TMS320C55x DSP Library Programmer's Reference

SPRU422G
November 2003

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products & application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Preface

Read This First

About This Manual

The Texas Instruments TMS320C55x™ DSPLIB is an optimized DSP Function Library for C programmers on TMS320C55x devices. It includes over 50 C-callable assembly-optimized general-purpose signal processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. By using these routines you can achieve execution speeds considerable faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP functions, TI DSPLIB can shorten significantly your DSP application development time.

Related Documentation

The MathWorks, Inc. *Matlab Signal Processing Toolbox User's Guide.* Natick, MA: The MathWorks, Inc., 1996. .

Lehmer, D.H. "Mathematical Methods in large-scale computing units." *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery, Cambridge, MA, 1949*. Cambridge, MA: Harvard University Press, 1951.

Oppenheim, Alan V. and Ronald W Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

Digital Signal Processing with the TMS320 Family (SPR012)

TMS320C55x DSP CPU Reference Guide (SPRU371)

TMS320C55x Optimizing C Compiler User's Guide (SPRU281)

Trademarks

TMS320, TMS320C55x, and C55x are trademarks of Texas Instruments.

Matlab is a trademark of Mathworks, Inc.

Contents

1 Introduction **1-1**
Introduction to the TMS320C55x DSP Library.
1.1 DSP Routines 1-2
1.2 Features and Benefits 1-2
1.3 DSPLIB: Quality Freeware That You Can Build On and Contribute To 1-2

2 Installing DSPLIB **2-1**
Describes how to install the DSPLIB.
2.1 DSPLIB Content 2-2
2.2 How to Install DSPLIB 2-3
2.2.1 De-Archive DSPLIB 2-3
2.2.2 Relocate Library File 2-3
2.3 How to Rebuild DSPLIB 2-4
2.3.1 For Full Rebuild of 55xdsp.lib 2-4
2.3.2 For Partial Rebuild of 55xdsp.lib (modification of a specific DSPLIB function, for example fir.asm) 2-4

3 Using DSPLIB **3-1**
Describes how to use the DSPLIB.
3.1 DSPLIB Arguments and Data Types 3-2
3.1.1 DSPLIB Arguments 3-2
3.1.2 DSPLIB Data Types 3-2
3.2 Calling a DSPLIB Function from C 3-3
3.3 Calling a DSPLIB Function from Assembly Language Source Code 3-3
3.4 Where to Find Sample Code 3-3
3.5 How DSPLIB is Tested - Allowable Error 3-4
3.6 How DSPLIB Deals with Overflow and Scaling Issues 3-4
3.7 Where DSPLIB Goes From Here 3-6

4 Function Descriptions **4-1**
Provides descriptions for the TMS320C55x DSPLIB functions.
4.1 Arguments and Conventions Used 4-2
4.2 DSPLIB Functions 4-3
acorr 4-7
add 4-9

atan2_16 4-10
atan16 4-11
bexp 4-13
cbrev 4-14
cbrev32 4-15
cfft 4-16
cfft32 4-19
cfft off-place 4-21
cfir 4-23
cifft 4-27
cifft32 4-29
cifft off-place 4-31
convol 4-34
convol1 4-36
convol2 4-38
corr 4-40
dlms 4-42
dlmsfast 4-44
expn 4-48
fir 4-49
fir2 4-52
firdec 4-55
firinterp 4-57
firlat 4-59
firs 4-61
fltoq15 4-64
hilb16 4-65
iir32 4-69
iircas4 4-71
iircas5 4-73
iircas51 4-75
iirlat 4-77
ldiv16 4-79
log_10 4-80
log_2 4-82
logn 4-83
maxidx 4-84
maxidx34 4-86
maxval 4-87
maxvec 4-87
minidx 4-88
minval 4-89
minvec 4-90
mmul 4-91

mtrans 4-92
mul32 4-93
neg 4-94
neg32 4-95
power 4-96
q15tofl 4-97
rand16 4-97
rand16init 4-99
recip16 4-100
rfft 4-102
rifft 4-104
sine 4-105
sqrt_16 4-106
sub 4-107

5 DSPLIB Benchmarks and Performance Issues 5-1

Describes benchmarks and performance issues for the DSPLIB functions.

5.1 What DSPLIB Benchmarks are Provided 5-2
5.2 Performance Considerations 5-2

6 Software Updates and Customer Support 6-1

Details the software updates and customer support issues for the TMS320C55x DSPLIB.

6.1 DSPLIB Software Updates 6-2
6.2 DSPLIB Customer Support 6-2

A Overview of Fractional Q Formats A-1

Describes the fractional Q formats used by the DSPLIB functions.

A.1 Q3.12 Format A-2
A.2 Q.15 Format A-2
A.3 Q.31 Format A-2

B Calculating the Reciprocal of a Q15 Number B-1

Provides the calculations used to find the inverse of a fractional Q15 number.

Figures

4-1 dbuffer Array in Memory at Time j 4-25
4-2 x Array in Memory 4-26
4-3 r Array in Memory 4-26
4-4 x Array in Memory 4-34
4-5 r Array in Memory 4-35
4-6 h Array in Memory 4-35
4-7 x Array in Memory 4-37
4-8 r Array in Memory 4-37
4-9 h Array in Memory 4-37
4-10 x Array in Memory 4-39
4-11 r Array in Memory 4-39
4-12 h Array in Memory 4-39
4-13 x Buffer 4-46
4-14 dbuffer 4-47
4-15 h Buffers 4-47
4-16 dbuffer Array in Memory at Time j 4-51
4-17 x Array in Memory 4-51
4-18 r Array in Memory 4-52
4-19 dbuffer Array in Memory at Time j 4-54
4-20 x Array in Memory 4-54
4-21 r Array in Memory 4-55
4-22 dbuffer Array in Memory at Time j 4-63
4-23 x Array in Memory 4-63
4-24 r Array in Memory 4-64
4-25 dbuffer Array in Memory at Time j 4-67
4-26 x Array in Memory 4-68
4-27 r Array in Memory 4-68

Tables

4-1 Function Descriptions 4-2
4-2 Summary Table 4-3
A-1 Q3.12 Bit Fields A-2
A-2 Q.15 Bit Fields A-2
A-3 Q.31 Low Memory Location Bit Fields A-2
A-4 Q.31 High Memory Location Bit Fields A-2

Chapter 1

Introduction

The Texas Instruments TMS320C55x DSP Library (DSPLIB) is an optimized DSP Function Library for C programmers on TMS320C55x devices. It includes over 50 C-callable assembly-optimized general-purpose signal processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. By using these routines you can achieve execution speeds considerable faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP functions, TI DSPLIB can shorten significantly your DSP application development time.

Topic **Page**

1.1 DSP Routines 1-2

1.2 Features and Benefits 1-2

1.3 DSPLIB: Quality Freeware That You Can Build On and Contribute To 1-2

1.1 DSP Routines

The TI DSPLIB includes commonly used DSP routines. Source code is provided to allow you to modify the functions to match your specific needs.

The routines included within the library are organized into eight different functional categories:

- Fast-Fourier Transforms (FFT)
- Filtering and convolution
- Adaptive filtering
- Correlation
- Math
- Trigonometric
- Miscellaneous
- Matrix

1.2 Features and Benefits

- Hand-coded assembly optimized routines
- C-callable routines fully compatible with the TI C55x compiler
- Fractional Q15-format operand supported
- Complete set of examples on usage provided
- Benchmarks (time and code) provided
- Tested against Matlab™ scripts

1.3 DSPLIB: Quality Freeware That You Can Build On and Contribute To

DSPLIB is a free-of-charge product. You can use, modify, and distribute TI C55x DSPLIB for usage on TI C55x DSPs with no royalty payments. See section 3.7, *Where DSPLIB Goes From Here*, for details.

Chapter 2

Installing DSPLIB

This chapter describes how to install the DSPLIB.

Topic **Page**

2.1 DSPLIB Content **2-2**

2.2 How to Install DSPLIB **2-3**

2.3 How to Rebuild DSPLIB **2-4**

2.1 DSPLIB Content

The TI DSPLIB software consists of 4 parts:

1) a header file for C programmers:

   dsplib.h

2) One object library:

   55xdsp.lib

3) One source library to allow function customization by the end user

   55xdsp.src

4) Example programs and linker command files used under the “55x_test” sub-directory.

2.2 How to Install DSPLIB

Note:

Read the README.1ST file for specific details of release.

2.2.1 De-Archive DSPLIB

DSPLIB is distributed in the form of an executable self-extracting ZIP file (55xdsplib.exe). The zip file automatically restores the DSPLIB individual components in the same directory you execute the self extracting file. Following is an example on how to install DSPLIB, just type:

55xdsplib.exe -d

The DSPLIB directory structure and content you will find is:

55xdsplib (dir)	
55xdsp.lib	: use for standards short-call mode
blt55x.bat	: re-generate 55xdsp.lib based on 55xdsp.src
examples(dir)	: contains one subdirectory for each routine included in the library where you can find complete test cases
include(dir)	
dsplib.h	: include file with data types and function prototypes
tms320.h	: include file with type definitions to increase TMS320 portability
misc.h	: include file with useful miscellaneous definitions
doc(dir)	
55x_src (dir)	: contains assembly source files for functions

2.2.2 Relocate Library File

Copy the C55x DSPLIB object library file, 55xdsp.lib, to your C5500 runtime support library folder.

For example, if your TI C5500 tools are located in *c:\ti\c5500\cgtools\bin* and c runtime support libraries (rts55.lib etc.) in *c:\ti\c5500\cgtools\lib,* copy 55xdsplib.lib to this folder. This allows the C55x compiler/linker to find 55xdsp.lib.

2.3 How to Rebuild DSPLIB

2.3.1 For Full Rebuild of 55xdsp.lib

To rebuild 55xdsp.lib, execute the blt55x.bat. This will overwrite any existing 55xdsp.lib.

2.3.2 For Partial Rebuild of 55xdsp.lib (modification of a specific DSPLIB function, for example fir.asm)

1) Extract the source for the selected function from the source archive:

   ar55 x 55xdsp.src fir.asm

2) Re-assemble your new fir.asm assembly source file:

   asm55 –g fir.asm

3) Replace the object , fir.obj, in the dsplib.lib object library with the newly formed object:

   ar55 r 55xdsp.lib fir.obj

Chapter 3

Using DSPLIB

This chapter describes how to use the DSPLIB.

Topic **Page**

3.1 DSPLIB Arguments and Data Types 3-2

3.2 Calling a DSPLIB Function from C 3-3

3.3 Calling a DSPLIB Function from Assembly Language Source Code 3-3

3.4 Where to Find Sample Code 3-3

3.5 How DSPLIB is Tested — Allowable Error 3-4

3.6 How DSPLIB Deals with Overflow and Scaling Issues 3-4

3.7 Where DSPLIB Goes From Here 3-6

3.1 DSPLIB Arguments and Data Types

3.1.1 DSPLIB Arguments

DSPLIB functions typically operate over vector operands for greater efficiency. Though these routines can be used to process short arrays or scalars (unless a minimum size requirement is noted) , the execution times will be longer in those cases.

Vector stride is always equal 1: vector operands are composed of vector elements held in consecutive memory locations (vector stride equal to 1).

Complex elements are assumed to be stored in a Re-Im format.

In-place computation is allowed (unless specifically noted): Source operand can be equal to destination operand to conserve memory.

3.1.2 DSPLIB Data Types

DSPLIB handles the following fractional data types:

Q.15 (DATA) : A Q.15 operand is represented by a *short* data type (16 bit) that is predefined as *DATA*, in the *dsplib.h* header file.

Q.31 (LDATA) : A Q.31 operand is represented by a *long* data type (32 bit) that is predefined as L*DATA*, in the *dsplib.h* header file.

Q.3.12 : Contains 3 integer bits and 12 fractional bits.

Unless specifically noted, DSPLIB operates on Q15-fractional data type elements. Appendix A presents an overview of Fractional Q formats

3.2 Calling a DSPLIB Function from C

In addition to installing the DSPLIB software, to include a DSPLIB function in your code you have to:

- Include the *dsplib.h* include file
- Link your code with the DSPLIB object code library, *55xdsp.lib* or *55xdspx.lib*.
- Use a correct linker command file describing the memory configuration available in your C55x board.

A project file has been included for each function in the examples folder. You can reference function_t.c files for calling a DSPLIB function from C.

The examples presented in this document have been tested using the Texas Instruments C55x Simulator. Customization may be required to use it with a different simulator or development board.

Refer to the *TMS320C55x Optimizing C Compiler User's Guide* (SPRU281).

3.3 Calling a DSPLIB Function from Assembly Language Source Code

The TMS320C55x DSPLIB functions were written to be used from C. Calling the functions from assembly language source code is possible as long as the calling-function conforms with the Texas Instruments C55x C compiler calling conventions. Refer to the *TMS320C55x Optimizing C Compiler User's Guide*, if a more in-depth explanation is required.

Realize that the TI DSPLIB is not an optimal solution for assembly-only programmers. Even though DSPLIB functions can be invoked from an assembly program, the result may not be optimal due to unnecessary C-calling overhead.

3.4 Where to Find Sample Code

You can find examples on how to use every single function in DSPLIB, in the *examples* subdirectory. This subdirectory contains one subdirectory for each function. For example, the *examples/araw* subdirectory contains the following files:

- *araw_t.c*: main driver for testing the DSPLIB acorr (raw) function.
- *test.h*: contains input data(a) and expected output data(yraw) for the acorr (raw) function as. This test.h file is generated by using Matlab scripts.

test.c: contains function used to compare the output of araw function with the expected output data.

ftest.c: contains function used to compare two arrays of float data types.

ltest.c: contains function used to compare two arrays of long data types.

ld3.cmd: an example of a linker command you can use for this function.

3.5 How DSPLIB is Tested - Allowable Error

Version 1.0 of DSPLIB is tested against Matlab scripts. Expected data output has been generated from Matlab that uses double-precision (64-bit) floating-point operations (default precision in Matlab). Test utilities have been added to our test main drivers to automate this checking process. Note that a maximum absolute error value (MAXERROR) is passed to the test function, to set the trigger point to flag a functional error.

We consider this testing methodology a good first pass approximation. Further characterization of the quantization error ranges for each function (under random input) as well as testing against a set of fixed-point C models is planned for future releases. We welcome any suggestions you may have on this respect.

3.6 How DSPLIB Deals with Overflow and Scaling Issues

One of the inherent difficulties of programming for fixed-point processors is determining how to deal with overflow issues. Overflow occurs as a result of addition and subtraction operations when the dynamic range of the resulting data is larger than what the intermediate and final data types can contain.

The methodology used to deal with overflow should depend on the specifics of your signal, the type of operation in your functions, and the DSP architecture used. In general, overflow handling methodologies can be classified in five categories: saturation, input scaling, fixed scaling, dynamic scaling, and system design considerations.

It's important to note that a TMS320C55x architectural feature that makes overflow easier to deal with is the presence of *guard bits in all four accumulators.* The 40-bit accumulators provide eight guard bits that allow up to 256 consecutive multiply-and-accumulate (MAC) operations before an accumulator overrun – a very useful feature when implementing, for example, FIR filters.

There are 4 specific ways DSPLIB deals with overflow, as reflected in each function description:

Scaling implemented for overflow prevention: In this type of function, DSPLIB scales the intermediate results to prevent overflow. Overflow should not occur as a result. Precision is affected but not significantly. This is the case of the FFT functions, in which scaling is used after each FFT stage.

No scaling implemented for overflow prevention: In this type of function, DSPLIB does not scale to prevent overflow due to the potentially strong effect in data output precision or in the number of cycles required. This is the case, for example, of the MAC-based operations like filtering, correlation, or convolutions. The best solution on those cases is to design your system , for example your filter coefficients with a gain less than 1 to prevent overflow. In this case, overflow could happen unless you input scale or you design for no overflow.

Saturation implemented for overflow handling: In this type of function, DSPLIB has enabled the TMS320C55x 32-bit saturation mode (SATD bit = 1). This is the case of certain basic math functions that require the saturation mode to be enabled.

Not applicable: In this type of function, due to the nature of the function operations, there is no overflow.

DSPLIB reporting of overflow conditions (overflow flag): Due to the sometimes unpredictible overflow risk, most DSPLIB functions have been written to return an overflow flag *(oflag)* as an indication of a potentially dangerous 32-bit overflow. However, because of the guard-bits, the C55x is capable of handling intermediate 32-bit overflows and still produce the correct final result. Therefore, the oflag parameter should be taken in the context of a warning but not a definitive error.

As a final note, DSPLIB is provided also in source format to allow customization of DSPLIB functions to your specific system needs.

3.7 Where DSPLIB Goes From Here

We anticipate DSPLIB to improve in future releases in the following areas:

Increased number of functions: We anticipate the number of functions in DSPLIB will increase. We welcome user-contributed code. If during the process of developing your application you develop a DSP routine that seems like a good fit to DSPLIB, let us know. We will review and test your routine and possibly include it in the next DSPLIB software release. Your contribution will be acknowledged and recognized by TI in the *Acknowledgments* section. Use this opportunity to make your name known by your DSP industry peers. Simply email your contribution To Whom It May Concern: dsph@ti.com and we will contact you.

Increased Code portability: DSPLIB looks to enhance code portability across different TMS320-based platforms. It is our goal to provide similar DSP libraries for other TMS320™ devices, working in conjunction with C55x compiler intrinsics to make C-developing easier for fixed-point devices. However, it's anticipated that a 100% portable library across TMS320 devices may not be possible due to normal device architectural differences. TI will continue monitoring DSP industry standardization activities in terms of DSP function libraries.

Chapter 4

Function Descriptions

This chapter provides descriptions for the TMS320C55x DSPLIB functions.

Topic **Page**

4.1 Arguments and Conventions Used 4-2
4.2 DSPLIB Functions 4-3

4.1 Arguments and Conventions Used

The following convention has been followed when describing the arguments for each individual function:

Table 4-1. Function Descriptions

Argument	Description
x,y	argument reflecting input data vector
r	argument reflecting output data vector
nx,ny,nr	arguments reflecting the size of vectors x,y, and r respectively. In functions where nx = nr = nr, only nx has been used.
h	Argument reflecting filter coefficient vector (filter routines only)
nh	Argument reflecting the size of vector h
DATA	data type definition equating a short, a 16-bit value representing a Q15 number. Usage of DATA instead of short is recommended to increase future portability across devices.
LDATA	data type definition equating a long, a 32-bit value representing a Q31 number. Usage of LDATA instead of long is recommended to increase future portability across devices.
ushort	Unsigned short (16 bit). You can use this data type directly, because it has been defined in dsplib.h

4.2 DSPLIB Functions

The routines included within the library are organized into 8 different functional categories**:**

FFT
Filtering and convolution
Adaptive filtering
Correlation
Math
Trigonometric
Miscellaneous
Matrix

Table 4-2 lists the functions by these 8 functional catagories.

