Synthesis and Optimization of DSP Algorithms

George A. Constantinides,
Peter Y.K. Cheung and
Wayne Luk

Kluwer Academic Publishers

2

Background

This chapter provides some of the necessary background required for the rest of this book. In particular, since this book is likely to be of interest both to DSP engineers and digital designers, a basic introduction to the essential nomenclature within each of these fields is provided, with references to further material as required.

Section 2.1 introduces microprocessors and field-programmable gate arrays. Section 2.2 then covers the discrete-time description of signals using the z-transform. Finally, Section 2.3 presents the representation of DSP algorithms using computation graphs.

2.1 Digital Design for DSP Engineers

2.1.1 Microprocessors vs. Digital Design

One of the first options faced by the designer of a digital signal processing system is whether that system should be implemented in hardware or software. A software implementation forms an attractive possibility, due to the mature state of compiler technology, and the number of good software engineers available. In addition microprocessors are mass-produced devices and therefore tend to be reasonably inexpensive. A major drawback of a microprocessor implementation of DSP algorithms is the computational throughput achievable. Many DSP algorithms are highly parallelizable, and could benefit significantly from more fine-grain parallelism than that available with general purpose microprocessors. In response to this acknowledged drawback, general purpose microprocessor manufacturers have introduced extra single-instruction multiple-data (SIMD) instructions targetting DSP such as the Intel MMX instruction set [PW96] and Sun's VIS instruction set [TONH96]. In addition, there are microprocessors specialized entirely for DSP such as the well-known Texas Instruments DSPs [TI]. Both of these implementations allow higher throughput than that achievable with a general purpose processor, but there is still a significant limit to the throughput achievable.

The alternative to a microprocessor implementation is to implement the algorithm in custom digital hardware. This approach brings dividends in the form of speed and power consumption, but suffers from a lack of mature high-level design tools. In digital design, the industrial state of the art is register-transfer level (RTL) synthesis [IEE99, DC]. This form of design involves explicitly specifying the cycle-by-cycle timing of the circuit and the word-length of each signal within the circuit. The architecture must then be encoded using a mixture of data path and finite state machine constructs. The approaches outlined in this book allow the production of RTL-synthesizable code directly from a specification format more suitable to the DSP application domain.

2.1.2 The Field-Programmable Gate Array

There are two main drawbacks to designing an application-specific integrated circuit (ASICs) for a DSP application: money and time. The production of state of the art ASICs is now a very expensive process, which can only realistically be entertained if the market for the device can be counted in millions of units. In addition, ASICs need a very time consuming test process before manufacture, as 'bug fixes' cannot be created easily, if at all.

The Field-Programmable Gate Array (FPGA) can overcome both these problems. The FPGA is a programmable hardware device. It is mass-produced, and therefore can be bought reasonably inexpensively, and its programmability allows testing in-situ. The FPGA can trace its roots from programmable logic devices (PLDs) such as PLAs and PALs, which have been readily available since the 1980s. Originally, such devices were used to replace discrete logic series in order to minimize the number of discrete devices used on a printed circuit board. However the density of today's FPGAs allows a single chip to replace several million gates [Xil03]. Under these circumstances, using FPGAs rather than ASICs for computation has become a reality.

There are a range of modern FPGA architectures on offer, consisting of several basic elements. All such architectures contain the 4-input lookup table (4LUT or simply LUT) as the basic logic element. By configuring the data held in each of these small LUTs, and by configuring the way in which they are connected, a general circuit can be implemented. More recently, there has been a move towards heterogeneous architectures: modern FPGA devices such as Xilinx Virtex also contain embedded RAM blocks within the array of LUTs, Virtex II adds discrete multiplier blocks, and Virtex II pro [Xil03] adds PowerPC processor cores.

Although many of the approaches described in this book can be applied equally to ASIC and FPGA-based designs, it is our belief that programmable logic design will continue to increase its share of the market in DSP applications. For this reason, throughout this book, we have reported results from these methods when applied to FPGAs based on 4LUTs.

