

EMBEDDED TECHNOLOGY™
S E R I E S

DSP Software Development Techniques for Embedded and Real-Time Systems

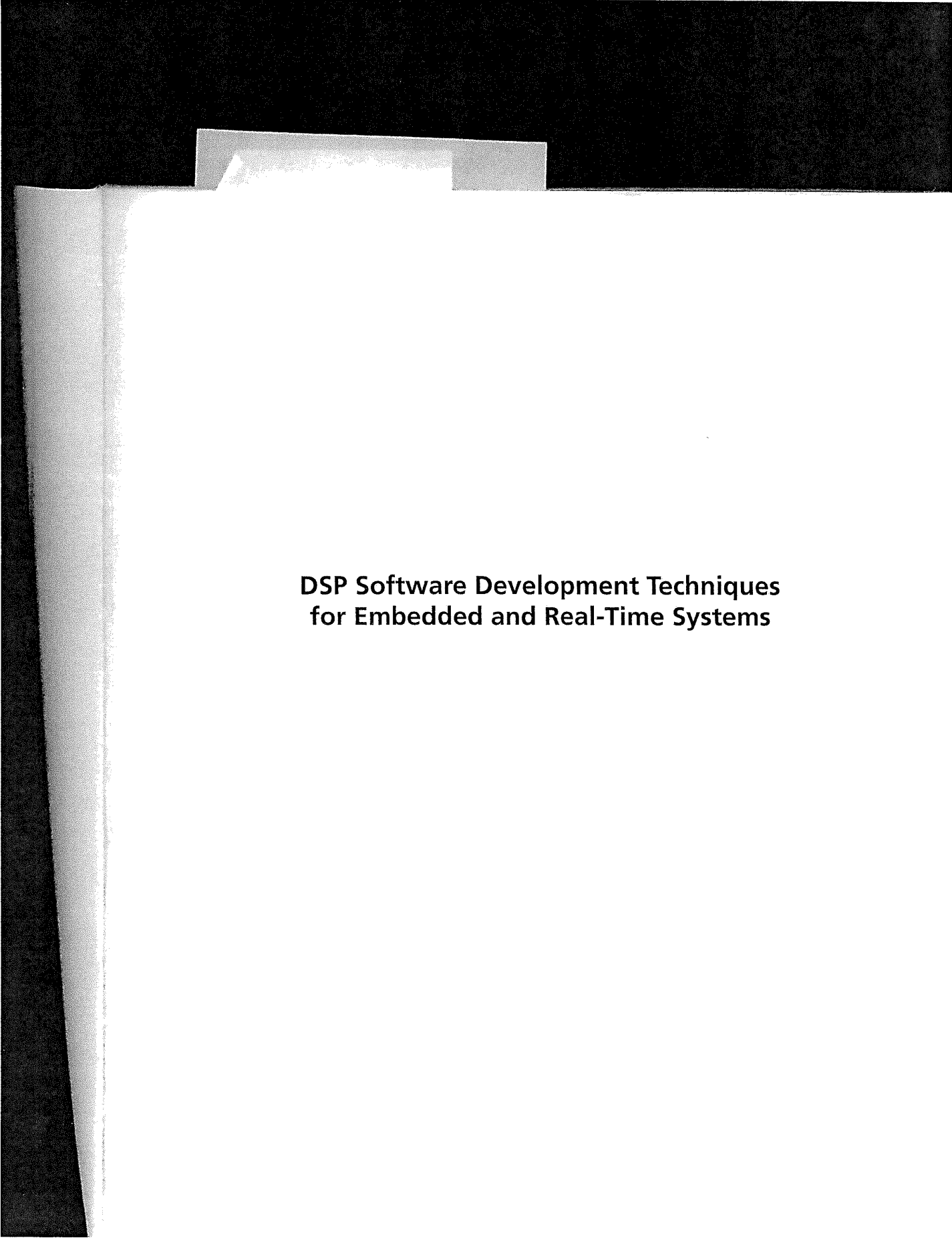
Robert Ushana



CD-ROM Contains
Code Composer Studio™ Development
Tools and Getting Started with DSP Guide



Newnes



DSP Software Development Techniques for Embedded and Real-Time Systems



DSP Software Development Techniques for Embedded and Real-Time Systems

by Robert Oshana



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier



Newnes is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Copyright © 2006, Elsevier Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com.uk. You may also complete your request online via the Elsevier homepage (<http://www.elsevier.com>), by selecting "Customer Support" and then "Obtaining Permissions."



