Kurt Baudendistel. “DSP Implementations of Speech Processing.”
2000 CRC Press LLC. <http://www.engnetbase.com>.

DSP Implementations of Speech Processing

Kurt Baudendistel
Momentum Data Systems

Implementations of digital speech processing algorithms in software can be distinguished from those resulting from general-purpose algorithms basically in the *type of arithmetic* and the *algorithmic constructs* used in their realization. In addition, many speech processing algorithms are realized with Programmable Digital Signal Processors (PDSPs) as the software development target—this leads to important considerations in the languages and paradigms used to realize the algorithms.

Although they are important topics in their own right, this section does not discuss the historical development of PDSPs to explain why these devices provide the architectural features that they do, and it does not provide a primer on PDSP architectures, either in general or in specific. Brief synopses of these topics are presented in the text, however, where they are appropriate.

49.1 Software Development Targets

PDSPs were developed as specialized microprocessors in the late 1970s in response to the needs of speech processing algorithms, and the vast majority of these devices have remained to this day basically audio-rate and, hence, speech processing devices [1]–[5]. These processors present a unique venue in which to both examine and implement speech processing algorithms since even a cursory examination of the device architectures quickly reveals the strong synergy between PDSP features and speech processing algorithms. As a result, specialized but restricted software development skills are necessary to realize speech processing algorithms on these devices.

Within the context of speech processing application realization, PDSPs which provide *fixed-point* data processing capabilities in hardware rather than floating-point capabilities are significantly more important. The simple reason for this is that fixed-point PDSPs are significantly less expensive than floating-point PDSPs but still provide the required computational capabilities for this class of applications. The fixed-point hardware capabilities are used to realize various *types of arithmetic* or *data abstractions* for the infinite-precision mathematical constructs used in algorithms. These abstractions include the well-known *integer arithmetic*, as well as various forms of *fixed-point arithmetic* that

use fixed shifts to control the scale of values within a computation and *block floating-point arithmetic* which performs run-time scale manipulations under programmer control.

General-purpose microprocessors also present an implementation medium that is well suited to many speech processing operations, although not one so well tailored to the task as that provided by PDSPs. All of the algorithmic structures presented here can be realized via microprocessors, and in fact many software libraries have been specifically designed to allow such realization [6]–[7].

49.2 Software Development Paradigms

As with general-purpose algorithms, a single software development paradigm cannot be described under which speech processing algorithms are always implemented. A small set of such paradigms do exist, however, and they are distinguished by just a few salient features.

Imperative vs. Applicative Language

Imperative programming languages specify a program as a sequence of commands to be performed in the order given. All of the familiar high-level programming languages, such as C, C++, or FORTRAN, as well as the assembly languages of most PDSPs, are imperative.

Applicative programming languages, on the other hand, describe a program via a collection of relationships that must be maintained between variables. Applicative languages intended to be programmed directly by the user such as Silage, SIGNAL, LUCID/Lustre, and Esterel, as well as the assembly language of data-flow PDSPs, can all be used to specify speech processing algorithms in a non-imperative manner, but their use to date in real applications is quite limited. And, although usually described as a hardware-description language and used as an intermediate language generated by other tools rather than directly by programmers, from the point of view of this discussion VHDL is an applicative language that can be used to describe speech processing algorithms directly.

Graphical programming environments such as Ptolemy, GOSPL, COSSAP, and SPW also provide an applicative "language" in which speech processing algorithms can be described. However, these environments universally rely on atomic elements that are programmed with a separate paradigm, usually an imperative one.

Most speech processing applications are implemented using imperative languages, and this programming model will be used here. Note, however, that the important distinguishing features of speech processing algorithms, arithmetic and algorithmic constructs, are applicable within any programming paradigm.

High Level Language vs. Assembly Language

Given that an imperative programming paradigm is to be used, the choice of a High Level Language (HLL) or assembly language as an implementation vehicle seems very straightforward [8]–[9]. The common wisdom holds that (1) assembly language should be chosen where execution speed is of the essence, in realizing "signal-processing kernels", since HLL compilers cannot produce object code of the same efficiency as can be obtained with hand-coded assembly language. However, (2) a high-level language should be used otherwise, in the realization of "control code", since this allows effective software development and the use of a *top-down* code development strategy.

In PDSP implementations, however, this sensible arrangement is often not possible. The reason for this is that use of a HLL compiler and run-time system makes untoward demands, relatively speaking, on an embedded system where resources such as registers, memory, and instruction cycles are quite scarce. In particular:

1. The settings in the control registers of the processor are often different between signal-processing and control code, and the device is more often than not " in the wrong mode".

2. The run-time memory organization demanded by a high-level language, typically including a stack on which automatic variables are to be allocated but lacking memory bank control, is one that most system designers are not willing to provide.
3. The standard function-call mechanism of a high-level language does not fit well with the customized register usage demanded in embedded systems programming.

