

Overview of the ADSP-21K Optimized DSP Library

2.1 Manual Contents

The ADSP-21K Optimized DSP Library is a high performance scalar math, vector and digital signal processing library containing over 385 hand-optimized routines for the Analog Devices ADSP-21K family of Digital Signal Processors. The routines are organized into 50 functional categories. The documentation for each function (contained within Chapter 5) includes the name, description, algorithm, call synopsis, domain, accuracy, execution times, and any applicable notes. The functional categories for the ADSP-21K Optimized DSP Library functions include:

- Scalar Power Functions
- Simple Scalar Trigonometric Functions
- Simple Scalar Functions
- Scalar Inverse Trigonometric Functions
- Scalar Hyperbolic Functions
- Scalar Inverse Hyperbolic Functions
- Scalar Fix, Float & Truncation Functions
- Simple Functions

- Power Functions
- Simple Trigonometric Functions
- Arc Trigonometric Function
- Hyperbolic Function
- Inverse Hyperbolic Functions
- Averaging and Summing Functions
- Comparison Functions
- Vector Fix, Float and Truncation Functions
- Limiting Functions
- Logical Functions
- 2-Vector & 1-Scalar Functions
- 2-Vector & 2-Scalar Math Functions
- 3-Vector Math Functions
- 3-Vector & 1-Scalar Functions
- 4-Vector Math Functions
- 5-Vector Math Functions
- Maximum/Minimum Functions
- Gather/Scatter Functions
- Comparison Functions
- Conversion Functions
- Other Functions
- Complex Functions
- Complex Filtering Correlation Functions
- Complex Vector Magnitude Functions
- Complex Conversion Functions
- Complex Vector Simple Functions
- Complex Vector Fundamental Functions
- Complex Vector Real Vector Functions
- Complex Vector Conjugation Functions
- Complex Vector Complex Scalar Functions
- Convolution Functions
- Correlation Functions
- Accumulating Spectrum Functions
- Filtering Functions (Both FIR & IIR)
- Windowing Functions
- FFT Functions
- Integration Functions
- Distribution Generation Functions
- Real and Complex Matrix Functions
- Statistical Functions

- Compander Functions
- Misc. Memory and Checksum Testing Functions

2.2 Definition of a Vector Routine

We define vectors as aggregates of data of the same type (of type complex, real or integer) which typically relate to a phenomenon or problem you are attempting to analyze. If you compute the arithmetic mean of a series of numbers, for example, you are performing a vector operation, as all data points for a defined array are being summed and divided by the number of total data points, **n**, to compute the average.

For purposes of the Wideband Optimized DSP Library, vectors are considered to be composed of floating-point elements, integer elements, or unsigned integer elements. From these elemental types Wideband engineers have constructed a variety of efficient routines to perform scalar and vector functions. Later, an aggregate of floating-point numbers are used to define complex data types, which are extended to define complex vector operations.

Throughout all 21K Optimized DSP Library routines, numbers are held in memory as a vector (which we define as an array of numbers held in consecutive memory locations) or a scalar (which we define as one memory location for real numbers, and 2 consecutive memory locations in the case of a scalar complex number).

2.3 Vector Versus Scalar Operations

Vector operations are designed to perform repetitive tasks on a large grouping of data. This is opposed to performing a scalar operation, where a routine is called once, supplied with one argument (typically) and produces one result.

For instance, the cosine routine, called **cos**, accepts one input argument each time the routine is called. It will produce one answer for each argument supplied, but the routine must be repetitively called to compute multiple cosines. This is inefficient when computing more than 1 cosine.

Alternately, the vector version of the cosine routine, called **vcos**, also accepts one argument and produces one answer. The difference is that the **vcos** vector routine can accept an array of input arguments which produces one answer each traversal through the code.

The two routines differ in that the **vcos** vector routine is specifically built to calculate multiple iterations of the cosine, as the code has an optimized cosine execution loop built into its core. The scalar routine, **cos** is built to be called only once, and traversed only once per call.

