Newnes

INCLUDES

FREE
NEWNES ONLINE
MEMBERSHIP

DIGITAL MEDIA PROCESSING

DSP Algorithms Using C

- Streamline your programming with decreased algorithm development times
- Covers all the latest algorithms needed for constrained systems
- Case studies on WiMAX, GPS, and portable media players demonstrate real-world applications

Hazarathaiah Malepati

Digital Media Processing

DSP Algorithms Using C

Hazarathaiah Malepati

ELSEVIER

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

The Boulevard, Langford Lane
Kidlington, Oxford, OX5 1GB, UK

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data

Malepati, Hazarathaiah.
Digital media processing : DSP algorithms using C / by Hazarathaiah Malepati.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-85617-678-1 (alk. paper)
1. Multimedia systems. 2. Embedded computer systems—Programming. 3. Signal processing—Digital techniques.
4. C (Computer program language). I. Title.
QA76.575.M3152 2919
006.7–dc22

2009050460

British Library Cataloguing-in-Publication Data

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

Printed in the United States
10 11 12 13 14 10 9 8 7 6 5 4 3 2 1

This book is dedicated to my late father Mastanaiah Malepati, whose vision and hard work shaped my career a lot.

This page intentionally left blank

Contents

On the Website

Part 5 Embedded Systems

Preface

The title of this book could well have been *Digital Media Processing Algorithms: Efficient Implementation Techniques in C*, as it is not only about digital media processing algorithms, but also contains many implementation techniques for most algorithms. The main purpose of it is to fill the gap between theory and techniques taught at universities and that are required by the software industry in the digital processing of data, signal, speech, audio, images, and video on an embedded processor. The book serves as a bridge to transit from the technical institute to the embedded software development industry. Many powerful algorithms in current cutting-edge technologies are analyzed, and simulation and implementation techniques are presented.

Digital media processing demands efficient programming in order to optimize functionality. Data, signal, image, audio, and video processing—some or all of which are present in all electronic devices today—are complex programming environments. Optimized algorithms (step-by-step directions) are difficult to create, but they can make all the difference when developing a new application. This book discusses the most recent algorithms available to maximize your programming, while simultaneously keeping in mind memory and real-time constraints of the architecture with which you are working. General implementation concepts can be integrated into many architectures that you find yourself working with on a specific project.

My interest in writing a book on digital media processing algorithms derives from reading literature in the field and working on those algorithms. This book cannot replace the literature on the background theory related to the algorithms; in fact, what is written here is largely incomplete without it. Although I do not rigorously discuss the theory and derivation of equations and theorems, a brief introduction and basic mathematics are provided for most of the algorithms presented.

Typically, developers of embedded software modules want to know the basic functionality of an algorithm and simulation techniques, in addition to whether any techniques are available to efficiently implement a particular algorithm. Most developers are proficient with equations and algorithms as a result of university training; however, the efficient implementation of such algorithms requires industry experience. But employers, of course, expect developers to immediately begin work. Often they provide training for writing quality software, but not for writing efficient software. Software engineers learn how to do this in time, such as during the course of working on a few efficiently implemented modules or observing a senior engineer's implementation methods. Many such techniques to efficiently simulate and implement digital media processing algorithms are described in this book.

Today many algorithms are available on the Internet, and the software for a number of them is available in the public domain. But the information available on the web is theory oriented, and we may obtain only pieces of the software here and there and not the complete solution. Sometimes, we can obtain the complete software for a particular algorithm that works well, but it may be inefficient for use in a particular project. Consequently, users have to enhance software efficiency by purchasing it from a third-party source. What's here provides the information needed to develop efficient software for many algorithms from scratch.

The book is aimed at graduate and postgraduate students in various engineering subdisciplines and software industry junior-level employees developing embedded systems software. Only college-level knowledge of mathematics is required to understand the equations and calculations. Knowledge of ANSI C is a prerequisite for this book. Knowledge of microcontroller, microprocessor, or digital signal processing (DSP) architectures will provide an added advantage so that you can understand implementation skills a bit faster.

Unlike other DSP algorithm books that concentrate mainly on basic operations, such as the Fourier transforms and digital filters, this book covers many algorithms commonly used in media processing. For most of them, this book provides full details of flow, implementation complexity, and efficient implementation techniques using ANSI C. In addition, simulation results are provided for selected algorithms.

This book uses the Analog Devices, Inc. (ADI) Blackfin processor (BF5xx series) as the reference embedded processor, and it discusses implementation complexity of all algorithms covered with respect to this amazing general-purpose DSP processor. The *Pcode* notation (meaning pseudocode or program code) is used to flag simulation code.