2.1.3 Arithmetic on FPGAs

Two arithmetic operations together dominate DSP algorithms: multiplication and addition. For this reason, we shall take the opportunity to consider how multiplication and addition are implemented in FPGA architectures. A basic understanding of the architectural issues involved in designing adders and multipliers is key to understanding the area models derived in later chapters of this book.

Many hardware architectures have been proposed in the past for fast addition. As well as the simple ripple-carry approach, these include carry-look-ahead, conditional sum, carry-select, and carry-skip addition [Kor02]. While the ASIC designer typically has a wide choice of adder implementations, most modern FPGAs have been designed to support fast ripple-carry addition. This means that often, 'fast' addition techniques are actually slower than ripple-carry in practice. For this reason, we restrict ourselves to ripple carry addition.

Fig. 2.1 shows a portion of the Virtex II 'slice' [Xil03], the basic logic unit within the Virtex II FPGA. As well as containing two standard 4LUTs, the slice contains dedicated multiplexers and XOR gates. By using the LUT to generate the 'carry propagate' select signal of the multiplexer, a two-bit adder can be implemented within a single slice.

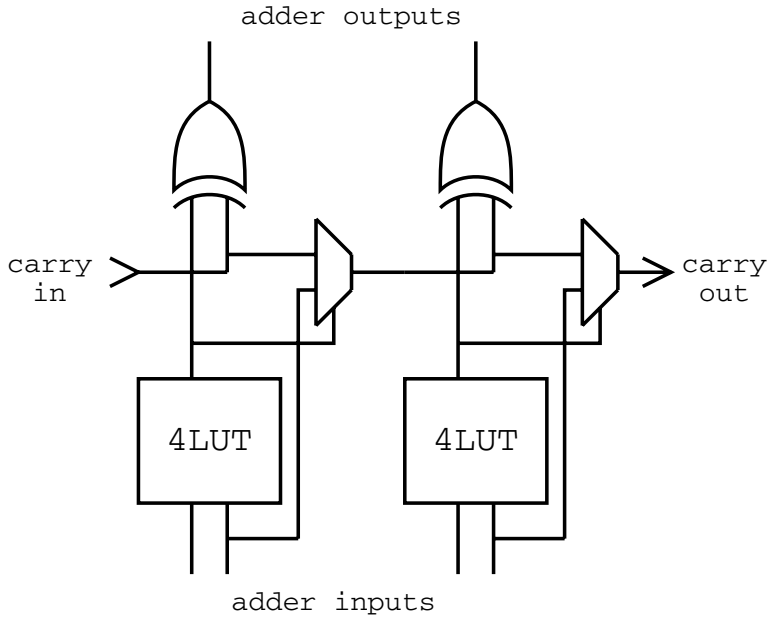


Fig. 2.1. A Virtex II slice configured as a 2-bit adder

In hardware arithmetic design, it is usual to separate the two cases of multiplier design: when one operand is a constant, and when both operands may vary. In the former case, there are many opportunities for reducing the hardware cost and increasing the hardware speed compared to the latter case. A constant-coefficient multiplication can be re-coded as a sum of shifted versions of the input, and common sub-expression elimination techniques can be applied to obtain an efficient implementation in terms of adders alone [Par99] (since shifting is free in hardware). General multiplication can be performed by adding partial products, and general multipliers essentially differ in the ways they accumulate such partial products. The Xilinx Virtex II slice, as well as containing a dedicated XOR gate for addition, also contains a dedicated AND gate, which can be used to calculate the partial products, allowing the 4LUTs in a slice to be used for their accumulation.

2.2 DSP for Digital Designers

A signal can be thought of as a variable that conveys information. Often a signal is one dimensional, such as speech, or two dimensional, such as an image. In modern communication and computation, such signals are often stored digitally. It is a common requirement to process such a signal in order to highlight or supress something of interest within it. For example, we may wish to remove noise from a speech signal, or we may wish to simply estimate the spectrum of that signal.