2.4 Minimum Array Sizes To Be Provided to Routines

All the routines in the Optimized ADSP-21K DSP Library (exclusive of the scalar routines) are optimized to perform repetitive operations on large groups of data. However, note that due to the extensive use of parallel instructions within certain sections of code, some routines require at least 2 data points to avoid problems associated with filling and draining the ADSP-21K processor instruction pipeline. Consult the "Minimum Vector Size" Column shown in the benchmark tables in Chapter 6 to ascertain the minimum vector argument size needed to execute a specific routine. In many cases, an array of size of length 1 is acceptable, while in other cases the array should be at least of length 2.

2.5 Optimization Principles

The key to the ADSP-21K Optimized DSP Library's speed is based on a number of principles:

- A special assembly language instruction called LCNTR or Loop Counter is implemented in all vector routines so that no overhead becomes associated with repetitive traversals of a main loop outside of the initial cycles necessary set up the zero overhead loop.
- Parallel instructions are used whenever possible. Current compiler technology has not yet reached the level of optimization inherent in a careful, hand assembled code. Part of this is due to the difficulty of building a compiler which takes full advantage of the ADSP-21K native parallel instruction set. Implementing vector operations in assembler code demands an implementation that avoids pipeline delays and illegal use of registers.
- Hand optimized implementations, as found in the 21K Optimized DSP Library, contains sophisticated loop and branching structures which even the most intelligent compiler would be hard pressed to match. This results in performance increases which may be up to 200% faster for involved algorithms over the same code written in fully optimized C.
- Careful coding techniques insure that the processor pipeline is always loaded with instructions necessary to perform a computation. Techniques are implemented which avoid the use of inefficient constructs and unnecessary branching.
- The inner loops are kept to a minimum size to by careful algorithm design techniques.

2.6 Basic Striding Concepts

Vector operations typically work on large aggregates of data. Most of the time, an end user will choose to perform a given operation on all data points received. However, sometimes a user may wish to work with only selected portions of data.

In such cases, the user would instruct the computer to skip over, or ignore, a desired portion of data, and only read every *n* data points. This skipping of data is called *strid-*

ing. We define this as a controlled way to skip reading (thus operating on) selected memory locations (data) without having to first process data before discarding it. In this manner, precious CPU cycles are saved as specific data is designated as not needing processing, and may be ignored.

All vector orientated routines in the Wideband ADSP-21K Optimized DSP Library (designated as having arguments such as **a**, **b**, **c**, **d**, and **e**, etc.in the function calls), have strides associated with them. These strides are typically indicated by the designators **i**, **j**, **k**, **l**, and **h**, in the argument synopsis. For instance, the routine **vadd** adds the elements of array **a** and array **b** and places the results in array **c**. The **vadd** routine is typically called as follows:

vadd (a , 1 , b , 1 , c , 1 , 6)

The integer following each array representation (**a**, **b**, **c**) represents the stride length assigned to the preceding array. In the case of the **vadd** example just shown, the stride length is 1. The preceding call states that the **vadd** routine is to add every corresponding element in input vector **a** to every corresponding input element in vector **b** and place the sum in the corresponding memory location defined by the index in array **c**. This is demonstrated as follows:

TABLE 2

Vector Add - Vadd (a, 1, b, 1, c, 1, 6) - All Strides Equal

Index Of Array	Vector a Value	Stride i Value	Vector b Value	Stride j Value	Vector C Value	Stride k Value
0	3	1	4	1	7	1
1	6	1	8	1	14	1
2	3	1	5	1	8	1
3	7	1	3	1	10	1
4	2	1	5	1	7	1
5	9	1	7	1	16	1

This, however, can be changed. If we wished to skip a portion of input data and only sample at 1/2 the rate, (decimating by 2) this could be accomplished by skipping every other data point. In effect, we would leapfrog the odd samples. We could do this with the following call to **vadd**:

vadd (a , 2 , b , 2 , c , 2 , 6)

The preceding call states that the **vadd** routine is to add every other element in input vector **a** to every other item in input vector **b** and place the sum in every other position in output array **c**. In effect, only half the data points will be sampled. Six represents the number of data samples to be considered by the routine. This can be demonstrated as

follows (with the shaded boxes indicating which elements are used as both input and output values):

TABLE 3

Vector Add - Vadd (a, 2, b, 2, c, 2, 6) - All Strides Equal But 1/2 Data Skipped

Index Of Array	Vector a Value	Stride i Value	Vector b Value	Stride j Value	Vector C Value	Stride k Value
0	3	2	4	2	7	2
1	6	2	8	2	0	2
2	3	2	5	2	8	2
3	7	2	3	2	0	2
4	2	2	5	2	7	2
5	9	2	7	2	0	2