Table 4-2. Summary Table

(a) FFT

Functions	Description
void cfft (DATA *x, ushort nx, type)	Radix-2 complex forward FFT - MACRO
void cfft32 (LDATA *x, ushort nx, type);	32-bit forward complex FFT
void cifft (DATA *x, ushort nx, type)	Radix-2 complex inverse FFT - MACRO
void cifft32 (LDATA *x, ushort nx, type);	32-bit inverse complex FFT
void cbrev (DATA *x, DATA *r, ushort n)	Complex bit-reverse function
void cbrev32 (LDATA *a, LDATA *r, ushort)	32-bit complex bit reverse
void rfft (DATA *x, ushort nx, type)	Radix-2 real forward FFT - MACRO
void rifft (DATA *x, ushort nx, type)	Radix-2 real inverse FFT - MACRO

(b) Filtering and Convolution

Functions	Description
ushort fir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)	FIR direct form
ushort fir2 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)	FIR direct form (Optimized to use DUAL-MAC)
ushort firs (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh2)	Symmetric FIR direct form (generic routine)
ushort cfir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)	Complex FIR direct form

Table 4-2. Summary Table (Continued)

ushort convol (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)	Convolution
ushort convol1 (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)	Convolution (Optimized to use DUAL-MAC)

(b) Filtering and Convolution (Continued)

Functions	**Description**
ushort convol2 (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)	Convolution (Optimized to use DUAL-MAC)
ushort iircas4 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)	IIR cascade direct form II. 4 coefficients per biquad.
ushort iircas5 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)	IIR cascade direct form II. 5 coefficients per biquad
ushort iircas51 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)	IIR cascade direct form I. 5 coefficients per biquad
ushort iirlat (DATA *x, DATA *h, DATA *r, DATA *pbuffer, int nx, int nh)	Lattice inverse IIR filter
ushort firlat (DATA *x, DATA *h, DATA *r, DATA *pbuffer, int nx, int nh)	Lattice forward FIR filter
ushort firdec (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nh, ushort nx, ushort D)	Decimating FIR filter
ushort firinterp (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nh, ushort nx, ushort I)	Interpolating FIR filter
ushort hilb16 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)	FIR Hilbert Transformer
ushort iir32 (DATA *x, LDATA *h, DATA *r, LDATA *dbuffer, ushort nbiq, ushort nr)	Double-precision IIR filter

(c) Adaptive filtering

Functions	**Description**
ushort dlms (DATA *x, DATA *h, DATA *r, DATA *des, DATA *dbuffer, DATA step, ushort nh, ushort nx)	LMS FIR (delayed version)
ushort oflag = dlmsfast (DATA *x, DATA *h, DATA *r, DATA *des, DATA *dbuffer, DATA step, ushort nh, ushort nx)	Adaptive delayed LMS filter (fast implemented)

(d) Correlation

Functions	**Description**
ushort acorr (DATA *x, DATA *r, ushort nx, ushort nr, type)	Autocorrelation (positive side only) - MACRO

Table 4-2. Summary Table (Continued)

ushort corr (DATA *x, DATA *y, DATA *r, ushort nx, ushort ny, type)	Correlation (full-length)

(e) Trigonometric

Functions	**Description**
ushort sine (DATA *x, DATA *r, ushort nx)	sine of a vector
ushort atan2_16 (DATA *i, DATA *q, DATA *r, ushort nx)	Four quadrant inverse tangent of a vector
ushort atan16 (DATA *x, DATA *r, ushort nx)	Arctan of a vector

(f) Math

Functions	**Description**
ushort add (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)	Optimized vector addition
ushort expn (DATA *x, DATA *r, ushort nx)	Exponent of a vector
short bexp (DATA *x, ushort nx)	Exponent of all values in a vector
ushort logn (DATA *x, LDATA *r, ushort nx)	Natural log of a vector
ushort log_2 (DATA *x, LDATA *r, ushort nx)	Log base 2 of a vector
ushort log_10 (DATA *x, LDATA *r, ushort nx)	Log base 10 of a vector
short maxidx (DATA *x, ushort ng, ushort ng_size)	Index for maximum magnitude in a vector
short maxidx34 (DATA *x, ushort nx)	Index of the maximum element of a vector ≤ 34
short maxval (DATA *x, ushort nx)	Maximum magnitude in a vector
void maxvec (DATA *x, ushort nx, DATA *r_val, DATA *r_idx)	Index and value of the maximum element of a vector
short minidx (DATA *x, ushort nx)	Index for minimum magnitude in a vector
short minval (DATA *x, ushort nx)	Minimum element in a vector
void minvec (DATA *x, ushort nx, DATA *r_val, DATA *r_idx)	Index and value of the minimum element of a vector
ushort mul32 (LDATA *x, LDATA *y, LDATA *r, ushort nx)	32-bit vector multiply
short neg (DATA *x, DATA *r, ushort nx)	16-bit vector negate
short neg32 (LDATA *x, LDATA *r, ushort nx)	32-bit vector negate
short power (DATA *x, LDATA *r, ushort nx)	sum of squares of a vector (power)
void recip16 (DATA *x, DATA *r, DATA *rexp, ushort nx)	Vector reciprocal

Table 4-2. Summary Table (Continued)

void ldiv16 (LDATA *x, DATA *y, DATA *r, DATA *rexp, ushort nx)	32-bit by 16-bit long division

(f) Math (Continued)

Functions	**Description**
ushort sqrt_16 (DATA *x, DATA *r, short nx)	Square root of a vector
short sub (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)	Vector subtraction

(g) Matrix

Functions	**Description**
ushort mmul (DATA *x1, short row1, short col1, DATA *x2, short row2, short col2, DATA *r)	matrix multiply
ushort mtrans (DATA *x, short row, short col, DATA *r)	matrix transponse

(h) Miscellaneous

Functions	**Description**
ushort fltoq15 (float *x, DATA *r, ushort nx)	Floating-point to Q15 conversion
ushort q15tofl (DATA *x, float *r, ushort nx)	Q15 to floating-point conversion
ushort rand16 (DATA *r, ushort nr)	Random number generation
void rand16init(void)	Random number generation initialization

acorr — *Autocorrelation*

Function

ushort oflag = acorr (DATA *x, DATA *r, ushort nx, ushort nr, type)
(defined in araw.asm, abias.asm , aubias.asm)

Arguments

x [nx]	Pointer to real input vector of nx real elements. nx ≥ nr
r [nr]	Pointer to real output vector containing the first nr elements of the positive side of the autocorrelation function of vector a. r must be different than a (in-place computation is not allowed).
nx	Number of real elements in vector x
nr	Number of real elements in vector r
type	Autocorrelation type selector. Types supported: If type = raw, r contains the raw autocorrelation of x If type = bias, r contains the biased autocorrelation of x If type = unbias, r contains the unbiased autocorrelation of x
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred If oflag = 0, a 32-bit overflow has not occurred

Description

Computes the first nr points of the positive side of the autocorrelation of the real vector x and stores the results in real output vector r. The full-length autocorrelation of vector x will have 2*nx-1 points with even symmetry around the lag 0 point (r[0]). This routine provides only the positive half of this for memory and computational savings.

Algorithm

Raw Autocorrelation

$$r[j] = \sum_{k=0}^{nx-j-1} x[j+k]\,x[k] \qquad 0 \le j \le nr$$

Biased Autocorrelation

$$r[j] = \frac{1}{nx}\sum_{k=0}^{nx-j-1} x[j+k]\,x[k] \qquad 0 \le j \le nr$$

Unbiased Autocorrelation

$$r[j] = \frac{1}{(nx - abs(j))}\sum_{k=0}^{nx-j-1} x[j+k]\,x[k] \qquad 0 \le j \le nr$$

Overflow Handling Methodology No scaling implemented for overflow prevention

Special Requirements none

Implementation Notes

Special debugging consideration: This function is implemented as a macro that invokes different autocorrelation routines according to the type selected. As a consequence the acorr symbol is not defined. Instead the acorr_raw, acorr_bias, acorr_unbias symbols are defined.

Autocorrelation is implemented using time-domain techniques

Example See examples/abias, examples/aubias, examples/araw subdirectories

Benchmarks (preliminary)

Cycles† **Abias:**

Core:
nr even: [(4 * nx - nr * (nr + 2) + 20) / 8] * nr
nr odd: [(4 * nx - (nr - 1) * (nr + 1) + 20) / 8] * (nr - 1) + 10
nr = 1: (nx + 2)

Overhead:
nr even: 90
nr odd: 83
nr = 1: 59

Araw:

Core:
nr even: [(4 * nx - nr * (nr + 2) + 28) / 8] * nr
nr odd: [(4 * nx - (nr - 1) * (nr + 1) + 28) / 8] * (nr - 1) + 13
nr = 1: (nx + 1)

Overhead:
nr even: 34
nr odd: 35
nr = 1: 30

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

Cycles†	**Aubias:**	
	Core:	
	nreven:	[(8 * nx - 3 * nr * (nr + 2) + 68) / 8] * nr
	nr odd:	[(8 * nx - 3 * (nr-1) * (nr+1) + 68)/8] * (nr - 1) + 33
	nr = 1:	nx + 26
	Overhead:	
	nr even:	64
	nr odd:	55
	nr = 1:	47
Code size (in bytes)	**Abias:**	226
	Araw:	178
	Aubias:	308

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

add — *Vector Add*

Function

ushort oflag = add (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)
(defined in add.asm)

Arguments

x[nx] — Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)

y[nx] — Pointer to input data vector 2 of size nx

r[nx] — Pointer to output data vector of size nx containing

(x+y) if scale = 0

(x+y) /2 if scale = 1

nx — Number of elements of input and output vectors.
nx ≥ 4

scale — Scale selection

If scale = 1, divide the result by 2 to prevent overflow

If scale = 0, do not divide by 2

oflag — Overflow flag.

If oflag = 1, a 32-bit overflow has occurred

If oflag = 0, a 32-bit overflow has not occurred

Description

This function adds two vectors, element by element.

Algorithm for $(i = 0;\ i < nx;\ i++) \quad z(i) = x(i) + y(i)$

Overflow Handling Methodology Scaling implemented for overflow prevention (user selectable)

Special Requirements none

Implementation Notes none

Example See examples/add subdirectory

Benchmarks (preliminary)

Cycles†	Core:	3 * nx
	Overhead:	23
Code size (in bytes)	60	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

atan2_16 *Arctangent 2 Implementation*

Function ushort oflag = atan2_16 (DATA *i, DATA *q, DATA *r, ushort nx)
(defined in arct2.asm)

Arguments

q[nx]	Pointer to quadrature input vector of size nx.
i[nx]	Pointer to in-phase input vector of size nx
r[nx]	Pointer to output data vector (in Q15 format) number representation of size nx containing. In-place processing allowed (r can be equal to x) on output, r contains the arctangent of (q/I) /π
nx	Number of elements of input and output vectors.
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred If oflag = 0, a 32-bit overflow has not occurred

Description This function calculates the arctangent of the ratio q/I, where -1 <= atan2_16(Q/I) <= 1 representing an actual range of $-\pi$ < atan2_16(Q/I) < π. The result is placed in the resultant vector r. Output scale factor correction = π. For example, if:

y = [0x1999, 0x1999, 0x0, 0xe667, 0x1999] (equivalent to [0.2, 0.2, 0, -0.2, 0.2] float)

x = [0x1999, 0x3dcc, 0x7ffff, 0x3dcc c234] (equivalent to [0.2, 0.4828, 1, 0.4828, –0.4828] float)

atan2_16(y, x, r,4) should give:

r = [0x2000, 0x1000, 0x0, 0xf000, 0x7000] equivalent to [0.25, 0.125, 0, –0.125, 0.875]*π

Algorithm for $(j = 0;\ j < nx;\ j++) \quad r(j) = atan2(q[j],\ i[j])$

Overflow Handling Methodology Not applicable

Special Requirements Linker command file: you must allocate .data section (for polynomial coefficients)

Implementation Notes none

Example See examples/arct2 subdirectory

Benchmarks (preliminary)

Cycles†	18 + 62 * nx
Code size (in bytes)	170 program; 10 data; 4 stack

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

atan16 — *Arctangent Implementation*

Function ushort oflag = atan16 (DATA *x, DATA *r, ushort nx)
(defined in atant.asm)

Arguments

x[nx]	Pointer to input data vector of size nx. x contains the tangent of r, where |x| < 1.
r[nx]	Pointer to output data vector of size nx containing the arctangent of x in the range [$-\pi/4$, $\pi/4$] radians. In-place processing allowed (r can be equal to x) atan(1.0) = 0.7854 or 6478h

nx	Number of elements of input and output vectors.
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred If oflag = 0, a 32-bit overflow has not occurred

Description This function calculates the arc tangent of each of the elements of vector x. The result is placed in the resultant vector r and is in the range [-π/2 to π/2] radians. For example,
if x = [0x7fff, 0x3505, 0x1976, 0x0] (equivalent to tan(π/4), tan(π/8), tan(π/16), 0 in float):
atan16(x,r,4) should give
r = [0x6478, 0x3243, 0x1921, 0x0] equivalent to [π/4, π/8, π/16, 0]

Algorithm for $(i = 0;\ i < nx;\ i++)\quad r(i) = atan(x(i))$

Overflow Handling Methodology Not applicable

Special Requirements Linker command file: you must allocate .data section (for polynomial coefficients)

Implementation Notes

$atan(x),$ with $0 \le x \le 1,$ output scaling factor $= \pi.$

Uses a polynomial to compute the arctan (x) for |x| <1. For |x| > 1, you can express the number x as a ratio of 2 fractional numbers and use the atan2_16 function.

Example See examples/atant subdirectory

Benchmarks (preliminary)

Cycles†	14 + 8 * nx
Code size (in bytes)	43 program; 6 data

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

bexp — *Block Exponent Implementation*

Function

short r = bexp (DATA *x, ushort nx)
(defined in bexp.asm)

Arguments

x [nx]	Pointer to input vector of size nx
r	Return value. Maximum exponent that may be used in scaling.
nx	Length of input data vector

Description

Computes the exponents (number of extra sign bits) of all values in the input vector and returns the minimum exponent. This will be useful in determining the maximum shift value that may be used in scaling a block of data.

Algorithm Not applicable

Overflow Handling Methodology Not applicable

Special Requirements none

Implementation Notes none

Example See examples/bexp subdirectory

Benchmarks (preliminary)

Cycles	Core:	3 * nx
	Overhead:	4
Code size (in bytes)	19	

cbrev — *Complex Bit Reverse*

Function

void cbrev (DATA *a, DATA *r, ushort n)
(defined in cbrev.asm)

Arguments

x[2*nx]	Pointer to complex input vector x.
r[2*nx]	Pointer to complex output vector r.
nx	Number of complex elements of vectors x and r.

To bit-reverse the input of a complex FFT, nx should be the complex FFT size.

To bit-reverse the input of a real FFT, nx should be half the real FFT size.

Description

This function bit-reverses the position of elements in complex vector x into output vector r. In-place bit-reversing is allowed. Use this function in conjunction with FFT routines to provide the correct format for the FFT input or output data. If you bit-reverse a linear-order array, you obtain a bit-reversed order array. If you bit-reverse a bit-reversed order array, you obtain a linear-order array.

Algorithm

Not applicable

Overflow Handling Methodology

Not applicable

Special Requirements

Alignment of input database address: (n+1) LSBs must be zeros, where $n = \log_2(nx)$.

Ensure that the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).

Implementation Notes

x is read with bit-reversed addressing and r is written in normal linear addressing.

Off-place bit-reversing (x = r) requires half as many cycles as in-place bit-reversing (x <> r). However, this is at the expense of doubling the data memory requirements.

Example

See examples/cfft and examples/rfft subdirectories

Benchmarks (preliminary)

Cycles†	Core: 2 * nx (off-place) 4 * nx + 6 (in-place) Overhead: 17
Code size (in bytes)	81 (includes support for both in-place and off-place bit-reverse)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

cbrev32 — *32-Bit Complex Bit Reverse*

Function void cbrev32(LDATA *a, LDATA *r, ushort)
(defined in cbrev32.asm)

Arguments

x[2*nx]	Pointer to complex input vector x.
r[2*x]	Pointer to complex output vector r.
nx	Number of complex elements in vector x.

To bit-reverse the output of a complex (i)FFT, nx should be the complex (i)FFT size.

To bit-reverse the output of a real (i)FFT, nx should be half the real (i)FFT size.

Description This function bit-reverses the position of elements in complex vector x into output vector r. In-place bit-reversing is allowed. Use this function in conjunction with (i)FFT routines to provide the correct format for the (i)FFT input or output data. If you bit-reverse a linear-order array, you obtain a bit-reversed order array. Ifyou bit-reverse a bit-reversed order array, you obtain a linear-order array.

Algorithm Not applicable

Overflow Handling Methodology Not applicable

Special Requirements

Alignment of input database address: (n+1) LSBs must be zeros, where n = log 2 (nx).

Ensure that the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).

Implementation Notes x is read in normal linear addressing and r is written with bit-reversed addressing.

Example See example/c(i)fft32 subdirectory

Benchmarks

Cycles[†]	Core:	
	5*nx	(off-place)
	11*nx	(in-place)
Code size (in bytes)	75 (includes support for both in-place and off-place bit-reverse)	

[†] Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

cfft — *Forward Complex FFT*

Function void cfft (DATA *x, ushort nx, type);
(defined in cfft.asm)

Arguments

x [2*nx]	Pointer to input vector containing nx complex elements (2*nx real elements) in normal order. On output, vector contains the nx complex elements of the FFT(x) in bit-reversed order. Complex numbers are stored in interleaved Re-Im format.
nx	Number of complex elements in vector x. Must be between 16 and 1024.
type	FFT type selector. Types supported: If type = SCALE, scaled version selected If type = NOSCALE, non-scaled version selected

Description Computes a complex nx-point FFT on vector x, which is in normal order. The original content of vector x is destroyed in the process. The nx complex elements of the result are stored in vector x in bit-reversed order. The twiddle table is in bit-reversed order.

Algorithm (DFT)

$$y[k] = \frac{1}{(scale\ factor)} * \sum_{i=0}^{nx-1} x[i] * \left(\cos\left(\frac{2*\pi*i*k}{nx}\right) + j\sin\left(\frac{2*\pi*i*k}{nx}\right)\right)$$

Overflow Handling Methodology If type = SCALE is selected, scaling before each stage is implemented for overflow prevention

Special Requirements

This function requires the inclusion of file “twiddle.inc” which contains the twiddle table (automatically included).

The twiddle table must be located in internal memory since it is accuired by the C55x coefficient bus.

Input data section is aligned on 32-bit boundary.

Input data section and twiddle table in different DARAM blocks:

- Twiddle table can be placed in an SARAM block in a different block than the input section.
- Input table is 32-bit aligned in SARAM or DARAM blocks.

Data memory alignment (reference cfft.cmd in examples/cfft directory):

- Alignment of input vector on a 32-bit boundary (2 of the LSBs of the byte address must be zero)
- Ensure that the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).
- For best performance the data buffer has to be in a DARAM block.
- If the twiddle table and the data buffer are in the same block then the radix-2 kernal is 7 cycles and the radix-4 kernel is not affected.

Implementation Notes

The implementations are optimized for MIPS, not for code size. They implement the decimation-in-time (DIT) FFT algorithm.

The NOSCALE version is implemented using radix-2 butterflies. The first two stages are replaced by a single radix-4 stage.

The SCALE version is implemented using only radix-2 stages. This routine prevents overflow by scaling by 2 before each FFT stage.

Example See examples/cfft subdirectory

Benchmarks

6 cycles (radix-2 butterfly - used in both scaled and non-scaled versions)

10 cycles (radix-4 butterfly – used in the first 2 stages of a non-scaled version)

CFFT - SCALE

FFT Size	Cycles[†]	Code Size (in bytes)
16	367	490
32	665	490
64	1331	490
128	2829	490
256	6183	490
512	13638	490
1024	30043	490

[†] Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

CFFT - NOSCALE

FFT Size	Cycles[†]	Code Size (in bytes)
16	299	361
32	558	361
64	1149	361
128	2500	361
256	5563	361
512	12434	361
1024	27689	361

[†] Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

cfft32 — *32-Bit Forward Complex FFT*

Function void cfft32 (LDATA *x, ushort nx, type);

Arguments

x[2*nx] — Pointer to input vector containing nx complex elements (2*nx real elements) in normal-order. On output, vector x contains the nx complex elements of the FFT(x) in bit-reversed order. Complex numbers are stored in the interleaved Re-Im format.

nx — Number of complex elements in vector x. Must be between 4 and 1024.

type — FFT type selector. Types supported:

If type = SCALE, scaled version selected

If type = NOSCALE, non-scaled version selected

Description Computes a complex nx-point FFT on vector x, which is in normal order. The original content of vector x is destroyed in the process. The nx complex elements of the result are stored in vector x in bit-reversed order.

Algorithm (DFT)

$$y[k] = \frac{1}{(scale\ factor)} * \sum_{i=0}^{nx-1} x[i] * \left(\left(\frac{2 * \pi * i * k}{nx}\right) - j \sin\left(\frac{2 * \pi * i * k}{nx}\right)\right)$$

Overflow Handling Methodology If scale==1, scaling before each stage is implemented for overflow prevention.

Special Requirements

This function requires the inclusion of fine “twiddle.inc” which contains the twiddle table (automatically included).

Data memory alignment (reference cfft.cmd in examples/cfft32 directory):

- Alignment of input database address: (n+1) LSBs must be zeros, where n=log2(nx).
- Ensure t hat the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).

For best performance, the data buffer has to be in a DARAM block.

Implementation Notes

Radix-2 DIT version of the FFT algorithm is implemented. The implementation is optimized for MIPS, not for code size.

If scale == 0, the first two stages are combined and implemented in radix-4 for MIPS optimization.

If scale == 1, the first two stages are not combined, but they are separately implemented to save multiplication operations for MIPS optimization. The last stage is also separately implemented because it doesn't need scaling operation.

Example See example/cfft32 subdirectory

Benchmarks

12 cycles for radix-2 butterfly in non-scaled version; 15 cycles for radix-2 butterfly in scaled version

21 cycles for radix-4 butterfly in non-scaled version

10 cycles for stage 1 loop in scaled version; 10 cycles for group 1 of stage 2 loop in scaled version; 13 cycles for group 2 of stage 2 in scaled version

CFFT32 - SCALE

FFT Size	Cycles†	Code Size (in bytes)
16	715	504
32	1712	504
64	4038	504
128	9412	504
256	21618	504
512	48960	504

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

CFFT – NOSCALE

FFT Size	Cycles†	Code Size (in bytes)
16	601	337
32	1461	337
64	3460	337
128	8083	337
256	18594	337
512	42161	337

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

cfft off-place — *Forward Complex FFT*

Function int* cfft_offplace (DATA *x, DATA *y, ushort nx, type);

Arguments

x [2*nx]	Pointer to input vector containing nx complex elements (2*nx real elements) in normal order. Complex numbers are stored in interleaved Re-Im format.
y[2*nx]	Pointer to vector of size 2*nx used in out-of-place computation.
Nx	Number of complex elements in vector x. Must be between 8 and 1024.
Type	FFT type selector. Types supported: If type = SCALE, scaled version selected If type = NOSCALE, non-scaled version selected

Description Computes a complex nx-point FFT on vector x, which is in normal order. The implementation is out-of-place, beginning with the third stage. The function provides a pointer to the array containing the output.