By convention, the value of a discrete-time signal x can be represented by a sequence $x[n]$. The index n corresponds to a multiple of the sampling period T, thus $x[n]$ represents the value of the signal at time nT. The z transform (2.1) is a widely used tool in the analysis and processing of such signals.

$$X(z) = \sum_{n=-\infty}^{+\infty} x[n]z^{-n} \tag{2.1}$$

The z transform is a linear transform, since if $X_1(z)$ is the transform of $x_1[n]$ and $X_2(z)$ is the transform of $x_2[n]$, then $\alpha X_1(z) + \beta X_2(z)$ is the transform of $\alpha x_1[n] + \beta x_2[n]$ for any real α, β. Perhaps the most useful property of the z transform for our purposes is its relationship to the convolution operation. The output $y[n]$ of any linear time-invariant (LTI) system with input $x[n]$ is given by (2.2), for some sequence $h[n]$.

$$y[n] = \sum_{k=-\infty}^{+\infty} h[k]x[n-k] \tag{2.2}$$

Here $h[n]$ is referred to as the impulse response of the LTI system, and is a fixed property of the system itself. The z transformed equivalent of (2.2), where $X(z)$ is the z transform of the sequence $x[n]$, $Y(z)$ is the z transform

of the sequence $y[n]$ and $H(z)$ is the z transform of the sequence $h[n]$, is given by (2.3). In these circumstances, $H(z)$ is referred to as the transfer function.

$$Y(z) = H(z)X(z) \tag{2.3}$$

For the LTI systems discussed in this book, the system transfer function $H(z)$ takes the rational form shown in (2.4). Under these circumstances, the values $\{z_1, z_2, \ldots, z_m\}$ are referred to as the *zeros* of the transfer function and the values $\{p_1, p_2, \ldots, p_n\}$ are referred to as the *poles* of the transfer function.

$$H(z) = K\frac{(z^{-1} - z_1^{-1})(z^{-1} - z_2^{-1})\ldots(z^{-1} - z_m^{-1})}{(z^{-1} - p_1^{-1})(z^{-1} - p_2^{-1})\ldots(z^{-1} - p_n^{-1})} \tag{2.4}$$

2.3 Computation Graphs

Synchronous Data Flow (SDF) is a widely used paradigm for the representation of digital signal processing systems [LM87b], and underpins several commerical tools such as Simulink from The MathWorks [SIM]. A simple example diagram from Simulink is shown in Fig. 2.2. Such a diagram is intuitive as a form of data-flow graph, a concept we shall formalize shortly. Each node represents an operation, and conceptually a node is ready to execute, or 'fire', if enough data are present on all its incoming edges.

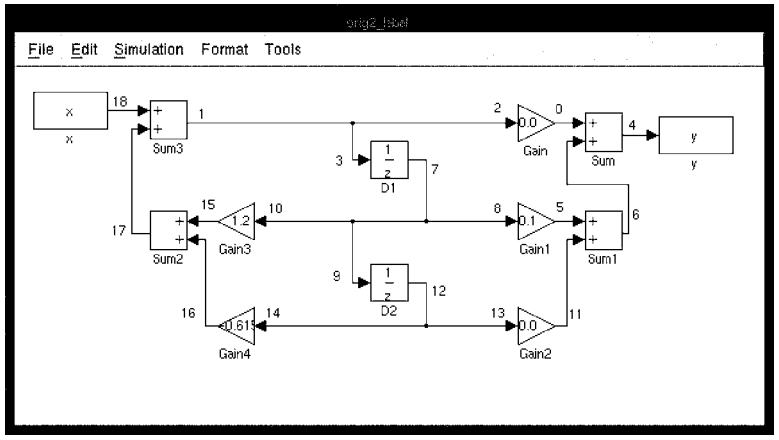


Fig. 2.2. A simple Simulink block diagram

In some chapters, special mention will be made of linear time invariant (LTI) systems. Individual computations in an LTI system can only be one of several types: constant coefficient multiplication, unit-sample delay, addition, or branch (fork). Of course the representation of an LTI system can be of a

hierarchical nature, in terms of other LTI systems, but each leaf node of any such representation must have one of these four types. A flattened LTI representation forms the starting point for many of the optimization techniques described.

We will discuss the representation of LTI systems, on the understanding that for differentiable nonlinear systems, used in Chapter 4, the representation is identical with the generalization that nodes can form any differentiable function of their inputs.

The representation used is referred to as a *computation graph* (Definition 2.1). A computation graph is a specialization of the data-flow graphs of Lee *et al.* [LM87b].