The availability of test vectors is very important for testing the functionality of any algorithm. Test vectors, look-up tables, and simulation results for most of the standalone algorithms described in this book are available on the companion website at *www.elsevier.direct/companions*. In addition, a final part, Embedded Systems, can be found there along with Appendices A and B, References, and Exercises.

Disclaimer

An algorithm can be implemented on an embedded processor in more than one way. Performance metrics vary according to implementation method. Sometimes there may be a flaw in a particular implementation of a given algorithm, even though we get the best performance with it. It may not be possible to test rigorously for all possible flaws in a given time frame. The program code provided in this book is tested for only a few cases, and it provides selected ways of implementing algorithms and corresponding simulation code. The code may contain bugs. In particular, cryptographic systems are very vulnerable to changes in algorithm flow and implementation as well as software and hardware bugs. Neither the author nor the publisher is responsible for system failures due to the use of any of the techniques or program codes presented in this book. In addition, a few techniques provided may be patented by either ADI or another company; check with the patent office before attempting to incorporate any of the implementation methods discussed when developing your own software.

Acknowledgments

I am very thankful to Analog Devices, Inc. (ADI) and its employees for giving me the opportunity to write this book. ADI is a great place to work and to achieve career goals.

In particular, I am very much indebted to Yosi Stein and Rick Gentile, without whom I may not have succeeded in completing this book. The theme for the book originated while working with Yosi at ADI. My dream of writing it came true with the constant support and encouragement I received from Rick Gentile. I am proud to say Rick and Yosi are the heart and soul of this book.

It is with great pleasure that I thank Boris Liberol for reading every page and providing material on loop-filter and motion compensation for the video coding chapter; Chalil Mohammed for providing sections for the audio coding chapter; and Gabby Yi for providing material on motion estimation. David Katz and Rick Gentile generously gave me permission to take a few sections from their book, *Embedded Media Processing*.

I thank Rick Gentile, Pushparaj Domenic, Gabby Yi, and Bijesh Poyil for reviewing selected sections, and external reviewers Seth Benton and Kenton Williston for reviewing some portions of the material and for giving valuable suggestions for improving the book. I thank Goulin Pan, An Wei, and Boris Learner for spending their precious time with me to clarify a few digital media processing concepts.

I am especially grateful to S.V. Narasimhan, V.U. Reddy, and K.V.S. Hari for their guidance. It is with them that I first began my journey into digital media processing.

I thank N. Sridhara, P. Rama Prabhu, Pushparaj Domenic, Yosi Stein, Joshua Kablotsky, Gordon Sterling, and Rick Gentile for giving me a chance to work with them as part of their team.

I offer my heartfelt thanks to *Analog Dialogue* editor Scott Wayne for forwarding this material to Newnes–Elsevier, and to acquisitions editor Rachel Roumeliotis at Newnes for accepting and preparing the contract for this book. I am very thankful to this book's project manager Marilyn E. Rash, copyeditor Barbara A. Kohl, and proofreader Samantha Molineaux-Graham for enhancing the material here by far from my original writing.

Last, but not least, I thank my family for their support and encouragement during this intense period of brainstorming: my mother Mastanamma for her love and sacrifices and the effort she made in shaping my career; my sister Madhavi, brother-in-law Venkateswarulu, father-in-law Guruvaiah, and mother-in-law Swarajyam have been very supportive and taken care of family responsibilities while I was engaged in this endeavor.

Above all, I would like to thank my wife Sunitha Rani for her love, patience, and constant support throughout this project, and my beautiful daughter Akshara Mahalakshmi, who stayed with her grandparents while I was writing this book. I missed her a lot and hope she will forgive me for not being with her during this time.

CHAPTER 1

Introduction

1.1 Digital Media Processing

Digital media processing as it is currently understood and further developed in this book is described in the following subsections.

1.1.1 Digital Media Defined

In this book, *media* comprises data, text, signal, voice, audio, image, or video information, and *digital media* is the digital representation of analog media information. In our daily lives, we typically use many types of media for various purposes, including the following:

- telephoning (voice)
- listening to music (audio)
- watching TV (audio/video)
- camera use (image/video)
- e-mailing (text/images)
- online shopping (text/data/images)
- money transfer (text/data)
- navigating websites (text/image)
- conferencing (voice/video)
- body scanning with ultrasound and/or magnetic resonance imaging (MRI) (signal/image)
- driving vehicles using GPS (signal/audio/video), and so on

Applications that use media are continually increasing.

