discuss two approaches here.

**Enhanced conventional DSP processor.** The first approach is to extend the traditional DSP architecture by adding more parallel computational units to the data path, such as a second multiplier or adder. This approach requires an extended instruction set that takes advantage of the additional hardware by encod-

ing even more operations in a single instruction and executing them in parallel. We refer to this type of processor as an *enhanced conventional DSP* because the approach is an extension of the established DSP architectural style.

The Lucent Technologies DSP16210, which has two multipliers, an arithmetic logic unit, an adder (separate from the ALU), and a bit manipulation unit, is a prime example of this approach. Lucent also equipped the DSP16210 with two 32-bit data buses, enabling it to retrieve four 16-bit data words from memory in every instruction cycle (assuming the words are retrieved in pairs). These wider buses keep the dual multipliers and other functional units from starving for data. The DSP16210, which executes at 100 native MIPS, offers a strong boost in performance while maintaining a cost and energy footprint similar to previous generations of DSP processors. It is specifically targeted at high-performance telecommunications applications, and it includes specialized hardware to accelerate common telecommunications algorithms.

**Multiple-instruction issue.** Another way to get more work out of every cycle is to issue more than one instruction per instruction cycle. This is common in high-end GPPs, which are often 2- or even 4-way superscalar (they can issue and execute up to 2 or 4 instructions per cycle). It's a relatively new technique in the DSP world, however, and has mostly been implemented using VLIW (very long instruction word) rather than superscalar architectures.

A prime example of this approach is the much-publicized Texas Instruments TMS320C6201. This VLIW processor pulls in up to 256 bits of instruction words at a time, breaks them into as many as eight 32-bit subinstructions, and passes them to its eight independent computational units. In the best case, all eight units are active simultaneously, and the processor executes eight subinstructions in parallel.

The TMS320C6201 has a projected clock rate of 200 MHz, which translates into a peak MIPS rating of 1,600. The catch here is that each subinstruction is extremely simple (by DSP standards). Thus, it may take several TMS320C6201 instructions to specify the same amount of work that a conventional DSP can specify in a single instruction. In addition, it is often not possible to keep all eight execution units running in parallel; more typically, five or six will be active at any one time. The performance gain afforded by using this VLIW approach combined with a high clock rate is substantial. However, it is not nearly as high as you might expect from comparing the 1,600 MIPS rating with the 100 MIPS rating of the Lucent DSP16210. This disparity arises because a typical DSP16210 instruction accomplishes more work than a typical TMS320C6201 instruction, a critical distinction that simple metrics such as MIPS fail to capture.

Like most VLIW processors, the TMS-320C6201 consumes much more energy than traditional DSP processors, and requires relatively large amount of program memory. For these reasons, the chip is not well suited for portable applications. TI gave up energy and memory efficiency in exchange for ultrahigh performance, producing a processor intended for line-powered applications, such as modem banks, where it can take the place of several lower performance DSP processors.

### GPPS GET DSP

In the past few years, GPP developers have begun enhancing their processor designs with DSP extensions. For example, Intel added the Pentium's MMX extensions, which specifically support DSP and other multimedia tasks. MMX also gives the Pentium SIMD (single instruction, multiple data) capabilities, significantly improving the processor's speed on DSP algorithms.

Many low- and moderate-performance GPPs are now available in versions that include DSP hardware, resulting in hybrid GP/DSP processors. Hitachi, for example, offers a DSP-enhanced version of its SH-2 microcontroller, called the SH-DSP. Advanced RISC Machines, a vendor of licensable microcontroller cores for use in custom chips, recently introduced a licensable DSP coprocessor core called Piccolo. Piccolo is intended to augment ARM's low-end GPP, the ARM7, with signal-processing capabilities. In short, just about all of the major GPP vendors are adding DSP enhancements to their processors in one form or another, and the distinction between GPPs and DSP processors is not quite as clear as it once was.

### WHICH ONE IS BETTER?

There are a variety of metrics you can use to judge processor performance. The most often cited is speed, but other metrics, such as energy consumption or memory usage, can be equally important, especially in embedded-system applications. Like developers using GPPs, DSP engineers must be able to accurately compare many facets of processor performance so that they can decide which processor to choose.

In light of the ever-increasing number of processor families for DSP applications, it has become more difficult than ever for system designers to choose the processor that will provide the best performance in a given application.

In the past, DSP designers have relied on MIPS or similar metrics for a rough idea of the relative horsepower provided by various DSP chips. Unfortunately, as processor architectures have diversified, traditional metrics such as MIPS have become less relevant and often downright misleading. Engineers must be wary

Unfortunately, as processor architectures have diversified, traditional metrics such as MIPS have become less relevant and often downright misleading.