We could perform one final optimization on this routine by assigning the output vector **c** a stride of 1. This avoids filling half the memory of output array **c** with zeroes for those memory locations that are not assigned results. The call would be as follows:

vadd (a, 2, b, 2, c, 1, 6)

This would be demonstrated as follows:

TABLE 4

Vector Add - Vadd (a, 2, b, 2, c, 1, 6) - All Strides Not Equal And 1/2 Data Skipped

Index Of Array	Vector a Value	Stride i Value	Vector b Value	Stride j Value	Vector C Value	Stride k Value
0	3	2	4	2	7	2
1	6	2	8	2	8	2
2	3	2	5	2	7	2
3	7	2	3	2	0	2
4	2	2	5	2	0	2
5	9	2	7	2	0	2

In this case, the output vector **c** has results shown in array positions 0, 1 and 2 even though the values in vector **a** and **b** positions 0, 2, and 4 are used as the input arguments.

You may stride through arrays in any stride increment you choose as long as the size of your stride is not larger than the size of its associated array. For instance, you may not specify that you expect to process 10 data points by setting **n** equal to 10 and then specify 11 as any one of the striding arguments. Also, the integer representing the stride value must be within the valid range of single precision integers acceptable to the ADSP-21K processor: $-2^{31} \leq x \leq 2^{31}-1$

2.7 Introduction To The Trigonometric Functions

Trigonometric functions are used throughout signal processing, engineering and scientific applications. Indeed, it is often impossible to solve a scientific or engineering problems without utilizing a trigonometric relationship of some type. However, due to the need for accuracy and speed, what once began thousands of years ago as the simple calculation of the ratios of the sides of a right triangle has evolved in the modern era of computing into a fine art form.

Trigonometric functions are historically defined as mathematical expressions representing the ratios of the sides of a right triangle. From the definition of the sides of a right triangle, the trigonometric functions representing the acute angles were defined: sine, cosine, tangent, cosecant, secant, and cotangent. With the advent of the concept of the inverse function, the hyperbolic functions and the power functions (log, power, log2, log 10, \exp^{10} , \exp^2 , etc.) the number of functions expressing the periodic functions quickly expanded.

There are multiple ways to compute these relationships. Most lay persons are used to conceiving of these computations as little more than function buttons on a scientific calculator. The more mathematically sophisticated users, such as an average engineer, has had enough exposure to calculus to know that the trigonometric functions can be computed using a power series. For instance, in calculus, we learn that the sin function can be computed as a Maclaurin's series expansion as follows:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad \text{where } (|x| < \infty)$$

For a computer however, this is a poor method to calculate the sine function as the algorithm will converge slowly on the answer. Also, the accuracy of the function is dependent on the number of terms in the polynomial, with a greater number of terms resulting in a more accurate answer.

2.8 Introduction To The Approximation Process

Fortunately, the advent of digital computers in the later 1940s stimulated research into the problem of computing these function using numerical approximations. In the late 1950s and the early 1960s much research was conducted at the Los Alamos and Argonne National Laboratories, and at the National Bureau of Standards into the methodologies for numerical approximation of these functions. The result of these research studies was the development of a series of approximations, which are, in reality, simple n degree polynomial equations with coefficients for each factor which are used to compute, to a given degree of accuracy, an approximation of a particular function.

These approximations are polynomials which mimic, if you will, the mathematical behavior of the function over a defined input range, called the **domain**. Each approximation has a number of coefficients associated with it. Generally, the greater the degree of accuracy required by the user, the larger the number of coefficients, and the longer the algorithm will take to execute. For example, the sine function can be approximated

using the following rational approximation created by Carlson and Goldstein at Los Alamos Laboratories in 1955:

$$\sin x = x \left(1 + a_2 x^2 + a_4 x^4 + a_6 x^6 + a_8 x^8 + a_{10} x^{10} \right) \quad \text{where } 0 \leq x \leq \frac{\pi}{2}$$

TABLE 5

Carlson and Goldstein Sine Function Approximation Coefficients

$a_2 = -0.49999999963$
$a_4 = 0.0416666418$
$a_6 = -0.0013888397$
$a_8 = 0.0000027526$
$a_{10} = -0.0000000239$

where the answer is accurate to $\pm 2 \times 10^{-9}$

Note that the function has a limited domain (e.g., a range of valid arguments you can supply to the function) and a limited accuracy ($\pm 2 \times 10^{-9}$). Generally speaking, if more accurate results are demanded, coefficients can be added to the approximation and a longer polynomial generated to increase the precision of the computation. If, however, execution speed is of primary importance and accuracy is less importance, the number of approximation coefficients can be limited, resulting in an algorithm which executes faster but returns less precision.