1.1.2 Why Digital Media Processing Is Required

In all of the previously mentioned applications, media is sent or received. As a sender or receiver, we typically use the media (talking, listening, watching, mailing, etc.) without experiencing difficulties in perceiving (with our eyes, ears, etc.) or delivering (talking, mailing, texting, etc.) the media. In reality, the media that we send or receive passes through many physical channels and each one adds noise (due to interference, interruptions, switching, lightning, topographic obstacles, etc.) to the original media. In addition, users may want to protect the media (from others), enhance it (improve the original), compress it (for storing/transmitting with less bandwidth), or even work with it (for analysis, detection, extraction, classification, etc.). Digital media processing using appropriate algorithms then is required at both the transmitting and receiving ends to prevent and/or eliminate noise and to achieve application-specific objectives mentioned here.

1.1.3 How Digital Media Is Processed

A software-based digital media processing system is comprised of three entities: an algorithm (that which processes), a software language (to implement the processing), and embedded hardware (to execute the processing). Examples of embedded hardware are digital signal processors (DSPs), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs). In this book, the Analog Devices, Inc. Blackfin

DOI: 10.1016/B978-1-85617-678-1.00001-6

DSP is the reference embedded processor (see Appendix A on the companion website) for executing algorithms. The algorithms are implemented in the C language. Algorithm examples are discussed in the next section.

1.2 Media-Processing Algorithms

In this book, digital media processing algorithms are divided into four categories: data, signal and image, speech and audio, and video. Each category of algorithms are discussed in great detail in various chapters of this book.

1.2.1 Data Processing

Digital systems handle media signals (e.g., data, voice, audio, image, video, text, graphics, and communication signals) by representing them with 1s and 0s, known as binary digits (bits). There are many advantages to digital representation of signals. For example, providing integrity and authenticity to the signal using data security algorithms becomes possible once the signal is digitized. It is also possible to protect data from random and burst errors using data error correction algorithms. In some cases, it is even possible to compress the digital media data using source-coding techniques to minimize the required data transmission or storage bandwidth.

Part 1 of this book covers the most popular algorithms used for data security, error correction, and compression. For all algorithms, a brief introduction, complete details of algorithm flow, C simulation for core algorithm functions, efficient techniques to implement data processing algorithms on the embedded processor, and algorithm computational cost (in terms of clock cycles and memory) for implementing on the reference embedded processor ADI-BF53x (2005) are provided.

Chapter 2 is focused on the most widely used data security algorithms in practice. The algorithms covered include triple data encryption algorithms (TDEA), advanced encryption standard (AES), keyed-hash message authentication code (HMAC), and elliptic curve digital signature algorithm (ECDSA). In addition, cryptography basics and pseudorandom-number generation methods are briefly discussed.

Chapter 3 discusses various data-error detection and correction algorithms. Error detection based on checksum and cyclic redundancy check (CRC) computation is discussed. Both block codes and convolutional codes for error correction and corresponding decoding methods are discussed in detail. The algorithms covered include CRC32, Hamming (N, K), BCH (N, K), Reed-Solomon (RS) (N, K) error correction codes, RS (N, K) erasures correction codes, trellis coded modulation (TCM), turbo codes, low-density parity check (LDPC) codes, Viterbi decoding, maximum *a posteriori* (MAP) decoding, and sum-product (SP) decoding algorithms. Chapter 4 discusses efficient simulation and implementation techniques for all error correction algorithms discussed in Chapter 3.

Widely used data entropy coding methods are discussed in Chapter 5. Variable length codes and arithmetic coding approaches for entropy coding are discussed. The algorithms covered include the MPEG2 VLD, H.264 UVLC and CAVLC, JPEG2000 MQ-coder, and H.264 CABAC.

1.2.2 Digital Signal and Image Processing

We process raw signals using signal processing algorithms to get the desired signal output. Signal processing algorithms have many applications—telecommunications, medical, aerospace, radar, sonar, and weather forecasting, to name the most common. Part 2 of this book is dedicated to signals and systems, time-frequency transformation algorithms, filtering algorithms, multirate signal-processing techniques, adaptive signal processing algorithms, and digital communication algorithms. The later chapters of Part 2 are devoted to image processing tools and advanced image processing algorithms.

In Chapter 6, background theory of digital signal processing algorithms is discussed. We will cover signal representation, types of signals, sampling theorem, signal time-frequency representation (using Fourier series, Fourier transform, Laplace transform, z-transform, and discrete cosine transform [DCT]), linear time invariant (LTI) systems, and convolution operation.