Recognizing the importance of preserving what has been written, Elsevier prints its books on acid-free paper whenever possible.

Library of Congress Cataloging-in-Publication Data

Application submitted.

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN-13: 978-0-7506-7759-2

ISBN-10: 0-7506-7759-7

For information on all Newnes publications,
visit our website at www.books.elsevier.com.

05 06 07 08 09 10 10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER

BOOK AID
International

Sabre Foundation

*Dedicated to
Susan, Sam, and Noah*

Table of Contents

Acknowledgments	iv
Introduction: Why Use a DSP?	xi
What's on the CD-ROM?	xvii
Chapter	
1 Introduction to Digital Signal Processing	1
2 Overview of Embedded and Real-Time Systems	19
3 Overview of Embedded Systems Development Life Cycle Using DSP ..	35
4 Overview of Digital Signal Processing Algorithms	59
5 DSP Architectures	123
6 Optimizing DSP Software	159
7 Power Optimization Techniques Using DSP	229
8 Real-Time Operating Systems for DSP	260
9 Testing and Debugging DSP Systems	321
10 Managing the DSP Software Development Effort	351
11 Embedded DSP Software Design Using Multicore System on a Chip (SoC) Architectures	389
12 The Future of DSP Software Technology	411
Appendixes	
A Software Performance Engineering of an Embedded DSP System Application	419
B More Tips and Tricks for DSP Optimization	433
C Cache Optimization in DSP and Embedded Systems	479
D Specifying Behavior of Embedded DSP Systems	507
E Analysis Techniques for Real-time DSP Systems	525
F DSP Algorithmic Development—Rules and Guidelines	539
About the Author	569
Index	571

Introduction: Why Use a DSP?

In order to understand the usefulness of programmable Digital Signal Processing, I will first draw an analogy and then explain the special environments where DSPs are used.

A DSP is really just a special form of microprocessor. It has all of the same basic characteristics and components; a CPU, memory, instruction set, buses, etc. The primary difference is that each of these components is customized slightly to perform certain operations more efficiently. We'll talk about specifics in a moment, but in general, a DSP has hardware and instruction sets that are optimized for high-speed numeric processing applications and rapid, real-time processing of analog signals from the environment. The CPU is slightly customized, as is the memory, instruction sets, buses, and so forth.

I like to draw an analogy to society. We, as humans, are all processors (cognitive processors) but each of us is specialized to do certain things well; engineering, nursing, finance, and so forth. We are trained and educated in certain fields (specialized) so that we can perform certain jobs efficiently. When we are specialized to do a certain set of tasks, we expend less energy doing those tasks. It is not much different for microprocessors. There are hundreds to choose from and each class of microprocessor is specialized to perform well in certain areas. A DSP is a specialized processor that does signal processing very efficiently. And, like our specialty in life, because a DSP specializes in signal processing, it expends less energy getting the job done. DSPs, therefore, consume less time, energy and power than a general-purpose microprocessor when carrying out signal processing tasks.

When you specialize a processor, it is important to specialize those areas that are commonly and frequently put to use. It doesn't make sense to make something efficient at doing things that are hardly ever needed! Specialize those areas that result in the biggest bang for the buck!

But before I go much further, I need to give a quick summary of what a processor must do to be considered a digital signal processor. It must do two things very well. First, it must be good at math and be able to do millions (actually billions) of multiplies and adds per second. This is necessary to implement the algorithms used in digital signal processing.

