Why Are You Reading This Book?

Well, assuming that you find your pleasure reading elsewhere, we hope you’re reading this book for at least one of the following reasons:

- You’d like to understand more about what embedded media processing entails, and what demands it places on designers who are more familiar with conventional systems based on microcontrollers (MCUs) or digital signal processors (DSPs).
- You need to design an embedded multimedia system and want to know how to choose the proper processor for the application.
- You’ve already chosen a processor, and you want to know how to start architecting your system so that you can avoid problems at the outset.
- You’re in the middle of designing a multimedia system, but you’ve run into some snags with trying to get all of your data flows and memory accesses to line up properly.
- You’re related to one of the authors.

In this book we concern ourselves with embedded media processing (EMP) systems—that is, applications involving large data blocks (whether image, video, audio, speech, or some combination of these), a need for signal processing, and (often) a real-time nature. By “embedded,” we mean a specialized processor that’s part of a larger system and is targeted for a specific application (as opposed to a personal computer, which is intended to do some of everything). A further requirement of many
EMP systems is portability, so battery life is a key factor, and it’s often necessary to reach a compromise between power dissipation and computational performance.

So What’s All the Excitement About Embedded Multimedia Systems?

With the multimedia revolution in full swing, we're becoming accustomed to toting around cell phones, PDAs, cameras and MP3 players, concentrating our daily interactions into the palms of our hands. But given the usefulness of each gadget, it’s surprising how often we upgrade to “the latest and greatest” device. This is, in part, due to the fact that the cell phone we bought last year can’t support the new video clip playback feature touted in this year’s TV ads.

After all, who isn't frustrated after discovering that his portable audio player gets tangled up over the latest music format? And which overworked couple has the time, much less the inclination, to figure out how to get the family vacation travelogue off their Mini-DV camcorder and onto a DVD or hard disk?

As Figure 1.1 implies, we’ve now reached the point where a single gadget can serve as a phone, a personal organizer, a camera, an audio player, and a web-enabled portal to the rest of the world.

But still, we’re not happy.

Let’s add a little perspective: we used to be satisfied just to snap a digital picture and see it on our computer screen. Just 10 years ago, there were few built-in digital camera features, the photo resolution was comparatively low, and only still pictures were an option. Not that we were complaining, since previously our only digital choice involved scanning 35-mm prints into the computer.

In contrast, today we expect multi-megapixel photos, snapped several times per second, that are automatically white-balanced and color-corrected. What’s more, we demand seamless transfer between our camera and other media nodes, a feature made practical only because the camera can compress the images before moving them.

Clearly, consumer appetites demand steady improvement in the “media experience.” That is, people want high-quality video and audio streams in small form factors, with
low power requirements (for improved battery life) and at low cost. This desire leads
to constant development of better compression algorithms that reduce storage require-
ments while increasing audio/video resolution and frame rates.

To a large extent, the Internet drives this evolution. After all, it made audio, images
and streaming video pervasive, forcing transport algorithms to become increasingly
clever at handling ever-richer media across the limited bandwidth available on a
network. As a result, people today want their portable devices to be Net-connected,
high-speed conduits for a never-ending information stream and media show. Unfor-
tunately, networking infrastructure is upgraded at a much slower rate than bandwidth
demands grow, and this underscores the importance of excellent compression ratios
for media-rich streams.

It may not be readily apparent, but behind the scenes, processors have had to evolve
dramatically to meet these new and demanding requirements. They now need to run
at very high clock rates (to process video in real time), be very power efficient (to prolong battery life), and comprise very small, inexpensive single-chip solutions (to save board real estate and keep end products price-competitive). What’s more, they need to be software-reprogrammable, in order to adapt to the rapidly changing multimedia standards environment.

**A Simplified Look at a Media Processing System**

Consider the components of a typical media processing system, shown in Figure 1.2. Here, an input source presents a data stream to a processor’s input interface, where it is manipulated appropriately and sent to a memory subsystem. The processor core(s) then interact with the memory subsystem in order to process the data, generating intermediate data buffers in the process. Ultimately, the final data buffer is sent to its destination via an output subsystem. Let’s examine each of these components in turn.

![Figure 1.2 Components of a typical media processing system](image)

**Core Processing**

Multimedia processing—that is, the actual work done by the media processor core—boils down into three main categories: format coding, decision operating and overlaying.

Software *format coders* separate into three classifications. *Encoders* convert raw video, image, audio and/or voice data into a compressed format. A digital still camera (DSC) provides a good example of an encoding framework, converting raw image sensor data into compressed JPEG format. *Decoders*, on the other hand, convert
a compressed stream into an approximation (or exact duplicate) of the original uncompressed content. In playback mode, a DSC decodes the compressed pictures stored in its file system and displays them on the camera’s LCD screen. Transcoders convert one media format into another one, for instance MP3 into Windows Media Audio 9 (WMA9).

Unlike the coders mentioned above, decision operators process multimedia content and arrive at some result, but do not require the original content to be stored for later retrieval. For instance, a pick-and-place machine vision system might snap pictures of electronic components and, depending on their orientation, size and location, rotate the parts for proper placement on a circuit board. However, the pictures themselves are not saved for later viewing or processing. Decision operators represent the fastest growing segment of image and video processing, encompassing applications as diverse as facial recognition, traffic light control, and security systems.

Finally, overlays blend multiple media streams together into a single output stream. For example, a time/date stamp might be instantiated with numerous views of surveillance footage to generate a composited output onto a video monitor. In another instance, graphical menus and icons might be blended over a background video stream for purposes of annotation or user input.

Considering all of these system types, the input data varies widely in its bandwidth requirements. Whereas raw audio might be measured in tens of kilobits/second (kb/s), compressed video could run several megabits per second (Mbps), and raw video could entail tens of megabytes per second (Mbytes/s). Thus, it is clear that the media processor needs to handle different input formats in different ways. That’s where the processor’s peripheral set comes into play.


**Chapter 1**

**Input/Output Subsystems—Peripheral Interfaces**

Peripherals are classified in many ways, but a particularly useful generalization is to stratify them into functional groups like those in Table 1.1. Basically, these interfaces act to help control a subsystem, assist in moving and storing data, or enable connectivity with other systems or modules in an application.