Thus, more often than not, HLL programming is not currently utilized in PDSP systems. This will change, however, as PDSP HLL compilers become more sophisticated and as PDSP architectures become more "microprocessor-like".

Specialized vs. Standard High Level Languages

Specialized languages are often developed as dialects of standard high level programming languages by the authors of compilers. DSP/C, for example, is an extension of the C language that contains special vector and signal processing operations [10]. While they appear to be quite useful for target code development for speech processing applications, the lack of general support means that these languages are not often used for either algorithm or target code development.

Extensible languages, on the other hand, allow "dialects" of standard programming languages to be created by the end-user. C++ and Ada allow the construction of specialized arithmetic support via `class` and `generic` constructs, respectively. While these languages are quite useful for algorithm development, they generally cannot produce efficient realizations of the kind desired in target code for speech processing applications.

More often than not, when standard high level languages are used, they are simply augmented by libraries of operations. The Signal Processing Toolbox for Matlab™ and the Basic Operators for the C language used in standard speech codecs are good examples of these [6]–[7].

Block vs. Single-Sample Processing

Speech coding applications lend themselves quite well to *block processing*, where individual time-domain signal samples are buffered into vectors or *frames* [11]. This is often done for algorithmic reasons, as in LPC analysis, but significant performance gains can be realized by choosing this processing structure as well, when this is possible.[1]

Buffered data can be processed much more efficiently than single samples with typical PDSP architectures because the overhead associated with data transfer and instruction pipelining in these devices can be amortized over the entire vector rather than occurring for each sample. For example, the ubiquitous multiply-accumulate operation can be performed in a single instruction cycle by most PDSPs, but only *within the instruction execution pipeline*, meaning that overhead of several instruction cycles are required to set up for this level of performance. In single-sample processing, this instruction execution rate cannot be achieved.

Frames can be processed *in toto* or divided into *subframes* that are to be processed individually. This technique provides algorithmic flexibility without sacrificing the significant performance enhancement to be achieved with block processing.

Static vs. Dynamic Run-Time Operation

Two disparate philosophies on the operation of any real-time software system are particularly evident in speech processing implementations. *Static* and *dynamic* here indicate that run-time

[1] Not all algorithms can use block processing—modems and other signaling systems with very low delay requirements cannot. This technique is generally useful, however, for speech processing applications.

resource requirements, outlined in Table 49.1, can be computed and known at compile-time or only at run-time, respectively. Of course, some mix of these two philosophies can be found in any system, but the emphasis will usually be placed on one or the other.

TABLE 49.1 Static vs. Dynamic Operation

Resource	Static operation	Dynamic operation
Memory allocation	Global	Stack/heap
⟶ Address computation	Fixed	Stack-relative dynamic
Vector size	Fixed	Data dependent
Execution time	Fixed	Data dependent
⟶ Branch paths	Time-equivalent	Time-disparate
⟶ Wait-state insertion[a]	Must be computed	Can be ignored
Data transfer	Polling possible	DMA required
⟶ Fifo buffers	Not necessary	Required
⟶ Fifo overflow	Impossible	Possible
Operating system	Not typical	Typical

[a] Wait states may be inserted by an interlocked pipeline.

Exact vs. Approximate Arithmetic

The terms *exact* and *approximate* here refer to the concern on the part of the programmer as to whether the results produced by a given arithmetic operation are fully specified by the programmer in a *bit-exact* manner, or whether the best, approximate numerical performance that can be produced by a particular processor is acceptable [12]. For example, IEEE floating-point arithmetic is exact, while machine-dependent floating-point formats can be considered approximate from the point of view of a programmer porting code to that architecture from another. The most important form of exact arithmetic for speech processing applications is that provided by the Basic Operators, which are used in the C language specification provided as part of modern speech coding standards [6]–[7]. As a general rule, the integral and fractional fixed-point arithmetic forms, discussed in Section 49.4, can be considered exact and approximate, respectively.

Approximate arithmetic is much simpler to specify than exact arithmetic, but it is harder to evaluate. In the former case, implementation details are left up to the target architecture, but if the numerical performance of a particular realization does not meet some criteria, gross changes are required in the source code. The problem here is that the criteria are not defined as part of the source code and must be supplied elsewhere. Exact arithmetic, on the other hand, requires excruciating detail in the specification of the algorithm from the outset, but no evaluation of the realization is required since this realization must adhere to the specification.

Approximate arithmetic is the form promoted by the C language where, for example, the data type `int` does not define the precision of the integer or the results of operations that overflow.[2] It is also the form preferred by software developers working in a native code development environment.[3] Exact arithmetic, on the other hand, is preferred by developers who produce standards and who work in cross-code development environments because it eases the task of porting the algorithm from one environment to the other. Care must be exercised in this case, however, as any cross-development

[2] This is not to say that the C language cannot be *used* to realize exact arithmetic, which it often is through the machine-dependent declaration of data types such as `int16` and `int32`, but rather that the language was not *designed* for use with exact arithmetic.