Signal time-frequency representation and signal filtering are discussed thoroughly in most digital signal processing textbooks, including this one. In Chapter 7, we discuss implementation aspects of the fast Fourier transform (FFT), DCT, finite-impulse response (FIR) filters, and infinite impulse response (IIR) filters.

C simulation is provided for all algorithms. Comparative algorithm costs (in terms of clock cycles and memory) for implementation on the reference embedded processor are discussed.

Chapter 8 discusses adaptive signal processing algorithms (minimum mean square error [MMSE] criterion, least mean square [LMS], recursive least squares [RLS], linear prediction [LP], Levinson-Durbin algorithm and lattice filters), multirate signal processing building blocks (e.g., decimation, interpolation, polyphase filter implementation of decimation and interpolation, and filter banks), and wavelet signal processing (multiresolution analysis and discrete wavelet transform). The C fixed-point implementation of the LMS algorithm is also presented.

Chapter 9 discusses the digital communication environment (channel capacity, noise measurement, modulation techniques), single-carrier communication, multicarrier communication system building blocks (discrete multitone [DMT] and orthogonal frequency division multiplexing [OFDM] transceivers), channel estimation algorithms (for both wireline and wireless), channel equalizers (minimum mean square [MMS] equalizer, decision-feedback [DF] equalizer, Viterbi equalizer, and turbo equalizer) and synchronization algorithms (frequency offset estimation, symbol timing recovery, and frame synchronization). As most digital communication algorithms involve basic signal-processing tasks (e.g., DFT, filtering), no exclusive C simulation is provided for these algorithms. However, a few techniques to efficiently implement commonly used basic mathematic operations such as division and square root on fixed-point processors are discussed, and C-simulation code is provided for those basic operations.

Image processing plays an important role in medical imaging, digital photography, computer graphics, multimedia communications, automotive, and video surveillance, to name the most common applications. Image processing tools are basically algorithms used to process the image to achieve aims specific to the application, such as improving image quality, creating special effects, compressing images for storage or fast transmission, and correcting abnormalities in the captured image (sometimes the capturing device itself introduces artifacts in the image due to hardware limitations or lens distortion). Image processing tools are also used in classifying images, detecting objects in the image, and extracting useful information from captured images.

Chapter 10 is focused on discussing and simulating widely used image processing tools such as color conversion, color enhancement, brightness and contrast correction, edge enhancement, noise reduction, edge detection, image scaling, image object corners detection, dilation and erosion morphological operators, and the Hough transform.

Advanced image processing algorithms such as image rotation, image stabilization, object detection (e.g., the human face, vehicle license plates), 2D image filtering, fisheye correction, and image compression techniques (DCT-based JPEG and wavelet-based JPEG2000), are discussed in Chapter 11. The C-simulation code and algorithm costs (in terms of processor clock cycles and memory) are also provided for image rotation and 2D image filtering algorithms.

1.2.3 Speech and Audio Processing

Speech and audio coding are very important topics in the field of multimedia storage and communication systems. Example audio- and speech-coding applications are telecommunications, digital audio broadcasting (DAB), portable media players, military applications, cinema, home entertainment systems, and distance learning. Human speech processing has many other applications, such as voice detection and speech recognition. Part 3 is dedicated to discussion of algorithms related to speech processing, speech coding, audio coding, and audio post-processing, among others.

In Chapter 12, we discuss sound and audio signals, and explore how audio data is presented to the processor from a variety of audio converters. Next, the formats in which audio data is stored and processed are described. Selected software building blocks for embedded audio systems are also discussed. Because efficient data movement is essential for overall system optimization, data buffering as it applies to speech and audio algorithms is examined. There are many speech coding algorithms in the literature and this chapter briefly discusses a few methods. Various speech compression standards are also briefly addressed. Finally, the Voice over Internet Protocol (VoIP) and the purpose of the jitter buffer in VoIP communication systems are discussed.

Audio coding methods are discussed in Chapter 13. While audio requires less processing power in general than video processing, it should be considered equally important. Recent applications such as wireless, Internet, and multimedia communication systems have created a demand for high-quality digital audio delivery at low bit rates. The technologies behind various audio coding techniques are discussed, followed by examination of MPEG-4 AAC codec modules and encoder and decoder architectures. Various commercially available audio codecs and their implementation costs (in terms of cycles and memory) are presented. Finally, we discuss a few audio post-processing techniques for enhancing the listening experience.