Algorithm (DFT)

$$y[k] = \frac{1}{(scale\ factor)} * \sum_{i=0}^{nx-1} x[i] * \left(\cos\left(\frac{2*\pi*i*k}{nx}\right) - j\sin\left(\frac{2*\pi*i*k}{nx}\right)\right)$$

Overflow Handling Methodology If type is SCALE, scaling before each stage is implemented for overflow prevention.

Special Requirements

Data memory alignment (reference cfft.cmd):

- 32-bit alignment for vectors x and y.
- Ensure that the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).
- For best performance, the data buffer x and the buffer y both must be in DARAM blocks, and in different DARAM blocks. The kernel then takes 5 cycles to execute.

This function requires the inclusion of file “twiddle.inc”, which contains the twiddle table (automatically included).

Implementation Notes

Radix-2 DIT version of the FFT algorithm is implemented. The implementation is optimized for MIPS, not for code size. In the no-scaling version, the first two stages are computed as radix-4 stage. In the scaling version, the first two radix-2 stages are unrolled.

This routine prevents overflow by scaling by 2 before each FFT stage, assuming that the parameter "SCALE" is passed.

The routine is an out-of-place implementation of the FFT, beginning with the third stage. The results of the computation on values from array x are stored in array y. In the next stage, the computation is done on values read from array y, and the results are stored in x, and so on. The nx complex elements of the final result are stored, depending on whether the total number of stages is even or odd, in either vector x or y, in bit-reversed order.

Example See examples/cfft subdirectory

Benchmarks

5 cycles (radix-2 butterfly)

10 cycles (radix-4 butterfly)

1 cycle/complex value (scaling)

CFFT_offplace_SCALE

FFT Size	Cycles†	Code Size (in bytes)
16	390	610
32	694	610
64	1326	610
128	2730	610
256	5802	610
512	12566	610
1024	27318	610

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

CFFT_offplace_NOSCALE

FFT Size	Cycles†	Code Size (in bytes)
16	337	444
32	590	444
64	1133	444
128	2354	444
256	5073	444
512	11126	444
1024	24469	444

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

cfir — *Complex FIR Filter*

Function

ushort oflag = cfir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)

Arguments

x[2*nx] — Pointer to input vector of nx complex elements.

h[2*nh] — Pointer to complex coefficient vector of size nh in normal order. For example, if nh=6, then h[nh] = {h0r, h0i, h1r, h1i h2r, h2i, h3r, h3i, h4r, h4i, h5r, h5i} where h0 resides at the lowest memory address in the array.

This array must be located in internal memory since it is accessed by the C55x coefficient bus.

r[2*nx] — Pointer to output vector of nx complex elements.
In-place computation (r = x) is allowed.

dbuffer[2*nh + 2]	Pointer to delay buffer of length nh =2 * nh + 2 In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed. The first element in this array is present for alignment purposes, the second element is special in that it contains the array index-1 of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to 0 (like all the other array entries) for the first block only.
nx	Number of complex input samples
nh	The number of complex coefficients of the filter. For example, if the filter coefficients are {h0, h1, h2, h3, h4, h5}, then nh = 6. Must be a minimum value of 3. For smaller filters, zero pad the coefficients to meet the minimum value.
oflag	Overflow error flag (returned value) If oflag = 1, a 32-bit data overflow has occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes a complex FIR filter (direct-form) using the coefficients stored in vector h. The complex input data is stored in vector x. The filter output result is stored in vector r. This function maintains the array dbuffer containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for both block-by-block (nx ≥ 2) and sample-by-sample filtering (nx = 1). In-place computation (r = x) is allowed.

Algorithm

$$r[j] = \sum_{k=0}^{nh-1} h[k]\, x[j-k] \qquad 0 \le j \le nx$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements nh must be a minimum value of 3. For smaller filters, zero pad the h[] array.

Implementation Notes The first element in the dbuffer array is present only for alignment purposes. The second element in this array (index=0) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index - 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4-1 shows the array in memory with an entry index of 2. The newest entry in the dbuffer is denoted by x(j-0), which in this case would occupy index = 3 in the array. The next newest entry is x(j-1), and so on. It is assumed that all x() entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

Figure 4-1, Figure 4-2, and Figure 4-3 show the dbuffer, x, and r arrays as they appear in memory.

Figure 4-1. dbuffer Array in Memory at Time j

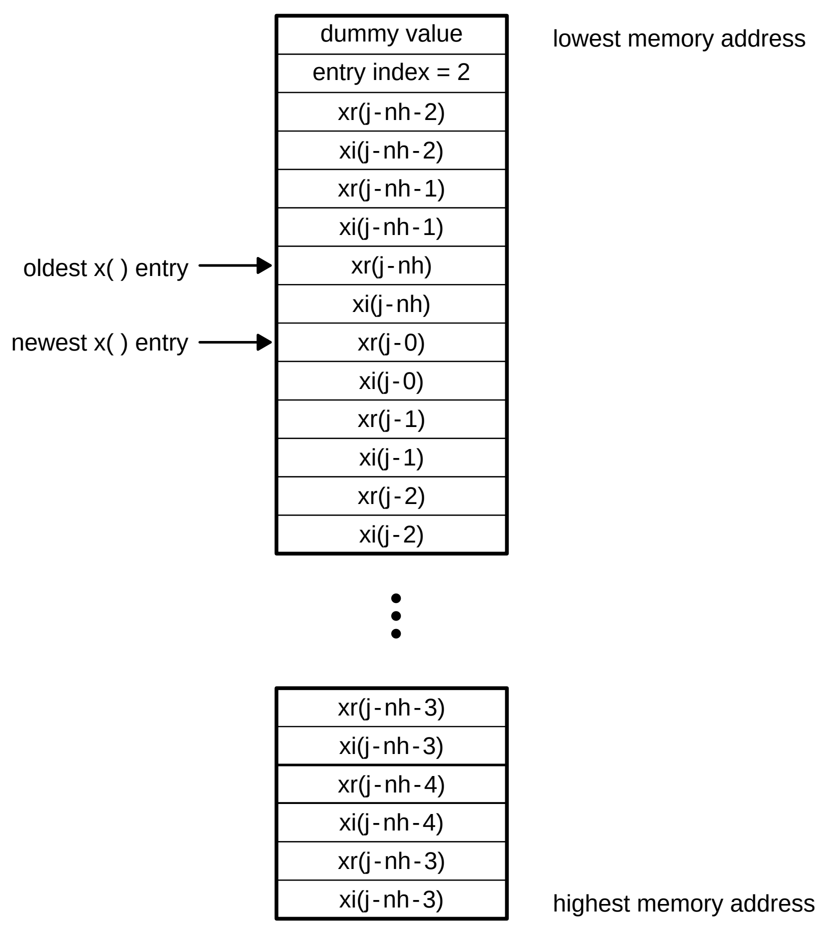


Figure 4-2. x Array in Memory

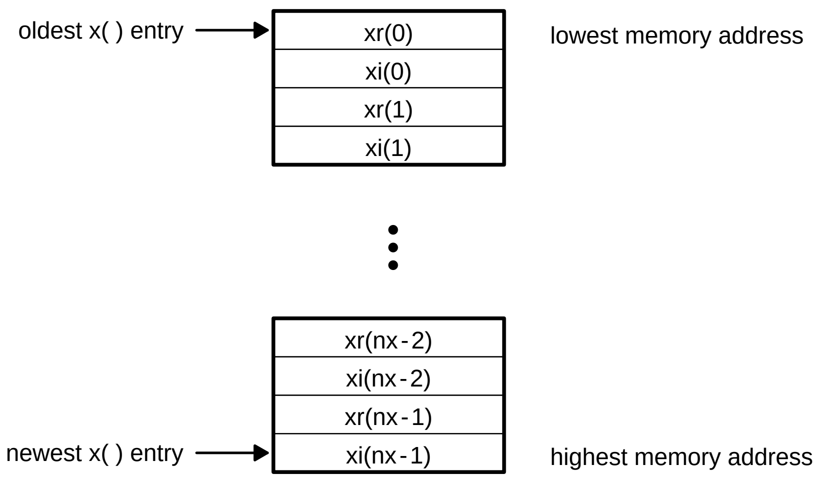
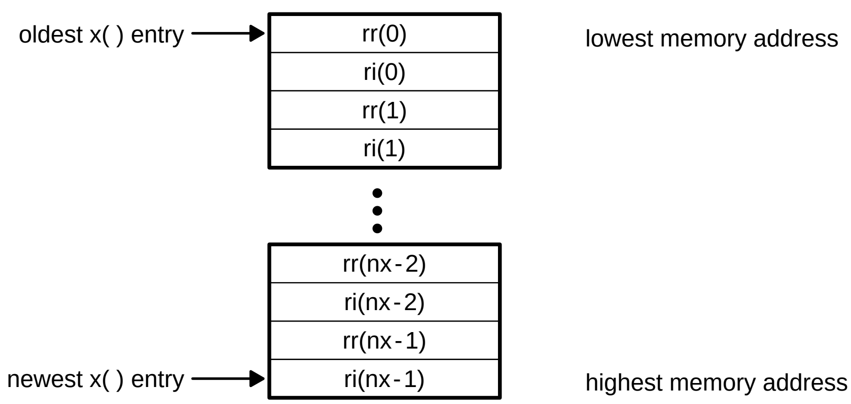


Figure 4-3. r Array in Memory



Example See examples/cfir subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx * [8 + 2(nh-2)]
	Overhead:	51
Code size (in bytes)	136	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

cifft — *Inverse Complex FFT*

Function

void cifft (DATA *x, ushort nx, type);
(defined in cifft.asm)

Arguments

x [2*nx]	Pointer to input vector containing nx complex elements (2*nx real elements) in normal order. On output, vector contains the nx complex elements of the IFFT(x) in bit-reversed order. Complex numbers are stored in interleaved Re-Im format.
nx	Number of complex elements in vector x. Must be between 16 and 1024.
type	FFT type selector. Types supported: If type = SCALE, scaled version selected If type = NOSCALE, non-scaled version selected

Description

Computes a complex nx-point IFFT on vector x, which is in normal order. The original content of vector x is destroyed in the process. The nx complex elements of the result are stored in vector x in bit-reversed order.

Algorithm

(IDFT)

$$y[k] = \frac{1}{(scale\ factor)} * \sum_{i=0}^{nx-1} x[w] * \left(\cos\left(\frac{-2 * \pi * i * k}{nx}\right) + j \sin\left(\frac{-2 * \pi * i * k}{nx}\right)\right)$$

Overflow Handling Methodology

If type = SCALE is selected, scaling before each stage is implemented for overflow prevention

Special Requirements

This function requires the inclusion of file “twiddle.inc” which contains the twiddle table (automatically included).

The twiddle table must be located in internal memory since it is accuired by the C55x coefficient bus.

Input data section is aligned on 32-bit boundary.

Input data section and twiddle table in different DARAM blocks:

- Twiddle table can be placed in an SARAM block in a different block than the input section.
- Input table is 32-bit aligned in SARAM or DARAM blocks.

Data memory alignment (reference cfft.cmd in examples/cfft directory):

- Alignment of input vector on a 32-bit boundary (2 of the LSBs of the byte address must be zero)
- Ensure that the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).
- For best performance the data buffer has to be in a DARAM block.
- If the twiddle table and the data buffer are in the same block then the radix-2 kernal is 7 cycles and the radix-4 kernel is not affected.

Implementation Notes

The implementations are optimized for MIPS, not for code size. They implement the decimation-in-time (DIT) FFT algorithm.

The NOSCALE version is implemented using radix-2 butterflies. The first two stages are replaced by a single radix-4 stage.

The SCALE version is implemented using only radix-2 stages. This routine prevents overflow by scaling by 2 before each FFT stage.

Example See examples/cifft subdirectory

Benchmarks (preliminary)

6 cycles (radix-2 butterfly - used in both scaled and non-scaled versions)

10 cycles (radix-4 butterfly – used in non-scaled version)

CIFFT - SCALE

FFT Size	Cycles†	Code Size (in bytes)
16	381	490
32	678	490
64	1343	490
128	2840	490
256	6187	490
512	13640	490
1024	30037	490

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

CFFT - NOSCALE

FFT Size	Cycles[†]	Code Size (in bytes)
16	308	356
32	569	356
64	1162	356
128	2515	356
256	5572	356
512	12435	356
1024	27691	356

[†] Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

cifft32 — *32-Bit Inverse Complex FFT*

Function void cifft32 (LDATA *x, ushort nx, type);

Arguments

x[2*nx]	Pointer to input vector containing nx complex elements (2*nx real elements) in normal-order. On output, vector x contains the nx complex elements of the iFFT(x) in bit-reversed order. Complex numbers are stored in the interleaved Re-Im format.
nx	Number of complex elements in vector x. Must be between 4 and 1024.
type	FFT type selector. Types supported: If type = SCALE, scaled version selected If type = NOSCALE, non-scaled version selected

Description Computes a complex nx-point iFFT on vector x, which is in normal-order. The original content of vector x is destroyed in the process. The nx complex elements of the result are stored in vector x in bit-reversed order.

Algorithm (iDFT)

$$y[k] = \frac{1}{(scale\ factor)} * \sum_{i=0}^{nx-1} x[i] * \left(\cos\left(\frac{2*\pi*i*k}{nx}\right) + j\sin\left(\frac{2*\pi*i*k}{nx}\right)\right)$$

Overflow Handling Methodology If scale == 1, scaling before each stage is implemented for overflow prevention.

Special Requirements

This function requires the inclusion of fine “twiddle.inc” which contains the twiddle table (automatically included).

Data memory alignment (reference cifft.cmd in examples/cifft32 directory):

- Alignment of input database address: (n+1) LSBs must be zeros, where n = log2(nx).
- Ensure t hat the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).

For best performance, the data buffer has to be in a DARAM block.

Implementation Notes

Radix-2 DIT version of the iFFT algorithm is implemented. The implementation is optimized for MIPS, not for code size.

If scale == 0, the first two stages are combined and implemented in radix-4 for MIPS optimization.

If scale == 1, the first two stages are not combined, but they are separately implemented to save multiplication operations for MIPS optimization. The last stage is also separately implemented because it doesn’t need scaling operation.

Example See example/cifft32 subdirectory

Benchmarks

12 cycles for radix-2 butterfly in non-scaled version; 15 cycles for radix-2 butterfly in scaled version

21 cycles for radix-4 butterfly in non-scaled version

10 cycles for stage 1 loop in scaled version; 10 cycles for group 1 of stage 2 loop in scaled version; 13 cycles for group 2 of stage 2 in scaled version

CIFFT32 - SCALE

iFFT Size	Cycles†	Code Size (in bytes)
16	715	504
32	1712	504
64	4038	504
128	9412	504
256	21618	504
512	48960	504

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

CFFT32 - NOSCALE

iFFT Size	Cycles†	Code Size (in bytes)
16	601	337
32	1461	337
64	3460	337
128	8083	337
256	18594	337
512	42161	337

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

cifft off-place

Forward Complex IFFT

Function

int* cifft_offplace (DATA *x, DATA *y, ushort nx, type);

Arguments

x [2*nx]	Pointer to input vector containing nx complex elements (2*nx real elements) in normal order. Complex numbers are stored in interleaved Re-Im format.
y[2*nx]	Pointer to vector of size 2*nx used in out-of-place computation.
Nx	Number of complex elements in vector x. Must be between 8 and 1024.

Type FFT type selector. Types supported:

If type = SCALE, scaled version selected
If type = NOSCALE, non-scaled version selected

Description Computes a complex nx-point FFT on vector x, which is in normal order. The implementation is out-of-place, beginning with the third stage. The function provides a pointer to the array containing the output.

Algorithm (DFT)

$$y[k] = \frac{1}{(scale\ factor)} * \sum_{i=0}^{nx-1} x[i] * \left(\cos\left(\frac{2 * \pi * i * k}{nx}\right) + j\sin\left(\frac{2 * \pi * i * k}{nx}\right)\right)$$

Overflow Handling Methodology If type is SCALE, scaling before each stage is implemented for overflow prevention.

Special Requirements

Data memory alignment (reference cfft.cmd):

- 32-bit alignment for vectors x and y.
- Ensure that the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).
- For best performance, the data buffer x and the buffer y both must be in DARAM blocks, and in different DARAM blocks. The kernel then takes 5 cycles to execute.

This function requires the inclusion of file “twiddle.inc”, which contains the twiddle table (automatically included).

Implementation Notes

Radix-2 DIT version of the IFFT algorithm is implemented. The implementation is optimized for MIPS, not for code size. In the no-scaling version, the first two stages are computed as radix-4 stage. In the scaling version, the first two radix-2 stages are unrolled.

This routine prevents overflow by scaling by 2 before each IFFT stage, assuming that the parameter “SCALE” is passed.

The routine is an out-of-place implementation of the IFFT, beginning with the third stage. The results of the computation on values from array x are stored in array y. In the next stage, the computation is done on values read from array y, and the results are stored in x, and so on. The nx complex elements of the final result are stored, depending on whether the total number of stages is even or odd, in either vector x or y, in bit-reversed order.

Example See examples/cfft subdirectory

Benchmarks

5 cycles (radix-2 butterfly)

10 cycles (radix-4 butterfly)

1 cycle/complex value (scaling)

CIFFT_offplace_SCALE

FFT Size	Cycles†	Code Size (in bytes)
16	386	608
32	687	608
64	1322	608
128	2723	608
256	5798	608
512	12559	608
1024	27314	608

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

CIFFT_offplace_NOSCALE

FFT Size	Cycles†	Code Size (in bytes)
16	340	442
32	595	442
64	1140	442
128	2363	442
256	5084	442
512	11139	442
1024	24484	442

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

convol — *Convolution*

Function ushort oflag = convol (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)

Arguments

x[nr+nh-1]	Pointer to input vector of nr + nh - 1 real elements.
h[nh]	Pointer to input vector of nh real elements.
r[nr]	Pointer to output vector of nr real elements.
nr	Number of elements in vector r. In-place computation (r = x) is allowed (see Description section for comment).
nh	Number of elements in vector h.
oflag	Overflow error flag (returned value) If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes the real convolution of two real vectors x and h, and places the results in vector r. Typically used for block FIR filter computation when there is no need to retain an input delay buffer. This function can also be used to implement single-sample FIR filters (nr = 1) provided the input delay history for the filter is maintained external to this function. In-place computation (r = x) is allowed, but be aware that the r output vector is shorter in length than the x input vector; therefore, r will only overwrite the first nr elements of the x.

Algorithm

$$r[j] = \sum_{k=0}^{nh-1} h[k]\, x[j-k] \qquad 0 \le j \le nr$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements none

Implementation Notes Figure 4-4, Figure 4-5, and Figure 4-6 show the x, r, and h arrays as they appear in memory.

Figure 4-4. x Array in Memory

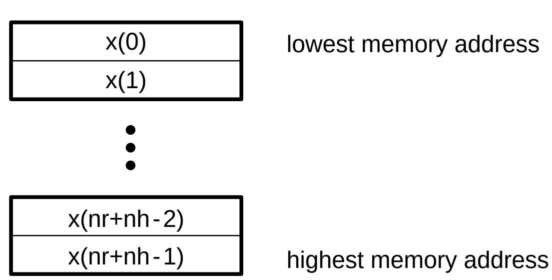


Figure 4-5. r Array in Memory

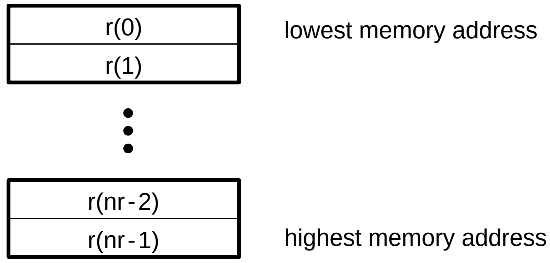
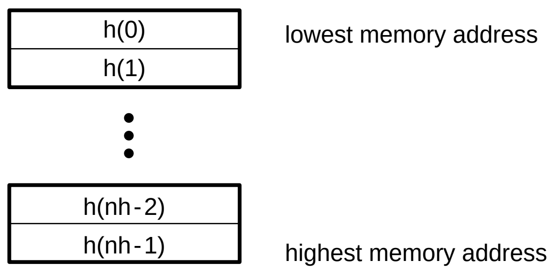


Figure 4-6. h Array in Memory



Example See examples/convol subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nr * (1 + nh)
	Overhead:	44
Code size (in bytes)	88	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

convol1 — *Convolution (fast)*

Function ushort oflag = convol1 (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)

Arguments

x[nr+nh-1]	Pointer to input vector of nr+nh-1 real elements.
h[nh]	Pointer to input vector of nh real elements.
r[nr]	Pointer to output vector of nr real elements. In-place computation (r = x) is allowed (see Description section for comment).
nr	Number of elements in vector r. Must be an even number.
nh	Number of elements in vector h.
oflag	Overflow error flag (returned value)

- If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result.
- If oflag = 0, a 32-bit overflow has not occurred.