<table>
<thead>
<tr>
<th>Subsystem Control</th>
<th>Storage</th>
<th>Connectivity</th>
<th>Data Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Low-speed Serial Interfaces</td>
<td>• Asynchronous Memory Interface</td>
<td>• USB 2.0 On-the-Go/Host/Device</td>
<td>• Low-speed Serial Interfaces</td>
</tr>
<tr>
<td>• Programmable Timers</td>
<td>• ATAPI / Serial ATA</td>
<td>• PCI</td>
<td>• USB 2.0 Full Speed (12 Mbps)</td>
</tr>
<tr>
<td>• General-purpose I/O</td>
<td>• Flash Storage Card Interfaces</td>
<td>• IEEE 802.3 (Ethernet)</td>
<td>• 10BaseT Ethernet</td>
</tr>
<tr>
<td>• Real-Time Clock</td>
<td>• SDRAM/DDR Interface</td>
<td>• IEEE 802.11 a/b/g (WiFi)</td>
<td>• IEEE 802.11b</td>
</tr>
<tr>
<td>• Watchdog Timer</td>
<td></td>
<td>• IEEE 1394 (Firewire)</td>
<td>• Synchronous Serial Audio/Data Ports</td>
</tr>
<tr>
<td>• Host Interface</td>
<td></td>
<td>• Asynchronous Memory Interface</td>
<td>• Host Interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Flash Storage Card Interfaces</td>
<td>• IEEE 802.11a/g</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 100BaseT Ethernet</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• USB 2.0 High Speed (480 Mbps)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• IEEE 1394 (Firewire)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Parallel Video/Data Ports</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Gigabit Ethernet</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PCI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PCI Express</td>
</tr>
</tbody>
</table>

Let’s look now at some examples of each interface category.

**Subsystem Control**

**Low-Speed Serial Interfaces**

UART—*Universal Asynchronous Receiver/Transmitter*—As its name suggests, this full-duplex interface needs no separate clock or frame synchronization lines. Instead, these are decoded from the bit stream in the form of start bit, data bits, stop bits, and optional parity bits. UARTs are fairly low-speed (kbps to Mbps) and have high overhead, since every data word has control and error checking bits associated with it. UARTs can typically support RS-232 modem implementations, as well as IrDA™ functionality for close-range infrared transfer.
SPI—*Serial Peripheral Interface*—This is a synchronous, moderate-speed (tens of Mbps), full-duplex master/slave interface developed by Motorola. The basic interface consists of a clock line, an enable line, a data input (“Master In, Slave Out”) and a data output (“Master Out, Slave In”). SPI supports both multimaster and multislave environments. Many video and audio codecs have SPI control interfaces, as do many EEPROMs. We’ll talk more about SPI in Chapter 5.

I²C—*Inter-IC Bus*—Developed by Philips, this synchronous interface requires only two wires (clock and data) for communication. The phase relationships between the two lines determines the start and completion of data transfer. There are primarily three speed levels: 100 kbps, 400 kbps and 3.4 Mbps. Like SPI, I²C is very commonly used for the control channel in video and audio converters, as well as in some ROM-based memories.

*Programmable Timers*—These multifunction blocks can generate programmable pulse-width modulated (PWM) outputs that are useful for one-shot or periodic timing waveform generation, digital-to-analog conversion (with an external resistor/capacitor network, for instance), and synchronizing timing events (by starting several PWM outputs simultaneously). As inputs, they’ll typically have a width-capture capability that allows precise measurement of an external pulse, referenced to the processor’s system clock or another timebase. Finally, they can act as event counters, counting external events or internal processor clock cycles (useful for operating system ticks, for instance).

*Real-Time Clock (RTC)*—This circuit is basically a timer that uses a 32.768 kHz crystal or oscillator as a time base, where every $2^{15}$ ticks equals one second. In order to use more stable crystals, sometimes higher frequencies are employed instead; the most common are 1.048 MHz and 4.194 MHz. The RTC can track seconds, minutes, hours, days, and even years—with the functionality to generate a processor alarm interrupt at a particular day, hour, minute, second combination, or at regular intervals (say, every minute). For instance, a real-time clock might wake up a temperature sensor to sample the ambient environment and relay information back to the MCU via I/O pins. Then, a timer’s pulse-width modulated (PWM) output could increase or decrease the speed of a fan motor accordingly.

*Programmable Flags/GPIO (General Purpose Inputs/Outputs)*—These all-purpose pins are the essence of flexibility. Configured as inputs, they convey status information from the outside world, and they can be set to interrupt upon receiving an
edge-based or level-based signal of a given polarity. As outputs, they can drive high or low to control external circuitry. GPIO can be used in a “bit-banging” approach to simulate interfaces like I²C, detect a keypress through a key matrix arrangement, or send out parallel chunks of data via block writes to the flag pins.

**Watchdog Timer**—This peripheral provides a way to detect if there’s a system software malfunction. It’s essentially a counter that is reset by software periodically with a count value such that, in normal system operation, it never actually expires. If, for some reason, the counter reaches 0, it will generate a processor reset, a nonmaskable interrupt, or some other system event.

**Host Interface**—Often in multimedia applications an external processor will need to communicate with the media processor, even to the point of accessing its entire internal/external memory and register space. Usually, this external host will be the conduit to a network, storage interface, or other data stream, but it won’t have the performance characteristics that allow it to operate on the data in real time. Therefore the need arises for a relatively high-bandwidth “host port interface” on the media processor. This port can be anywhere from 8 bits to 32 bits wide and is used to control the media processor and transfer data to/from an external processor.

**Storage**

**External Memory Interface (Asynchronous and SDRAM)**—An external memory interface can provide both asynchronous memory and SDRAM memory controllers. The asynchronous memory interface facilitates connection to FLASH, SRAM, EEROM and peripheral bridge chips, whereas SDRAM provides the necessary storage for computationally intensive calculations on large data frames. We’ll talk much more about these memory interfaces in Chapter 2.

It should be noted that, while some designs may employ the external memory bus as a means to read in raw multimedia data, this is often a suboptimal solution. Because the external bus is intimately involved in processing intermediate frame buffers, it will be hard pressed to manage the real-time requirements of reading in a raw data stream while writing and reading intermediate data blocks to and from L1 memory. This is why the video port needs to be decoupled from the external memory interface, with a separate data bus.

**ATAPI/Serial ATA**—These are interfaces used to access mass storage devices like hard disks, tape drives, and optical drives (CD/DVD). Serial ATA is a newer standard that
encapsulates the venerable ATAPI protocol, yet in a high-speed serialized form, for increased throughput, better noise performance, and easier cabling. We’ll talk more about these interfaces in Chapter 2.

*Flash Storage Card Interfaces*—These peripherals originally started as memory cards for consumer multimedia devices like cameras and PDAs. They allow very small footprint, high density storage and connectivity, from mass storage to I/O functions like wireless networking, Bluetooth and Global Positioning System (GPS) receivers. They include CompactFlash, Secure Digital (SD), MemoryStick, and many others. Given their rugged profile, small form factor and low power requirements, they’re perfect for embedded media applications. We’ll discuss these cards further in Chapter 2.