[3] *Native* indicates that code for a particular processor is developed on that processor, while *cross* indicates that the host and target processors are different.

introduces inherent biases into an implementation that may be difficult or impractical to realize on a particular target processor [7].

It is well-known that a trade-off always exists between numerical and execution performance, as discussed in Section 49.4. It is not so well-known, however, that approximate arithmetic will always allow an equivalent or better balance to be struck in this trade-off than exact arithmetic. This is because the excruciating detail provided as part of an exact specification supplies not a minimum numerical requirement, but an exact one. In the case where a particular architecture can provide more precision than is specified, extra code must be inserted to remove that precision, resulting in less efficient execution performance. And, precisely because of this, an exact specification is in fact always targeted to a particular PDSP or microprocessor architecture—no algorithm can be specified in an exact manner and be truly portable or architecturally neutral.

49.3 Assembly Language Basics

Assembly languages for PDSPs are closely matched with the PDSP architecture for which they are designed, but they all share common elements [1]–[5]. In particular, multiple processing units must be programmed at the same time:

- adder
- multiplier
- fixed-point logic, such as shifter(s), rounding logic, saturation logic, etc.
- address generation unit
- program memory, for instruction fetch or data fetch
- data memories, perhaps multiple

In some cases, these units operate by default. For example, instruction fetches occur each machine cycle unless program memory is otherwise used. And in other cases, these units are utilized in combination. For example, (1) the DSP56000 multiply-accumulate instructions and (2) all address generation and memory fetch operations are indivisible and not pipelined. In all other cases, however, these processing units must be programmed within the *instruction execution pipeline* in which the outputs of one processing unit are connected directly to the inputs of another.

Coding Paradigms

Distinct coding paradigms are required by the architectures of various PDSPs, basically determined by the pipeline of that device, in order to perform this programming [13, 14]. Several assembly language forms are presented by PDSPs to realize these coding paradigms:

Data stationary coding specifies ultimately the *data* that is operated on by an instruction, but not the *time* at which the operation takes place—the latter is implicit in the form of the instruction. For example, the AT&T DSP32 instruction

```
*r0++ = a0 = *r1++ + *r2++
```
(49.1)

specifies the locations in memory from which the addends should be read and to which the sum should be written, but it is implicit that the sum will be written to memory in the third instruction cycle following this one.

Because of such delays, *illegal* and *erroneous* instruction combinations can be written that cause conflicts in the use of data from both memory and registers—the former can be detected by the assembler, but the latter will simply produce data manipulations different from those intended by programmer.

Time stationary coding specifies the operations that should occur at the *time* that this instruction is executed, while the *data* to be used is whatever is present in the "pipeline registers" at this time. For example, the AT&T DSP16 instruction

```
a1 = a0 + y        y = *r0++
```
(49.2)

specifies that a sum should occur at this time between the named registers and that a memory read should occur to the **y** register in parallel. No illegal or erroneous instruction combinations are possible in this case.

Interlocked coding solves the instruction combination problems of data stationary coding by automatically introducing extra machine cycles or *wait states* to ensure that conflicts do not occur. While this is convenient for the programmer, it does not produce more efficient execution than pure data stationary coding—on the contrary, it encourages programmers to be less savvy about their product.

Data flow coding is appropriate for machines that realize an applicative paradigm directly, such as the Hughes DFSP or the NEC μPD7281.

It must be pointed out that a mixture of these coding paradigms is often used in real PDSPs for control of different processing units. For example, the AT&T DSP16, while ostensibly a time-stationary device, utilizes a form of interlocking to allow multiple accesses to the same memory bank in a single instruction cycle [1].

Assembly Languages Forms

Within the four coding paradigms presented above, several assembly language forms can be utilized. First, either an infix form as given in Eq. 49.1 or the traditional assembly language prefix form using instruction mnemonics, as shown in Eq. 49.3 for the Motorola DSP56000, can be used:

```
clr  a
```
(49.3)

Second, the instruction may consist of a single field, as in Eq. 49.1 or Eq. 49.3, or it may contain multiple fields to be executed in parallel, as in Eq. 49.2 or Eq. 49.4:

```
mac     x0, y0,a     x:(r0)+,x0     y:(r4)+,y0
```
(49.4)

Note, however, that even within the multiple fields more than one operation is specified—in both Eq. 49.2 and Eq. 49.4 address register updates are specified along with the memory move. Pure *horizontal microcode,* in which a dedicated field in each instruction word controls a particular processing unit, is used in only a few modern PDSP architectures, but the multiple-field instructions are similar.

Additionally, all PDSPs contain "mode registers" which control operation of particular elements of the device. For example, the `auc` register of the AT&T DSP16 controls the multipler-shift, and thus the type of arithmetic realized by this processor's `p=x*y` instruction. Such mode registers, while prevalent and powerful in extending the effective instruction encoding space of a PDSP, are quite difficult to manage in large programming systems, especially in the design of function libraries.