Description Computes the real convolution of two real vectors x and h, and places the results in vector r. This function utilizes the dual-MAC capability of the C55x to process in parallel two output samples for each iteration of the inner function loop. It is, therefore, roughly twice as fast as CONVOL, which implements only a single-MAC approach. However, the number of output samples (nr) must be even. Typically used for block FIR filter computation when there is no need to retain an input delay buffer. This function can also be used to implement single-sample FIR filters (nr = 1) provided the input delay history for the filter is maintained external to this function. In-place computation (r = x) is allowed, but be aware that the r output vector is shorter in length than the x input vector; therefore, r will only overwrite the first nr elements of the x.

Algorithm

$$r[j] = \sum_{k=0}^{nh-1} h[k]\, x[j-k] \qquad 0 \le j \le nr$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements

nr must be an even value.

The vector h[nh] must be located in internal memory since it is accessed using the C55x coefficient bus, and that bus does not have access to external memory.

Implementation Notes Figure 4-7, Figure 4-8, and Figure 4-9 show the x, r, and h arrays as they appear in memory.

Figure 4-7. x Array in Memory

Figure 4-8. r Array in Memory

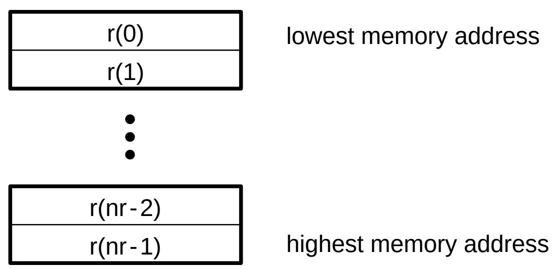
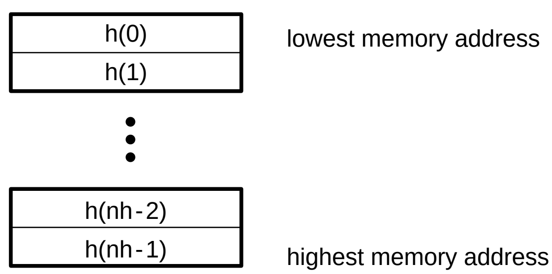


Figure 4-9. h Array in Memory



Example See examples/convol1 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nr/2 * [3+(nh-2)]
	Overhead:	58
Code size (in bytes)	101	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

convol2 — *Convolution (fastest)*

Function ushort oflag = convol2 (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)

Arguments

x[nr+nh-1]	Pointer to input vector of nr + nh - 1 real elements.
h[nh]	Pointer to input vector of nh real elements.
r[nr]	Pointer to output vector of nr real elements. In-place computation (r = x) is allowed (see Description section for comment). This array must be aligned on a 32-bit boundary in memory.
nr	Number of elements in vector r. Must be an even number.
nh	Number of elements in vector h.
oflag	Overflow error flag (returned value) If oflag = 1, a 32-bit data overflow has occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes the real convolution of two real vectors x and h, and places the results in vector r. This function utilizes the dual-MAC capability of the C55x to process in parallel two output samples for each iteration of the inner function loop. It is, therefore, roughly twice as fast as CONVOL, which implements only a single-MAC approach. However, the number of output samples (nr) must be even. In addition, this function offers a small performance improvement over CONVOL1 at the expense of requiring the r array to be 32-bit aligned in memory. Typically used for block FIR filter computation when there is no need to retain an input delay buffer. This function can also be used to implement single-sample FIR filters (nr = 1) provided the input delay history for the filter is maintained external to this function. In-place computation (r = x) is allowed, but be aware that the r output vector is shorter in length than the x input vector; therefore, r will only overwrite the first nr elements of the x.

Algorithm

$$r[j] = \sum_{k=0}^{nh-1} h[k]\,x[j-k] \qquad 0 \le j \le nr$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements

nr must be an even value.

The vector h[nh] must be located in internal memory since it is accessed using the C55x coefficient bus, and that bus does not have access to external memory.

The vector r[nr] must be 32-bit aligned in memory.

Implementation Notes Figure 4-10, Figure 4-11, and Figure 4-12 show the x, r, and h arrays as they appear in memory.

Figure 4-10. x Array in Memory

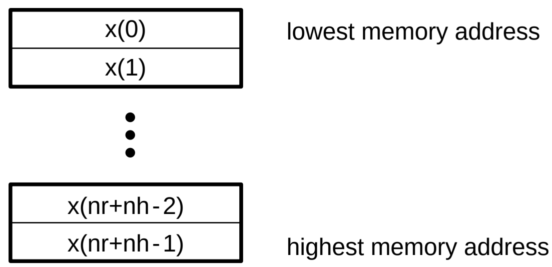


Figure 4-11.r Array in Memory

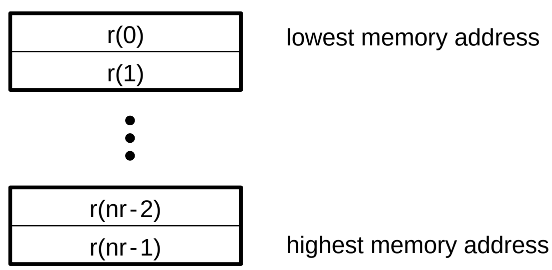
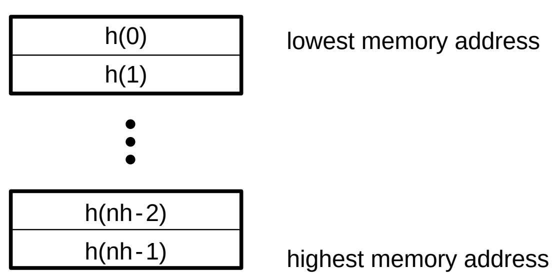


Figure 4-12. h Array in Memory



Example See examples/convol2 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nr/2 * (1 + nh)
	Overhead:	24
Code size (in bytes)	100	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

corr *Correlation, full-length*

Function ushort oflag = corr (DATA *x, DATA *y, DATA *r, ushort nx, ushort ny, type)

Arguments

x [nx]	Pointer to real input vector of nx real elements.
x [ny]	Pointer to real input vector of ny real elements.
r[nx+ny-1]	Pointer to real output vector containing the full-length correlation (nx + ny - 1 elements) of vector x with y. r must be different than both x and y (in-place computation is not allowed).
nx	Number of real elements in vector x
ny	Number of real elements in vector y
type	Correlation type selector. Types supported: If type = raw, r contains the raw correlation If type = bias, r contains the biased-correlation If type = unbias, r contains the unbiased-correlation
oflag	Overflow flag If oflag = 1, a 32-bit overflow has occurred If oflag = 0, a 32-bit overflow has not occurred

Description Computes the full-length correlation of vectors x and y and stores the result in vector r. using time-domain techniques.

Algorithm Raw correlation

$$r[j] = \sum_{k=o}^{nr-j-1} x[j+k] * y[k] \qquad 0 \le j \le nr = nx + ny - 1$$

Biased correlation

$$r[j] = \frac{1}{nr} \sum_{k=o}^{nr-j-1} x[j+k] * y[k] \qquad 0 \le j \le nr = nx + ny - 1$$

Unbiased correlation

$$r[j] = \frac{1}{(nx - abs(j))} \sum_{k=o}^{nr-j-1} x[j+k] * y[k] \qquad 0 \le j \le nr = nx + ny - 1$$

Overflow Handling Methodology No scaling implemented for overflow prevention

Special Requirements none

Implementation Notes

Special debugging consideration: This function is implemented as a macro that invokes different correlation routines according to the *type* selected. As a consequence the *corr* symbol is not defined. Instead the corr_raw, corr_bias, corr_unbias symbols are defined.

Correlation is implemented using time-domain techniques

Benchmarks (preliminary)

Cycles	Raw:	2 times faster than C54x
	Unbias:	2.14 times faster than C54x
	Bias:	2.1 times faster than C54x
Code size (in bytes)	Raw:	318
	Unbias:	417
	Bias:	356

dlms — *Adaptive Delayed LMS Filter*

Function

ushort oflag = dlms (DATA *x, DATA *h, DATA *r, DATA *des, DATA *dbuffer, DATA step, ushort nh, ushort nx)
(defined in dlms.asm)

Arguments

x[nx]	Pointer to input vector of size nx
h[nh]	Pointer to filter coefficient vector of size nh. h is stored in reversed order : h(n-1), ... h(0) where h[n] is at the lowest memory address. Memory alignment: h is a circular buffer and must start in a k-bit boundary(that is, the k LSBs of the starting address must be zeros) where k = log2(nh)
r[nx]	Pointer to output data vector of size nx. r can be equal to x.
des[nx]	Pointer to expected output array
dbuffer[nh+2]	Pointer to the delay buffer structure. The delay buffer is a structure comprised of an index register and a circular buffer of length nh + 1. The index register is the index into the circular buffer of the oldest data sample.
nh	Number of filter coefficients. Filter order = nh - 1. nh ≥ 3
nx	Length of input and output data vectors
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred If oflag = 0, a 32-bit overflow has not occurred

Description

Adaptive delayed least-mean-square (LMS) FIR filter using coefficients stored in vector h. Coefficients are updated after each sample based on the LMS algorithm and using a constant step = 2*μ. The real data input is stored in vector dbuffer. The filter output result is stored in vector r .

LMS algorithm uses the previous error and the previous sample (delayed) to take advantage of the C55x LMS instruction.

The delay buffer used is the same delay buffer used for other functions in the C55x DSP Library. There is one more data location in the circular delay buffer than there are coefficients. Other C55x DSP Library functions use this delay buffer to accommodate use of the dual-MAC architecture. In the DLMS function, we make use of the additional delay slot to allow coefficient updating as well as FIR calculation without a need to update the circular buffer in the interim operations.

The FIR output calculation is based on x(i) through x(i-nh+1). The coefficient update for a **delayed** LMS is based on x(i-1) through x(i-nh). Therefore, by having a delay buffer of nh+1, we can perform all calculations with the given delay buffer containing delay values of x(i) through x(i-nh). If the delay buffer was of length nh, the oldest data sample, x(i-nh), would need to be updated with the newest data sample, x(i), sometime after the calculation of the first coefficient update term, but before the calculation of the last FIR term.

Algorithm

FIR portion

$$r[j] = \sum_{k=0}^{nh-1} h[k] * x[i-k] \qquad 0 \le i \le nx - 1$$

Adaptation using the previous error and the previous sample:

$$e(i) = des(i-1) - r(i-1)$$
$$h_k(i+1) = h_k(i) + 2 * \mu * e(i-1) * x(i-k-1)$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements Minimum of 2 input and desired data samples. Minimum of 2 coefficients

Implementation Notes

Delayed version implemented to take advantage of the C55x LMS instruction.

Effect of using delayed error signal on convergence minimum:
For reference, the following is the algorithm for the regular LMS (non-delayed):

FIR portion

$$r[j] = \sum_{k=0}^{nh-1} h[k] * x[i-k] \qquad 0 \le i \le nx - 1$$

Adaptation using the current error and the current sample:

$$e(i) = des(i) - r(i)$$
$$h_k(i+1) = h_k(i) + 2 * \mu * e(i) * x(i-k)$$

Example See examples/dlms subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx * (7 + 2*(nh - 1)) = nx * (5 + 2 * nh)
	Overhead:	26
Code size (in bytes)	122	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

dlmsfast *Adaptive Delayed LMS Filter (fast implemented)*

Function ushort oflag = dlmsfast (DATA *x, DATA *h, DATA *r, DATA *des, DATA *dbuffer, DATA step, ushort nh, ushort nx)
This function is implemented for better performance on large number of filter orders.
(defined in dlmsfast.asm)

Arguments

x[nx] Pointer to input vector of size nx.

h[2*nh] Pointer to filter coefficient array of size 2*nh. This array contains two coefficient buffers h_coef and h_scratch. The upated coefficients in different time slot are stored into these two buffers alternatively. The final updated coefficients are stored in h_coef.

> h_coef is stored in reversed order: h_coef(n-1), ... h_coef(0) where h_coef(n-1) is at the lowest memory address of the first half of array h.
>
> h_scratch is stored in reversed order : h_scratch(n-1), ... h_scratch(0) where h_scratch(n-1) is at the lowest memory address of the second half of array h.
>
> Memory alignment: h must be aligned in 32 bytes boundary.

r[nx] Pointer to output data vector of size nx. r can be equal to x.

des[nx] Pointer to expected output array.

dbuffer[nh+3]	Pointer to the delay buffer structure. The delay buffer is a structure comprised of an index register and a circular buffer of length nh+2. The index register is the index into the circular buffer of the oldest data sample. Memory alignment: dbuffer must be aligned in 32 bytes boundary.
nh	Number of filter coefficients. Filter order = nh-1. nh has to be a even number. nh $\geq$ 10.
nx	Length of input and output data vectors. nx has to be a even number.
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred If oflag = 0, a 32-bit overflow has not occurred

Description

Adaptive delayed least-mean-square (LMS) FIR filter using coefficients stored in vector h. Coefficients are updated after each sample based on the LMS algorithm and using a constant step = 2*μ. The real data input is stored in vector dbuffer. The filter output result is stored in vector r.

Unlike the DLMS function in DSPLIB, which uses C55x LMS instruction to do partial filtering and addition of delta h to the coefficient, this fast LMS algorithm is implemented by doing coefficient updating and filtering separately to get better cycle count.

In this implementation, two input data are processed as a pair. The filtering operation uses dual-MAC to process two time slots of data and two set of coefficients are updated corresponding to these two time slots.

The delay buffer used is the same delay buffer used for other functions in the C55x DSP Library. There is two more data location in the circular delay buffer than there are coefficients. Other C55x DSP Library functions use this delay buffer to accommodate use of the dual-MAC architecture. In the DLMS function, we make use of the additional delay slots to allow coefficient updating as well as FIR calculation without a need to update the circular buffer in the interim operations.

The first time slot of FIR output calculation is based on x(i) through x(i-nh+1). While the coefficient update for a **delayed** LMS is based on x(i-1) through x(i-nh). The second time slot of FIR output is based on x(i+1) through x(i-nh+2). While the coefficient update for the delayed LMS is based on x(i) through x(i-nh+1). Therefore, by having a delay buffer of nh+2, we can perform all calculations with the given delay buffer containing delay values of x(i) through x(i-nh+1).

Algorithm

FIR portion:

$$r[i] = \sum_{k=0}^{nh-1} h[k] * x[i-k] \qquad 0 \le i \le nx-1$$

Adaptation using the previous error and the previous sample:

$$e(i) = des(i-1) - r(i-1)$$
$$h_k(i+1) = h_k(i) + 2 * \mu * e(i-1) * x(i-k-1)$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements

This function is good for large number of input data.

Minimum of 10 coefficients. Coefficient buffer need to be aligned on 32 bytes boundary.

dbuffer need to be aligned on 32 bytes memory boundary.

Coefficient buffer and dbuffer need to be put into different block of memory to get better performance.

Implementation Notes

Filtering and coefficient updating are implemented separately. Figure 4-13, Figure 4-14, and Figure 4-15 show the x buffer, dbuffer, and h buffers.

Figure 4-13. x Buffer

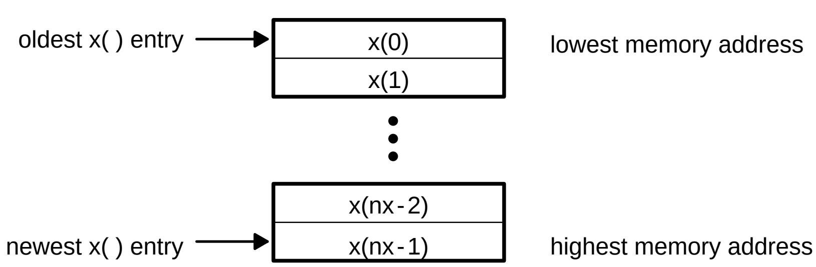


Figure 4-14. dbuffer

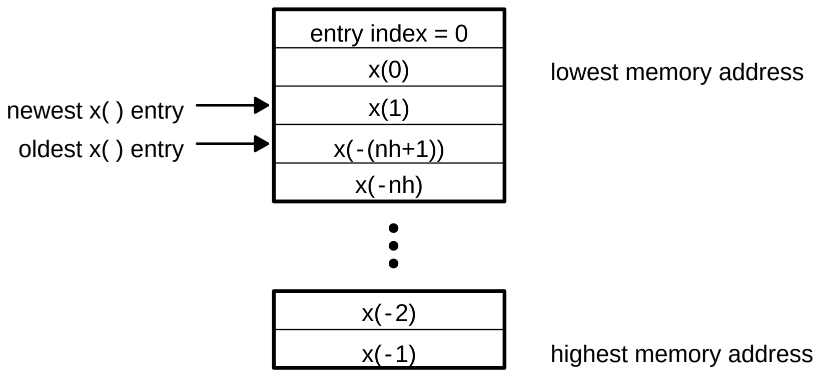
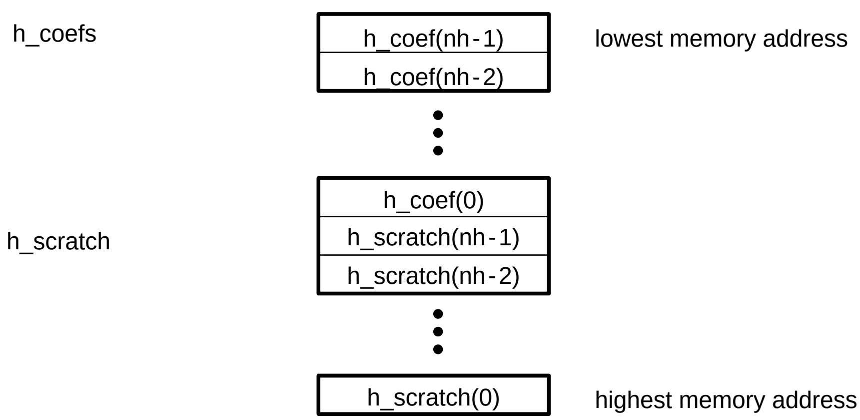


Figure 4-15. h Buffers



Effect of using delayed error signal on convergence minimum. For reference, the following is the algorithm for the regular LMS (non-delayed):

FIR portion

$$r[i] = \sum_{k=0}^{nh-1} h[k] * x[i-k] \qquad 0 \le i \le nx-1$$

Adaptation using the current error and the current sample

$$e(i) = des(i) - r(i)$$
$$h_k(i+1) = h_k(i) + 2 * \mu * e(i) * x(i-k)$$

Example See examples/dlmsfast subdirectory

Benchmarks

Cycles†	Core:	nx/2 * (26 + 3*nh)
	Overhead:	71
Code size (in bytes)	322	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

expn — *Exponential Base e*

Function
ushort oflag = expn (DATA *x, DATA *r, ushort nx)
(defined in expn.asm)

Arguments

x[nx]	Pointer to input vector of size nx. x contains the numbers normalized between (-1,1) in q15 format
r[nx]	Pointer to output data vector (Q3.12 format) of size nx. r can be equal to x.
nx	Length of input and output data vectors
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred If oflag = 0, a 32-bit overflow has not occurred

Description
Computes the exponent of elements of vector x using Taylor series.

Algorithm
for $(i = 0;\ i < nx;\ i++)\quad y(i) = e^{x(i)}$ where $-1 < x(i) < 1$

Overflow Handling Methodology Not applicable

Special Requirements Linker command file: you must allocate .data section (for polynomial coefficients)

Implementation Notes Computes the exponent of elements of vector x. It uses the following Taylor series:

$$\exp(x) = c0 + (c1 * x) + (c2 * x^2) + (c3 * x^3) + (c4 * x^4) + (c5 * x^5)$$

where
c0 = 1.0000
c1 = 1.0001
c2 = 0.4990
c3 = 0.1705
c4 = 0.0348
c5 = 0.0139

Example See examples/expn subdirectory

Benchmarks (preliminary)

Cycles†	Core:	11 * nx
	Overhead:	18
Code size (in bytes)	57	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

fir *FIR Filter*

Function ushort oflag = fir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)

Arguments

x[nx] Pointer to input vector of nx real elements.

h[nh] Pointer to coefficient vector of size nh in normal order. For example, if nh=6, then h[nh] = {h0, h1, h2, h3, h4, h5} where h0 resides at the lowest memory address in the array.

This array must be located in internal memory since it is accessed by the C55x coefficient bus.

r[nx] Pointer to output vector of nx real elements. In-place computation (r = x) is allowed.

dbuffer[nh+2]	Pointer to delay buffer of length nh = nh + 2 In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed. The first element in this array is special in that it contains the array index-1 of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to 0 (like all the other array entries) for the first block only.
nx	Number of input samples
nh	The number of coefficients of the filter. For example, if the filter coefficients are {h0, h1, h2, h3, h4, h5}, then nh = 6. Must be a minimum value of 3. For smaller filters, zero pad the coefficients to meet the minimum value.
oflag	Overflow error flag (returned value) If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes a real FIR filter (direct-form) using the coefficients stored in vector h. The real input data is stored in vector x. The filter output result is stored in vector r. This function maintains the array dbuffer containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for both block-by-block (nx ≥ 2) and sample-by-sample filtering (nx = 1). In place computation (r = x) is allowed.