The second thing it must do well is to guarantee real time. Let's go back to our real-life example. I took my kids to a movie recently and when we arrived, we had to wait in line to purchase our tickets. In effect, we were put into a queue for processing, standing in line behind other moviegoers. If the line stays the same length and doesn't continue to get longer and longer, then the queue is real-time in the sense that the same number of customers are being processed as there are joining the queue. This queue of people may get shorter or grow a bit longer but does not grow in an unbounded way. If you recall the evacuation from Houston as Hurricane Rita approached, that was a queue that was growing in an unbounded way! This queue was definitely not real time and it grew in an unbounded way, and the system (the evacuation system) was considered a failure. Real-time systems that cannot perform in real time are failures.

If the queue is really big (meaning, if the line I am standing in at the movies is really long) but not growing, the system may still not work. If it takes me 50 minutes to move to the front of the line to buy my ticket, I will probably be really frustrated, or leave altogether before buying my ticket to the movies (my kids will definitely consider this a failure). Real-time systems also need to be careful of large queues that can cause the system to fail. Real-time systems can process information (queues) in one of two ways: either one data element at a time, or by buffering information and then processing the "queue." The queue length cannot be too long or the system will have significant latency and not be considered real time.

If real time is violated, the system breaks and must be restarted. To further the discussion, there are two aspects to real time. The first is the concept that for every sample period, one input piece of data must be captured, and one output piece of data must be sent out. The second concept is latency. Latency means that the delay from the signal being input into the system and then being output from the system must be preserved as immediate.

Keep in mind the following when thinking of real-time systems: producing the correct answer too late is wrong! If I am given the right movie ticket and charged the correct amount of money after waiting in line, but the movie has already started, then the system is still broke (unless I arrived late to the movie to begin with). Now go back to our discussion.

So what are the "special" things that a DSP can perform? Well, like the name says, DSPs do signal processing very well. What does "signal processing" mean? Really, it's a set of algorithms for processing signals in the digital domain. There are analog equivalents to these algorithms, but processing them digitally has been proven to be more efficient. This has been a trend for many many years. Signal processing algorithms are the basic building blocks for many applications in the world; from cell phones to MP3 players, digital still cameras, and so on. A summary of these algorithms is shown in the following table.

Algorithm	Equation
Finite Impulse Response Filter	$y(n) = \sum_{k=0}^M a_k x(n-k)$
Infinite Impulse Response Filter	$y(n) = \sum_{k=0}^M a_k x(n-k) + \sum_{k=1}^N b_k y(n-k)$
Convolution	$y(n) = \sum_{k=0}^N x(k)h(n-k)$
Discrete Fourier Transform	$X(k) = \sum_{n=0}^{N-1} x(n) \exp[-j(2\pi/N)nk]$
Discrete Cosine Transform	$F(u) = \sum_{x=0}^{N-1} c(u) \cdot f(x) \cdot \cos\left[\frac{\pi}{2N}u(2x+1)\right]$

One or more of these algorithms are used in almost every signal processing application. Finite Impulse Response Filters and Infinite Impulse Response Filters are used to remove unwanted noise from signals being processed, convolution algorithms are used for looking for similarities in signals, discrete Fourier transforms are used for representing signals in formats that are easier to process, and discrete cosine transforms are used in image processing applications. We'll discuss the details of some of these algorithms later, but there are some things to notice about this entire list of algorithms. First, they all have a summing operation, the function. In the computer world, this is equivalent to an accumulation of a large number of elements which is implemented using a "for" loop. DSPs are designed to have large accumulators because of this characteristic. They are specialized in this way. DSPs also have special hardware to perform the "for" loop operation so that the programmer does not have to implement this in software, which would be much slower.

The algorithms above also have multiplications of two different operands. Logically, if we were to speed up this operation, we would design a processor to accommodate the multiplication and accumulation of two operands like this very quickly. In fact, this is what has been done with DSPs. They are designed to support the multiplication and accumulation of data sets like this very quickly; for most processors, in just one cycle. Since these algorithms are very common in most DSP applications, tremendous execution savings can be obtained by exploiting these processor optimizations.