**Connectivity**

Interfacing to PCs and PC peripherals remains essential for most portable multimedia devices, because the PC constitutes a source of constant Internet connectivity and near-infinite storage. Thus, a PC’s 200-Gbyte hard drive might serve as a “staging ground” and repository for a portable device’s current song list or video clips. To facilitate interaction with a PC, a high-speed port is mandatory, given the substantial file sizes of multimedia data. Conveniently, the same transport channel that allows portable devices to converse in a peer-to-peer fashion often lets them dock with the “mother ship” as a slave device.

*Universal Serial Bus (USB) 2.0*—Universal Serial Bus is intended to simplify communication between a PC and external peripherals via high-speed serial communication. USB 1.1 operated only up to 12 Mbps, and USB 2.0 was introduced in 2000 to compete with IEEE 1394, another high-speed serial bus standard. USB 2.0 supports Low Speed (1.5 Mbps), Full Speed (12 Mbps) and High Speed (480 Mbps) modes, as well as Host and On-the-Go (OTG) functionality. Whereas a USB 2.0 Host can master up to 127 peripheral connections simultaneously, OTG is meant for a peer-to-peer host/device capability, where the interface can act as an ad hoc host to a single peripheral connected to it. Thus, OTG is well-suited to embedded applications where a PC isn’t needed. Importantly, USB supports Plug-and-Play (automatic configuration of a plugged-in device), as well as hot pluggability (the ability to plug in a device without first powering down). Moreover, it allows for bus-powering of a plugged-in device from the USB interface itself.
**Chapter 1**

**PCI—Peripheral Component Interconnect**—This is a local bus standard developed by Intel Corporation and used initially in personal computers. Many media processors use PCI as a general-purpose “system bus” interface to bridge to several different types of devices via external chips (e.g., PCI to hard drive, PCI to 802.11, etc.). PCI can offer the extra benefit of providing a separate internal bus that allows the PCI bus master to send or retrieve data from an embedded processor’s memory without loading down the processor core or peripheral interfaces.

**Network Interface**—In wired applications, Ethernet (IEEE 802.3) is the most popular physical layer for networking over a LAN (via TCP/IP, UDP and the like), whereas IEEE 802.11a/b/g is emerging as the prime choice for wireless LANs. Many Ethernet solutions are available either on-chip or bridged through another peripheral (like asynchronous memory or USB).

**IEEE 1394 ("Firewire")**—IEEE 1394, better known by its Apple Computer trademark “Firewire,” is a high-speed serial bus standard that can connect with up to 63 devices at once. 1394a supports speeds up to 400 Mbps, and 1394b extends to 800 Mbps. Like USB, IEEE 1394 features hot pluggability and Plug-and-Play capabilities, as well as bus-powering of plugged-in devices.

**Data Movement**

**Synchronous Serial Audio/Data Port**—Sometimes called a “SPORT,” this interface can attain full-duplex data transfer rates above 65 Mbps. The interface itself includes a data line (receive or transmit), clock, and frame sync. A SPORT usually supports many configurations of frame synchronization and clocking (for instance, “receive mode with internally generated frame sync and externally supplied clock”). Because of its high operating speeds, the SPORT is quite suitable for DSP applications like connecting to high-resolution audio codecs. It also features a multichannel mode that allows data transfer over several time-division-multiplexed channels, providing a very useful mode for high-performance telecom interfaces. Moreover, the SPORT easily supports transfer of compressed video streams, and it can serve as a convenient high bandwidth control channel between processors.

**Parallel Video/Data Port**—This is a parallel port available on some high-performance processors. Although implementations differ, this port can, for example, gluelessly transmit and receive video streams, as well as act as a general-purpose 8- to 16-bit I/O port for high-speed analog-to-digital (A/D) and digital-to-analog (D/A) converters.
Moreover, it can act as a video display interface, connecting to video encoder chips or LCD displays. On the Blackfin processor, this port is known as the “Parallel Peripheral Interface,” or “PPI.”

**Memory Subsystem**

As important as it is to get data into (or send it out from) the processor, even more important is the structure of the memory subsystem that handles the data during processing. It’s essential that the processor core can access data in memory at rates fast enough to meet the demands of the application. Unfortunately, there’s a tradeoff between memory access speed and physical size of the memory array.

Because of this, memory systems are often structured with multiple tiers that balance size and performance. Level 1 (L1) memory is closest to the core processor and executes instructions at the full core-clock rate. L1 memory is often split between Instruction and Data segments for efficient utilization of memory bus bandwidth. This memory is usually configurable as either SRAM or cache. Additional on-chip L2 memory and off-chip L3 memory provide additional storage (code and data)—with increasing latency as the memory gets further from the processor core.

In multimedia applications, on-chip memory is normally insufficient for storing entire video frames, although this would be the ideal choice for efficient processing. Therefore, the system must rely on L3 memory to support relatively fast access to large buffers. As we’ll see in Chapter 7, the processor interface to off-chip memory constitutes a major factor in designing efficient media frameworks, because L3 access patterns must be planned to optimize data throughput.

**Laying the Groundwork for an EMP Application**

OK, so you’re starting a brand new application, and your boss mumbled something about “…streaming media gadget that’s better than our competition’s.” Not surprisingly, selecting a processor for multimedia applications is a complex endeavor. It involves a thorough analysis of the processor's core architecture and peripheral set, a solid understanding of how video and audio data flows through the system, and an appreciation for what level of processing is attainable at an acceptable level of power dissipation.
Chapter 1

What Kind(s) of Media Am I Dealing With?

This is a central question, because it drives many other decisions. Here, you need to answer questions about data bandwidth, whether the data pattern is bursty or sustained, and the formats or protocols you’ll need to support.

What Do I Need to Do With the Data?

This question relates to the processor performance. Among the first measures that system designers should analyze when evaluating a processor for its performance are the number of instructions performed each second, the number of operations accomplished in each processor clock cycle and the efficiency of the computation units.

As processing demands outpace technological advances in processor core evolution, there comes a point at which a single processor will not suffice for certain applications. This is one reason to consider using a dual-core processor. Adding another processor core not only effectively doubles the computational load capability of the processor, but also has some surprising structural benefits that aren’t immediately obvious. We’ll consider these in Chapter 7.

The merits of each of the aforementioned metrics can be determined by running a representative set of benchmarks on the processors under evaluation. The results will indicate whether the real-time processing requirements exceed the processor’s capabilities and, equally as important, whether there will be sufficient capacity available to handle new or evolving system requirements.

Are My System Needs Likely to Change Over Time, or Will This Be a Static System?

If changes will likely be necessary to accommodate new media formats, user interface features, or the like, then a programmable solution (e.g., DSP, MCU, FPGA, Convergent Processor) will probably be required. If system requirements are firm and unlikely to change, a fixed-function ASIC might be suitable instead.

Is This a Portable Application?

Battery-powered systems dictate a whole new set of application requirements. They necessitate a power-efficient processing solution in a compact form factor. Often, this restriction involves a tradeoff between processing performance and power efficiency, and this is a realm where processors not explicitly suited for multimedia applications
Embedded Media Processing

will tend to fall short, since they’ll burn more power in accomplishing tasks for which they’re not optimized.