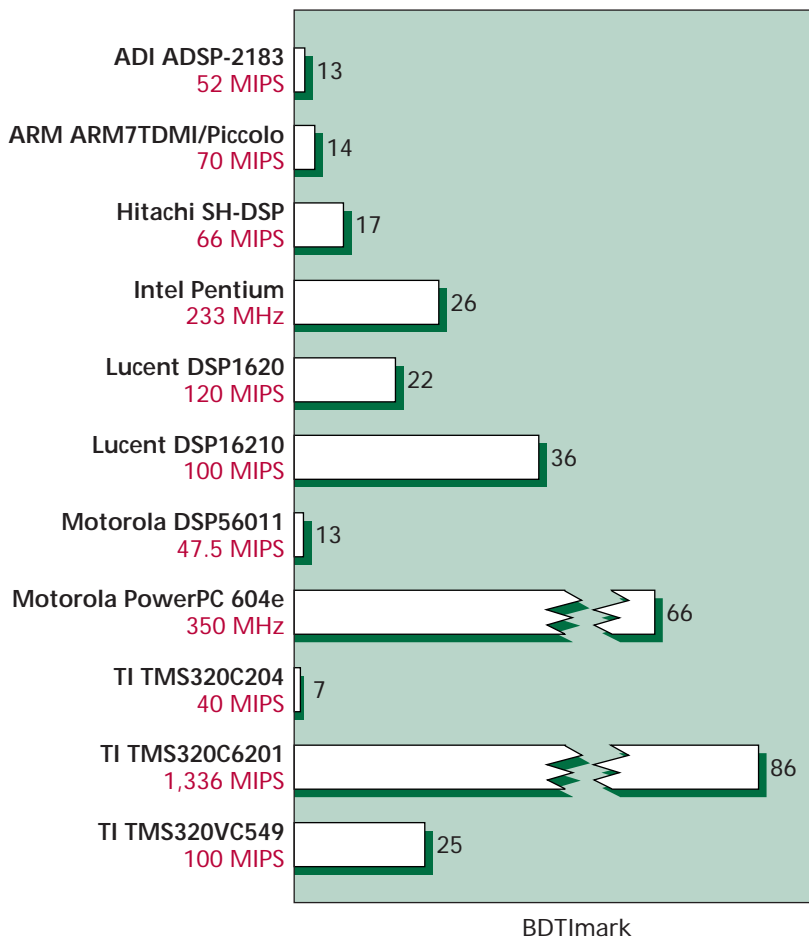


Figure 3. Comparison of selected commercial DSP, general-purpose, and hybrid GP/DSP processors by BDTImarks and MIPS (or MHz for GPPs). BDTImark is a composite measure based on measurements from a set of DSP-specific benchmarks. Note how MIPS ratios do not necessarily provide a good indication of comparative DSP performance. Scores for other processors are available at <http://www.bdti.com>.

of the performance claims presented in sales brochures; all MIPS are not created equal.

The root problem with simple metrics like MIPS is that they don't actually measure performance, because performance is a function of more than just the number of instructions executed per second. Performance is also a function of how much work is accomplished in each instruction, which depends on the processor's specific architecture and instruction set. Thus, when processor vendors cite MIPS ratings, they are leaving out a crucial piece of information: the amount of work each instruction performs.

Clearly, engineers need a way to gauge processor performance on DSP algorithms that isn't tied to a specific architecture. But what's the best way to do that?

One possibility would be to implement complete DSP applications on each processor under consideration, and compare the amount of time each requires to complete the given task. This method is often used for benchmarking general-purpose computer systems that run applications written in a high-level language. Once developers finish an application, they can easily

recompile it for different computers and measure the run time. However, for reasons discussed earlier, DSP systems are usually programmed (at least to some degree) in assembly language.

This makes full-application benchmarks unattractive for DSP performance comparisons for two main reasons:

- If the application is programmed in a high-level language, the quality of the compiler will greatly affect performance results. Hence, the benchmark would measure both the processor and the compiler.
- Although it's certainly possible to develop and optimize entire applications in assembly code, it is impractical for the purposes of comparing processor performance because the application would have to be recoded and optimized on every processor under consideration.

### Benchmarks

Fortunately, one of the characteristics of DSP applications is that the majority of the processing effort is often concentrated in a few relatively small pieces (kernels) of the program, which can be isolated and used as the basis for benchmark comparisons.

Berkeley Design Technology Inc. (BDTI), a DSP technology analysis and software development firm, has developed a DSP benchmarking methodology based on a group of DSP algorithm kernels.

Algorithm kernels, such as the FIR filter algorithm, form the heart of many DSP applications. They make good benchmark candidates because of their inherent relevance to DSP and because they are small enough to implement and optimize in assembly code in a reasonable amount of time.

Over the past six years, BDTI has implemented their suite of kernel-based benchmarks, the BDTI Benchmarks, on a wide variety of processors. By looking at benchmark results, designers can see exactly which algorithm kernels a given processor performs efficiently. Given information about the relative importance of each algorithm in the overall application (we refer to this information as an *application profile*), system designers can accurately assess which DSP is best for the application under consideration.