There are also inherent structures in DSP algorithms that allow them to be separated and operated on in parallel. Just as in real life, if I can do more things in parallel, I can get more done in the same amount of time. As it turns out, signal processing algorithms have this characteristic as well. So, we can take advantage of this by putting multiple orthogonal (nondependent) execution units in our DSPs and exploit this parallelism when implementing these algorithms.

DSPs must also add some reality to the mix of these algorithms shown above. Take the IIR filter described above. You may be able to tell just by looking at this algorithm that there is a feedback component that essentially feeds back previous outputs into the calculation of the current output. Whenever you deal with feedback, there is always an inherent stability issue. IIR filters can become unstable just like other feedback systems. Careless implementation of feedback systems like the IIR filter can cause the output to oscillate instead of asymptotically decaying to zero (the preferred approach). This problem is compounded in the digital world where we must deal with finite word lengths, a key limitation in all digital systems. We can alleviate this using saturation checks in software or use a specialized instruction to do this for us. DSPs, because of the nature of signal processing algorithms, use specialized saturation underflow/overflow instructions to deal with these conditions efficiently.

There is more I can say about this, but you get the point. Specialization is really all it's about with DSPs; these devices are specifically designed to do signal processing really well. DSPs may not be as good as other processors when dealing with nonsignal processing centric algorithms (that's fine; I'm not any good at medicine either). So it's important to understand your application and pick the right processor.

With all of the special instructions, parallel execution units and so on designed to optimize signal processing algorithms, there is not much room left to perform other types of general-purpose optimizations. General-purpose processors contain optimization logic such as branch prediction and speculative execution, which provide performance improvements in other types of applications. But some of these optimizations don't work as well for signal processing applications. For example, branch prediction works really well when there are a lot of branches in the application. But DSP algorithms do not have a lot of branches. Much signal processing code consists of well defined functions that execute off a single stimulus, not complicated state machines requiring a lot of branch logic.

Digital signal processing also requires optimization of the software. Even with the fancy hardware optimizations in a DSP, there is still some heavy-duty tools support required—specifically, the compiler—that makes it all happen. The compiler is a nice tool for taking a language like C and mapping the resultant object code onto this specialized microprocessor. Optimizing compilers perform a very complex and difficult task of producing code that fully “entitles” the DSP hardware platform. We'll talk a lot about optimizing compilers later on in the book.

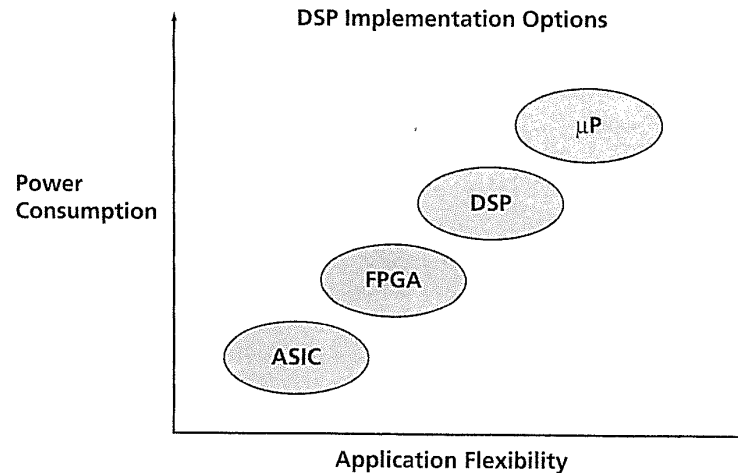
There is no black magic in DSPs. As a matter of fact, over the last couple of years, the tools used to produce code for these processors have advanced to the point where you can write much of the code for a DSP in a high level language like C or C++ and let the compiler map and optimize the code for you. Certainly, there will always be special things you can do, and certain hints you need to give the compiler to produce the optimal code, but it's really no different from other processors. As a matter of fact, we'll spend a couple of chapters talking about how to optimize DSP code to achieve optimal performance, memory, and power.