Choosing the proper battery system for the application is also key—for energy density considerations, as well as for the weight of the end product, time between recharges, and even safety concerns.

**Does my Application Require a Fixed-Point or Floating-Point Device?**

Processor computation arithmetic is divided into two broad categories: fixed-point and floating-point. In general, the cutting-edge fixed-point families tend to be faster, more power-conscious and cost-sensitive, while floating-point processors offer high precision at a wide dynamic range. *Dynamic range* refers to the ratio between the largest and smallest numbers that can be represented in a numeric format, whereas *precision* refers to the granularity with which a fraction can be defined.

As illustrated in Figure 1.3, designers whose applications require only a small amount of floating-point functionality are caught in a “gray zone,” often forced to move to higher-cost floating-point devices. Today, however, some fixed-point processors

---

Figure 1.3  *The choice between a fixed-point and floating-point processor isn’t always clear*
are running at such high clock speeds that, given the right architecture, it becomes possible to emulate floating-point operations. This approach allows the designer to trade off floating-point computational efficiency for low cost and low-power operation. Obviously, this approach will not work for a truly float-intensive application, but it presents an appealing opportunity for designers “stuck in the gray zone.” We’ll talk much more about fixed-point versus floating-point in Chapter 5, because audio is a typical “gray zone” application that can often be implemented in either format.

**How Does the Data Get Into and/or Out of the Chip?**

The right peripheral mix saves time and money by eliminating the need for external circuitry to support the required interfaces. The processor of choice should have a flexible, ample peripheral set. Refer to the “Peripheral Interfaces” discussion earlier in this chapter.

**How Do I Develop on the Processor?**

This is a very important question that depends as much on the needs of the individual developers as it does on the final application. Many companies have built an experience base with certain development tools packages, and they may have constructed a large code repository that they would like to leverage. Also, a strong development tools suite can dramatically increase productivity and decrease development and debug time. For instance, a powerful C/C++ compiler can facilitate easy porting of existing source code onto a new platform, and multiple optimization choices (for example, optimize for speed, power, or code density) can help designers achieve their development goals without having to invest heavily in understanding the finer points of a processor’s instruction set and architecture. Figure 1.4 shows a screenshot of the VisualDSP++ development environment available for debug and development on Blackfin processors.

**Do I Need an Operating System?**

Operating systems perform a wide variety of tasks, from recognizing and processing high-level I/O, to tracking and managing files, to controlling multiple simultaneous processes and ensuring that they don’t interfere with one another. As with development tools, many developers have strong preferences with regard to operating system (OS) choices. If a company has a breadth of applications built on top of a particular OS, there’s likely a strong preference to choose their next processor based partly on that OS’s availability on the new platform. Often, an application will require an OS
that’s designed for a particular type of environment. For instance, some operating systems guarantee low latency (for time-critical applications), control multiple simultaneously running applications (multitasking), allow multiple users concurrent access to the system (multiuser), and/or allow several instances of an application to run at the same time (multithreading). Of course, many applications don’t require an OS at all, so this issue is not always a chief concern.

**What Are the Different Ways to Benchmark a Processor?**

There are many different kinds of benchmarks that can be run for comparison. The problem is, it’s very hard to find uniformity among all processors in a space, as far as the ways that vendors measure performance. Couple that with the fact that there are myriad ways to perform the same task on a single processor, and you can see how
hard it can be to get a true objective measure of performance differences between candidate devices. That’s why a good approach is to evaluate performance from several different angles:

- **Independent organizations.** These companies or consortiums attempt to create objective benchmarks for processor comparisons for specific groups of tasks. For instance, Berkeley Design Technologies, Inc. (BDTI) has a suite of signal processing kernels that are widely used for comparing DSP performance of processors. Likewise, EEMBC, the Embedded Microprocessor Benchmark Consortium, measures the capabilities of embedded processors according to several application-specific algorithms.

- **Vendor collateral.** Vendor-supplied data sheets, application notes and code examples might be the most obvious way to obtain comparative information. Unfortunately, however, you’ll be hard pressed to find two vendors who run identical tests, in large part because each wants to make their processor stand out against the competition. Therefore, you might find that they’ve taken shortcuts in algorithm kernels or made somewhat unrealistic assumptions in power measurements. On the other hand, some vendors go to great lengths to explain how measurements were taken and how to extrapolate that data for your own set of conditions.

- **Bench testing.** If you want something done right, do it yourself! There are certain basic performance measurements that you can make across processor platforms that will give you a good idea of data flow limitations, memory access latencies, and processor bottlenecks. We’ll discuss this further in Chapter 7.

Also, keep in mind that benchmarks don’t always tell the whole story. Sometimes, slightly tweaking an algorithm to eliminate potentially unnecessary restrictions can make a huge difference in performance. For instance, true IEEE 854 floating-point emulation can be very expensive in terms of fixed-point computing power. However, relaxing a few constraints (such as representation of special-case numbers) can improve the floating-point emulation benchmarks considerably, as we’ll see in Chapter 5.

**How Much Am I Willing to Spend?**

This is an important question, but you’re likely to arrive at a misleading answer unless you consider the total system design cost. That is, Processor A might cost more than
Processor B at face value, but it could offer more peripheral connectivity (whereas B requires add-on interface chips), come in a package allowing two-layer board design (contrasted with B’s package, which mandates expensive four or six-layer boards in order to route effectively), and have extra processing power that gives more headroom for future expandability.

**OK, So What Processor Choices Do I Have?**

Glad you asked. But there’s no short answer…

There are lots of processor options to consider for embedded media applications. Each has its own pros and cons, and device selection is a crucial milestone in designing the system. In other words, you should not just make a “default” decision to use whatever processor you’ve used in the past. Although reflexively using a “familiar” part can speed development time, this approach can also lead to design limitations that are often unforeseen until it’s too late in the process to address them appropriately. Let’s review the basic device choices available:

- Application-Specific Integrated Circuit (ASIC)
- Application-Specific Standard Product (ASSP)
- Field-Programmable Gate Array (FPGA)
- Microcontroller (MCU)
- Digital Signal Processor (DSP)
- DSP/MCU Combinations (discrete or unified)

**ASIC and ASSP**

ASICs are “hardwired” to fit an exact application space. Because of this, they can be optimized for both power and performance as part of the design process. The ASIC usually evolves from a prototype solution. After market requirements stabilize, an ASIC can be developed to reduce the cost of the end application for high unit volumes. Thus, for markets based on stable technologies and requirements, the ASIC is often a good choice.