If both accuracy and speed are of primary importance, algorithms can be developed which utilize lookup tables within processor memory to initiate fast computation of desired results. However, this approach uses valuable on-board memory which may be needed and reserved for other processes. Contact us if you have special needs of this type and we can advise you on how to proceed in your development effort.

2.9 Factors Contributing to Loss of Precision

The Wideband routines are designed to return 7.75 digits of precision internal to the routine. However, in some cases, slight lose of precision may be encountered when users enter very large numbers into the routines. In general, the accuracy of the final answer produced by an approximation is always dependent upon the approximation chosen to implement the function, the architecture of the processor, and the magnitude of the argument supplied to the routine. These 3 factors represent the key sources of loss of precision in answers returned from routines. Since you have control over only one factor (magnitude of the argument) let's take each source and examine how it can contribute to loss of precision so that you can carefull avoid some of the obvious pitfalls:

2.9.1 Processor Architecture

The SHARC family of DSP processors offers 32 bits which can be used to represent a floating-point number. In this scenario, 24 bits are used to represent the mantissa and 8

bits to represent the exponent. This means that a large number with over, say 7.5 to 8 digits of precision, cannot be accurately represented within the machine without starting to lose the last digit of precision. The reason is that more than 24-bits are needed internally to accurately maintain over 8 digits of precision.

2.9.2 Magnitude of Supplied Argument

The architecture limitations of the SHARC processor described above means that large numbers supplied to transcendental routines are not represented accurately within the machine, and hence the precision of the results of the routine should take this into account. As the number represented rises in magnitude, the machine is less able to represent the intended number perfectly. The result is a less precise answer

2.9.3 Polynomial Approximation and Reduction

All trigonometric functions computed on DSPs are based on the computation of approximations. An approximation is essentially a mathematical formula which consists of a polynomial formula (and sometimes a rational formula), a series of data points to be plugged into the polynomial formula, and a set of mathematical conditions which state that in order for the end user to get accurate returns, the argument must obey certain rules.

For instance, take a moment to examine the sample source code sine routine in Appendix A. Note that the routine computes a polynomial expansion of any argument supplied to the routine and returns an accurate answer. To do this, the user has to make sure that the number supplied to the routine falls within the domain of the -1.0×10^9 to $+1.0 \times 10^9$.

The input argument is then reduced by the internal logic of the algorithm so that its phase is unlocked. Since the transcendental functions are defined as relationships between sides of a right triangle, the ideal scenario is to present the routine with an argument which does not contain needless "extra" data.

For instance, in the sine routine, the very definition of the sine function places the domain of the function to be $0 \leq \text{argument} \leq 2\pi$. The sine starts at 0, reaches 1 at 90 degrees, is back down to zero at 180 degrees, reaches -1 at 270 degrees and is back to 0, at the start of the unit circle at 360 degrees. This is, in effect a nice way of saying that any number larger than 6.28318530718 or less than or equal to zero is excessive (or shall we say not important to the functioning of the algorithm), as we have already gone around the unit circle once.

The argument must therefore be reduced so that the true "significant" data as related to the ratios of relationships of sides of a right angle triangle are computed, as the definition of the transcendental functions relate to one time around the unit circle only. For example, if an end user were to supply a argument of 471,238.9804 to the sine routine, the end result should be the same as supplying the argument 1.5707963268. the result will be the answer of 1. The difference between the two is just the number of excessive times we have moved around the circle. Both arguments represent the sine of 90 degrees.

We have discussed the concept of reduction in detail as this is one of the keys to the computing accurate transcendental function. All Wideband routines reduce the arguments supplied to the routine (based on the domain listed for the function) to an acceptable interval which the approximation needs in order to guarantee that it can provide accuracy which it specified.