The results of BDTI's comprehensive benchmarking effort provide an extremely detailed, in-depth analysis of processors' performance on typical DSP algorithms. However, we still would like a single-number metric for quick comparisons. For this reason, BDTI introduced and trademarked a new composite speed metric, the *BDTI*mark, last year.

### The BDTImark

The BDTImark takes the execution time results from all of the BDTI Benchmarks and crunches them into one number. It provides a far more accurate



assessment of a processor's DSP performance than other simplified metrics (such as MIPS) because it is a measurement of execution speed on actual DSP algorithms. In Figure 3, we present the BDTI mark scores for a variety of DSP, GP, and hybrid DSP/GP processors, including those discussed earlier.

Clearly, a quick comparison of BDTI mark scores and MIPS ratings shows that a higher MIPS rating does not necessarily translate into better DSP performance.<sup>4</sup>

For example, consider the BDTI mark scores for the 100-MIPS DSP16210 and the 120-MIPS DSP1620. The DSP16210 is about 1.5 times faster, because it has an extra multiplier and other hardware that lets it do substantially more work in every instruction cycle. The TMS320C6201, shown here at 167 MHz (1,336 MIPS), achieves an impressive BDTI mark score, but is not 13 times faster than the DSP16210, as might be expected from the two processors' MIPS ratio.

Another point of interest is that the score for the Pentium is actually higher than the scores for many of the low- to moderate-performance DSP processors—a surprising result mostly attributable to the processor's 233-MHz clock rate. The Motorola PowerPC 604e performs even better; in fact, it is faster on the DSP benchmarks than nearly all the DSP processors.

This observation leads to a common question: Why use a DSP processor at all when the DSP capabilities of high-end GPPs such as the PowerPC 604e are becoming so strong?

The answer is, there's more to it than raw performance. Although high-end GPPs are able to perform DSP work at a rate comparable to many DSP processors, they achieve this performance by using complicated dynamic features. For this reason, high-end GPPs are not well suited for real-time applications—dynamic features cause real problems both in terms of guaranteeing real-time behavior and optimizing code. In addition, the theoretical peak performance of a high-end GPP may never be achieved in real-time DSP programs, because the programmer may have to assume worst-case behavior and write the software accordingly.

High-end GPPs also tend to cost substantially more money and consume more power than DSP processors, an unacceptable combination in, for example, highly competitive portable telecommunications applications. And, although software development tools for the most widely used GPPs are much more sophisticated than those of their DSP counterparts, they are not geared toward DSP software development and lack features that are essential in the DSP world.

It will be interesting to see how well the more recent additions to the DSP world—the hybrid GP/DSP processors—can penetrate the market. These processors, such as the ARM7/Piccolo and the SH-

DSP, don't suffer from the drawbacks that accompany high-end GPPs, but they also don't offer the same level of performance.

The bottom line is, if there is a high-performance GPP in the existing system (as in the case of a PC), it may be attractive to use for signal processing and to avoid adding a separate DSP processor. And, particularly in the case of GPPs enhanced with DSP extensions, it may be possible to get good DSP performance out of the system without adding a separate DSP processor. If you are building a DSP application from scratch, however, it is likely that a dedicated DSP or hybrid GP/DSP processor will be a better choice, for reasons of economy, lower power consumption, and ease of development.

Though high-performance GPPs have already begun to challenge DSP speed, DSP processors aren't likely to be supplanted in the near future because they are able to provide extremely strong signal processing performance with unmatched economy. ♦

#### References

1. R. Lyons, *Understanding Digital Signal Processing*, Addison Wesley Longman, Reading, Mass., 1997.
2. P. Lapsley et al., *DSP Processor Fundamentals*, IEEE Press, New York, 1997.
3. J. Hennessy and D. Patterson, *Computer Organization and Design*, Morgan Kaufmann, San Francisco, 1998.
4. *Buyer's Guide to DSP Processors*, Berkeley Design Technology Inc., Berkeley, Calif., 1994, 1995, and 1997.

*Jennifer Eyre is an engineer and a technical writer at Berkeley Design Technology Inc. where she analyzes and evaluates microprocessors used in DSP applications. Eyre received a BSEE and an MSEE from UCLA. She is a member of Eta Kappa Nu.*

*Jeff Bier is cofounder and general manager of BDTI, where he oversees DSP technology analysis and software development services. He has extensive experience in software, hardware, and design tool development for DSP and control applications. Bier received a BS from Princeton University and an MS from the University of California, Berkeley. He is a member of the IEEE Design and Implementation of Signal Processing Systems (DISPS) Technical Committee.*

Contact the authors at BDTI, 2107 Dwight Way, Second Floor, Berkeley, CA 94704; {eyre, bier}@bdti.com.