1.2.4 Video Processing

Advances in video coding technology and standardization, along with rapid development and improvements of network infrastructures, storage capacity, and computing power, are enabling an increasing number of video applications. Digitized video has played an important role in many consumer electronics applications, including DVD, portable media players, HDTV, video telephony, video conferencing, Internet video streaming, and distance learning, among others. As we move to high-definition video, the computing bandwidth required to process video increases manyfold, and more than 80% of total available embedded processor computing power is allocated for video processing.

Chapter 14 describes video signals, and various redundancies present in video frames are explored. Video coding building blocks (e.g., motion estimation/compensation, block transform, quantization, and variable-length coding) are briefly discussed, followed by a survey of various video coding standards and comparisons with respect to coding efficiency and costs. Computationally complex (high-cost) coding blocks are identified. Efficient ways of implementing video coders are discussed, followed by an examination of the two most widely adopted video coding standards—the MPEG-2 and H.264 decoder modules. Details of H.264-specific decoding modules (e.g., H.264 transform, intraprediction, loop filtering) are provided. Also discussed are a few techniques to efficiently implement the H.264 macroblock layer. A scalable video coding (based on the H.264 scalable extension standard) and its applications are discussed. Video processing, as stated before, when compared to other media processing, is very costly in terms of computation, memory, and data movement bandwidths. Video coding and system issues because of limited MIPS, memory, and system bus bandwidth are presented in Section 16.5 on the companion website, along with the use of proper frameworks to minimize power consumption in low-power video applications.

Video data is often processed after decompression and before sending it to the display for enhancement or rendering it suitable for playing on the screen. This part of the procedure is called "video post-processing." Chapter 15 is focused on video post-processing modules such as video scaling, video filtering, video enhancement, alpha blending, gamma correction, and video transcoding.

1.3 Embedded Systems and Applications

Embedded systems enable numerous digital devices used in daily life, and thus, are literally everywhere. Embedded computing systems have grown tremendously in recent years not only in popularity, but also in computational complexity. In all the applications listed in Table 1.1, digital embedded systems process some form of digital data. Digital media processing algorithms play an important role in all embedded system applications.

This book is focused on digital media and communication processing algorithms—that is, applications involving processing and communication of large data blocks (whether image, video, audio, speech, text blocks, or some combination of these), which often need real-time data processing. For an application, we choose a particular embedded processor along with a peripheral set only after studying its capabilities to run the algorithms of a particular application.

The last part of this book discusses embedded systems, media processing, and their applications. Embedded systems have several common characteristics that distinguish such systems from general-purpose computing systems. Unlike desktops, the embedded systems handle huge amount of data per second with very limited resources (e.g., arithmetic logic units [ALUs], memory, peripherals). In most cases, embedded systems handle very few tasks and usually these tasks must be performed in real time.

In Chapter 16 (see companion website), we discuss the important components of an embedded system (e.g., processor core, memory, and peripherals). Various types of memory and peripheral components are briefly

Table 1.1: Digital media processing applications

Digital Home	**Telecommunications**	**Consumer Electronics**
AV receivers	ADSL/VDSL	Digital camera
DVD/Blu-Ray players	Cable modems	Portable media players
TV/desktop audio/video	Wire/wireless smart phones	Portable DVD players
Sound bar	IP phone	Digital video recorder
Digital picture frame	Femto base stations	Personal GPS navigation
Video telephony	Software defined radio	Mobile TV
IP TV, IP phone, IP camera	WLAN, WiFi, WiMAX	Bluetooth
Door phone	Mobile TV	HD/ANC headphones
Smoke detector	Radar/sonar	Video game players
Network video recorder	Power line communication	Digital music instruments
CD clock radio	Video conferencing	
FM/satellite radio		
Automotives	**Industrial**	**Medical**
Advanced driver assistance	Power meter	Ultrasound
Automotive infotainment	Motor control	CT, MRI, PET
Digital audio/satellite radio	Active noise cancellation	Digital *x*-ray
Vision control	Barcode scanner	Pulse oximetry
Bluetooth hands-free phone	Flow meter	Digital stethoscope
Electronic stability control	Oscilloscope	Blood-pressure monitor
Safety/airbag control	**Security**	Lab diagnostic equipment
Crash detection	Surveillance IP networks	Heart rate monitor
	Fingerprint biometrics	
	Video doorbell	
	Video analytic server	

discussed. The necessity of software–hardware partitioning of embedded systems to handle complex applications is discussed, as well as possible ways to efficiently partition such a system. Finally, we discuss future embedded processor requirements to handle very complex embedded applications.