This is all transparent to the end user, as this takes place automatically in the routines themselves. For instance, when computing the sine, as shown in Appendix A, the incoming argument supplied to the routine is reduced from the domain of -1.0×10^9 to $+1.0 \times 10^9$ to an interval of 0 to $\frac{1}{\pi}$. This reduced argument, which represents the distilled essence of user's argument, is then passed to a polynomial (or sometimes a rational) approximation.

This approximation uses the distilled or reduced argument in a series of multiply and adds (and sometimes division) using a defined set of coefficients at the end of the routine in order to calculate the answer. This series of multiply and adds is called a Horner's expansion. In this manner the routine computes the sine. All transcendental routines examine the incoming argument to the routine, reduce the argument if necessary to the associated interval defined within the algorithm, compute the minmax results and then re-expand the result.

We state in the User's Manual that we calculate the transcendental routines to 7.75 digits of precision. What we are saying here is that the approximation coefficients and their associated domain reduction range which we have chosen to compute the transcendental function, (which were based on the work of Cody and Waite, Hart, and Abramovitz and Stegum), were chosen in such a manner as to attempt to reach 7.75 digits of precision for a given argument.

For instance, the 4 coefficients associated with the sine routine in reading the code closely discloses that the reduction range is 0 to $\frac{1}{\pi}$. This particular set of coefficients and their range reduction requirements were taken from Cody and Waite, which recommended this set for routines where the maximum precision of the mantissa is less than or equal to 24.

2.9.4 Implications

Why are discussing this in detail? The answer is that often times you supply an argument to a function which computes a transcendental function and the results loose their precision as the answer gets further from zero. This is directly related to the inability of the SHARC to represent large number accurately and the subsequent loss of precision. This means that the polynomial approximations and reduction processes start their computations with an argument which is not a precise representation of the original argument. This means that a valid input within a domain may not return a perfect answer if length of the argument exceeds 8 digits of precision.

2.9.5 References Used

The approximations used to implement the trigonometric functions with the ADSP-21K Optimized DSP Library were chosen from the classic work by William Cody and Will-

iam Waite entitled Software Manual for the Elementary Functions. Further research was done using the work of John Hart's Computer Approximations and Abramovitz and Stegun's Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Functions. The approximations were chosen in such a manner as to be congruent with the 32-bit architecture of the SHARC and its lack of a microcoded or native full precision reciprocal instruction.

2.10 Important Relationships When Computing Trigonometric Approximations

Although the exact approximation shown in Table 5 is not utilized in the Wideband library, it is useful for demonstrating a number of salient points regarding the computation of trigonometric functions:

- Since most libraries never disclose the exact polynomial used to compute the function, you should pay special attention to the benchmark accuracy claims provided by the developers. In our case, the benchmarks are conservative. The Wideband library maintains 7.75 decimal digits of precision throughout its calculations, which is consistent with the 32-bit calculation model.
- The speed at which the function computes the final answer is a function of the number of coefficients and the number of degrees of the polynomial used to approximate the function. Generally speaking, the greater the degree of the polynomial used and the greater the number of coefficients, the slower the routine will execute, but the more accurate the answer will be. The converse also holds true: generally speaking, the smaller the number of coefficients the faster the routine will execute and the less accurate the routine's answer will be. Beware of vendors who promise both speed and accuracy without suggesting that one affects the other. In the case of the Wideband routines, when presented with a scenario where speed must be balanced against accuracy, we have opted for accuracy.
- All trigonometric approximations must be chosen such that the approximation used to implement the function is fitted to the processor architecture. For instance, in the case of the Wideband trigonometric functions, we have chosen approximations whose coefficient ranges do not exceed the ability of the processor to represent them. This is analogous to saying that we have not chosen approximations which are designed for 64-bit machines (like those running at National Laboratories). Such implementations might use coefficients whose fine accuracy might not be representable in a 32-bit machine such as the ADSP-21K series machines. Further, we have chosen approximations which return the maximum amount of accuracy for a 32-bit processor and yet do not waste precious inner-loop cycles that would be necessary to support 64-bit precision. Extensive efforts went into choosing an algorithm that matches the processor's precision capabilities.
- This concept was further extended in choosing approximation algorithms that avoid the weaknesses of a specific class of processors. For example, CISC processors such as the x86 or 68000 series may easily implement algorithms which utilize native microcoded instruction sets for operations such as division, reciprocation and square root. The ADSP-21K family of DSP chips, however, uses a native microcoded instruction to compute a reciprocal to 4-bit precision based on a ROM lookup table. To gain full 32-bit accuracy, the initial approximation is processed through a soft-