Algorithm

$$r[j] = \sum_{k=0}^{nh-1} h[k]\, x[j-k] \qquad 0 \le j \le nx$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements nh must be a minimum value of 3. For smaller filters, zero pad the h[] array.

Implementation Notes The first element in the dbuffer array (index = 0) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index - 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4-16 shows the array in memory with an entry index of 2. The newest entry in the dbuffer is denoted by x(j-0), which in this case would occupy index = 3 in the array. The next newest entry is x(j-1), and so on. It is assumed that all x() entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

The dbuffer array actually contains one more history value than is needed to implement this filter. The value x(j-nh) does not enter into the calculations for for the output r(j). However, this value is required in other DSPLIB filter functions that utilize the dual-MAC units on the C55x, such as FIR2. Including this extra location ensures compatibility across all filter functions in the C55x DSPLIB.

Figure 4-16, Figure 4-17, and Figure 4-18 show the dbuffer, x, and r arrays as they appear in memory.

Figure 4-16. dbuffer Array in Memory at Time j

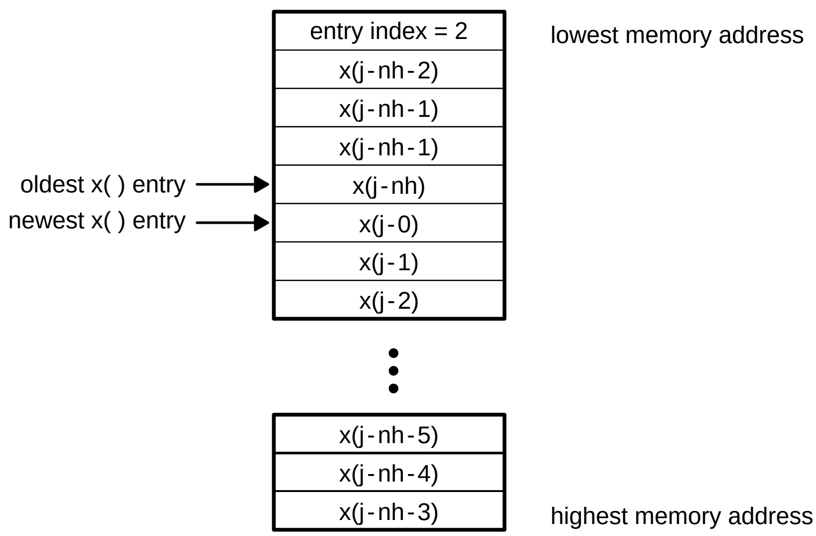


Figure 4-17. x Array in Memory

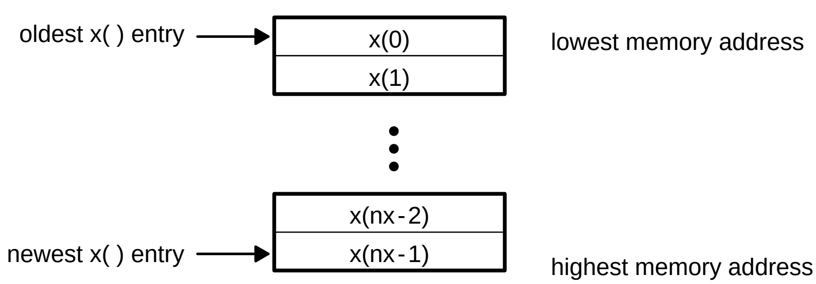
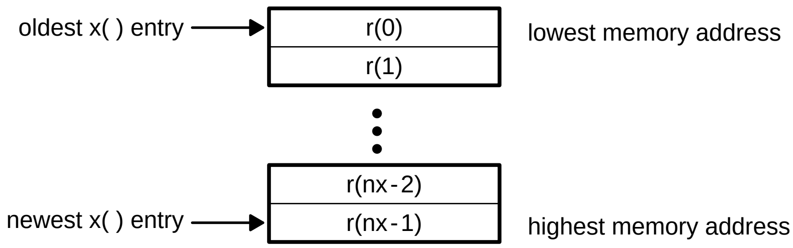


Figure 4-18. r Array in Memory



Example See examples/fir subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx * (2 + nh)
	Overhead:	25
Code size (in bytes)	107	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

fir2 *Block FIR Filter (fast)*

Function ushort oflag = fir2 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)

Arguments

x[nx] Pointer to input vector of nx real elements.

r[nx] Pointer to output vector of nx real elements. In-place computation (r = x) is allowed.

h[nh] Pointer to coefficient vector of size nh in normal order. For example, if nh=6, then h[nh] = {h0, h1, h2, h3, h4, h5} where h0 resides at the lowest memory address in the array.

This array must be located in internal memory since it is accessed by the C55x coefficient bus.

dbuffer[nh + 2]	Pointer to delay buffer of length nh = nh + 2 In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed. The first element in this array is special in that it contains the array index-1 of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to 0 (like all the other array entries) for the first block only.
nx	Number of input samples. Must be an even number.
nh	The number of coefficients of the filter. For example, if the filter coefficients are {h0, h1, h2, h3, h4, h5}, then nh = 6. Must be a minimum value of 3. For smaller filters, zero pad the coefficients to meet the minimum value.
oflag	Overflow error flag (returned value) If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes a real block FIR filter (direct-form) using the coefficients stored in vector h. This function utilizes the dual-MAC capability of the C55x to process in parallel two output samples for each iteration of the inner function loop. It is, therefore, roughly twice as fast as FIR, which implements only a single-MAC approach. However, the number of input samples (nx) must be even The real input data is stored in vector x. The filter output result is stored in vector r. This function maintains the array dbuffer containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for block-by-block filtering only (nx ≥ 2). It cannot be used for sample-by-sample filtering (nx = 1). In-place computation (r = x) is allowed.

Algorithm

$$r[j] = \sum_{k=0}^{nh-1} h[k]\, x[j-k] \qquad 0 \le j \le nx$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements

nh must be a minimum value of 3. For smaller filters, zero pad the h[] array.

nx must be an even value.

Coefficient array h[nh] must be located in internal memory since it is accessed using the C55x coefficient bus, and that bus does not have access to external memory.

Pointer to the output vector r should be aligned to the 32-bit boundary.

Implementation Notes The first element in the dbuffer array (index = 0) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index - 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4-19 shows the array in memory with an entry index of 2. The newest entry in the dbuffer is denoted by x(j-0), which in this case would occupy index = 3 in the array. The next newest entry is x(j-1), and so on. It is assumed that all x() entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

Figure 4-19, Figure 4-20, and Figure 4-21 show the dbuffer, x, and r arrays as they appear in memory.

Figure 4-19. dbuffer Array in Memory at Time j

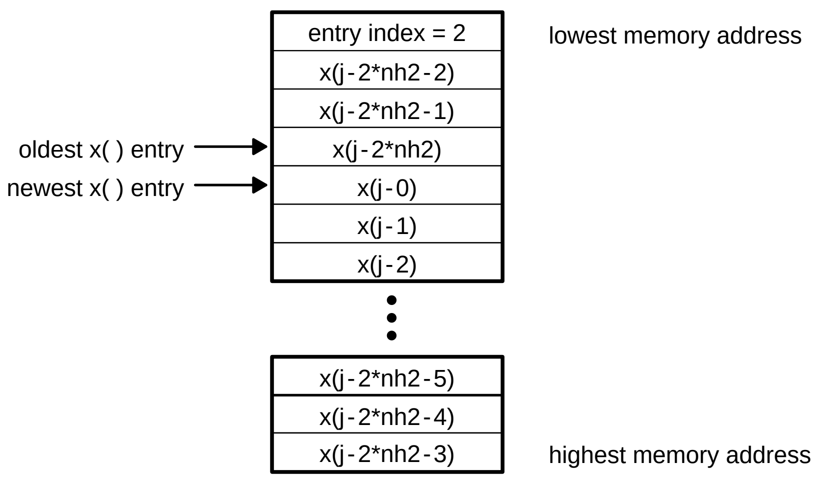


Figure 4-20. x Array in Memory

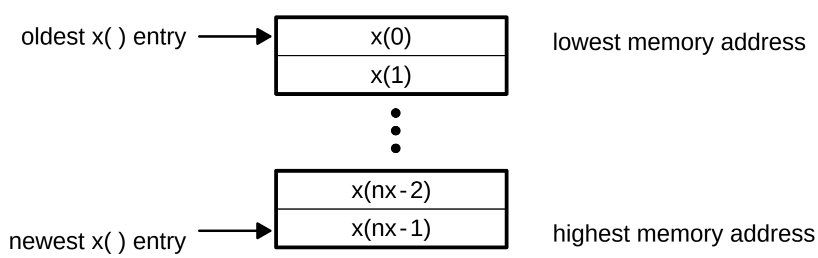
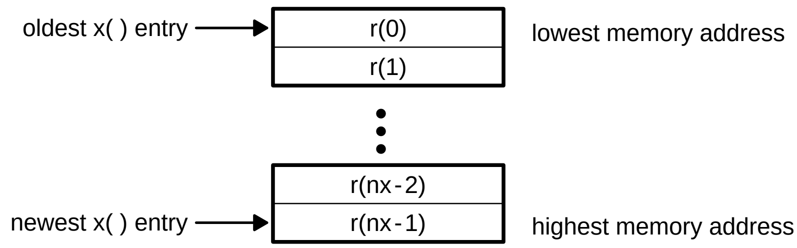


Figure 4-21. r Array in Memory



Example See examples/fir2 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	(nx/2) * [9+(nh-2)]
	Overhead:	32
Code size (in bytes)	134	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

firdec — *Decimating FIR Filter*

Function ushort oflag = firdec (DATA *x, DATA *h, DATA *r, DATA *dbuffer , ushort nh, ushort nx, ushort D)
(defined in decimate.asm)

Arguments

x [nx]	Pointer to real input vector of nx real elements.
h[nh]	Pointer to coefficient vector of size nh in normal order: H = b0 b1 b2 b3 …
r[nx/D]	Pointer to real input vector of nx/D real elements. In-place computation (r = x) is allowed
dbuffer[nh+1]	Delay buffer

In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves previous delayed input samples. It also preserves a ptr to the next new entry into the dbuffer. This ptr is preserved across function calls in dbuffer[0].

nx	Number of real elements in vector x
nh	Number of coefficients
D	Decimation factor. For example a D = 2 means you drop every other sample. Ideally, nx should be a multiple of D. If not, the trailing samples will be lost in the process.
oflag	Overflow error flag If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes a decimating real FIR filter (direct-form) using coefficient stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r. This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function can be used for both block-by-block and sample-by-sample filtering (nx = 1).

Algorithm

$$r[j] = \sum_{k=0}^{nh} h[k]\, x[j * D - k] \qquad 0 \le j \le nx$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements none

Implementation Notes none

Example See examples/decim subdirectory

Benchmarks (preliminary)

Cycles	Core:	(nx/D)*(10+nh+(D-1))
	Overhead	67
Code size (in bytes)	144	

firinterp — *Interpolating FIR Filter*

Function

ushort oflag = firinterp (DATA *x, DATA *h, DATA *r, DATA *dbuffer , ushort nh, ushort nx, ushort I)
(defined in interp.asm)

Arguments

x [nx]	Pointer to real input vector of nx real elements.
h[nh]	Pointer to coefficient vector of size nh in normal order: H = b0 b1 b2 b3 …
r[nx*I]	Pointer to real output vector of nx real elements. In-place computation (r = x) is allowed
dbuffer[(nh/I)+1]	Delay buffer of (nh/I)+1 elements In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves delayed input samples in dbuffer[1…(nh/I)+1]. It also preserves a ptr to the next new entry into the dbuffer. This ptr is preserved across function calls in dbuffer[0]. The delay buffer is only nh/I elements and holds only delayed x inputs. No zero-samples are inserted into dbuffer (since only non-zero products contribute to the filter output)
nx	Number of real elements in vector x and r
nh	Number of coefficients, with (nh/I) $\geq$ 3
I	Interpolation factor. I is effectively the number of output samples for every input sample. This routine can be used with I=1.
oflag	Overflow error flag If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes an interpolating real FIR filter (direct-form) using coefficient stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r. This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function can be used for both block-by-block and sample-by-sample filtering (nx = 1).

Algorithm

$$r[t] = \sum_{k=0}^{nh} h[k]x\left[\frac{t}{I-k}\right] \qquad 0 \le j \le nr$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements none

Implementation Notes none

Example See examples/decimate subdirectory

Benchmarks (preliminary)

Cycles	Core: If I > 1 nx*(2+I*(1+(nh/I))) If I=1 : nx*(2+nh) Overhead 72
Code size (in bytes)	164

firlat — *Lattice Forward (FIR) Filter*

Function ushort oflag = firlat (DATA *x, DATA *h, DATA *r, DATA *pbuffer, int nx, int nh)

Arguments

x [nx] Pointer to real input vector of nx real elements in normal order:
x[0]
x[1]
.
.
x[nx-2]
x[nx-1]

h[nh] Pointer to lattice coefficient vector of size nh in normal order:
h[0]
h[1]
.
.
h[nh-2]
h[nh-1]

r[nx] Pointer to output vector of nx real elements. In-place computation (r = x) is allowed.
r[0]
r[1]
.
.
r[nx-2]
r[nx-1]

pbuffer [nh] Delay buffer

In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.

pbuffer: procession buffer of nh length in order:
e′0[n-1]
e′1[n-1]
.
.
e′nh-2[n-1]
e′nh-1[n-1]

nx Number of real elements in vector x (input samples)

nh Number of coefficients

oflag Overflow error flag

If oflag = 1, a 32-bit data overflow has occurred in an intermediate or final result

If oflag = 0, a 32-bit overflow has not occurred.

Description Computes a real lattice FIR filter implementation using coefficient stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r. This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function can be used for both block-by-block and sample-by-sample filtering (nx=1)

Algorithm

$$e_0[n] = e'_0[n] = x[n],$$
$$e_i[n] = e_{i-1}[n] - hie'_{i-1}[n-1], \qquad i = 1, 2, \ldots, N$$
$$e'_i[n] = h_i e_{i-1}[n] + e'_{i-1}[n-1], \qquad i = 1, 2, \ldots, N$$
$$y[n] = e_N[n]$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements none

Implementation Notes none

Example See examples/firlat subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx{4 + 4(nh-1)]
	Overhead:	23
Code size (in bytes)	53	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

firs — *Symmetric FIR Filter*

Function ushort oflag = firs (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh2)

Arguments

x[nx]	Pointer to input vector of nx real elements.
r[nx]	Pointer to output vector of nx real elements. In-place computation (r = x) is allowed.
h[nh2]	Pointer to coefficient vector containing the first half of the symmetric filter coefficients. For example, if the filter coefficients are {h0, h1, h2, h2, h1, h0}, then h[nh2] = {h0, h1, h2} where h0 resides at the lowest memory address in the array. This array must be located in internal memory since it is accessed by the C55x coefficient bus.
dbuffer[2*nh2 + 2]	Pointer to delay buffer of length nh = 2*nh2 + 2 In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed. The first element in this array is special in that it contains the array index of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to 0 (like all the other array entries) for the first block only.
nx	Number of input samples

nh2	Half the number of coefficients of the filter (due to symmetry there is no need to provide the other half). For example, if the filter coefficients are {h0, h1, h2, h2, h1, h0}, then nh2 = 3. Must be a minimum value of 3. For smaller filters, zero pad the coefficients to meet the minimum value.
oflag	Overflow error flag (returned value) If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes a real FIR filter (direct-form) with nh2 symmetric coefficients using the FIRS instruction approach. The filter is assumed to have a symmetric impulse response, with the first half of the filter coefficients stored in the array h. The real input data is stored in vector x. The filter output result is stored in vector r. This function maintains the array dbuffer containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for both block-by-block (nx ≥ 2) and sample-by-sample filtering (nx = 1). In-place computation (r = x) is allowed.

Algorithm

$$r[j] = \sum_{k=0}^{nh2-1} h,\ldots,[k] * \ (x[j-k] + x[j+k-2*nh2+1]) \qquad 0 \le j \le nx$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements

nh must be a minimum value of 3. For smaller filters, zero pad the h[] array.

Coefficient array h[nh2] must be located in internal memory since it is accessed using the C55x coefficient bus, and that bus does not have access to external memory.

Implementation Notes The first element in the dbuffer array (index = 0) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index - 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4-22 shows the array in memory with an entry index of 2. The newest entry in the dbuffer is denoted by x(j-0), which in this case would occupy index = 3 in the array. The next newest entry is x(j-1), and so on. It is assumed that all x() entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

The dbuffer array actually contains one more history value than is needed to implement this filter. The value x(j-2*nh2) does not enter into the calculations for for the output r(j). However, this value is required in other DSPLIB filter functions that utilize the dual-MAC units on the C55x, such as FIR2. Including this extra location ensures compatibility across all filter functions in the C55x DSPLIB.

Figure 4-22, Figure 4-23, and Figure 4-24 show the dbuffer, x, and r arrays as they appear in memory.

Figure 4-22. dbuffer Array in Memory at Time j

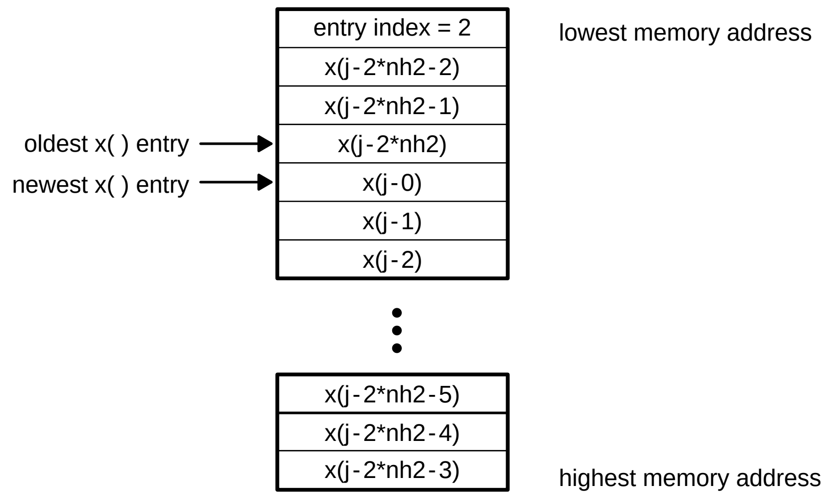


Figure 4-23. x Array in Memory

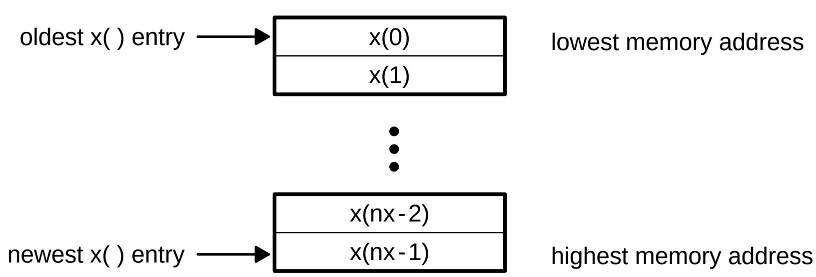
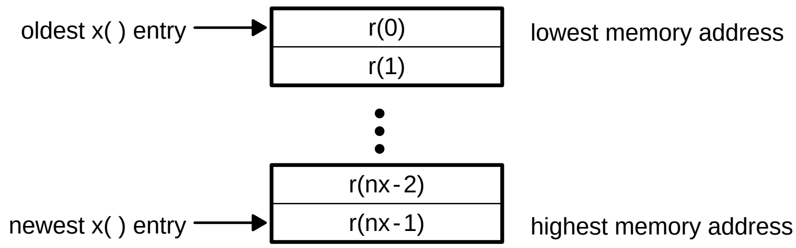


Figure 4-24. r Array in Memory



Example See examples/firs subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx[5 + (nh-2)]
	Overhead:	72
Code size (in bytes)	133	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

fltoq15 *Floating-point to Q15 Conversion*

Function ushort errorcode = fltoq15 (float *x, DATA *r, ushort nx)
(defined in fltoq15.asm)

Arguments

x[nx]	Pointer to floating-point input vector of size nx. x should contain the numbers normalized between (-1,1). The errorcode returned value will reflect if that condition is not met.
r[nx]	Pointer to output data vector of size nx containing the q15 equivalent of vector x.
nx	Length of input and output data vectors
errorcode	The function returns the following error codes: 1 - if any element is too large to represent in Q15 format 2 - if any element is too small to represent in Q15 format 3 - both conditions 1 and 2 were encountered

Description Convert the IEEE floating-point numbers stored in vector x into Q15 numbers stored in vector r. The function returns the error codes if any element x[i] is not representable in Q15 format.

All values that exceed the size limit will be saturated to a Q15 1 or -1 depending on sign (0x7fff if value is positive, 0x8000 if value is negative). All values too small to be correctly represented will be truncated to 0.