Definition 2.1. A *computation graph* $G(V, S)$ is the formal representation of an algorithm. V is a set of graph nodes, each representing an atomic computation or input/output port, and $S \subset V \times V$ is a set of directed edges representing the data flow. An element of S is referred to as a *signal*. The set S must satisfy the constraints on indegree and outdegree given in Table 2.1 for LTI nodes. The *type* of an atomic computation $v \in V$ is given by $\text{TYPE}(v)$ (2.5). Further, if V_G denotes the subset of V with elements of GAIN type, then $\text{COEF} : V_G \to \mathbb{R}$ is a function mapping the GAIN node to its coefficient.

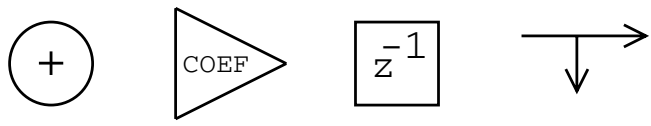
$$\text{TYPE} : V \to \{\text{INPORT}, \text{OUTPORT}, \text{ADD}, \text{GAIN}, \text{DELAY}, \text{FORK}\} \tag{2.5}$$

□

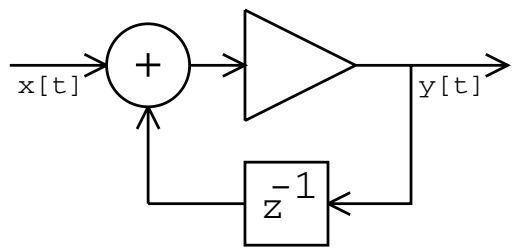
Table 2.1. Degrees of nodes in a computation graph

TYPE(v)	INDEGREE(v)	OUTDEGREE(v)
INPORT	0	1
OUTPORT	1	0
ADD	2	1
DELAY	1	1
GAIN	1	1
FORK	1	≥ 2

Often it will be useful to visualize a computation graph using a graphical representation, as shown in Fig. 2.3. Adders, constant coefficient multipliers and unit sample delays are represented using different shapes. The coefficient of a GAIN node can be shown inside the triangle corresponding to that node. Edges are represented by arrows indicating the direction of data flow. Fork nodes are implicit in the branching of arrows. INPORT and OUTPORT nodes are also implicitly represented, and usually labelled with the input and output names, $x[t]$ and $y[t]$ respectively in this example.



(a) some nodes in a computation graph



(b) an example computation graph

Fig. 2.3. The graphical representation of a computation graph

Definition 2.1 is sufficiently general to allow any multiple input, multiple output (MIMO) LTI system to be modelled. Such systems include operations such as FIR and IIR filtering, Discrete Cosine Transforms (DCT) and RGB to YCrCb conversion. For a computation to provide some useful work, its result must be in some way influenced by primary external inputs to the system. In addition, there is no reason to perform a computation whose result cannot influence external outputs. These observations lead to the definition of a well-connected computation graph (Definition 2.2). The computability property (Definition 2.4) for systems containing loops (Definition 2.3) is also introduced below. These definitions become useful when analyzing the properties of certain algorithms operating on computation graphs. For readers from a computer science background, the definition of a recursive system (Definition 2.3) should be noted. This is the standard DSP definition of the term which differs from the software engineering usage.

Definition 2.2. A computation graph $G(V, S)$ is *well-connected* iff (a) there exists at least one directed path from at least one node of type INPORT to each node $v \in V$ and (b) there exists at least one directed path from each node in $v \in V$ to at least one node of type OUTPORT. □

Definition 2.3. A *loop* is a directed cycle (closed path) in a computation graph $G(V, S)$. The *loop body* is the set of all vertices $V_1 \subset V$ in the loop. A computation graph containing at least one loop is said to describe a *recursive* system. □

Definition 2.4. A computation graph G is *computable* iff there is at least one node of type DELAY contained within the loop body of each loop in G. □