ware based Newton-Raphson algorithm. For trigonometric functions, this translates into expensive processor timing that should be avoided, if possible. The Wideband library attempts to avoid this through the judicious use of approximations that avoid the division process (when possible) and instead substitutes multiplication by reciprocal values. Further, when cases utilizing the reciprocal operation have to be implemented within code, special techniques are used to implement ensure maximum speed and accuracy. In effect, the library is coded in such a manner as to play into the strong suit of the ADSP-21K processor, which is multiplication, addition and accumulation.

2.11 32-Bit Computational Accuracy

All Wideband functions, unless otherwise specified in the “Accuracy” section of each descriptive manual page, are designed to return 32-bits of accuracy. The extended precision model of 40-bit calculation is not implemented, as performance would suffer from extended inner-loops lengths necessary to support precision calculations in the trigonometric function approximation process. All calculations, unless otherwise specifically stated, result in IEEE format accuracy: 8 bits for the exponent and 24 bits for the mantissa, for a total of 32 bits.

2.12 Complex Number Representation

Complex numbers are used throughout the scientific and engineering community. A complex number \mathbf{c} can be described by the equation:

$$c = a + ib$$

where \mathbf{a} and \mathbf{b} are real numbers and $i = \sqrt{-1}$

Complex numbers are atomic, but we can derive real-valued functions as follows:

$$a = \text{Re}\{c\} \quad , \text{ which derives the real part of } c \text{ and}$$

$$b = \text{Im}\{c\} \quad , \text{ which derives the imaginary part of } c.$$

Note that both \mathbf{a} and \mathbf{b} are real numbers.

When a computer processes problems utilizing complex data, it is necessary to understand the manner in which complex data is accessed and processed. Typical operations involving vectors utilize arrays, each element of which contains a floating-point number or integer which represents a data value. The same concept hold true for complex numbers, except that the arrays must hold both real and imaginary numbers in order to represent complex numbers. This is done by interleaving both the real and imaginary portions of the data. In effect, the array becomes twice as long as a real-valued array as both a

real and imaginary portion of data must be accounted for. This can be represented as follows:

TABLE 6
Complex Number Representation Within Arrays

Index Of Array	Real Array	Complex Array
0	1st Real Number	1st Number - Real Portion
1	2nd Real Number	1st Number - Imaginary Portion
2	3rd Real Number	2nd Number - Real Portion
3	4th Real Number	2nd - Imaginary Portion

Note that striding concepts discussed earlier in the “Basic Striding Concepts” section also hold true for complex data. For example, if you provide a stride of one as an argument to a routine, each iteration of a routine will result in striding over both the real and imaginary portions for each iteration of the main loop. This what we meant by the term *atomic*: each complex number, although basically composed of two floating-point numbers in memory, is treated as one number for computing purposes. However, a stride of 1 typically results in 2 numbers being accessed or computed each iteration of the inner loop: a real number and an imaginary number.

When discussing complex numbers in the equations section of this manual, we composed a method of conveying to the user the exact nature of the algorithm that was being examined without unnecessary confusing detail regarding the indices of the vectors. We chose to describe the real and imaginary portions of a complex number using the **Re** and **Im** nomenclature. For instance, let us suppose you wish to add two complex numbers together to produce a resultant third complex number. The routine that you would call to do this is **cvadd**. If you look up **cvadd** in the complex section, you’ll see the following equation:

$$Re\{C_{mk}\} = Re\{A_{mi}\} + Re\{B_{mj}\} \quad m = \{0, 1, 2, \dots, n - 1\}$$

$$Im\{C_{mk}\} = Im\{A_{mi}\} + Im\{B_{mj}\}$$

The first line of the equation states that the real portion (*Re*) of resultant complex output vector **c** is equal to the addition of the real portion (*Re*) of input complex vector **a** and the real portion (*Re*) of input complex vector **b**. **Re** always represent the real portion of the complex number.