Algorithm Not applicable

Overflow Handling Methodology Saturation implemented for overflow handling

Special Requirements none

Implementation Notes none

Example See examples/expn subdirectory

Benchmarks (preliminary)

Cycles†	Core:	17 * nx (if x[n] ==0) 23 * nx (if x[n] is too small for Q15 representation) 32 * nx (if x[n] is too large for Q15 representation) 38 * nx (otherwise)
	Overhead:	23
Code size (in bytes)	157	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

hilb16 *FIR Hilbert Transformer*

Function ushort oflag = hilb16 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)

Arguments

x[nx]	Pointer to input vector of nx real elements.
h[nh]	Pointer to coefficient vector of size nh in normal order. H= {h0, h1, h2, h3, h4, …} Every odd valued filter coefficient has to 0, i.e. h1 = h3 = … = 0. And H = {h0, 0, h2, 0, h4, 0, …} where h0 resides at the lowest memory address in the array. This array must be located in internal memory since it is accessed by the C55x coefficient bus.

r[nx]	Pointer to output vector of nx real elements. In-place computation (r = x) is allowed.
dbuffer[nh + 2]	Pointer to delay buffer of length nh = nh + 2 In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed. The first element in this array is special in that it contains the array index-1 of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to zero (like all the other array entries) for the first block only.
nx	Number of real elements in vector x (input samples)
nh	The number of coefficients of the filter. For example if the filter coefficients are {h0, h1, h2, h3, h4, h5}, then nh = 6. Must be a minimum value of 6. For smaller filters, zero pad the coefficients to meet the minimum value.
oflag	Overflow error flag (returned value) If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes a real FIR filter (direct-form) using the coefficients stored in vector h. The real input data is stored in vector x. The filter output result is stored in vector r. This function maintains the array dbuffer containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for both block-by-block (nx >= 2) and sample-by-sample filtering (nx = 1). In place computation (r = x) is allowed.

Algorithm

$$r[j] = \sum_{k=0}^{nh-1} h[k]x[j-k] \qquad 0 \le j \le nx$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements

Every odd valued filter coefficient has to be 0. This is a requirement for the Hilbert transformer. For example, a 6 tap filter may look like this: H = [0.867 0 –0.324 0 –0.002 0]

Always pad 0 to make nh as a even number. For example, a 5 tap filter with a zero pad may look like this: H = [0.867 0 –0.324 0 –0.002 0]

nh must be a minimum value of 6. For smaller filters, zero pad the H[] array.

Implementation Notes The first element in the dbuffer array (index = 0) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index - 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4-25 shows the array in memory with an entry index of 2. The newest entry in the dbuffer is denoted by x(j-0), which in this case would occupy index = 3 in the array. The next newest entry is x(j-1), and so on. It is assumed that all x() entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

The dbuffer array actually contains one more history value than is needed to implement this filter. The value x(j-nh) does not enter into the calculations for for the output r(j). However, this value is required in other DSPLIB filter functions that utilize the dual-MAC units on the C55x, such as FIR2. Including this extra location ensures compatibility across all filter functions in the C55x DSPLIB.

Figure 4-25, Figure 4-26, and Figure 4-27 show the dbuffer, x, and r arrays as they appear in memory.

Figure 4-25. dbuffer Array in Memory at Time j

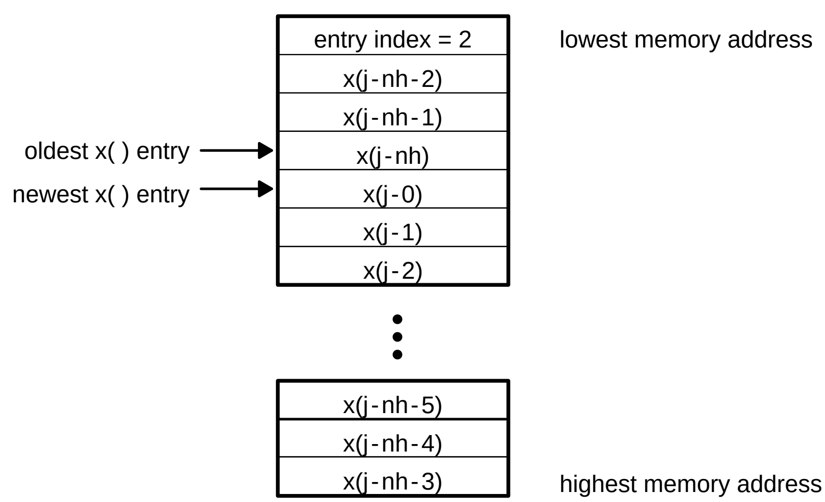


Figure 4-26. x Array in Memory

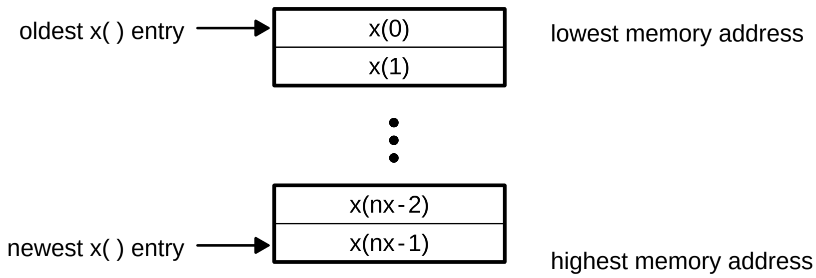
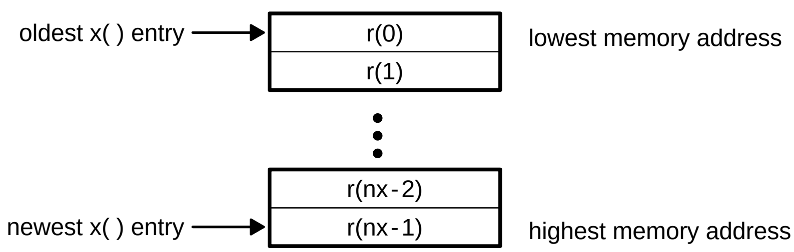


Figure 4-27. r Array in Memory



Example See examples/hilb16 subdirectory

Benchmarks (preliminary)

Cycles	Core:	nx*(2+nh/2)
	Overhead:	28
Code size (in bytes)	108	

iir32 *Double-precision IIR Filter*

Function

ushort oflag = iir32 (DATA *x, LDATA *h, DATA *r, LDATA *dbuffer, ushort nbiq, ushort nr)
(defined in iir32.asm)

Arguments

x [nr] Pointer to input data vector of size nr

h[5*nbiq] Pointer to the 32-bit filter coefficient vector with the following format. For example for nbiq= 2, h is equal to:

b21 – high	**beginning of biquad 1**
b21 – low	
b11 – high	
b11 – low	
b01 – high	
b01 – low	
a21 – high	
a21 – low	
a11 – high	
a11 – low	
b22 – high	beginning of biquad 2 coefs
b22 – low	
b12 – high	
b12 – low	
b02 – high	
b02 – low	
a22 – high	
a22 – low	
a12 – high	
a12 – low	

r[nr] Pointer to output data vector of size nr. r can be equal or less than x.

dbuffer[2*nbiq+2]	Pointer to address of 32-bit delay line dbuffer. Each biquad has 3 consecutive delay line elements. For example for nbiq = 2:

d1(n-2) - low **beginning of biquad 1**
d1(n-2) - high
d1(n-1) – low
d1(n-1) – high

d2(n-2) - low beginning of biquad 2
d2(n-2) - high
d2(n-1) – low
d2(n-1) – high

In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.

Memory alignment: none required for C5510. This is a group of circular buffers. Each biquad's delay buffer is treated separately. The Buffer Start Address (BSAxx) updated to a new location for each biquad.

nbiq	Number of biquads
nr	Number of elements of input and output vectors
oflag	Overflow flag.

If oflag = 1, a 32-bit overflow has occurred
If oflag = 0, a 32-bit overflow has not occurred

Description

Computes a cascaded IIR filter of nbiquad biquad sections using 32-bit coefficients and 32-bit delay buffers. The input data is assumed to be single-precision (16 bits).

Each biquad section is implemented using Direct-form II. All biquad coefficients (5 per biquad) are stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r .

This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function is more efficient for block-by-block filter implementation due to the C-calling overhead. However, it can be used for sample-by-sample filtering (nx = 1).

Algorithm (for biquad)

$$d(n) = x(n) - a1 * d(n-1) - a2 * d(n-2)$$
$$y(n) = b0 * d(n) + b1 * d(n-1) + b2 * d(n-2)$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements none

Implementation Notes See program iircas32.asm

Example See examples/iir32 subdirectory

Benchmarks (preliminary)

Cycles	Core:	nx*(7+ 31*nbiq)
	Overhead:	77
Code size (in bytes)	203	

iircas4 *Cascaded IIR Direct Form II Using 4 Coefficients per Biquad*

Function ushort oflag = iircas4 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)
(defined in iir4cas4.asm)

Arguments

x [nx]	Pointer to input data vector of size nx
h[4*nbiq]	Pointer to filter coefficient vector with the following format: h = a11 a21 b21 b11 ....a1i a2i b2i b1i where i is the biquad index (a21 is the a2 coefficient of biquad 1). Pole (recursive) coefficients = a. Zero (non-recursive) coefficients = b
r[nx]	Pointer to output data vector of size nx. r can be equal than x.

dbuffer[2*nbiq]	Pointer to address of delay line d. Each biquad has 2 delay line elements separated by nbiq locations in the following format: d1(n-1), d2(n-1),..di(n-1) d1(n-2), d2(n-2)...di(n-2) where i is the biquad index (d2(n-1) is the (n-1)th delay element for biquad 2). In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed. Memory alignment: this is a circular buffer and must start in a k-bit boundary(that is, the k LSBs of the starting address must be zeros) where k = log2 (2*nbiq).
nbiq	Number of biquads
nx	Number of elements of input and output vectors
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred If oflag = 0, a 32-bit overflow has not occurred

Description Computes a cascade IIR filter of nbiq biquad sections. Each biquad section is implemented using Direct-form II. All biquad coefficients (4 per biquad) are stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r.

This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function is more efficient for block-by-block filter implementation due to the C-calling overhead. However, it can be used for sample-by-sample filtering (nx = 1).

Algorithm (for biquad)

$$d(n) = x(n) - a1 * d(n-1) - a2 * d(n-2)$$
$$y(n) = d(n) + b1 * d(n-1) + b2 * d(n-2)$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements none

Implementation Notes none

Example See examples/iircas4 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx * (2 + 3 * nbiq)
	Overhead:	44
Code size (in bytes)	122	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

iircas5 — *Cascaded IIR Direct Form II (5 Coefficients per Biquad)*

Function

ushort oflag = iircas5 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)
(defined in iircas5.asm)

Arguments

x [nx]	Pointer to input data vector of size nx
h[5*nbiq]	Pointer to filter coefficient vector with the following format: h = a11 a21 b21 b01 b11 ... a1i a2i b2i b0i b1i where i is the biquad index a21 is the a2 coefficient of biquad 1). Pole (recursive) coefficients = a. Zero (non-recursive) coefficients = b
r[nx]	Pointer to output data vector of size nx. r can be equal than x.
dbuffer[2*nbiq]	Pointer to address of delay line d. Each biquad has 2 delay line elements separated by nbiq locations in the following format: d1(n-1), d2(n-1),..di(n-1) d1(n-2), d2(n-2)...di(n-2) where i is the biquad index(d2(n-1) is the (n-1)th delay element for biquad 2). In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed. Memory alignment: this is a circular buffer and must start in a k-bit boundary(that is, the k LSBs of the starting address must be zeros) where k = log2 (2*nbiq).
nbiq	Number of biquads

nx	Number of elements of input and output vectors
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred If oflag = 0, a 32-bit overflow has not occurred

Description

Computes a cascade IIR filter of nbiq biquad sections. Each biquad section is implemented using Direct-form II. All biquad coefficients (5 per biquad) are stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r.

This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function is more efficient for block-by-block filter implementation due to the C-calling overhead. However, it can be used for sample-by-sample filtering (nx = 1).

The usage of 5 coefficients instead of 4 facilitates the design of filters with a unit gain of less than 1 (for overflow avoidance), typically achieved by filter coefficient scaling.

Algorithm

(for biquad)

$$d(n) = x(n) - a1 * d(n-1) - a2 * d(n-2)$$
$$y(n) = b0 * d(n) + b1 * d(n-1) + b2 * d(n-2)$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements none

Implementation Notes none

Example See examples/iircas5 subdirectory

Benchmarks

(preliminary)

Cycles†	Core:	nx * (5 + 5 * nbiq)
	Overhead:	60
Code size (in bytes)	126	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

iircas51 — *Cascaded IIR Direct Form I (5 Coefficients per Biquad)*

Function

ushort oflag = iircas51 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)
(defined in iircas51.asm)

Arguments

x [nx]	Pointer to input data vector of size nx
h[5*nbiq]	Pointer to filter coefficient vector with the following format: h = b01 b11 b21 a11 a21 ....b0i b1i b2i a1i a2I where i is the biquad index (a21 is the a2 coefficient of biquad 1). Pole (recursive) coefficients = a. Zero (non-recursive) coefficients = b
r[nx]	Pointer to output data vector of size nx. r can be equal than x.
dbuffer[4*nbiq]	Pointer to address of delay line dbuffer. Each biquad has 4 delay line elements stored consecutively in memory in the following format: x1(n-1), x1(n-2), y1(n-1), y1(n-2) ... xi(n-2), xi(n-2), yi(n-1),yi(n-2) where i is the biquad index(x1(n-1) is the (n-1)th delay element for biquad 1). In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed. Memory alignment: No need for memory alignment.
nbiq	Number of biquads
nx	Number of elements of input and output vectors
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes a cascade IIR filter of nbiq biquad sections. Each biquad section is implemented using Direct-form I. All biquad coefficients (5 per biquad) are stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r.

This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function is more efficient for block-by-block filter implementation due to the C-calling overhead. However, it can be used for sample-by-sample filtering (nx = 1).

The usage of 5 coefficients instead of 4 facilitates the design of filters with a unit gain of less than 1 (for overflow avoidance), typically achieved by filter coefficient scaling.

Algorithm (for biquad)

$$y(n) = b0 * x(n) + b1 * x(n-1) + b2 * x(n-2) - a1 * y(n-1) - a2 * y(n-2)$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements none

Implementation Notes none

Example See examples/iircas51 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx * (5 + 8 * nbiq)
	Overhead:	68
Code size (in bytes)	154	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

iirlat — *Lattice Inverse (IIR) Filter*

Function ushort oflag = iirlat (DATA *x, DATA *h, DATA *r, DATA *pbuffer, int nx, int nh)

Arguments

x [nx]	Pointer to real input vector of nx real elements in normal order: x[0] x[1] . . x[nx-2] x[nx-1]
h[nh]	Pointer to lattice coefficient vector of size nh in normal order with the first element zero-padded: 0 h[0] h[1] . . h[nh-2] h[nh-1]
r[nx]	Pointer to output vector of nx real elements. In-place computation (r = x) is allowed. r[0] r[1] . . r[nx-2] r[nx-1]
pbuffer[nh]	Delay buffer In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.
nx	Number of real elements in vector x (input samples)

nh	Number of coefficients
oflag	Overflow error flag

If oflag = 1, a 32-bit data overflow has occurred in an intermediate or final result.

If oflag = 0, a 32-bit overflow has not occurred.

Description Computes a real lattice IIR filter implementation using coefficient stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r . This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function can be used for both block-by-block and sample-by-sample filtering (nx = 1)

Algorithm

$$e_N[n] = x[n],$$
$$e_{i-1}[n] = e_i[n] - hie'_{i-1}[n-1], \qquad i = N, (N-1), \ldots, 1$$
$$e'_i[n] = -k_i e_{i-1} + e'_{i-1}[n-1], \qquad i = N, (N-1), \ldots, 1$$
$$y[n] = e_0[n] = e'_0[n]$$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements none

Implementation Notes none

Example See examples/iirlat subdirectory

Benchmarks (preliminary)

Cycles†	Core:	4 * (nh - 1) * nx
	Overhead:	24
Code size (in bytes)	54	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

ldiv16 — *32-bit by 16-bit Long Division Function*

Function void ldiv16 (LDATA *x, DATA *y, DATA *r, DATA *rexp, ushort nx)

Arguments

x [nx]	Pointer to input data vector 1 of size nx x[0] x[1] . . x[nx-2] x[nx-1]
r[nx]	Pointer to output data buffer r[0] r[1] . . r[nx-2] r[nx-1]
rexp[nx]	Pointer to exponent buffer for output values. These exponent values are in integer format. rexp[0] rexp[1] . . rexp[nx-2] rexp[nx-1]
nx	Number of elements of input and output vectors

Description This routine implements a long division function of a Q31 value divided by a Q15 value. The reciprocal of the Q15 value, y, is calculated then multiplied by the Q31 value, x. The result is returned as an exponent such that:

r[i] * rexp[i] = true reciprocal in floating-point

Algorithm The reciprocal of the Q15 number is calculated using the following equation:

$$Ym = 2 * Ym - Ym^2 * Xnorm$$

If we start with an initial estimate of Ym, the equation converges to a solution very rapidly (typically 3 iterations for 16-bit resolution).

The initial estimate can be obtained from a look-up table, from choosing a midpoint, or simply from linear interpolation. The method chosen for this problem is linear interpolation and is accomplished by taking the complement of the least significant bits of the Xnorm value.

The reciprocal is multiplied by the Q31 number to generate the output.

Overflow Handling Methodology none

Special Requirements none

Implementation Notes none

Example See examples/ldiv16 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	4 * nx
	Overhead:	14
Code size (in bytes)	91	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

log_10 *Base 10 Logarithm*

Function ushort oflag = log_10 (DATA *x, LDATA *r, ushort nx)
(defined in log_10.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r[nx]	Pointer to output data vector (Q31 format) of size nx.
nx	Length of input and output data vectors
oflag	Overflow flag.

If oflag = 1, a 32-bit overflow has occurred.

If oflag = 0, a 32-bit overflow has not occurred.

Description Computes the log base 10 of elements of vector x using Taylor series.

Algorithm for $(i = 0;\ i < nx;\ i++)\quad y(i) = \log 10x(i)$ where $-1 < x(i) < 1$

Overflow Handling Methodology No scaling implemented for overflow prevention

Special Requirements none

Implementation Notes y = 0.4343 * ln(x) with x = M(x)*2^P(x) = M*2^P
y = 0.4343 * (ln(M) + ln(2)*P)
y = 0.4343 * (ln(2*M) + (P-1)*ln(2))
y = 0.4343 * (ln((2*M-1)+1) + (P-1)*ln(2))
y = 0.4343 * (f(2*M-1) + (P-1)*ln(2))
with f(u) = ln(1+u).

We use a polynomial approximation for f(u) :
f(u) = (((((C6*u+C5)*u+C4)*u+C3)*u+C2)*u+C1)*u+C0
for 0<= u <= 1.

The polynomial coefficients Ci are as follows :
C0 = 0.000 001 472
C1 = 0.999 847 766
C2 = -0.497 373 368
C3 = 0.315 747 760
C4 = -0.190 354 944
C5 = 0.082 691 584
C6 = -0.017 414 144

The coefficients Bi used in the calculation are derived from the Ci as follows:

B0	Q30	1581d	0062Dh
B1	Q14	16381d	03FFDh
B2	Q15	-16298d	0C056h
B3	Q16	20693d	050D5h
B4	Q17	-24950d	09E8Ah
B5	Q18	21677d	054ADh
B6	Q19	-9130d	0DC56h

Example See examples/log_10 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	35 * nx
	Overhead:	36
Code size (in bytes)	162	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

log_2 — *Base 2 Logarithm*

Function ushort oflag = log_2 (DATA *x, LDATA *r, ushort nx)
(defined in log_2.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r[nx]	Pointer to output data vector (Q31 format) of size nx.
nx	Length of input and output data vectors
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes the log base 2 of elements of vector x using Taylor series.

Algorithm for $(i = 0;\ i < nx;\ i++)\quad y(i) = \log 12x(i)$ where $0 < x(i) < 1$

Overflow Handling Methodology No scaling implemented for overflow prevention

Special Requirements none

Implementation Notes y = 1.4427 * ln(x) with x = M(x)*2^P(x) = M*2^P
y = 1.4427 * (ln(M) + ln(2)*P)
y = 1.4427 * (ln(2*M) + (P-1)*ln(2))
y = 1.4427 * (ln((2*M-1)+1) + (P-1)*ln(2))
y = 1.4427 * (f(2*M-1) + (P-1)*ln(2))
with f(u) = ln(1+u).

We use a polynomial approximation for f(u) :
f(u) = (((((C6*u+C5)*u+C4)*u+C3)*u+C2)*u+C1)*u+C0
for 0<= u <= 1.