49.4 Arithmetic

The most fundamental problem encountered during the implementation of speech processing algorithms is that the algorithm must be realized (1) using the finite-precision arithmetic capabilities of real processors rather than the infinite-precision available in mathematic formulae (2) under typically severe cost constraints in terms of the processing capabilities of the target system [15]–[17].

Any arbitrary level of arithmetic performance can be achieved by any processor, but the cost of this performance in terms of machine cycles can be prohibitive, and so an engineering trade-off is required.

Finite-precision arithmetic effects can be broadly classified as *representational* and *operational errors*:

- The bit pattern used to represent a finite-precision value can be of many forms, but all restrict the *range* of values over which a representation can be provided as well as the *precision* or number of bits used for the representation of a given value. No forms of arithmetic allow values outside the range to be represented, but some invoke an exception handler when such is requested. This is not appropriate in most speech processing systems, however, and in this case a finite-precision representation must be provided to approximate this value.
  The difference between an infinite-precision value and its finite-precision representation is the representational error, and there are two sources of such error: *truncation error* results from finite precision and *overflow error* results from range violations.
- Finite-precision operators used to transform values can also introduce error. In the case of simple arithmetic operators, this is equivalent to representational error, but it is often useful to conceptualize more complicated operators, such as an FIR or IIR filter, and to characterize the error introduced by that entity.

The engineering trade-off thus becomes an exercise in balancing the *numerical performance* of a realization of an algorithm in terms of truncation error and overflow error under the considerations introduced by possibly wide variance in input signal strengths or *dynamic range*, against implementation cost constraints in terms of target processor choice and available machine cycles on that processor. Because of the importance of this trade-off, it is important to examine different *types of arithmetic* and to evaluate the numerical performance and implementation cost of each type.

For example, floating-point arithmetic produces adequate numerical performance for most speech processing applications. However, the cost of floating-point processors is often prohibitive in dollar terms, and the cost of realizing floating-point arithmetic on a less expensive, fixed-point processor is prohibitive in terms of machine cycles. For this reason, some other type of arithmetic is often a better choice even though it may be numerically inferior and much harder to implement.

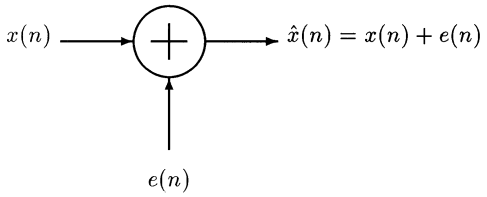
Regardless of the type of arithmetic chosen, however, it will be used in speech processing applications as a proxy or *abstraction* for the real-valued, infinite-precision arithmetic of mathematics. An important aspect that must be considered in evaluating finite-precision arithmetic types, then, is the effectiveness of the abstraction they provide for real-valued arithmetic. For example, all arithmetic needed for speech processing applications can be provided by integers, but determining what bit pattern to use to represent π or how to add two values of different scales can be quite difficult with this data abstraction.

Arithmetic Errors as Noise

Considering that most numerical values used in a speech processing algorithm are *signals*, in that they take on distinct values at distinct sample points, the difference between a finite-precision realization and the infinite-precision mathematical model on which it is based can be considered an *error* or *noise signal* that is injected into an algorithm at the point at which that arithmetic is used, as illustrated in Fig. 49.1. Given this model for the error as simply a noise source, finite-precision arithmetic effects can be analyzed in a manner similar to that used for other noise sources in a signal processing system.

An important corollary to this fact is that speech processing algorithms should be, and typically are, designed to be robust in the presence of arithmetic noise, just as they are designed to be robust in the presence of other noise sources.

The model for the noise that is injected at each point is a function of the type of arithmetic used in that operation, however. This noise model is an important element in understanding the motivation for using various types of arithmetic, and is presented where appropriate in the sections that follow.



$x(n)$:infinite-precision signal
$e(n)$:error signal
$\hat{x}(n)$:finite-precision signal

FIGURE 49.1: Noise model of arithmetic error.

Floating Point

A *floating-point number* consists of a sign bit, a mantissa, and an exponent, and it presents a well-known model for realizing an approximation to real-valued arithmetic, where the value of the number V is given by

$$V = M \cdot \beta^E \tag{49.5}$$

with β the radix of the representation, usually 2, and M and E the effective values of the signed mantissa and the exponent, respectively. A wide variety of floating-point formats exist, especially for PDSPs, of which the IEEE 754 Floating-Point Standard is the most widely utilized for general-purpose processors. These different formats are distinguished chiefly by the precision of the exponent and the mantissa, and the behavior of the arithmetic at the limits of the representable range.