The second line of the equation states that the imaginary portion (*Im*) of resultant complex output vector **c** is equal to the addition of the imaginary portion (*Im*) of input complex vector **a** and the imaginary portion (*Im*) of input complex vector **b**. **Im** always represent the imaginary portion of the complex number.

2.13 Setting Test Code Array Size & Data Type **COMPLEX**

Wideband has defined a data type called **COMPLEX** in its assembly language routines which defines a complex number as composed of 2 contiguous pieces of data (real and

imaginary), located in adjacent locations in memory. Since the **COMPLEX** data type has two values (real and imaginary) for every memory location normally reserved by the **n** argument (the count of elements), a mechanism had to be invented to resolve the proper definition of memory allocation for arrays of varying types.

In summary, here are the rules which you should follow:

- When specifying the number of elements you wish to process, **n**, think in terms of the whole elements, regardless of the element type. For example, consider processing 1000 floating-point elements with vector add (**vadd**) as you would consider processing 1000 complex elements with complex vector add (**cvadd**).
- When working with complex numbers, think of each complex number as one number which internally consists of a real and an imaginary portion. The complex type data structure is atomic. Adjacent memory location contain a real and an imaginary portion, which are considered to be one complex number. You do not need to double the number **n** (thinking that you must consider compensating for each real and imaginary component) when specifying calling parameters for the routine in your C code. Embedded in the assembly routine is code which will automatically do this for you.
- However, when you reach the point of supplying test data to a complex routine, you must consider the size of the arrays as an important consideration. The array size for complex types will always be twice the size of the real types, and you must allocate the array as type **COMPLEX** or an array of floats that is twice the size of the number of complex elements. Examine the test code supplied with the complex routines, and you'll notice that the number of floating-point elements provided for a given calculation is usually twice the size **n**. Remember to also consider the effects on input data vector lengths when increasing the stride ranges on either the input or output arrays of complex number.

2.14 Execution Timing

The routines within the Wideband library have been designed to execute at maximum speed. Since many users are often running processes which are time dependent, they need to know exactly how long it takes for a given routine to execute. Timing benchmarks are included in the library. This section explains how we derived the timing benchmarks so that no ambiguity exists as to execution speed of the library.

The speed with which a routine will execute is dependent upon the following:

- The language the algorithm is coded in - the Wideband library routines are coded in ADSP-21K assembler, as assembler code is always more efficient (as a native language then straight C code).
- The degree to which the programmer utilized all capabilities of the processors assembly language - the Wideband library utilizes parallel instruction whenever possible.
- The implementation of the algorithm. In many cases the algorithms are very simple and easy to implement. In these cases the inner-loop of the routine is generally short, and there isn't much programming skill utilized to implement the algorithm. An example of a very simple algorithm is the vector absolute value routine (**vabs**),

where the inner-loop is very short. More complex algorithms, such as those implemented in the complex and trigonometric algorithms, are much more difficult to implement. In the case of the complex functions, complex data types are used, where the programmer must have mathematical and coding skills sufficient to handle both the real and imaginary portion of the complex number, with a minimum loss of execution speed. In the case of the trigonometric functions, considerable math skills must be utilized throughout the development process, along with various insights into the coding process to guarantee that the arguments are properly reduced, the correct approximation chosen, and a sufficient degree of accuracy is obtained.

- Execution speed is dependent on the clock cycle the processor utilizes. Obviously, a processor utilizing a 40 MHz clock is going to execute code faster than a processor using a 33 MHz clock.
- Execution speed is also dependent on where you place your program data and what type of memory you are using. The ADSP-21062 has a 2 Mbit on-board SRAM cache while the ADSP-21060 has a 4 Mbit on-board SRAM cache. Since instructions are 48-bits long on this machine, this translated into roughly 40 Kilobytes of available on-board cache for the ADSP-21062 and 80Kbytes of available instruction memory for the ADSP-21060. As all Wideband routines are individually linkable, this leaves plenty of room to fill the zero wait-state memory with an optional real-time kernel or important code of your choice.
- Performance can further be increased by carefully choosing hardware CPU subsystems which have memory configurations which match your performance needs. Some subsystem implementations using the ADSP-21K family of processors utilize large banks of DRAM memory, with little or no SRAM. Processors attached to such memory will have difficulty achieving the performance associated with fast memory such as SRAM or exploiting the ADSP-21K family's true performance capabilities.
- Be sure to observe the type of memory your board currently uses and make software changes as necessary. Programmers using a mix of SRAM or DRAM should be sure to consult the Analog Devices Linker Manual to be sure to link important code pieces into zero-wait state SRAM or on-board cache, if possible. If the user exceeds the amount of zero-wait state memory, continuing linking to the next fastest memory available.