A cousin of the ASIC is the ASSP. The ASSP is dedicated to a specific application market but sold to many customers within that market, whereas an ASIC is usually designed and sold to a specific company. As a result, support collateral for ASSPs is usually much more extensive than with ASICs and, because of this, ASSPs are
often preferred over ASICs for companies with little in-house ASIC design expertise. ASICs and ASSPs are about equal in size and cost for a given application.

While a fixed-function ASIC/ASSP might very well embody an optimal solution for a specific EMP application—including the exact peripheral set necessary to succeed in the end market—it’s limited flexibility hinders its ability to accommodate new features or evolving requirements. This lack of programmability, in turn, limits a consumer’s options in the breadth of media formats available for recording or playback, for example.

Given today’s somewhat volatile mix of old, new and emerging media formats and standards, the decision to choose an ASIC at the outset is not so clear-cut. Evolving standards do not lend themselves to ASIC implementations. Neither do complicated algorithms requiring very high processor clock speeds. After all, software mistakes on a programmable processor can be rectified with just a “recompile,” but ASIC mistakes may mean a new chip spin at very high cost.

To keep pace with evolving performance, power and size requirements, there is a natural push towards developing devices in smaller silicon process geometries. However, as these geometries continue to shrink, ASIC development costs grow exponentially. Approaching 90-nanometer feature sizes, for instance, the IC mask set alone can cost around $1 million. This hefty development price tag, coupled with long design lead times, has given pause to companies pursuing an ASIC development path, as it makes it hard to keep pace in custom hardware design. In fact, this is a prime reason why EMP system developers are looking toward more programmable solutions.

**FPGA (Field-Programmable Gate Array)**

FPGAs help speed product development (thus shortening time to market), as there is a huge base of standard library cells available for signal and control processing applications. Because they basically comprise a string of optimized hardware blocks for a given collection of functions, they can achieve performance exceeding that of programmable processors. For instance, it’s common on FPGAs to do massively parallel computation that’s limited in scope, but not achievable on a standard microcontroller or DSP.

However, FPGAs are typically large, expensive, and power-hungry, when compared to programmable processing solutions and ASICs. Thus, although ideal for prototyping EMP designs, they aren’t well suited as the main processor for portable multimedia
applications. Also, while they afford considerable reconfigurability in implementing and modifying application blocks, FPGAs run up against a performance ceiling that is dictated by the number of gates in the package.

That said, it’s hard to beat an FPGA for flexibility in interfacing a system to the outside world. Most popular peripherals and industry-standard interfaces are available as standard blocks from FPGA vendors, and the configurable nature of FPGA logic serves as a “Get Out of Jail Free card” for adapting timing and logic nuances to allow the media processor to talk nicely with any external device. Finally, an additional advantage of FPGAs is that it’s straightforward to create an ASIC based on an FPGA implementation.

**MCU**

The typical MCU acts as a system controller. It excels in scenarios that involve many conditional operations, with frequent changes in program flow. MCU code is usually written in C or C++, and code density is paramount—algorithms are measured in terms of compiled code size. Memory systems are cache-based—automatically managing the flow of instructions into the core for execution—and the system usually runs from large memories with higher latencies than L1 memory.

Microcontrollers run the gamut in performance from 4-bit, 32 kHz models up beyond 32-bit, 500 MHz devices. However, the need to connect to media sources and targets often rules out any MCUs that aren’t 32-bit capable.

To understand why the 32-bit MCU is growing so popular in the multimedia arena, first consider that the 8-bit MCU does not possess the bandwidth and computational power required to guarantee real-time operation in these systems. This is not to say that 8-bit MCUs are not useful. On the contrary, they are still very popular across a wide array of markets, from cars to cameras. Their chief selling points are very low power consumption, tiny package, excellent code density, and dirt-cheap pricing ($1.00–$2.00). Many 8-bit MCUs today have added integrated flash memory, as well as Ethernet connectivity, making them very attractive for a wide range of small, focused tasks. Moreover, they often act as a companion to a 32-bit processor in more complex systems.

What about 16-bit microcontrollers? After all, these devices do offer an increase in throughput, performance, and integration. In addition, they consume low standby current and typically offer a larger on-chip memory than their 8-bit competitors.
As a natural countermeasure to the competition presented by the 32-bit MCU, 16-bit MCU vendors have tried to expand their internal bus sizes to allow faster operation without increasing the final cost of the solution. While this approach helps somewhat, the basic gap in performance still exists, because a fundamental problem for 8/16-bit MCUs, even those running at “fast” speeds, is that they still have limited processing capability and a small memory addressing range. For instance, some 8-bit MCUs have only a few data registers, with no separate accumulator. The presence of complete register files is very important because it eliminates the requirement to frequently transfer data between memory and the accumulator.

While 16-bit MCUs (starting around $2) might cost only about twice as much as 8-bit MCUs, it’s difficult to leverage legacy 8-bit design collateral. 16-bit devices usually require a completely new code development effort, as well as a new software tools suite. Given a transition from 8-bit MCUs, many developers would rather skip over the 16-bit path and jump straight to 32-bit devices (starting around $5), since they have to embark on a new learning curve and incur development cost anyway.

Viewed in this light, the actual cost of implementing a 32-bit MCU-based system is on par with that of many 16-bit solutions available today, while the performance advantage over the 16-bit MCU is significant. While the cost differential between 32-bit and 16-bit devices can be $3 or more, this gap is narrowed through increased peripheral integration and the built-in flexibility associated with much more computational headroom. Let’s explore further some reasons why 32-bit devices are increasing in popularity.

As Figure 1.5 indicates, many ubiquitous peripherals (high-speed USB 2.0, PCI, etc.) support such high data rates that it becomes unwieldy or impossible to handle these data streams on 8- and 16-bit MCUs. For example, in the area of network connectivity, a maximum data register size of 8 bits, or even 16 bits, hampers the ability to support a full set of network protocols. Additionally, with multiple 32-bit data address generation registers, great increases in performance can be achieved. These 32-bit MCU features result in denser compiled code, higher continuous bandwidth, and a more flexible programming model.

This point merits more explanation. With more bits in the processor’s data path and a 32-bit data and address register file, compiler support is greatly enhanced. In addition, a 32-bit device often supports both 16- and 32-bit opcodes. This allows more
flexibility in the instruction set architecture, which yields improvements in code density and allows many operations to be completed in a single processor clock cycle.

These features, in turn, reduce the dependency on hand-crafted assembly code, so that developers can program primarily in a high-level language like C. A C-based programming model translates directly into lower development and maintenance costs. It also allows a company’s legacy application code to transition into the 32-bit MCU environment in a much smoother manner.

(MIPS = Millions of Instructions Per Second)

Figure 1.5 **Broad range of MCUs available**
In an EMP application, MCUs can play one of many roles. For lower-end systems, they can serve as the sole system processor, mainly doing video decode and display for modest display resolutions (often QVGA or less) and video formats (MPEG-4 Simple Profile, for example, but not H.264). For mid- to high-end systems, a DSP or an MCU with specialized accelerators is needed to perform the massive high-speed computations necessary for supporting higher resolutions and more complex formats. However, the MCU still can serve as system controller in these applications, running an operating system and managing tasks throughout the system. Furthermore, because of their rich peripheral mix, they are often instrumental in bridging between the media processor and the outside world.