2.4 The Multiple Word-Length Paradigm

Throughout this book, we will make use of a number representation known as the *multiple word-length paradigm* [CCL01b]. The multiple word-length paradigm can best be introduced by comparison to more traditional fixed-point and floating-point implementations. DSP processors often use fixed-point number representations, as this leads to area and power efficient implementations, often as well as higher throughput than the floating-point alternative [IO96]. Each two's complement signal $j \in S$ in a multiple word-length implementation of computation graph $G(V, S)$, has two parameters n_j and p_j, as illustrated in Fig. 2.4(a). The parameter n_j represents the number of bits in the representation of the signal (excluding the sign bit), and the parameter p_j represents the displacement of the binary point from the LSB side of the sign bit towards the least-significant bit (LSB). Note that there are no restrictions on p_j; the binary point could lie outside the number representation, *i.e.* $p_j < 0$ or $p_j > n_j$.

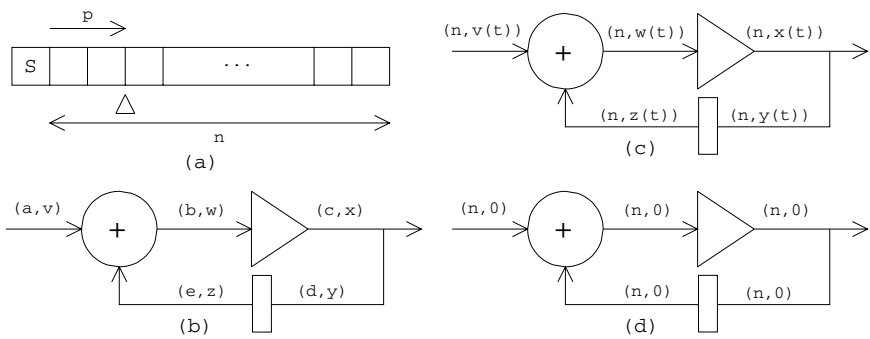


Fig. 2.4. The Multiple Word-Length Paradigm: (a) signal parameters ('s' indicates sign bit), (b) fixed-point, (c) floating-point, (d) multiple word-length

A simple fixed-point implementation is illustrated in Fig. 2.4(b). Each signal j in this block diagram representing a recursive DSP data-flow, is annotated with a tuple (n_j, p_j) showing the word-length n_j and scaling p_j of the signal. In this implementation, all signals have the same word-length and scaling, although shift operations are often incorporated in fixed-point designs, in order to provide an element of scaling control [KKS98]. Fig. 2.4(c) shows a standard floating-point implementation, where the scaling of each signal is a function of time.

A single uniform system word-length is common to both the traditional implementation styles. This is a result of historical implementation on single, or multiple, pre-designed fixed-point arithmetic units. Custom parallel hardware implementations can allow this restriction to be overcome for two reasons. Firstly, by allowing the parallelization of the algorithm so that different operations can be performed in physically distinct computational units. Secondly, by allowing the customization (and re-customization in FPGAs) of these computational units, and the shaping of the datapath precision to the requirements of the algorithm. Together these freedoms point towards an alternative implementation style shown in Fig. 2.4(d). This multiple word-length implementation style inherits the speed, area, and power advantages of traditional fixed-point implementations, since the computation is fixed-point with respect to each individual computational unit. However, by potentially allowing each signal in the original specification to be encoded by binary words with different scaling and word-length, the degrees of freedom in design are significantly increased.

An annotated computation graph $G'(V, S, A)$, defined in Definition 2.5, is used to represent the multiple word-length implementation of a computation graph $G(V, S)$.

Definition 2.5. An *annotated computation graph* $G'(V, S, A)$, is a formal representation of the fixed-point implementation of computation graph $G(V, S)$. A is a pair $(\mathbf{n}, \mathbf{p})$ of vectors $\mathbf{n} \in \mathbb{N}^{|S|}$, $\mathbf{p} \in \mathbb{Z}^{|S|}$, each with elements in one-to-one correspondence with the elements of S. Thus for each $j \in S$, it is possible to refer to the corresponding n_j and p_j. □

2.5 Summary

This chapter has introduced the basic elements in our approach. It has described the FPGAs used in our implementation, explained the desciption of signals using the z-transform, and the representation of algorithms using computation graphs. It has also provided an overview of the multiple word-length paradigm, which forms the basis of our design techniques described in the remaining chapters.