Floating-point arithmetic is usually used only in applications in which such arithmetic capabilities are provided in hardware by the processor—it is not often simulated via software by a processor that provides only fixed-point arithmetic capabilities, but rather another, similar data abstraction is used. While quite powerful and easy-to-use, floating-point arithmetic is actually of little practical value in the realization of speech processing algorithms.

Block Floating Point

A *block floating-point representation* of a vector of length N of numbers $\bar{v}$ consists of a single signed, 2's complement integer of precision B_e representing the exponent e for the block computed

as[4]

$$e = \max_{i \in N} \lceil \log_2 (|v_i|) \rceil \tag{49.6}$$

along with an array of N signed, 2s complement fractions of precision $B_m = b_m + 1$ representing the mantissas m_i to which the exponent can be applied to yield the represented values $\hat{v}_i$ as

$$\hat{v}_i = m_i \cdot 2^{b_m - e} \tag{49.7}$$

The precision of the exponent B_e and of the mantissas B_m are almost always chosen as the word length of the target machine, yielding a *single-precision block floating-point vector*.

Arithmetic on the exponent and mantissas in a block floating-point representation are controlled separately, since significant savings in computation can often be supplied directly by the programmer. For example, if a block floating-point vector is to be computed as the result of a correlation, it is known that the zeroeth lag will produce the value with the largest magnitude, and so the exponent for the vector can be immediately determined. In the absence of such direct support from the programmer, block floating-point computations require either (1) that high precision results be saved in a temporary buffer to be scaled after all values have been computed and the maximum exponent found or (2) that all results be computed twice—once to determine the exponent and a second time to compute the mantissas.

An array of length L of block floating-point vectors of length N can be constructed, yielding a construct consisting of L exponents and $L \cdot N$ mantissa values. This *segmented block floating-point representation* allows better representation of values over a wide dynamic range than is available with a single exponent. It is also quite suited to applications in which a segment of values is known to be of one scale that can be quite different from that of neighboring segments.

In the limit with $N = 1$, (segmented) block floating-point yields the *scalar (segmented) block floating-point representation* which is quite like the well-known (vector) floating-point representation, except that normalization occurs only on demand. This is an appropriate representation to use for quantities of large dynamic range in speech processing applications realized on fixed-point processors where true floating-point would be prohibitively expensive.

Fixed Point

A *fixed-point number* consists of a field of $B = b + 1$ data bits that is interpreted as a binary, 2's complement number relative to a scale factor or *size* that is multiplied by the field to yield a value. The two basic forms of fixed-point numbers are the *integral* and *fractional* forms, in which the *justification* of the data bits within the field determines how the value of a bit pattern is interpreted:

Justification	Field	Size	Value	Range
Right	Integer i	Stepsize Δ	$\Delta \cdot i$	$\left[-\Delta \cdot 2^b, \Delta \cdot 2^b\right)$
Left	Fraction f	Fieldsize ϕ	$\phi \cdot f$	$[-\phi, \phi)$

Regardless of the representation, note that the stepsize Δ and fieldsize ϕ are always related as $\phi = \Delta \cdot 2^b$ for quantities of precision $B = b + 1$.

Among other possible fixed-point representations, *center-justified* or *mixed numbers* are quite rare in speech processing applications, and all other common representations are easily derived from the integral and fractional forms.

[4]This is a simplified exponent definition used for purposes of illustration. The actual value used in any particular implementation will be machine dependent, but this is of no consequence except as regards the point at which exponent overflow or underflow occurs, rare occurrences in most systems.

Given the basic machine word length or precision, usually 16 or 24 bits, fixed-point PDSPs universally provide signed, single-precision multiplication producing a double-precision product, along with double-precision addition, which allows numerically efficient computation of a sum-of-products. *Multiple-precision operations* of greater precision, discussed below, must be simulated in software.

The additive operators (addition, subtraction, negation, and absolute value) are equivalent for any fixed-point representation, with the caveat that only numbers of the same type can be combined with the binary additive operators. That is, only numbers of the same precision, form, and size can be added together directly—other combinations require conversion of one or both quantities to another, possibly a third, type before the operation can take place. Given this equivalency, it can be seen that it is the kind of multiplication, controlled by the shift that occurs at the output of the multiplier and the input to the ALU in all processors, that determines the type of arithmetic realized by a device, as shown in Table 49.2.

TABLE 49.2 Multiplier-Shift Determines Processor Type

Processor type	Shift[a]
Integral	0
Fractional	1
Biquadratic[b]	2
Summation[b]	-N

[a] This value is a relative one—the value zero could just as easily have been assigned to the fractional machine.
[b] These names derive from the use of this type of arithmetic in second-order IIR filter sections and long summations, respectively.

Fixed-point PDSPs abound with shifters—at the ALU inputs, the multiplier output, accumulator outputs, and perhaps within an independent barrel shifter. Because of a dearth of instruction encoding space, however, these are often fixed or controlled from mode registers rather than instructions or general registers, as discussed in Section 49.3.