**DSP**

Digital signal processors emerged to address the performance gap that became evident in trying to apply MCUs to compute-intensive applications. DSPs have specialized architectural constructs specifically optimized to run in tight, efficient loops, performing as many multiply-accumulate (MAC) operations as possible in a single clock cycle. A DSP is ideal for high-performance number-crunching on data buffers such as those prevalent in multimedia applications. Achieving DSP performance goals usually requires writing optimized assembly code. Because DSP algorithms can typically fit in small, low-latency on-chip memory, code density is not generally of great concern.

DSPs can run at very high clock rates, and they are often compared in terms of how many millions of MAC operations they can perform per second. This metric is known as “mega MACS,” or MMACS. The MAC is a basic DSP operation used in digital filters, fast Fourier transforms (FFTs), and the like. Many DSPs have more than one MAC unit, and this provides them with increased signal processing capability. Figure 1.6 shows the recent strides made in signal processing performance, such that the cost per MMAC has plummeted, and the quantity of MMACS achievable on a single processor has skyrocketed.
In the world of multimedia processing, a DSP is not an ideal standalone processor. It is too focused on its math routines to “run the whole show,” so it relies on the slower, yet more “managerial,” microcontroller (MCU) to provide asynchronous system control functionality like a user interface and an operating system.

A few different design paths have emerged that incorporate an MCU and DSP into a system design solution. For starters, you can always use a separate DSP chip and MCU chip in the design. While this might be a little pricey and take up a fair amount of real estate, it allows maximum flexibility to size each chip appropriately to the system’s needs. Alternatively, some chip manufacturers take a heterogeneous multicore approach and couple separate DSP and MCU processors in a single package.

A limitation of these approaches, however, is that the designer must partition to a fixed share of control and DSP functions; once the DSP is “maxed out,” for instance, the MCU will be unable to take up the computational slack. Consequently, the final product might be highly integrated, yet inadequately structured to handle the latest media requirements. Another complexity this introduces is that separate DSP and MCU cores require separate sets of development tools, thus complicating the design environment.

Figure 1.6 The sharply declining costs of embedded processing performance

Discrete DSP + MCU

Source: Analog Devices, Inc.
1 MMACS = 1 Million Multiply-Accumulate operations/Second, a measure of processor performance
A trend emerging in parallel with the growth of faster 32-bit MCUs is the drive to incorporate more DSP functionality, such as instruction set extensions and MAC units, into the microcontroller device. This philosophy is only appropriate for straightforward signal processing applications, however, because MCU clock speeds and computation architectures are fundamentally not well suited for media-rate number crunching, and therefore these MCU+DSP add-on approaches lack the essential architectural basis required to serve as platforms for advanced media processing applications.

**Convergent Processor**

A processor design ideal for EMP applications combines an MCU and a DSP into a single device with a unified architecture optimized not only for computation on real-time multimedia data flows, but also for control-oriented tasks. Other key features of this *Convergent Processor* are high clock rate, low power dissipation per unit of processing (mW/MMACS), small form factor (high volumetric efficiency per unit of processing), and flexible programming model.

It is important to understand that this processor is not a DSP with an enhanced instruction set, nor is it a microcontroller with DSP extensions. Instead, it is an equally high-performance media processor and compiler-friendly processor to which both classes of developers can relate.

A Convergent Processor functions simultaneously as a 16-bit DSP and a 32-bit MCU, with the ability to devote all of its resources to control tasks, to computation tasks, or to some division between the two, depending on the real-time needs of the system. It replaces a two-chip solution, saving power and board space, as well as reducing system cost and complexity. For the consumer, this means smaller, cheaper multimedia gadgets with longer battery lives.

In today’s design paradigm, MCU and DSP programmers often inhabit two totally separate groups, interacting only at the “system boundary” level where their two functional worlds meet. This makes some sense, as the two groups of developers have evolved their own sets of design practices. For instance, the signal processing developer may relish the nitty-gritty details of the processor architecture and thrive on implementing tips and tricks to improve performance. On the other hand, the MCU programmer might well prefer a model of just turning on the device and letting it do the work.
The Convergent Processor satisfies both classes of engineers by supporting both direct memory access (DMA) and cache memory controllers for moving data through the system. Multiple high-speed DMA channels shuttle data between peripherals and memory systems, allowing the fine tuning controls sought by DSP programmers. Conversely, on-chip configurable instruction and data caches allow a hands-off approach of managing code and data in a manner very familiar to MCU programmers. Often, at the system integration level a combination of both approaches is ideal.

Another reason for the historical separation of MCU and DSP development groups is that the two processors have two separate sets of design imperatives. From a technical standpoint, engineers responsible for architecting a system are sometimes hesitant to mix a “control” application with signal processing on the same device. Their most common fear is that non-real-time tasks will interfere with time-critical tasks. For instance, the programmer who handles functions such as the graphical user interface (GUI) or the networking stack shouldn’t have to worry about hampering the real-time signal processing activities of the system. Of course the definition of “real-time” will vary based on the specific application. In an embedded application, the focus is on the time required to service an interrupt.

While the MCU control code is usually written in C and is library-based, the real-time DSP code is typically written in assembly format and is handcrafted to extract the most possible performance for a given application. Unfortunately, this optimization also limits the portability of an application and therefore propagates the need for divergent skill sets and tools suites between the two programming teams on future projects.

The Convergent Processor approach allows developers to create a C/C++ unified programming code base, leveraging existing application code from previous efforts. Because the processor is optimized for both control and signal-processing operations, compilers can generate code that is both “tight” (from a code density standpoint) and efficient (for computationally intensive signal processing applications). Any gap in compiler performance is closed by the high operating frequencies of these parts. Additionally, targeted assembly coding is still an option for optimizing critical processing loops.
The Convergent Processor allows utilization of a single tool chain for code development on a single, united platform. Thus, developers can learn one instruction set and maintain a single code base running on a single operating system. There’s one set of software application programming interfaces (APIs) and drivers, one debugger, one loader, one linker, one language, and so forth. A single learning curve means dramatic improvement in development productivity.

While this unified approach can greatly reduce the requirement of writing code in assembly, this fact alone doesn’t necessarily justify the switch to this unified platform. Operating system (OS) support is also key, because this allows software task layering. To achieve targeted performance, a prioritizable event/interrupt controller is crucial, with context switching through hardware-based stack and frame pointer support. With these architectural constructs, the Convergent Processor allows developers to create applications that include both worlds—system control and real-time signal processing—on the same device.