Chapter 17 (see companion website) briefly discusses various applications. Different embedded applications use different algorithms. The processing power and memory requirements vary from one application to another. We briefly talk about various modules present in a few embedded application sectors. The applications covered in this chapter include automotive, video surveillance, portable entertainment systems, digital communications, digital camera, and immigration and healthcare sectors.

1.4 Algorithm Implementation on DSP Architectures

In Section 1.2, various algorithms that are playing a critical role in diverse applications were mentioned. Although dozens of semiconductor companies are designing embedded processors with a range of architectural features to support different kinds of applications, no single architecture is efficient for processing all types of digital media processing algorithms. This is because processors designed with many pipeline stages (to execute in parallel multiple operations of numeric-intensive algorithms) do not efficiently handle algorithms that contain full-control operations. The architectures developed for executing the control code are not efficient at computing numeric-intensive algorithms. The architectural feature set of the reference embedded processor (see Appendix A on the companion website) is in between, and is good at handling both control and numeric-intensive algorithms.

In the following subsections, DSP architecture and its performance in executing various algorithms are briefly discussed. We also briefly describe a few algorithm implementation techniques.

1.4.1 DSP Architecture

A simplified block diagram of embedded DSP architecture is shown in Figure 1.1. The main architectural blocks of an embedded processor are the processor core (with register sets, ALU, data address generator [DAG], sequencer, etc.), memory (for holding instructions and data, for stack space, etc.), peripherals (e.g., serial peripheral interface [SPI], parallel peripheral interface [PPI], serial ports [SPORT], general-purpose timers, universal asynchronous receiver transmitter [UART], watchdog timer, and general-purpose I/O) and a few others (e.g., JTAG emulator, event controller, direct memory access [DMA] controller). Embedded processor peripherals and memory architectures are discussed in some detail in Chapter 16.

The peripheral features are important when we talk about the overall application. In this book, we assume that the architecture comes with all necessary peripherals to enable a particular application. Also, we assume that the program code and data required for algorithm processing are residing in the faster memory (or level 1, L1) memory, which can be accessed at the speed of the processor core. If we cannot fit data and program in L1 memory, then we store the extra data or program in L2/L3 memory and use DMA to get the data or program from L2/L3 memory without interrupting the processor core. From an algorithm-implementation point of view, the important things are processor core architecture, availability of L1 memory, and internal bus bandwidth.

Even more important than getting data into (or sending it out from) the processor, is the structure of the memory subsystem that handles the data during processing. It is essential that the processor core access data in memory at rates fast enough to meet application demands. L1 memory is often split between instruction and data segments for efficient utilization of memory bus bandwidth. Most DSP architectures support this Harvard-like architecture (in which data and instruction memories are accessed simultaneously, as shown in Figure 1.1) in combination with a hierarchical memory structure that views memory as a single, unified gigabyte 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.

The register file contains different register types (e.g., data registers, accumulators, address registers) to hold the information temporarily for ALU processing or for memory load/store purposes. The processor's computational units perform numeric processing for DSP algorithms and general control algorithms. Data moving in and out of the computational units go through the data register file. The processor's assembly language provides access to the data register file. The syntax lets programs move data to and from these registers and specify a computation's data format at the same time.

The DAGs generate addresses for data moving to and from memory. By generating addresses, the DAGs let programs refer to addresses indirectly using a DAG register instead of an absolute address.

The *program sequencer* controls the instruction execution flow, including instruction alignment and decoding. 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: jumps, conditional branches, function calls, interrupts, loops, and so on.

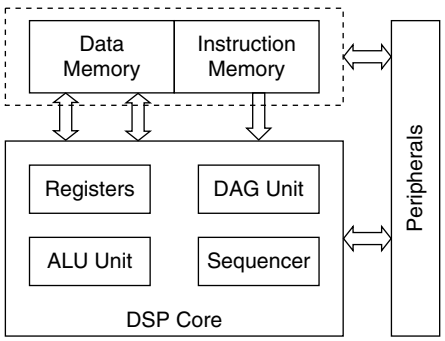

Figure 1.1: Simplified diagram of DSP architecture.

In the next subsection, we consider three algorithms with different processing flow requirements and discuss to what extent the benchmarks provided by processor manufacturers are useful in deciding which processor (from dozens of processors available today in the market) is suitable for a particular application.

1.4.2 Algorithm Complexity and DSP Performance

In this section, we consider three simple algorithms—dot product, RC4 stream cipher, and the H.264 CABAC encode-symbol-normalization process—and discuss embedded processor performance (with a particular architectural feature set) in executing those three algorithms.