The environment in which a DSP operates is important as well; not just the types of algorithms running on the DSP. Many (but not all) DSP applications are required to interact with the real world. This is a world that has a lot of stuff going on; voices, light, temperature, motion, and more. DSPs, like other embedded processors, have to *react* in certain ways within this real world. Systems like this are actually referred to as *reactive* systems. When a system is reactive, it needs to respond and control the *real* world, not too surprisingly, in *real-time*. Data and signals coming in from the real world must be processed in a timely way. The definition of timely varies from application to application, but it requires us to keep up with what is going on in the environment.

Because of this timeliness requirement, DSPs, as well as other processors, must be designed to respond to real-world events quickly, get data in and out quickly, and process the data quickly. We have already addressed the processing part of this. But believe it or not, the bottleneck in many real-time applications is not getting the data processed, but getting the data in and out of the processor quickly enough. DSPs are designed to support this real-world requirement. High speed I/O ports, buffered serial ports, and other peripherals are designed into DSPs to accommodate this. DSPs are, in fact, often referred to as data pumps, because of the speed in which they can process streams of data. This is another characteristic that makes DSPs unique.

DSPs are also found in many embedded applications. I'll discuss the details of embedded systems in Chapter 2. However, one of the constraints of an embedded application is scarce resources. Embedded systems, by their very nature, have scarce resources. The main resources I am referring to here are processor cycles, memory, power and I/O. It has always been this way, and always will. Regardless of how fast embedded processors run, how much memory can be fit on chip, and so on, there will always be applications that consume all available resources and then look for more! Also, embedded applications are very application-specific, not like a desktop application that is much more general-purpose.

At this point, we should now understand that a DSP is like any other programmable processor, except that it is specialized to perform signal processing really efficiently. So now the only question should be; why program anything at all? Can't I do all this signal processing stuff in hardware? Well, actually you can. There is a fairly broad spectrum of DSP implementation techniques, with corresponding trade-offs in flexibility, as well as cost, power, and a few other parameters. The graph below summarizes two of the main trade-offs in the programmable vs. fixed-function decision; flexibility and power.



An application-specific integrated circuit (ASIC) is a hardware only implementation option. These devices are programmed to perform a fixed-function or set of functions. Being a hardware only solution, an ASIC does not suffer from some of the programmable von Neumann-like limitations such as loading and storing of instructions and data. These devices run exceedingly fast in comparison to a programmable solution, but they are not as flexible. Building an ASIC is like building any other microprocessor, to some extent. It's a rather complicated design process, so you have to make sure the algorithms you are designing into the ASIC work and won't need to be changed for a while! You cannot simply recompile your application to fix a bug or change to a new wireless standard. (Actually, you could, but it will cost a lot of money and take a lot of time). If you have a stable, well-defined function that needs to run really fast, an ASIC may be the way to go.

Field-programmable gate arrays (FPGAs) are one of those in-between choices. You can program them and re-program them in the field, to a certain extent. These devices are not as flexible as true programmable solutions, but they are more flexible than an ASIC. Since FPGAs are hardware they offer similar performance advantages to other hardware-based solutions. An FPGA can be "tuned" to the precise algorithm, which is great for performance. FPGAs are not truly application specific, unlike an ASIC. Think of an FPGA as a large sea of gates where you can turn on and off different gates to implement your function. In the end, you get your application implemented, but there are a lot of spare gates laying around, kind of going along for the ride. These take up extra space as well as cost, so you need to do the trade-offs; are the cost, physical area, development cost and performance all in line with what you are looking for?

DSP and μP (microprocessor): We have already discussed the difference here, so there is no need to rehash it. Personally, I like to take the flexible route: programmability. I make a lot of mistakes when I develop signal processing systems; it's very

complicated technology! So, I like to know that I have the flexibility to make changes when I need to in order to fix a bug, perform an additional optimization to increase performance or reduce power (we talk a lot about this as well in this book), or change to the next standard. The entire signal processing field is growing and changing so quickly—witness the standards that are evolving and changing all the time—that I prefer to make the rapid and inexpensive upgrades and changes that only a programmable solution can afford.