The kind of multiplication realized by a processor also defines the kinds of data abstractions that are most useful given that machine architecture:

Q-notation is a natural extension of integer notation that is useful for right-justified arithmetic. A B-bit Qn fixed-point number is defined to have a binary point to the right of bit n, where bit 0 is the Least Significant Bit (LSB), yielding a stepsize $\Delta = 2^B$ a range $[-2^{B-n-1}, 2^{B-n-1})$. Multiplication is defined as producing a product with a precision and Q-value that are the sums of those of the multiplicands, respectively:

$$B_{x\star y} = B_x + B_y \tag{49.8}$$

$$n_{x\star y} = n_x + n_y \tag{49.9}$$

When precision is increased or reduced, it is naturally done on the left of a right-justified quantity, as with an integer. This seemingly simple operation is catastrophic when Q-notation is used to model real-valued arithmetic, however, since it produces overflow. Thus, precision must not be omitted at any point when Q-notation is in use—the term "a Qn number" should always be qualified as "a B-bit, Qn number".

Scaled fractions are a natural extension of fractions that are useful for left-justified arithmetic. A $b+1-$bit fractional number of fieldsize ϕ has a range $[-\phi, \phi)$, and multiplication is defined as producing a product with a precision and fieldsize that are the sum and product of those of the multiplicands, respectively:

$$b_{x\star y} = b_x + b_y \tag{49.10}$$

$$\phi_{x\star y} = \phi_x \cdot \phi_y \tag{49.11}$$

With this notation, biquadratic quantities can be seen to be simply scaled-fractions of fieldsize 2.0.

Precision is much less important for scaled-fractions than for Q-values. This is because increasing or reducing the precision of a left-justified quantity naturally occurs on the right, which simply raises or lowers the accuracy of the representation. Thus, while important as regards numerical performance, precision is not required in describing a quantity as "a scaled-fractional of fieldsize ϕ".

It should be pointed out that use of a right-justified data abstraction on a left-justified machine, or vice versa, is quite difficult.

Reduction describes the common response to overflow in fixed-point additive operations, where a sum is simply allowed to "wrap around" in the 2's complement representation:

$$\texttt{x+y} \equiv \mathrm{sgn}(x+y) \cdot [(|x+y| + \phi) \bmod 2\phi - \phi] \tag{49.12}$$

Saturation describes an alternate response to overflow where the result is set to the maximum representable value of the appropriate sign:

$$\texttt{x+y} \equiv \begin{cases} \phi - \Delta & x + y \geq \phi \\ x + y & -\phi \geq x + y < \phi \\ -\phi & x + y < -\phi \end{cases} \tag{49.13}$$

The bit patterns that result from saturation are `0x7f...f` and `0x80...0` in the cases of positive and negative overflow, respectively. Fixed-point PDSPs typically provide hardware to realize saturation because it gives a significant boost to the numerical performance of many speech processing algorithms in the presence of overflow. In most cases, when reduction arithmetic is in use *no* overflow can be tolerated, even in extremely unlikely situations, while *some* overflow can be tolerated with saturation arithmetic in most algorithms.

General-purpose microprocessors traditionally provide only a single overflow-detection bit. Fixed-point PDSPs, on the other hand, typically provide $N > 1$ overflow bits for each register that can be the destination of an additive operation in the ALU, usually termed *accumulators*. This feature allows summations of up to 2^N terms to be performed while the result can be saturated correctly if overflow does occur. The overflow bits are alternately called *secondary overflow bits*, *guard bits*, or *extension words* by different manufacturers.

For summations involving more than 2^N terms, it is often useful to determine if overflow occurred during the summation, even though enough information to saturate the result is not available—this capability is also required for the support of block floating-point operations. *Sticky* or *permanent* overflow bits are set when overflow occurs, but they are only cleared under programmer control, allowing such overflow detection. And, it is sometimes useful to provide such permanent overflow detection at a saturation value other than the range, as noted in Section 49.5.

Another option in the case of summations involving more than 2^N terms is to scale the inputs to the summation and then perform saturation at the end of the summation during a *rescaling* operation. As with all scaling operations, however, this one trades off overflow error for truncation error, and

it may introduce unacceptable noise levels. For example, in the case of a summation of K i.i.d. Gaussian random variables, prescaling introduces a $3\lceil \log_2 K \rceil$dB SNR degradation relative to an unscaled summation.