The high processing speeds that Convergent Processors can achieve translate into several tangible benefits. The first is time to market: there can be considerable savings in reducing or bypassing the code optimization effort if there’s plenty of processing headroom to spare. A second key benefit is reduced software maintenance, which can otherwise dominate a product’s lifecycle cost. Finally, for scalable device architectures, there’s the possibility of designing a system on the most capable processing family member, and then “rightsizing” the processor for the computational footprint of the final application.

A Look Inside the Blackfin Processor

If you haven’t guessed by now, the Convergent Processor will be our focus as we navigate through the complexities of embedded media processing. Although we will strive to offer generic principles and suggestions, we will use Analog Devices’ Blackfin processor as a basis for grounding our discussion in concrete examples. Therefore, let’s take a little time now to review some basic Blackfin processor architecture, in order to set the stage for our discussions throughout the rest of this book. If you’d like a more detailed treatment than what we present here, please refer to References 1, 2, and 3 in the Appendix.
System View

Figure 1.7 shows a simplified diagram of a representative Blackfin processor at a “system-on-chip” level. The processor core scales up beyond 600 MHz instruction execution rates. It connects to an on-chip Level 1 (L1) memory subsystem consisting of a mix of high-speed cache and SRAM for both instruction and data storage. Integrated peripherals allow connection to the outside world, and system control blocks aid management of processor operation. A System Interface Unit facilitates access between the core, peripherals, and external synchronous and asynchronous memory chips.

Of course, Figure 1.7 only shows a sample part in the Blackfin family. Variants include dual processor cores (each with their own L1 memory blocks), different on-chip memory configurations, and a multitude of peripheral block combinations.

**ADSP-BF533/532/531 Blackfin**

Figure 1.7  System view of representative single-core Blackfin processor
Computational Units

Figure 1.8 shows a functional diagram of a representative Blackfin processor core. At the computational heart of the core are two 16-bit multipliers and two 40-bit accumulators, which are collectively called multiply-accumulate, or MAC, units. Each MAC can perform a 16-bit by 16-bit multiply per cycle, with accumulation to a 40-bit result. Also, a 32-bit integer multiply can be performed in three cycles.

Additionally (no pun intended), there are two 40-bit arithmetic logic units (ALUs), which perform a traditional set of arithmetic and logical operations on 16-bit or 32-bit data.

Blackfin’s four 8-bit video ALUs perform useful byte-wise image and video manipulations, including byte alignment and packing, 16-bit and 8-bit adds with clipping, 8-bit averaging, and 8-bit subtract/absolute value/accumulate (SAA) operations.

Moreover, a 40-bit barrel shifter can deposit data and perform shifting, rotating, normalization, and extraction operations.

All together, these computational units process 8-, 16- or 32-bit data from the data register file, which consists of eight registers, R0 through R7. These registers can be used as any combination of sixteen 16-bit data registers or eight 32-bit registers. The upper and lower 16 bits of each register are accessible using the register name followed by “.H” for the upper 16 bits and “.L” for the lower 16 bits. For instance, “R0.L” accesses the lower 16 bits of data register 0.

For some instructions, two 16-bit ALU operations can be performed simultaneously on register pairs (a 16-bit high half and 16-bit low half of a compute register). By also using the second ALU, four 16-bit operations are possible at the same time.

A separate register file is also present for address generation. This address register file consists of pointer registers P0 through P5, in addition to a stack pointer (SP) and a frame pointer register (FP). The address registers are capable of 8-, 16- and 32-bit accesses, and they support a wide range of RISC-type addressing operations. A frame and stack can be set up anywhere in the memory space. Ideally these can be located somewhere inside the processor to improve performance, although this is not always practical, depending on required allocation size. The address arithmetic unit provides data address generation (DAG) functionality, allowing simultaneous dual fetches from memory. It contains a multi-ported register file consisting of four sets of 32-bit Index,
Modify, Length, and Base registers (for circular buffering), and eight additional 32-bit pointer registers (for C-style indexed stack manipulation).

**Memory Model**

Blackfin devices support a modified Harvard architecture in combination with a hierarchical memory structure that views memory as a single, unified 4-Gbyte address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space.
Chapter 1

Taken together, the Blackfin memory space is structured to balance cost and performance between fast, low-latency on-chip memory and larger, lower cost and lower performance off-chip memory systems. L1 on-chip memories typically operate at the full processor speed with little or no latency. At the L1 level, the instruction memory holds instructions only, the data memories hold data, and a dedicated scratchpad data memory stores stack and local variable information. Level 2 (L2) on-chip memories, when present, are usually bigger than L1 memories, but incur longer access latencies. The L2 memory is a unified instruction and data memory and can hold any mixture of code and data required by the system design. Level 3 (L3) external memory is larger and cheaper (per byte), but it has longer latencies than any on-chip memory.

DMA

The Blackfin processor uses direct memory access (DMA) to transfer data within memory spaces or between a memory space and a peripheral. The processor can specify data transfer operations and return to normal processing while the fully integrated DMA controller carries out the data transfers independent of processor activity. The DMA controller can perform data transfers between memory and on-chip peripherals (“Peripheral DMA”), as well as between and within L1/L2/L3 memory spaces (“Memory DMA” or “MemDMA”).

DMA transfers can be descriptor-based or register-based. Descriptor-based DMA transfers require a set of parameters stored within memory to initiate a DMA sequence. This sort of transfer allows the chaining together of multiple DMA sequences. In descriptor-based DMA operations, a DMA channel can be programmed to automatically set up and start another DMA transfer after the current sequence completes. Register-based DMA, on the other hand, allows the processor to directly program DMA registers to initiate a DMA transfer. Upon transfer completion, these registers can be automatically updated with their original setup values for continuous transfer, if needed.

The DMA engine is crucial to the successful operation of an EMP application. As such, we will be discussing DMA in much greater detail throughout the course of this book. We even devote an entire chapter to it—Chapter 3.

Instruction Flow

The program sequencer controls the instruction execution flow, including instruction alignment and decoding. The instruction pipeline, which holds the addresses of all
instructions currently being fetched, decoded and executed, is fully interlocked. This means that it automatically inserts stalls to guarantee correct operation when executing instructions that depend on data that’s not yet available (e.g., a result of a previous operation that hasn’t yet completed).

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. Generally, the processor executes instructions from memory in sequential order by incrementing the look-ahead address. However, when encountering one of the following structures, the processor will execute an instruction that is not at the next sequential address:

- **Loops.** One sequence of instructions executes several times with zero overhead. Dedicated loop registers are available to support nested loops without having to continually fetch the loop start and end addresses.
- **Subroutines.** The processor temporarily interrupts sequential flow to execute instructions from another part of memory.
- **Jumps.** Program flow transfers permanently to another part of memory.
- **Interrupts and Exceptions.** A runtime event or instruction triggers the execution of a subroutine.
- **Idle.** This instruction causes the processor to stop operating and hold its current state until an interrupt or wakeup occurs. Then, if appropriate, the processor services the interrupt and continues normal execution.