Dot Product

Dot product involves accumulation of sample-by-sample multiplication of elements from two sample arrays. The dot product, z, of two N-length sample arrays $x[]$ and $y[]$, can be computed as

$$z = \sum_{n=0}^{N-1} x[n]y[n] \tag{1.1}$$

A simple "for" loop C code that implements the dot product described by Equation (1.1) is shown in Pcode 1.1.

What is the cost (in terms of cycles and memory) of this dot-product algorithm for implementation on the embedded processor, given its processor core architecture? Clearly, we require two buffers of length 2*N bytes (assuming the elements are the 16-bit word type), each to hold the two input array buffers in memory.

In terms of computations, it involves N multiplications and N additions. If the embedded processor consumes one cycle for multiplication and one cycle for addition, then we require a total of $2N$ cycles (assuming a single ALU) to execute the corresponding dot-product code given in Pcode 1.1. What about the cycle cost of loading the data from memory to the data registers? Typically, many processors come with separate data load/store units; hence, we assume that the data loads happen parallel to compute operations and therefore they are free.

```
z = 0;
for(i = 0;i < N;i++)
         z += x[i] * y[i];
```

Pcode 1.1: Pseudo code for dot product.

Many embedded processors come with multiply–accumulate (MAC) units, and in this case we require only N cycles, as the dot product contains a total of N MAC operations. For this case, the two memory loads must happen in a single cycle.

Now, you may wonder whether this cycle count can be achieved with the C code ported to the processor assembly using the compiler or with the optimized assembly-level code written manually. Here, when we say that the cycle count is N for executing the dot product, it means that one MAC operation is mapped to a single processor instruction, which consumes exactly one cycle; only then can we describe the cycle count as N cycles for N MAC operations.

Is this the final cycle count for computing the dot product? Not exactly—in the dot-product case, it also depends on the number of MAC units that the processor comes with. For example, if the processor consists of four MAC units, then we require only N/4 cycles to complete the dot product. How is this possible? It is possible because we can execute four MAC operations in parallel on a four-MAC processor, as the dot product has no flow dependencies. However, we will have a problem with the data load unless we load 128 bits (four 16-bit words from array $x[]$ and another four 16-bit words from array $y[]$) of data to eight 16-bit registers in a single cycle.

For efficient compilation to run on a four-MAC processor, we unroll the dot-product loop in Pcode 1.1 by four times and reduce the loop count by a factor of 4 as shown in Pcode 1.2. Given that the dot product is a simple algorithm, most compilers can efficiently map the C code to the assembly language so that the difference between cycle estimation and actual cycles measured is negligible.

```
z1 = 0; z2 = 0; z3 = 0; z4 = 0;
for(i = 0;i < N/4;i += 4) {
          z1 += x[i]*y[i];              // MAC unit 1
          z2 += x[i + 1]*y[i + 1];      // MAC unit 2
          z3 += x[i + 2]*y[i + 2];      // MAC unit 3
          z4 += x[i + 3]*y[i + 3];      // MAC unit 4
}
z = z1 + z2 + z3 + z4;
```

Pcode 1.2: Pseudo code for dot product with loop unrolling four times.

Digital media processing algorithms are not just "dot products." Next, we consider another simple algorithm, the RC4 stream cipher.

RC4 Stream Cipher

The RC4 algorithm (see Section 2.1.6, RC4 Algorithm, for more details) is used as a stream cipher in low-security applications and as a pseudorandom number generator in many standard ciphers applications. RC4 is used in many commercial software packages, such as Lotus Notes and Oracle Secure SQL, and in network protocols, such as SSL, IPsec, WEP, and WPA. An RC4 simulation code is given in Pcode 1.3.

```
j = 0;
for (i = 0;i < N;i++) {            // N: data length in bytes
          k = i & 0xff;            // i mod 255
          r0 = SBox[k];
          r1 = j + r0;
          j = r1 & 0xff;           // i mod 255
          r1 = SBox[j];            // look-up table access with arbitrary offset
          Sbox[j] = r0;            // swap look-up table elements
          Sbox[k] = r1;
          r1 = r1 + r0;
          r1 = r1 & 0xff;          // i mod 255
          r1 = Sbox[r1];           // look-up table access with arbitrary offset
          in[i] = in[i] ^ r1;      // encrypt input message bytes
}
```

Pcode 1.3: Simulation code for RC4 stream cipher.