The nature of fixed-point PDSPs as single-precision multiply/double-precision add machines means that conversions between single- and double-precision quantities is quite common. *Extension* from single- to double-precision always takes place on the right for fixed-point quantities, except in the rare cases where integers are involved, and the extension is always with zeroes. Conversion from double- to single-precision, however, can be performed by *truncation* where the extra bits are simply removed,

$$(\ \texttt{x}\ \&\ (\ (\ \texttt{-1}\ )\ <<\ B) \tag{49.14}$$

where B is the basic machine precision, or by *rounding*:

$$(\ (\ \texttt{x}\ +(\ \texttt{1}\ <<\ B-1))\ \&\ (\ (\ \texttt{-1}\ )\ <<\ B) \tag{49.15}$$

Fixed-point PDSPs typically provide hardware to realize rounding because it gives a significant boost to the numerical performance of many speech processing algorithms. In most applications, it can be safely assumed that the low bits of the 2s complement value that are removed as part of a conversion operation are neither deterministic nor correlated and that they represent values that are uniformly distributed over the range $[0, \Delta)$. In this case, rounding produces errors that statistically are approximately zero-mean, while truncation produces errors with mean $\mu \approx \frac{1}{2}\Delta$, and this *bias error* can be significant in many situations.

Multiple-precision operations can be simulated in software in many ways, but usually one or more of the following formats is used to represent them:

Native format represents double- and higher-precision numbers as simply the appropriate bit pattern broken into multiple machine words. High-precision additive operations can be directly realized in this format using a carry flag, but multiplication of such quantities requires unsigned multiplication capabilities, which are lacking in most PDSPs and many general-purpose processors.

Double precision format (DPF) allows double-precision fractional multiplication, with double-precision inputs and double-precision output, to be realized using signed multiplier capabilities by representing a double-precision value as the concatenation of the high-order word with the low-order word logically right-shifted by one bit.

Double round format (DRF) allows double-precision multiplication, both integral and fractional, to be realized using signed multiplier capabilities by representing a double-precision value as the concatenation of the high-order word that would result from rounding the double-precision quantity to single-precision, with the original low-order word.

These representations are illustrated in Table 49.3.

TABLE 49.3 Double-Precision Formats

Format	High word	Low word
Native	`0x89AB`	`0xCDEF`
DPF	`0x89AB`	`0x66F7`
DRF	`0x89AC`	`0xCDEF`

49.5 Algorithmic Constructs

The second major distinction between implementations of digital speech processing algorithms and general-purpose algorithms concerns the algorithmic constructs used in their realization, and the most important of these are discussed below.

Delay Lines

Delay lines, which allow the storage of sample values from one operational cycle to the next, are an important component of speech processing systems, and they can be realized in a variety of ways with PDSPs:

Registers, including implicit pipeline registers, can be used to effectively realize short delays, including the one- and two-tap delays required in IIR filters.

Modulo addressing causes an address register to "wrap around" within a defined range to the start of a buffer when an attempt is made to increment that register past the end of the defined range. A delay line can be realized using modulo addressing by utilizing the location containing the expired data at a given step for the new data and by bumping the address register accordingly.
Most PDSPs do provide modulo-addressing capabilities, but often in only a limited manner. For example, strides greater than one or negative strides may not be supported, and the buffer may require a certain alignment in memory.

Writeback causes a delay line element to be written back to memory at a new location after it is read and used in a computation. While this technique is quite powerful as regards the rearrangement of data in memory, it is quite expensive in terms of memory bandwidth requirements.

These techniques are most useful for fixed delays, but equivalent methods can be used to realize variable delays.

Transforms

Modern PDSPs provide specialized support for transforms, and inverse transforms as well, especially the radix-2 FFT. This can include the ability to

- Compute both a sum and a difference on the same data in parallel.
- Compute addresses using *reverse-carry addition,* where the carry propagates to the right rather than to the left as in ordinary addition. This allows straightforward computation of the bit-reversed addresses needed to unscramble the results of many transform calculations.
- Detect overflow in fixed-point computations at a point other than the saturation point. This can be used to predict that overflow is likely to occur at the current transform stage based on the output of the previous stage *before computation of the current stage begins.* With this capability, the data can be scaled as part of the current processing stage if and only if it is necessary, efficiently producing an optimally scaled transform output.

Vector Structure Organization

Vectors of atomic components are always laid out simply as an array of the elements. When the components are not atomic, however, as with segmented block floating-point or complex quantities, an alternative is to organize the vector as two arrays: an exponent array and a mantissa array for segmented block floating-point quantities, or a real array and an imaginary array for complex quantities.

The choice of *interleaved* or *separate* arrays, as these two techniques are known, is a trade-off between resource demands, in terms of the number of address registers needed to access a single element, vs. flexibility, in terms of the order of access and stride control that is possible.

Zipping

Zipping is a generic term that is used to refer to the process of performing a sequence of multiply-accumulate operations on input arrays to realize the signal processing tasks of scaling, windowing, convolution, auto- and cross-correlation, and FIR filtering. The only real difference between these conceptually distinct tasks is (1) the choice of data or constant input arrays and (2) the order of access within these arrays.