**Event Handler**

Like many microcontrollers, Blackfin processors allow both interrupts and exceptions. Both kinds of events cause pipelined instructions to suspend execution in favor of servicing the triggering event. An interrupt is an event that changes normal processor instruction flow and is asynchronous to program flow. In contrast, an exception is a software-initiated event whose effects are synchronous to program flow. By trapping exceptions, the end system can guard against invalid or illegal programming.

The event management system supports nesting and prioritization. Consequently, several service routines may be active at any time, and a low priority event may be pre-empted by one of higher priority. The architecture employs a two-level event control mechanism. The system interrupt controller (SIC) works with the core event controller (CEC) to prioritize and control all system interrupts. The SIC provides a
mapping between the many peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core. This mapping is programmable, and individual interrupt sources can be masked in the SIC.

We’ll talk more about event handling in Chapter 4.

**Protection of Resources**

An important aspect of system control is task management. Increasingly, a real-time operating system (RTOS) is employed to handle the wide range of ongoing, concurrent tasks in a complex system. The RTOS simplifies the programming model by providing support for task scheduling and management, and it usually runs on an MCU, since DSPs do not have all of the RTOS-friendly features needed for efficient implementation.

Blackfin’s architecture, however, facilitates RTOS development with several important features. One is the provision of separate stack and frame pointers to reduce latency of OS calls and interrupt/exception handling. Another is the ability to restrict access to protected or reserved memory locations. This allows one task, via a paging mechanism, to block memory or instruction accesses by another task. An exception is generated whenever unauthorized access is made to a protected area of memory. The kernel will service this exception and take the appropriate action.

Yet a third RTOS-friendly feature is the existence of separate Supervisor and User modes. Supervisor mode has unrestricted use of all system and core resources, whereas User mode can only access a subset of system resources, thus providing a protected software environment. These crucial protection features prevent users from unknowingly or maliciously accessing or affecting shared parts of the system.

DSPs usually operate in the equivalent of Supervisor mode, allowing full access to all system resources at all times, whereas MCUs provide the analog of a User mode that allows applications to run on top of an OS. On a Blackfin device, both operating modes coexist under one unified architecture, so that a system can restrict user applications to accessing system resources only through the OS.

**Programming Model**

The Blackfin software development model enables high-performance DSP functionality within a framework matching that of typical RISC MCU devices. System-level and product-level application code can be written in C/C++ and layered on top of multiple standard real-time operating systems. Lower-level code, such as raw data
movement and processing, can be optimized in mixed assembly and C/C++ code, utilizing hand-tuned assembly libraries. Blackfin can be just as easily programmed in assembler (using an intuitive algebraic assembly language), compiled C/C++ code, or a mixture of both.

The Blackfin instruction set is optimized so that 16-bit opcodes represent the most frequently used instructions. Complex DSP instructions are encoded into 32-bit opcodes as multifunction instructions. A 64-bit multi-issue capability allows a 32-bit instruction to execute in parallel with two 16-bit instructions. This allows the programmer to use many of the core resources in a single instruction cycle.

**Power Management**

Along with the increased performance levels of embedded media processing applications comes higher clocking rates and, thus, higher power dissipation. This is why the processor needs an intelligent way to dynamically tailor its clock frequency commensurate with the tasks it’s performing. Consider a portable media player, for instance, profiled in Figure 1.9. The processor might need to run at 600 MHz for video processing, but only need about 200 MHz performance for audio processing. Therefore, if the device is acting as an MP3 player, why should it run at 600 MHz and discharge the battery faster? Moreover, lowering clock frequency allows a processor to run at lower operating voltage, thus maximizing battery life.

![Figure 1.9 Dynamic Power Management achieves optimal power levels by allowing both voltage and frequency tuning for the task at hand](image-url)
Control of power dissipation has long been a feature of embedded controllers. However, when the system requires DSP functionality as well, the power choices have been less than ideal. If discrete MCU and DSP chips are used in power-sensitive applications, a separate switching regulator must often be provided for each one, because the core voltages of the two devices frequently differ. This results in decreased power conversion efficiency and an increased design footprint, ultimately increasing layout complexity and solution cost. Moreover, when separate MCU and DSP cores are combined on one chip, the power solution is inherently nonoptimal, because it must service the needs of two completely independent processors with different loading profiles.

In contrast, the Blackfin processor contains an integrated dynamic power management (DPM) controller. Several intrinsic power modes are available to support a range of system performance levels. Additionally, clocks to unused peripherals and L2 memory are automatically disabled. The operating frequency can be adjusted over a wide range to satisfy stratifications in DSP/MCU processing needs. Finally, the voltage can be adjusted (either externally or through an integrated switching controller) to offer exponential savings in power dissipation.

Blackfin’s clock generation unit houses the phase-locked loop (PLL) and associated control circuitry. The PLL is highly programmable, allowing the user to control the processor’s performance characteristics and power dissipation dynamically.

Figure 1.10 shows a simplified block diagram of the clock generation unit. Generally speaking, an input crystal or oscillator signal (10 to 40 MHz) is applied to the CLKin pin to generate \( f_{\text{ref}} \). A selectable 1x-to-64x multiplier then multiplies this signal to generate the VCO frequency. Then, separate dividers independently generate core-clock (CCLK) and system/peripheral-clock (SCLK) frequencies. Depending on the specific device, CCLK can reach rates exceeding 600 MHz, while SCLK can attain a maximum of 133 MHz. Control logic ensures that the system clock frequency will not exceed the core-clock frequency.

The great advantage in this approach is that CCLK and SCLK can be changed “on-the-fly,” with very little cycle overhead. Thus, designers needn’t think twice about changing clock frequencies in order to meet different performance requirements for different segments of their code. The resulting linear savings in dynamic power dissipation comes at no implementation cost, from the designer’s perspective.
Another feature of the Clock Generation Unit is that it can be bypassed to allow the CLKin signal to pass straight through to CCLK. This capability permits use of a very low frequency CCLK during inactive operation intervals, to further reduce overall power dissipation.

We’ll elaborate much more on these power management details in Chapter 8. For now, we primarily wanted to introduce you to the CCLK and SCLK functionality and terminology, because we use it extensively in the chapters to come.

What’s Next?

We hope that this chapter has given you an overview of the nature of embedded media applications, what processing choices are available for EMP applications, and how to drive toward your final device decision from a “top-down” approach, guided by answering key questions about the application requirements. Furthermore, we introduced basic architectural concepts for the Convergent Processor model on which we’ll focus in subsequent chapters. In the next chapter, we’ll start by digging into the memory subsystem of a processor in an EMP application.