The general answer, as always, lies somewhere in between. In fact, many signal processing solutions are partitioned across a number of different processing elements. Certain parts of the algorithm stream—those that have a pretty good probability of changing in the near future—are mapped to a programmable DSP. Signal processing functions that will remain fairly stable for the foreseeable future are mapped into hardware gates (either an ASIC, an FPGA, or other hardware acceleration). Those parts of the signal processing system that control the input, output, user interface and overall management of the system heartbeat may be mapped to a more general-purpose processor. Complicated signal processing systems need the right combination of processing elements to achieve true system performance/cost/power trade-offs. We'll spend more time on this later in the book as well.

Signal processing is here to stay. It's everywhere. Any time you have a signal that you want to know more about, communicate in some way, make better or worse, you need to process it. The digital part is just the process of making it all work on a computer of some sort. If it's an embedded application you must do this with the minimal amount of resources possible. Everything costs money; cycles, memory, power—so everything must be conserved. This is the nature of embedded computing; be application specific, tailor to the job at hand, reduce cost as much as possible, and make things as efficient as possible. This was the way things were done in 1982 when I started in this industry, and the same techniques and processes apply today. The scale has certainly changed; computing problems that required supercomputers in those days are on embedded devices today!

This book will touch on these areas and more as it relates to digital signal processing. There is a lot to discuss and I'll take a practical rather than theoretical approach to describe the challenges and processes required to do DSP well.

DSP Architectures

General-purpose processors are designed to have broad functionality. These processors are designed to be used in a wide variety of applications. The performance goal of general-purpose processors is maximized performance over a broad range of applications. Specialized processors, on the other hand, are designed to take advantage of the limited functionality required by their applications to meet specific objectives. DSPs are a type of specialized processor. These processors are designed to focus on signal processing applications. Hence, there is little or no support to date for features such as virtual memory management, memory protection, and certain types of exceptions.

DSPs, as a type of specialized processor, have customized architectures to achieve high performance in signal processing applications. Traditionally, DSPs are designed to maximize performance for inner loops containing a product of sums. Thus, DSPs usually include hardware support for fast arithmetic, including single cycle multiply instructions (often both fixed-point and floating-point), large (extra wide) accumulators, and hardware support for highly pipelined, parallel computation and data movement. Support for parallel computation may include multiple pipelined execution units. To keep a steady flow of operands available, DSP systems usually rely on specialized, high bandwidth memory subsystems. Finally, DSPs feature specialized instructions and hardware for low overhead loop control and other specialized instructions and addressing modes that reduce the number of instructions necessary to describe the typical DSP algorithm.

Fast, Specialized Arithmetic

DSPs were designed to process certain mathematical operations very quickly. A seven tap FIR filter processing a 400 KHz signal, for example, must complete over 5.6 million multiplies and 5.6 million adds per second. DSP vendors recognized this bottleneck and have added specialized hardware to compute a multiply in only one cycle. This dramatically improves the throughput for these types of algorithms.

Another distinguishing feature of a DSP is its large accumulator register or registers. The accumulator is used to hold the summation of several multiplication operations without overflowing. The accumulator is several bits wider than a normal processor

register which can help avoid overflow during the accumulation of many intermediate results. The extra bits are called *guard bits*. In DSPs the multiplier, adder and accumulator are almost always cascaded so that the “multiply and accumulate” operation so common to DSP algorithms can be specified in a single specialized instruction. This instruction (typically called a *MAC*) is one example of how DSPs are optimized to reduce the number of instruction fetches necessary for DSP algorithms. I’ll describe others later in this chapter.