The polynomial coefficients Ci are as follows:
C0 = 0.000 001 472
C1 = 0.999 847 766
C2 = -0.497 373 368
C3 = 0.315 747 760
C4 = -0.190 354 944
C5 = 0.082 691 584
C6 = -0.017 414 144

The coefficients Bi used in the calculation are derived from the Ci as follows:

B0	Q30	1581d	0062Dh
B1	Q14	16381d	03FFDh
B2	Q15	-16298d	0C056h
B3	Q16	20693d	050D5h
B4	Q17	-24950d	09E8Ah
B5	Q18	21677d	054ADh
B6	Q19	-9130d	0DC56h

Example See examples/log_2 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	36 * nx
	Overhead:	37
Code size (in bytes)	166	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

logn — *Base e Logarithm (natural logarithm)*

Function ushort oflag = logn (DATA *x, LDATA *r, ushort nx)
(defined in logn.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r[nx]	Pointer to output data vector (Q31 format) of size nx.
nx	Length of input and output data vectors
oflag	Overflow flag.

If oflag = 1, a 32-bit overflow has occurred.

If oflag = 0, a 32-bit overflow has not occurred.

Description Computes the log base e of elements of vector x using Taylor series.

Algorithm for $(i = 0;\ i < nx;\ i++)\quad y(i) = \log nx(i)$ where $-1 < x(i) < 1$

Overflow Handling Methodology No scaling implemented for overflow prevention

Special Requirements none

Implementation Notes y = 0.4343 * ln(x) with x = M(x)*2^P(x) = M*2^P
y = 0.4343 * (ln(M) + ln(2)*P)
y = 0.4343 * (ln(2*M) + (P-1)*ln(2))
y = 0.4343 * (ln((2*M-1)+1) + (P-1)*ln(2))
y = 0.4343 * (f(2*M-1) + (P-1)*ln(2))
with f(u) = ln(1+u).

We use a polynomial approximation for f(u):
f(u) = (((((C6*u+C5)*u+C4)*u+C3)*u+C2)*u+C1)*u+C0
for 0<= u <= 1.

The polynomial coefficients Ci are as follows:
C0 = 0.000 001 472
C1 = 0.999 847 766
C2 = -0.497 373 368
C3 = 0.315 747 760
C4 = -0.190 354 944
C5 = 0.082 691 584
C6 = -0.017 414 144

The coefficients Bi used in the calculation are derived from the Ci as follows:

B0	Q30	1581d	0062Dh
B1	Q14	16381d	03FFDh
B2	Q15	-16298d	0C056h
B3	Q16	20693d	050D5h
B4	Q17	-24950d	09E8Ah
B5	Q18	21677d	054ADh
B6	Q19	-9130d	0DC56h

Example See examples/logn subdirectory

Benchmarks (preliminary)

Cycles†	Core:	26 * nx
	Overhead:	36
Code size (in bytes)	132	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

maxidx — *Index of the Maximum Element of a Vector*

Function short r = maxidx (DATA *x, ushort ng, ushort ng_size);
(defined in maxidx.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r	Index for vector element with maximum value.
ng	Number of groups.
ng_size	Size of group.

Description The vector x is divided in ng groups of size ng_size.

Size of x = ng x ng_size. ng_size must be an even number between 2 and 34. The larger ng_size, the better the performance.Returns the index of the maximum element of a vector x. The index is a number between 0 and nx - 1. In case of multiple maximum elements, r contains the index of the first maximum element found.

Example 1: size of x is 64.

Choose ng_size = 32, ng = 2

Example 2: size of x is 100.

Choose ng_size = 20, ng = 5

Example 3: size of x is 90.

Choose ng_size = 30, ng = 3

Algorithm Not applicable

Overflow Handling Methodology Not applicable

Special Requirements

ng_size is an even number between 2 and 34.

nx is an even number.

Input vector has to be 32-bit aligned.

Implementation Notes none

Example See examples/maxidx subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx/2 + ng16
	Overhead:	40
Code size (in bytes)	143	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

maxidx34 — *Index of the Maximum Element of a Vector ≤ 34*

Function short r = maxidx34 (DATA *x, ushort nx)
(defined in maxidx34.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r	Index for vector element with maximum value.
nx	Lenght of input data vector (nx ≤ 34).

Description Returns the index of the maximum element of a vector x. The index is a number between 0 and nx - 1. In case of multiple maximum elements, r contains the index of the first maximum element found.

Algorithm Not applicable

Overflow Handling Methodology Not applicable

Special Requirements Size of the vector, nx ≤ 34

nx is an even number.

Input vector has to be 32-bit aligned.

Implementation Notes none

Example See examples/maxidx34 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx/2
	Overhead:	42
Code size (in bytes)	26	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

maxval — *Maximum Value of a Vector*

Function short r = maxval (DATA *x, ushort nx)
(defined in maxval.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r	Maximum value of a vector
nx	Length of input data vector

Description Returns the maximum element of a vector x.

Algorithm Not applicable

Overflow Handling Methodology Not applicable

Special Requirements nx is an even number.

Input vector has to be 32-bit aligned.

Implementation Notes none

Example See examples/maxval subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx
	Overhead:	3
Code size (in bytes)	20	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

maxvec — *Index and Value of the Maximum Element of a Vector*

Function void maxvec (DATA *x, ushort nx, DATA *r_val, DATA *r_idx)
(defined in maxvec.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r_val	maximum value
r_idx	Index for vector element with maximum value
nx	Lenght of input data vector ($nx \geq 6$)

Description This function finds the index for vector element with maximum value. In case of multiple maximum elements, r_idx contains the index of the first maximum element found. r_val contains the maximum value.

Algorithm Not applicable

Overflow Handling Methodology Not applicable

Special Requirements none

Implementation Notes none

Example See examples/maxvec subdirectory

Benchmarks (preliminary)

Cycles	Core:	nx*3
	Overhead:	8
Code size (in bytes)	26	

minidx Index of the Minimum Element of a Vector

Function short r = minidx (DATA *x, ushort nx)
(defined in minidx.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r	Index for vector element with minimum value
nx	Length of input data vector

Description Returns the index of the minimum element of a vector x. In case of multiple minimum elements, r contains the index of the first minimum element found.

Algorithm Not applicable

Overflow Handling Methodology Not applicable

Special Requirements none

Implementation Notes none

Example See examples/minidx subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx * 3
	Overhead:	7
Code size (in bytes)	26	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

minval — *Minimum Value of a Vector*

Function short r = minval (DATA *x, ushort nx)
(defined in minval.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r	Minimum value of a vector
nx	Length of input data vector

Description Returns the minimum element of a vector x.

Algorithm Not applicable

Overflow Handling Methodology Not applicable

Special Requirements none

Implementation Notes none

Example See examples/minval subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx/2
	Overhead:	7
Code size (in bytes)	20	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

minvec — *Index and Value of the Minimum Element of a Vector*

Function void minvec (DATA *x, ushort nx, DATA *r_val, DATA *r_idx)
(defined in minvec.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r_val	Minimum value
r_idx	Index for vector element with minimum value
nx	Length of input data vector (nx $\geq$ 6)

Description This function finds the index for vector element with minimum value. In case of multiple minimum elements, r_idx contains the index of the first minimum element found. r_val contains the minimum value.

Algorithm Not applicable

Overflow Handling Methodology Not applicable

Special Requirements none

Implementation Notes none

Example See examples/minvec subdirectory

Benchmarks (preliminary)

Cycles	Core: nx*3 Overhead: 8
Code size (bytes)	26

mmul — *Matrix Multiplication*

Function

ushort oflag = mmul (DATA *x1,short row1,short col1,DATA *x2,short row2,short col2,DATA *r)
(defined in mmul.asm)

Arguments

x1[row1*col1]:	Pointer to input vector of size nx Pointer to input matrix of size row1*col1 ; row1 : ; : ; : ; r[row1*col2] : Pointer to output data vector of size row1*col2
row1	number of rows in matrix 1
col1	number of columns in matrix 1
x2[row2*col2]:	Pointer to input matrix of size row2*col2
row2	number of rows in matrix 2
col2	number of columns in matrix 2
r[row1*col2]	Pointer to output matrix of size row1*col2

Description

This function multiplies two matrices

Algorithm

Multiply input matrix A (M by N) by input matrix B (N by P) using 2 nested loops:

```
for i = 1 to M
   for k = 1 to P
   {
      temp = 0
      for j = 1 to N
         temp = temp + A(i,j) * B(j,k)
      C(i,k) = temp
      }
```

Overflow Handling Methodology Not applicable

Special Requirements Verify that the dimensions of input matrices are legal, i.e. col1 == row2

Implementation Notes In order to take advantage of the dual MAC architecture of the C55x, this implementation checks the size of the matrix x1. For small matrices x1 (row1 < 4 or col1 < 2), single MAC loops are used. For larger matrices x1 (row1 ≥ 4 and col1 ≥ 2), Dual MAC loops are more efficient and quickly make up for the additional initialization overhead.

Example See examples/mmul subdirectory

Benchmarks (preliminary)

Cycles† Core:

if(row1 < 4 || col1 < 2), use single MAC
((col1 + 2)*row1 + 4)*col2

if((row1==even)&&(row1 ≥ 4)&&(col1 ≥ 2)), use dual MAC
((col1 + 4)*0.5*row1 + 10)col2

if((row1==odd)&&(row1 ≥ 4)&&(col1 ≥ 2), use dual MAC
((col1 + 4)*0.5*(row1 - 1) + col1 + 12)col2

Overhead: 30

Code size (in bytes) 215

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

mtrans *Matrix Transpose*

Function ushort oflag = mtrans (DATA *x, short row, short col, DATA *r)
(defined in mtrans.asm)

Arguments

x[row*col]	Pointer to input matrix. In-place processing is not allowed.
row	number of rows in matrix
col	number of columns in matrix
r[row*col]	Pointer to output data vector

Description This function transposes matrix x

Algorithm

for i = 1 to M
  for j = 1 to N
    C(j,i) = A(i,j)

Overflow Handling Methodology Not applicable

Special Requirements none

Implementation Notes none

Example See examples/mtrans subdirectory

Benchmarks (preliminary)

Cycles†	Core:	(1 + col) * row
	Overhead:	23
Code size (in bytes)	65	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

mul32 — *32-bit Vector Multiplication*

Function ushort oflag = mul32 (LDATA *x, LDATA *y, LDATA *r, ushort nx)
(defined in mul32.asm)

Arguments

x[nx]	Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)
y[nx]	Pointer to input data vector 2 of size nx
r[nx]	Pointer to output data vector of size nx containing
nx	Number of elements of input and output vectors. Nx $\geq$ 4
oflag	Overflow flag

If oflag = 1, a 32-bit overflow has occurred
If oflag = 0, a 32-bit overflow has not occurred

Description This function multiplies two 32-bit Q31 vectors, element by element, and produces a 32-bit Q31 vector.

Algorithm

$$\text{for}(i = 0; i < nx; i++)$$
$$z(i) = x(i) * y(i)$$

Overflow Handling Methodology Scaling implemented for overflow prevention (user selectable)

Special Requirements none

Implementation Notes none

Example See examples/add subdirectory

Benchmarks

Cycles	Core:	4*nx + 4
	Overhead	21
Code size (in bytes)	73	

neg — *Vector Negate*

Function ushort oflag = neg (DATA *x, DATA *r, ushort nx)
(defined in neg.asm)

Arguments

x[nx] — Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)

r[nx] — Pointer to output data vector of size nx. In-place processing allowed

Special cases:

if x[I] = -1 = 32768 , then r = 1 = 321767 with oflag = 1

if x= 1 = 32767 , then r = -1 = 321768 with oflag = 1

nx — Number of elements of input and output vectors.
nx ≥ 4

oflag — Overflow flag.

If oflag = 1, a 32-bit overflow has occurred.

If oflag = 0, a 32-bit overflow has not occurred.

Caution: overflow in negation of a Q15 number can happen naturally when negating (-1).

Description This function negates each of the elements of a vector (fractional values).

Algorithm for $(i = 0;\ i < nx;\ i++)$ $\quad x(i) = -x(i)$

Overflow Handling Methodology Saturation implemented for overflow handling

Special Requirements none

Implementation Notes none

Example See examples/neg subdirectory

Benchmarks (preliminary)

Cycles†	Core:	4 * nx
	Overhead:	13
Code size (in bytes)	61	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

neg32 — *Vector Negate (double-precision)*

Function ushort oflag = neg32 (LDATA *x, LDATA *r, ushort nx)
(defined in neg.asm)

Arguments

x[nx] — Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)

r[nx] — Pointer to output data vector of size nx. In-place processing allowed

Special cases:

- if x = -1 = 32768 * 2^{16}, then r = 1 = 321767 * 2^{16} with oflag = 1
- if x= 1 = 32767 * 2^{16}, then r = -1 = 321768 * 2^{16} with oflag = 1

nx — Number of elements of input and output vectors.
nx ≥ 4

oflag — Overflow flag.

- If oflag = 1, a 32-bit overflow has occurred.
- If oflag = 0, a 32-bit overflow has not occurred.

Caution: overflow in negation of a Q31 number can happen naturally when negating (-1).

Description This function negates each of the elements of a vector (fractional values).

Algorithm for $(i = 0;\ i < nx;\ i++) \qquad x(i) = -x(i)$

Overflow Handling Methodology Saturation implemented for overflow handling

Special Requirements none

Implementation Notes none

Example See examples/neg32 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	4 * nx
	Overhead:	13
Code size (in bytes)	61	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

power — *Vector Power*

Function ushort oflag = power (DATA *x, LDATA *r, ushort nx)
(defined in power.asm)

Arguments

x[nx] Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)

r[1] Pointer to output data vector element in Q31 format
Special cases:

if x= -1 = 32768*2^{16} , then r = 1 = 321767*2^{16}
with oflag = 1

if x= 1 = 32767*2^{16} , then r = -1 = 321768*2^{16}
with oflag = 1

nx Number of elements of input vectors.
nx ≥ 4

oflag Overflow flag.

If oflag = 1, a 32-bit overflow has occurred.

If oflag = 0, a 32-bit overflow has not occurred.

Description This function calculates the power (sum of products) of a vector.

Algorithm Power = 0 for $(i = 0;\ i < nx;\ i++)$ power $+= x(i) * x(I)$

Overflow Handling Methodology No scaling implemented for overflow handling

Special Requirements none

Implementation Notes none

Example See examples/power subdirectory

Benchmarks (preliminary)

Cycles†	Core:	nx - 1
	Overhead:	12
Code size (in bytes)	54	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

q15tofl *Q15 to Floating-point Conversion*

Function ushort q15tofl (DATA *x, float *r, ushort nx)
(defined in q152fl.asm)

Arguments

x[nx]	Pointer to Q15 input vector of size nx.
r[nx]	Pointer to floating-point output data vector of size nx containing the floating-point equivalent of vector x.
nx	Length of input and output data vectors

Description Converts the Q15 stored in vector x to IEEE floating-point numbers stored in vector r.

Algorithm Not applicable

Overflow Handling Methodology Saturation implemented for overflow handling

Special Requirements none

Implementation Notes none

Example See examples/ug subdirectory

Benchmarks (preliminary)

Cycles†	Core:	7 * nx (if x[n] ==0) 32 * nx
	Overhead:	18
Code size (in bytes)	124	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

rand16 *Random Number Generation Algorithm*

Function ushort oflag= rand16 (DATA *r, ushort nr)

Arguments

*r	Pointer to the array where the 16-bit random numbers are stored
nr	Number of random numbers that are generated
oflag	Overflow error flag (returned value) If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result. If oflag = 0, a 32-bit overflow has not occurred.

Description This algorithm computes an array of random numbers based on the linear congruential method introduced by D. Lehmer in 1951. This is one of the fastest and simplest techniques of generating random numbers. The code shown here generates 16-bit integers, however, if a 32-bit value is desired the code can be modified to perform 32-bit multiplies using the defined constants RNDMULT and RNDINC. The disadvantage of this technique is that it is very sensitive to the choice of RANDMULT and RNDINC.

Algorithm $r[n] = [(r[n-1] * RNDMULT) + RNDINC] \% M$
where $0 \le n \le nr$ and $0 \le M \le 65536$

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements No special requirements.

Implementation Notes Rand16() is written so that it can be called from a C program. Prior to calling rand16(), rand16i() can be called to initialize the random number generator seed value. The C routine passes two parameters to rand16(): A pointer to the random number array *r and the count of random numbers (nr) desired. The random numbers are declared as short or 16 bit values. Two constants RNDMULT and RNDINC are defined in the function. The algorithm is sensitive to the choice of RNDMULT and RNDINC so exercise caution when changing these.

M	This value is based on the system that the routine runs. This routine returns a random number from 0 to 65536 (64K) and is NOT internally bounded. If you need a min/max limit, this must be coded externally to this routine.
RNDSEED	An arbitrary constant that can be any value between 0 and 64K. If 0 (zero) is chosen, then RNDINC should be some value greater than 1. Otherwise, the first two values will be 0 and 1. To change the set of random numbers generated by this routine, change the RNDSEED value. In this routine, RNDSEED is initialized to 21845, which is 65536/3.

RNDMULT	Should be chosen such that the last three digits fall in the pattern even_digit-2-1 such as xx821, xx421 etc. RNDMULT = 31821 is used in this routine.
RNDINC	In general, this constant can be any prime number related to M. Research shows that RNDINC (the increment value) should be chosen by the following formula: RNDINC = ((1/2 - (1/6 * SQRT(3))) * M). Using M=65536, RNDINC was picked as 13849.

The random seed initialized in rand16i() is used to generate the first random number. Each random number generated is used to generate the next number in the series. The random number is generated in the accumulator (32 bits) by using the multiply-accumulate (MAC) unit to do the computation. In the course of the algorithm if there is intermediate overflow, the overflow flag bit in status register is set. At the end of the algorithm, the overflow flag is tested for any intermediate overflow conditions.

Example See examples/rand16 subdirectory

Benchmarks

Cycles	Core:	13 + nr*2
	Overhead:	10
Code size (in bytes)	49	

C54x Benchmark for Comparison

Cycles	Core:	10 + nr*4
	Overhead:	16
Code size (in bytes)	56	

rand16init — *Random Number Generation Initialization*

Function void rand16init(void)

Arguments none

Description Initializes seed for 16-bit random number generator.

Algorithm Not applicable

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements Allocation of .bss section is required in linker command file.

Implementation Notes This function initializes a global variable rndseed in global memory to be used for the 16 bit random number generation routine (rand16)

Example See examples/rand16i subdirectory

Benchmarks

Cycles	6
Code size (in bytes)	9

C54x Benchmark for Comparison

Cycles	7
Code size (in bytes)	10

recip16 — *16-bit Reciprocal Function*

Function void recip16 (DATA *x, DATA *r, DATA *rexp, ushort nx)

Arguments

x[nx]	Pointer to input data vector 1 of size nx. x[0] x[1] . . x[nx-2] x[nx-1]
r[nx]	Pointer to output data buffer r[0] r[1] . . r[nx-2] r[nx-1]

rexp[nx]	Pointer to exponent buffer for output values. These exponent values are in integer format. rexp[0] rexp[1] . . rexp[nx-2] rexp[nx-1]
nx	Number of elements of input and output vectors

Description This routine returns the fractional and exponential portion of the reciprocal of a Q15 number. Since the reciprocal is always greater than 1, it returns an exponent such that:

$r[i] * r\exp[i] =$ true reciprocal in floating-point

Algorithm $Ym = 2 * Ym - Ym^2 * Xnorm$

If we start with an initial estimate of Ym, the equation converges to a solution very rapidly (typically 3 iterations for 16-bit resolution).

The initial estimate can be obtained from a look-up table, from choosing a midpoint, or simply from linear interpolation. The method chosen for this problem is linear interpolation and is accomplished by taking the complement of the least significant bits of the Xnorm value.

Overflow Handling Methodology none

Special Requirements none

Implementation Notes none

Example See examples/recip16 subdirectory

Benchmarks (preliminary)

Cycles†	Core:	33 * nx
	Overhead:	12
Code size (in bytes)	69	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

rfft — *Forward Real FFT (in-place)*

Function

void rfft (DATA *x, ushort nx, type);
(reference cfft.asm and unpack.asm)

Arguments

x [nx] — Pointer to input vector containing nx real elements. On output, vector x contains the first half (nx/2 complex elements) of the FFT output in the following order. Real FFT is a symmetric function around the Nyquist point, and for this reason only half of the FFT(x) elements are required.

On output x will contain the FFT(x) = y in the following format:

y(0)Re y(nx/2)im → DC and Nyquist
y(1)Re y(1)Im
y(2)Re y(2)Im
....
y(nx/2)Re y(nx/2)Im

Complex numbers are stored in Re-Im format

nx — Number of real elements in vector x. can take the following values.

nx = 16, 32, 64, 128, 256, 512, 2048

type — RFFT type selector. Types supported:

If type = SCALE, scaled version selected

If type = NOSCALE, non-scaled version selected

Description

Computes a Radix-2 real DIT FFT of the nx real elements stored in vector x in bit-reversed order. The original content of vector x is destroyed in the process. The first nx/2 complex elements of the FFT(x) are stored in vector x in normal-order.

Algorithm

(DFT)
See CFFT

Special Requirements

This function should work with unpack.asm and/or rfft.asm for proper result. See example in examples/rifft directory.