PDSPs are designed to implement this operation, above all others, efficiently—their performance here is what distinguishes them most from general-purpose and RISC microprocessors. Regardless of the coding paradigm used,[5] all PDSPs allow in a single instruction cycle the following:

- two memory accesses, either data-constant or data-data
- two address register updates
- a single-precision multiply
- a double-precision accumulate

Programming contortions are often required to achieve this throughput in the face of processor limitations and memory access penalties, but the holy grail single-cycle operation is always attainable.

Mathematical Functions

As in general-purpose programming, higher level mathematical functions can be realized within speech processing applications in one of three ways:

Bitwise computation can be used to build an exact representation one bit at a time. This technique is often used to implement single-precision division and square root functions, and some PDSPs even include special iterative instructions to accomplish these operations in a single cycle per output bit. For example, unsigned division can be realized for the Motorola 56000 as follows:

```
and   #$fe,ccr   ;  Clear quotient sign bit
rep   #24        ;  Form 24 bit quotient,          (49.16)
div   x0,a       ;    ...  one bit at a time.
```

Approximate computation, such as Newton's method, is often used to produce a double-precision result from a single-precision estimate.

Table lookup is often used, along with linear interpolation between sample points, especially for trigonometric, logarithmic, and inverse functions. Several PDSPs even include the necessary tables in ROM.

Looping Constructs

Loop counting can be done with general registers, and this is required in deeply nested loops, but hardware support is often provided by PDSPs for low- and zero-overhead loops. *Low-overhead loops* utilize a special counter register to realize a branching construct similar to the well-known decrement-and-branch instruction of the Motorola 68000 microprocessor. They are "low overhead"

[5] Coding paradigms are discussed in Section 49.3.

in that the cost of the loop is typically only that of the branch instruction per iteration—separate increment (or decrement) and test instructions are not needed. *Zero-overhead loops* go one step further and eliminate even the cost of the branch instruction per iteration. They do this via special-purpose hardware to perform the program counter manipulations normally handled in the branch instruction. There is an overhead cost at the start of the loop, but the cost per iteration is truly zero.

Loop reversal is an important concept that often allows more efficient coding of speech processing constructs. In its simplest form, a loop counter is run backward to allow more efficient counting of iterations, or an address register is run backward to allow it to be reused without having to reinitialize it. In both of these cases, the reversal of the counter or address register is only possible when there are no dependencies from one loop iteration to the next. More powerful, however, is to perform memory access via a temporary register to allow loops that need to run in one direction for algorithmic reasons to be coded in the opposite direction. This technique can be used to exploit the pipelined nature of PDSPs to great effect.

References

[1] AT&T Microelectronics, *DSP1610 Digital Signal Processor Information Manual*, 1992.

[2] Motorola, Inc., *DSP65000 Digital Signal Processor User's Manual*, 1990.

[3] Texas Instruments, Inc., *TMS320C25 User's Guide*, 1986.

[4] Analog Devices, Inc., *ADSP-2100 User's Guide*, 1988.

[5] NEC, Corp., *NEC μ PD7720 User's Manual*, 1984.

[6] *Draft Recommendation G.723 – Dual Rate Speech Coder for Multimedia Telecommunication Transmitting at 5.3 & 6.3 kbit/s*, International Telecommunication Union Telecommunications Standardization Sector (ITU) Study Group 15, 1995.

[7] *Draft Recommendation G.729 – Coding of Speech at 8 kbit/s using Conjugate-Structure Algebraic-Code-Excited Linear Prediction (CS-ACELP)*, ITU Study Group 15, 1995.

[8] Chassaing, C., *Digital Signal Processing with C and the TMS 320C30*, John Wiley & Sons, New York, 1992.

[9] Baudendistel, K., Code generation for the AT&T DSP32, in *Proc. ICASSP-90*, 1073-76, Apr. 1990.

[10] Leary, K. and Waddington, W., DSP/C: A standard high level language for DSP and numeric processing, in *Proc. ICASSP-90*, 1065-68, Apr. 1990.

[11] Sridharan, S. and Dickman, G., Block floating-point implementation of digital filters using the DSP56000, *Microprocessors and Microsystems*, 12, 299–308, July/Aug. 1988.

[12] Baudendistel, K., Compiler Development for Fixed-Point Processors, Ph.D. thesis, Georgia Institute of Technology, 1992.

[13] Madisetti, V. K., *VLSI Digital Signal Processors, An Introduction to Rapid Prototyping and Design Synthesis*, Butterworth-Heinemann, 1995.

[14] Lee, E.A., Programmable DSP architectures: Parts I & II, *IEEE ASSP Magazine*, 5 & 6, 4–19 & 4–14, Oct. 1988 & Jan. 1989.

[15] Oppenheim, A.V. and Schafer, R.W., *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[16] Jackson, L., Roundoff-noise analysis for fixed-point digital filters realized in cascade or parallel form, *IEEE Trans. Audio and Electroacoustics*, AU-18, 102-22, June 1970.

[17] Parks, T.W. and Burrus, C.S., *Digital Filter Design*, John Wiley & Sons, New York, 1987.