The MAC unit

At the core of the DSP CPU is the “MAC” unit (for multiply and accumulate), which gets its name from the fact that the unit consists of a multiplier that feeds an accumulator. To feed the multiplier most efficiently, a “data” bus is used for the “x” array, and a “coefficient” bus is used for the “a” array. This simultaneous use of two data buses is a fundamental characteristic of a DSP system. If the coefficients were constants, the user might want to store them in ROM. This is a common place to hold the program, and it allows the user to perform the MAC operations using coefficients stored in program memory and accessed via the program bus. Results of the MAC operation are built up in either of two accumulator registers. Both of these registers can be used equally. This allows the program to maintain two chains of calculations instead of one. The MAC unit is invoked with the MAC instruction. An example of a MAC instruction is shown in Figure 5.1. This one instruction is performing many operations in a single cycle; it reads two operands from memory, increments two pointers, multiplies the two operands together, and accumulates this product with the running total.

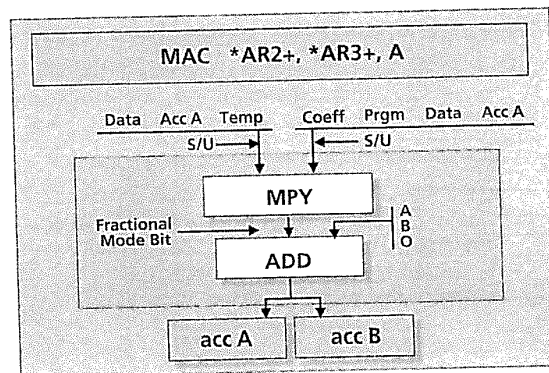


Figure 5.1 Multiply and accumulate functionality in a DSP (courtesy of Texas Instruments)

Parallel ALUs

Most DSP systems need to perform lots of general-purpose arithmetic (simple processes of addition and subtraction). For this type of process, a separate arithmetic logic unit (ALU) is often added to a DSP. The results of the ALU can be basic math or Boolean

functions. These results are sent to either of the accumulators seen before. The ALU performs standard bit manipulation and binary operations. The ALU also has an adder. The adder assures efficient implementation of simple math operations without interfering with the specialized MAC unit.

Inputs to the ALU are, as with the MAC unit, very broad to allow the user maximum flexibility in obtaining efficient code implementation. The ALU can use either accumulator and the data and/or coefficient buses as single or dual inputs (Figure 5.2). The ALU generates the standard status bits including overflow flags for both A and B, zero flags for both A and B, and a carry bit for both A and B.

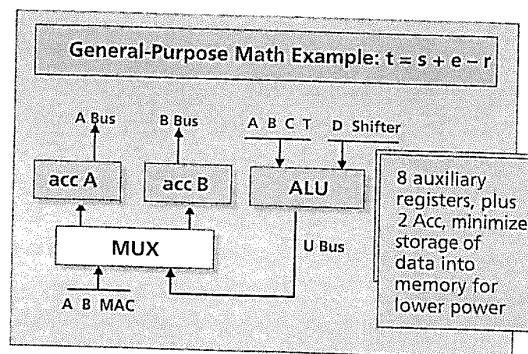


Figure 5.2 DSP with ALU (courtesy of Texas Instruments)

Numeric representation

While DSP units have traditionally favored fixed-point arithmetic, modern processors increasingly offer both fixed- and floating-point arithmetic. Floating-point numbers have many advantages for DSPs;

First, floating-point arithmetic simplifies programming by making it easier to use high level languages instead of assembly. With fixed-point devices, the programmer must keep track of where the implied binary point is. The programmer must also worry about performing the proper scaling throughout the computations to ensure the required accuracy. This becomes very error-prone and hard to debug as well as to integrate.

Floating-point numbers also offer greater dynamic range and precision than fixed-point. Dynamic range is the range of numbers that can be represented before an overflow or an underflow occurs. This range effectively indicates when a signal needs to be scaled. Scaling operations are expensive in terms of processor clocks and so scaling affects the performance of the application. Scaling data also causes errors due to truncation of data and rounding errors (also known as quantization errors). The dynamic range of a processor is determined by size of the exponent. So for an 8-bit exponent the range of magnitudes that can be represented would be:

$$1 * 2^{*(-127)} \dots 2 * 2^{*(127)}$$