2.15 How Timing Benchmarks Are Reported

Given all these considerations, the technical staff at Wideband decided to report performance benchmarks based on counting the exact number of instructions internal to each routine. Reporting the number of instruction per inner-loop plus the setup times for entry into and exit from a routine is less ambiguous than reporting cycles per second. In this manner, a user has a benchmark that is valid for all configurations. If a user knows that a routine will be executed by a 33 MHz processor instead of a 40 MHz processor valid planning for execution timing can be made.

Therefore, for every routine, we report two numbers as benchmarks: an **overhead number** and an **inner-loop number**. For instance, a typical timing benchmark looks as follows:

EXECUTION TIME $23 + 2*(N-1)$ cycles

- The first number (in this case 23) is the total number of instructions needed to enter the routine and exit the routine. This is called the **overhead number**. This includes the entry setup time to load values off of the stack, save any necessary registers, and the cycles necessary to setup the loop counter(s). This entry overhead number is added to the exit overhead number, which is derived from the total number of instructions used to flush any parallel instruction pipeline, store final answers, branch to exit points, restore saved registers, and return to the calling routine. Note that the average number represents a processor execution overhead expense, measured in cycles that is incurred only once during the call to the routine.
- The second number (in this case $2*(N-1)$) indicates the size of the inner-loop of the routine. This is called the **inner-loop number**. This parameter indicates that a computation will take place every $2*(N-1)$ clock cycles. To derive the number of clock cycles necessary to execute your problem, subtract 1 from the number of data items you desire to process (called **n**) and multiply the difference by 2. Note that the inner-loop cycle cost is incurred each time a traversal of the inner-loop is made. Therefore, this parameter represents the bulk of the processing time a processor will expend upon a problem.

2.16 Embedding Wideband Routines In PROMs, EPROMS and EEPROMs

All Wideband routines may be individually linked and embedded in ROM systems. This leads to three important implications:

- All routines are individually linkable. If you are a programmer working on project utilizing embedded techniques, you may embed a series of Wideband routines within a PROM without having to embed the entire library. This makes sense, as most programmers will use only a small number of the available Wideband routines within their code, for any given project. Valuable memory space is thus conserved, freeing space for other custom applications.
- There are no inter-routine dependencies within the code. Each routine stands alone, without the need to link in other supporting routines in order to make the routine work. For instance, in a function such as the root mean square, there is no need to link in a division and square root routine as separate entities as they are already included in the native code.
- Consult the "Size in Words" column within the timing benchmarks found in Chapter Six of this manual. All sizing parameters reported are in hexadecimal format.

2.17 How Setup and Execution Timing Tables Were Derived

Chapter 6 documents the routine names, routine definitions, the (entry and exit) overhead, and the number of cycles per element. The cycle and overhead counts were arrived at by examining the assembler code itself and counting the number of cycles necessary to enter and exit the code, and the number of clock cycles required in the inner-loop. Most ADSP-21K family assembler instructions are executed in one proces-

sor cycle. These issues were considered when tabulating the code count for the timing tables, and numbers are added accordingly. We have tried to make the timing specifications conservative and fair.

2.18 Element Count **m** For The Corrected Number of Executable Iterations

The statement $m = \{0, 1, 2, \dots, n-1\}$ shown in the complex add example (**cvadd**) is a way of stating the number of iterations that you wish the routine to process. The value **n** is always an integer and represents the number of elements you wish the routine to process. This number should represent the number of data pieces that you expect the process to execute. This is also convenient for C language programs which implement a "zero-origin" counting system. For example, if you are thinking of adding 2 vectors, both consisting of arrays which have 100 numbers each, you should call your routine with **n** = 100. While you are probably mentally conceiving your algorithms executing 1,2,3,4,5...99,100 it is, in reality, actually executing as 0,1,2,3,4...99.

The value **m** stands for the corrected number of iterations that you plan on having the routine execute. The value **m** also is used to indicate which memory position is currently being pointed to within an array.