Memory alignment: input data (x) must be aligned at even word boundary.

Data memory alignment (referenced rfft.cmd in examples/rfft directory):

- Alignment of input database address: (n+1) LSBs must be zeros, where n = $\log_2(nx)$.

- Ensure that the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).
- For best performance, the data buffer has to be in a DARAM block.
- If the twiddle table and the data buffer are in the same block, then the radix-2 kernel is 7 cycles and the radix-4 kernel is not affected.
- The twiddle table should be aligned to the nx word boundary; i.e., the lowest $\log_2(nx)$ least significant bits of the twiddle table starting address should be zero.

Implementation Notes Implemented as a complex FFT of size nx/2 followed by an unpack stage to unpack the real FFT results. Therefore, Implementation Notes for the cfft function apply to this case.

Notice that normally an FFT of a real sequence of size N, produces a complex sequence of size N (or 2*N real numbers) that will not fit in the input sequence. To accommodate all the results without requiring extra memory locations, the output reflects only half of the spectrum (complex output). This still provides the full information because an FFT of a real sequence has even symmetry around the center or nyquist point(N/2).

When scale = 1, this routine prevents overflow by scaling by 2 at each FFT intermediate stages and at the unpacking stage.

Example See examples/rfft Zip file

rifft — *Inverse Real FFT (in-place)*

Function

void rifft (DATA *x, ushort nx, type);
(reference cifft.asm)

Arguments

x [nx]	Pointer to input vector x containing nx real elements. The unpacki routine should be called to unpack the rfft sequence before calling the bit reversal routine. (See examples directory for calling sequence) On output, the vector x contains nx complex elements corresponding to RIFFT(x) or the signal itself.
nx	Number of real elements in vector x. nx can take the following values. nx =16, 32, 64, 128, 256, 512, 1024
type	RFFT type selector. Types supported: If type = SCALE, scaled version selected If type = NOSCALE, non-scaled version selected

Description

Computes a Radix-2 real DIT IFFT of the nx real elements stored in vector x in bit-reversed order. The original content of vector x is destroyed in the process. The first nx/2 complex elements of the IFFT(x) are stored in vector x in normal-order.

Algorithm

(IDFT)
See CIFFT

Special Requirements

This function should work with unpacki.asm and/or cbrev.asm for proper result. See example in examples/rifft directory.

Memory alignment: input data (x) must be aligned at even word boundary.

Data memory alignment (referenced rfft.cmd in examples/rfft directory):

- Alignment of input database address: (n+1) LSBs must be zeros, where n = $\log_2$(nx).
- Ensure that the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).
- For best performance, the data buffer has to be in a DARAM block.

- If the twiddle table and the data buffer are in the same block, then the radix-2 kernel is 7 cycles and the radix-4 kernel is not affected.
- The twiddle table should be aligned to the nx word boundary; i.e., the lowest $\log_2$(nx) least significant bits of the twiddle table starting address should be zero.

Implementation Notes Implemented as a complex IFFT of size nx/2 followed by an unpack stage to unpack the real IFFT results. Therefore, Implementation Notes for the cfft function apply to this case.

Notice that normally an IFFT of a real sequence of size N, produces a complex sequence of size N (or 2*N real numbers) that will not fit in the input sequence. To accommodate all the results without requiring extra memory locations, the output reflects only half of the spectrum (complex output). This still provides the full information because an IFFT of a real sequence has even symmetry around the center or nyquist point(N/2).

When scale = 1, this routine prevents overflow by scaling by 2 at each IFFT intermediate stages and at the unpacking stage.

Example See examples/rifft subdirectory

sine — *Sine*

Function ushort oflag = sine (DATA *x, DATA *r, ushort nx)
(defined in sine.asm)

Arguments

x[nx]	Pointer to input vector of size nx. x contains the angle in radians between [$-\pi$, π] normalized between (-1,1) in q15 format x = xrad /π For example: 45^o = π/4 is equivalent to x = 1/4 = 0.25 = 0x200 in q15 format.
r[nx]	Pointer to output vector containing the sine of vector x in q15 format
nx	Number of elements of input and output vectors. nx $\geq$ 4
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred. If oflag = 0, a 32-bit overflow has not occurred.

Description Computes the sine of elements of vector x. It uses Taylor series to compute the sine of angle x.

Algorithm for $(i = 0;\ i < nx;\ i++)$ $\quad y(i) = \sin(x(i))$ where $x(i) - \frac{xrad}{\pi}$

Overflow Handling Methodology Not applicable

Special Requirements none

Implementation Notes Computes the sine of elements of vector x. It uses the following Taylor series to compute the angle x in quadrant 1 (0-π/2).

sin(x) = c1*x + c2*x^2 + c3*x^3 + c4*x^4 + c5*x^5

c1 = 3.140625x
c2 = 0.02026367
c3 = - 5.3251
c4 = 0.5446778
c5 = 1.800293

The angle x in other quadrant is calculated by using symmetries that map the angle x into quadrant 1.

Example See examples/sine subdirectory

Benchmarks (preliminary)

Cycles†	Core:	19 * nx
	Overhead:	17
Code size (in bytes)	93 program; 3 data	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

sqrt_16 *Square Root of a 16-bit Number*

Function ushort oflag = sqrt_16 (DATA *x, DATA *r, short nx)
(defined in sqrtv.asm)

Arguments

x[nx]	Pointer to input vector of size nx.
r[nx]	Pointer to output vector of size nx containing the sqrt(x).
nx	Number of elements of input and output vectors.
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred. If oflag = 0, a 32-bit overflow has not occurred.

Description Calculates the square root for each element in input vector x, storing results in output vector r.

Algorithm for $(i = 0;\ i < nx;\ i++)$ $r[i] = \sqrt{(x(i)}$ $0 \le i \le nx$

Overflow Handling Methodology Not applicable

Special Requirements none

Implementation Notes The square root of a number(x) can be calculated using Newton's method. An initial approximation is guessed and then the approximation gets recomputed using the formula,

$$new\ approximation = old\ approximation - \frac{(old\ approximation^2 - x)}{2}.$$

The new approximation then becomes the old approximation and the process is repeated until the desired accuracy is reached.

Example See examples/sqrtv subdirectory

Benchmarks (preliminary)

Cycles†	Core:	35 * nx
	Overhead:	14
Code size (in bytes)	84 program; 5 data	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

sub *Vector Subtract*

Function short oflag = sub (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)
(defined in sub.asm)

Arguments

x[nx]	Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)
y[nx]	Pointer to input data vector 2 of size nx
r[nx]	Pointer to output data vector of size nx containing (x-y) if scale =0 (x-y)/2 if scale =1
nx	Number of elements of input and output vectors. nx ≥ 4

scale	Scale selection If scale = 1, divide the result by 2 to prevent overflow. If scale = 0, do not divide by 2.
oflag	Overflow flag. If oflag = 1, a 32-bit overflow has occurred. If oflag = 0, a 32-bit overflow has not occurred.

Description This function subtracts two vectors, element by element.

Algorithm for $(i = 0;\ i < nx;\ i++)\qquad z(i) = x(i) - y(i)$

Overflow Handling Methodology Scaling implemented for overflow prevention (user selectable)

Special Requirements none

Implementation Notes none

Example See examples/sub subdirectory

Benchmarks (preliminary)

Cycles†	Core:	3 * nx
	Overhead:	23
Code size (in bytes)	60	

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).

Chapter 5

DSPLIB Benchmarks and Performance Issues

All functions in the DSPLIB are provided with execution time and code size benchmarks. While developing the included functions, we tried to compromise between speed, code size, and ease of use. However, with few exceptions, the highest priority was given to optimize for speed and ease of use, and last for code size.

Even though DSPLIB can be used as a first estimation of processor performance for a specific function, you should know that the generic nature of DSPLIB may add extra cycles not required for customer specific usage.

Topic **Page**

5.1 What DSPLIB Benchmarks are Provided 5-2

5.2 Performance Considerations 5-2

5.1 What DSPLIB Benchmarks are Provided

DSPLIB documentation includes benchmarks for instruction cycles and memory consumption. The following benchmarks are typically included:

- Calling and register initialization overhead
- Number of cycles in the kernel code: Typically provided in the form of an equation that is a function of the data size parameters. We consider the kernel (or core) code, the instructions contained between the _start and _end labels that you can see in each of the functions.
- Memory consumption: Typically program size in bytes is reported. For functions requiring significant internal data allocation, data memory consumption is also provided. When stack usage for local variables is minimum, that data consumption is not reported.

For functions in which it is difficult to determine the number of cycles in the kernel code as a function of the data size parameters, we have included direct cycle count for specific data sizes.

5.2 Performance Considerations

Benchmark cycles presented assume best case conditions, typically assuming:

- 0 wait-state memory external memory for program and data
- data allocation to on-chip DARAM
- no pipeline hits

A linker command file showing the memory allocation used during testing and benchmarking in the Code Composer C55x Simulator is included under the example subdirectory.

Remember, execution speed in a system is dependent on where the different sections of program and data are located in memory. Be sure to account for such differences, when trying to explain why a routine is taking more time that the reported DSPLIB benchmarks.

Chapter 6

Software Updates and Customer Support

This chapter details the software updates and customer support issues for the TMS320C55x DSPLIB.

Topic	Page
6.1 DSPLIB Software Updates	**6-2**
6.2 DSPLIB Customer Support	**6-2**

6.1 DSPLIB Software Updates

C55x DSPLIB Software updates will be periodically released, incorporating product enhancement and fixes.

DSPLIB Software Updates will be posted as they become available in the same location you download this information. Source Code for previous releases will be kept public to prevent any customer problem in case we decide to discontinue or change the functionality of one of the DSPLIB functions. Make sure to read the readme.1st file available in the root directory of every release.

6.2 DSPLIB Customer Support

If you have any questions or want to report problems or suggestions regarding the C55x DSPLIB, contact Texas Instruments at dsph@ti.com.

We encourage the use of the software report form (report.txt) contained in the DSPLIB root directory to report any problem associated with the C55x DSPLIB.

Appendix A

Overview of Fractional Q Formats

Unless specifically noted, DSPLIB functions use Q15 format or to be more exact Q0.15. In a Q*m*.*n* format, there are *m* bits used to represent the two's complement integer portion of the number, and *n* bits used to represent the two's complement fractional portion. *m+n+1* bits are needed to store a general Q*m*.*n* number. The extra bit is needed to store the sign of the number in the most-significant bit position. The representable integer range is specified by $(-2^m, 2^m)$ and the finest fractional resolution is 2^{-n}.

For example, the most commonly used format is Q.15. Q.15 means that a 16-bit word is used to express a signed number between positive and negative 1. The most-significant binary digit is interpreted as the sign bit in any Q format number. Thus in Q.15 format, the decimal point is placed immediately to the right of the sign bit. The fractional portion to the right of the sign bit is stored in regular two's complement format.

Topic **Page**

A.1 Q3.12 Format A-2
A.2 Q.15 Format A-2
A.3 Q.31 Format A-2

A.1 Q3.12 Format

Q.3.12 format places the sign bit after the fourth binary digit from the right, and the next 12 bits contain the two's complement fractional component. The approximate allowable range of numbers in Q.3.12 representation is (-8,8) and the finest fractional resolution is $2^{-12} = 2.441 \times 10^{-4}$.

Table A-1. Q3.12 Bit Fields

Bit	15	14	13	12	11	10	9	…	0
Value	S	I3	I2	I1	Q11	Q10	Q9	…	Q0

A.2 Q.15 Format

Q.15 format places the sign bit at the leftmost binary digit, and the next 15 leftmost bits contain the two's complement fractional component. The approximate allowable range of numbers in Q.15 representation is (-1,1) and the finest fractional resolution is $2^{-15} = 3.05 \times 10^{-5}$.

Table A-2. Q.15 Bit Fields

Bit	15	14	13	12	11	10	9	…	0
Value	S	Q14	Q13	Q12	Q11	Q10	Q9	…	Q0

A.3 Q.31 Format

Q.31 format spans two 16-bit memory words. The 16-bit word stored in the lower memory location contains the 16 least-significant bits, and the higher memory location contains the most-significant 15 bits and the sign bit. The approximate allowable range of numbers in Q.31 representation is (-1,1) and the finest fractional resolution is $2^{-31} = 4.66 \times 10^{-10}$.

Table A-3. Q.31 Low Memory Location Bit Fields

Bit	15	14	13	12	…	3	2	1	0
Value	Q15	Q14	Q13	Q12	…	Q3	Q2	Q1	Q0

Table A-4. Q.31 High Memory Location Bit Fields

Bit	15	14	13	12	…	3	2	1	0
Value	S	Q30	Q29	Q28	…	Q19	Q18	Q17	Q16

Calculating the Reciprocal of a Q15 Number

The most optimal method for calculating the inverse of a fractional number (Y=1/X) is to normalize the number first. This limits the range of the number as follows:

$$0.5 \le Xnorm < 1$$
$$-1 \le Xnorm \le -0.5 \qquad (1)$$

The resulting equation becomes

$$Y = \frac{1}{(Xnorm * 2^{-n})}$$

or

$$Y = \frac{2^n}{Xnorm} \qquad (2)$$

where n = 1, 2, 3, …, 14, 15

Letting $Ye = 2^n$:

$$Ye = 2^n \qquad (3)$$

Substituting (3) into equation (2):

$$Y = Ye * \frac{1}{Xnorm} \qquad (4)$$

Letting $Ym = \frac{1}{Xnorm}$:

$$Ym = \frac{1}{Xnorm} \qquad (5)$$

Substituting (5) into equation (4):

$$Y = Ye * Ym \qquad (6)$$

For the given range of Xnorm, the range of Ym is:

$$1 \le Ym < 2$$
$$-2 \le Ym \le -1 \qquad (7)$$

To calculate the value of Ym, various options are possible:

a) Taylor Series Expansion

b) 2nd,3rd,4th,.. Order Polynomial (Line Of Best Fit)

c) Successive Approximation

The method chosen in this example is (c). Successive approximation yields the most optimum code versus speed versus accuracy option. The method outlined below yields an accuracy of 15 bits.

Assume $Ym(new) = $ exact value of $\frac{1}{Xnorm}$:

$$Ym(new) = \frac{1}{Xnorm} \tag{c1}$$

or

$$Ym(new) * X = 1 \tag{c2}$$

Assume $Ym(old) = $ estimate of value $\frac{1}{X}$:

$$Ym(old) * Xnorm = 1 + Dyx$$

or

$$Dxy = Ym(old) * Xnorm - 1 \tag{c3}$$

where Dyx = error in calculation

Assume that Ym(new) and Ym(old) are related as follows:

$$Ym(new) = Ym(old) - Dy \tag{c4}$$

where Dy = difference in values

Substituting (c2) and (c4) into (c3):

$$Ym(old) * Xnorm = Ym(new) * Xnorm + Dxy$$

$$(Ym(new) + Dy) * Xnorm = Ym(new) * Xnorm + Dxy$$

$$Ym(new) * Xnorm + Dy * Xnorm = Ym(new) * Xnorm + Dxy$$

$$Dy * Xnorm = Dxy$$

$$Dy = Dxy * \frac{1}{Xnorm} \tag{c5}$$

Assume that 1/Xnorm is approximately equal to Ym(old):

$$Dy = Dxy * Ym(old) \text{ (approx)} \tag{c6}$$

Substituting (c6) into (c4):

$$Ym(new) = Ym(old) - Dxy * Ym(old) \tag{c7}$$

Substituting for Dxy from (c3) into (c7):

$$Ym(new) = Ym(old) - (Ym(old) * Xnorm - 1) * Ym(old)$$

$$Ym(new) = Ym(old) - Ym(old)^2 * Xnorm + Ym(old)$$

$$Ym(new) = 2 * Ym(old) - Ym(old)^2 * Xnorm \tag{c8}$$

If after each calculation we equate Ym(old) to Ym(new):

$$Ym(old) = Ym(new) = Ym$$

Then equation (c8) evaluates to:

$$Ym = 2 * Ym - Ym^2 * Xnorm \qquad \text{(c9)}$$

If we start with an initial estimate of Ym, then equation (c9) converges to a solution very rapidly (typically 3 iterations for 16-bit resolution).

The initial estimate can be obtained from a look-up table, from choosing a midpoint, or simply from linear interpolation. The method chosen for this problem is linear interpolation and accomplished by taking the complement of the least significant bits of the Xnorm value.

Index

16-bit reciprocal function 4-100
32-bit by 16-bit long division function 4-79
32-bit vector multiplication 4-93

A

acorr 4-7
adaptive delayed LMS filter 4-42
  fast implemented 4-44
add 4-9
arctangent 2 implementation 4-10
arctangent implementation 4-11
atan16 4-11
atan2_16 4-10
autocorrelation 4-7

B

base 10 logarithm 4-80
base 2 logarithm 4-82
base e logarithm 4-83
bexp 4-13
block exponent implementation 4-13
block FIR filter (fast) 4-52

C

cascaded IIR direct form I 4-75
cascaded IIR direct form II 4-71, 4-73
cbrev 4-14
cbrev32 4-15
cfft 4-16
cfft off-place 4-21
cfft32 4-19
cfir 4-23
cifft 4-27
cifft32 4-29
complex bit reverse 4-14
  32-bit 4-15
complex FIR filter 4-23
conversion
  floating-point to Q15 (fltoq15) 4-64
  Q15 to floating-point (q15tofl) 4-97
convol 4-34
convol1 4-36
convol2 4-38
convolution 4-34
convolution (fast) 4-36
convolution (fastest) 4-38
corr 4-40
correlation
  auto (acorr) 4-7
  full-length (corr) 4-40

D

decimating FIR filter 4-55
dlms 4-42
dlmsfast 4-44
double-precision IIR filter 4-69
DSPLIB
  arguments 3-2
  calling a function from assembly language source code 3-3
  calling a function from C 3-3
  content 2-2
  data types 3-2
  dealing with overflow and scaling issues 3-4
  how to install 2-3
  how to rebuild 2-4

E

expn 4-48

exponential base e 4-48

F

FFT
  forward complex
    cfft 4-16
    cfft off-place 4-21
    cfft32 4-19
  forward real, in-place (rfft) 4-102
  inverse complex
    cifft 4-27
    cifft32 4-29
  inverse real, in-place (rifft) 4-104

fir 4-49

FIR filter 4-49
  block filter, fast (fir2) 4-52
  complex (cfir) 4-23
  decimating (firdec) 4-55
  direct form (fir) 4-49
  Hilbert Transformer 4-65
  interpolating (firinterp) 4-57
  lattice forward (firlat) 4-59
  symmetric (firs) 4-61

FIR Hilbert Transformer 4-65

fir2 4-52

firdec 4-55

firinterp 4-57

firlat 4-59

firs 4-61

floating-point to Q15 conversion 4-64

fltoq15 4-64

forward complex FFT 4-16
  32-bit 4-19
  off-place 4-21

forward complex IFFT 4-31

forward real FFT
  in-place 4-102

H

hilb16 4-65

I

IIR filter
  cascaded, direct form I (iircas51) 4-75
  cascaded, direct form II (iircas4) 4-71
  cascaded, direct form II (iircas5) 4-73
  double-precision (iir32) 4-69
  lattice inverse (iirlat) 4-77

iir32 4-69

iircas4 4-71

iircas5 4-73

iircas51 4-75

iirlat 4-77

index and value of maximum element of a vector 4-87

index and value of minimum element of a vector 4-90

index of maximum element of a vector 4-84

index of maximum element of a vector less than or equal to 34 4-86

index of minimum element of a vector 4-88

interpolating FIR filter 4-57

inverse complex FFT 4-27
  32-bit 4-29

inverse real FFT
  in-place 4-104

L

lattice forward (FIR) filter 4-59

lattice inverse (IIR) filter 4-77

ldiv16 4-79

log_10 4-80

log_2 4-82

logarithm
  base 10 (log_10) 4-80
  base 2 (log_2) 4-82
  base e (logn) 4-83

logn 4-83

M

matrix multiplication 4-91

matrix transpose 4-92

maxidx 4-84

maxidx34 4-86

maximum element of a vector
  index (maxidx) 4-84
  index and value (maxvec) 4-87
maximum element of a vector less than or equal to 34, index (maxidx34) 4-86
maximum value of a vector 4-87
maxval 4-87
maxvec 4-87
minidx 4-88
minimum element of a vector
  index (minidx) 4-88
  index and value (minvec) 4-90
minimum value of a vector 4-89
minval 4-89
minvec 4-90
mmul 4-91
mtrans 4-92
mul32 4-93

N

natural logarithm (logn) 4-83
neg 4-94
neg32 4-95

O

off-place cifft 4-31

P

power 4-96

Q

Q15 to floating-point conversion 4-97
q15tofl 4-97

R

rand16 4-97
rand16init 4-99
random number generation
  algorithm 4-97
  initialization 4-99
recip16 4-100
rfft 4-102
rifft 4-104

S

sine 4-105
sqrt_16 4-106
square root of a 16-bit number 4-106
sub 4-107
symmetric FIR filter 4-61

V

vector add 4-9
vector negate 4-94
vector negate, double-precision 4-95
vector power 4-96
vector subtract 4-107