In the iterative procedure of computing RC4 encrypted data using Pcode 1.3, the computation of a new j value requires updated (swapped) S-box values. Thus, computing many j values and swapping them at the same time is not possible due to the dependency of j on updated S-box values. The RC4 algorithm is sequential in nature, although no jumps are present. Even if multiple compute units are available with the processor, we cannot use them in this case for parallel implementation of the algorithm. See Section 2.1.6, RC4 Algorithm, for cycle costs and memory requirements to implement RC4 on the reference embedded processor.

Unlike the dot product, the execution of algorithms, such as RC4 on deep-pipeline processors, may not be efficient in terms of cycles. RC4 can be computed efficiently on microcontrollers with a two-stage pipeline in fewer cycles, compared to DSPs with 10 or more pipeline stages.

In the case of algorithms with frequently occurring conditional branches (e.g., the H.264 CABAC encode symbol normalization process described in Section 5.5), the performance of deep-pipeline DSPs worsens. As shown in Pcode 1.4, the normalization process has many conditional jumps in a "while loop." This process is costly in terms of cycles, as it performs normalization 1 bit at a time with many jumps. Avoiding jumps is the only solution to reduce cycle cost (see Section 5.5 for details).

In summary, DSPs are good at handling FFTs, filters, and matrix operations, and are less effective at handling both control code and sequential algorithms. Simple pipeline processors (e.g., ARM) are good at handling control and sequential algorithms, and less effective at handling signal processing tasks such as transforms, filtering operations, and so on.

In brief, the dot-product benchmark provided by the DSP manufacturer may not provide much useful information because the application at hand rarely contains dot-product kinds of operations. To efficiently run

```
while(pBAC->Range < 256) {// Low, Range, Outstanding bits (or Obits) are CABAC params
      if(pBAC->Low >= 512) {
              pBAC->Low -= 512;
              write_bits(1,1);
              if(pBAC->Obits > 0) {
                      write_bits(0,pBAC->Obits);        // bit-fifo write
                      pBAC->Obits = 0;
              }
      }
      else if(pBAC->Low < 256) {
               write_bits(0,1);
               if(pBAC->Obits > 0){
                       write_bits(1,pBAC->Obits);       // bit-fifo write
                       pBAC->Obits = 0;
               }
      }
      else{
               pBAC->Obits++;
               pBAC->Low -= 256;
      }
      pBAC->Range = pBAC->Range << 1;
      pBAC->Low = pBAC->Low << 1;
}
```

Pcode 1.4: Simulation code for H.264 CABAC encode symbol normalization.

any algorithm on a particular digital signal processor, we need to dedicate some time to understanding the underlying mathematical structure of the algorithm and then tune it to write efficient code for that processor. A few techniques to map algorithms to DSPs are discussed in the next section.

1.4.3 Algorithm Implementation Techniques

Digital data is efficiently processed with an embedded processor by optimizing the corresponding program at both the algorithm flow level and the instruction level. The algorithms are optimized for throughput, memory usage, I/O bandwidth, and power dissipation. In this subsection, we discuss algorithm-level optimization using various techniques for increasing throughput. In most cases, there is a trade-off between throughput and memory.

Algorithm code is optimized at the instruction level to eliminate pipeline stalls due to data dependencies, to minimize the overhead of control code such as jumps and software loop overheads, and to efficiently handle data movement within the system. Instruction-level optimization techniques vary by processor. Compilers also perform some degree of instruction-level optimization. Typically we see a 10 to 20% gain with instruction-level optimization (measured by a decrease in core clock cycles). When optimizing the code at the instruction level, complete knowledge of the algorithm structure may not be necessary.

On the other hand, program-flow optimization at the algorithm level requires knowledge of the algorithm's mathematical structure and properties. Compilers cannot achieve algorithm-level program optimization. Minimizing the number of computations and balancing the CPU and load/store bandwidth are possible with algorithm-level optimization. We can achieve algorithm-level optimization using multiple approaches. A few of these methods considered in this section include changing the algorithm flow, using look-up tables, using algorithm-flow statistics, using symmetry and periodicity, reusing already-computed data, and approximating mathematic functionality. The amount of cycle savings depends on a particular algorithm and its flow. For the algorithms discussed in this book, the amount of cycle savings achieved with algorithm-level optimization ranges from 20 to 80%.

Is Optimizing All the Program Code Worthwhile?

Before we proceed, we ask whether optimizing all the program code is worthwhile. The answer is that it depends on processor capabilities and application demands. Usually, we start optimizing the most critical modules in C, and if the MIPS budget is not met, we continue to optimize other critical modules. If we are still not within the MIPS budget, then we start writing assembly language and optimizing it. For example, consider a video decoder (see Chapter 14 for details); it has many layers and modules (see Figure 14.15). In the slice layer, we decode