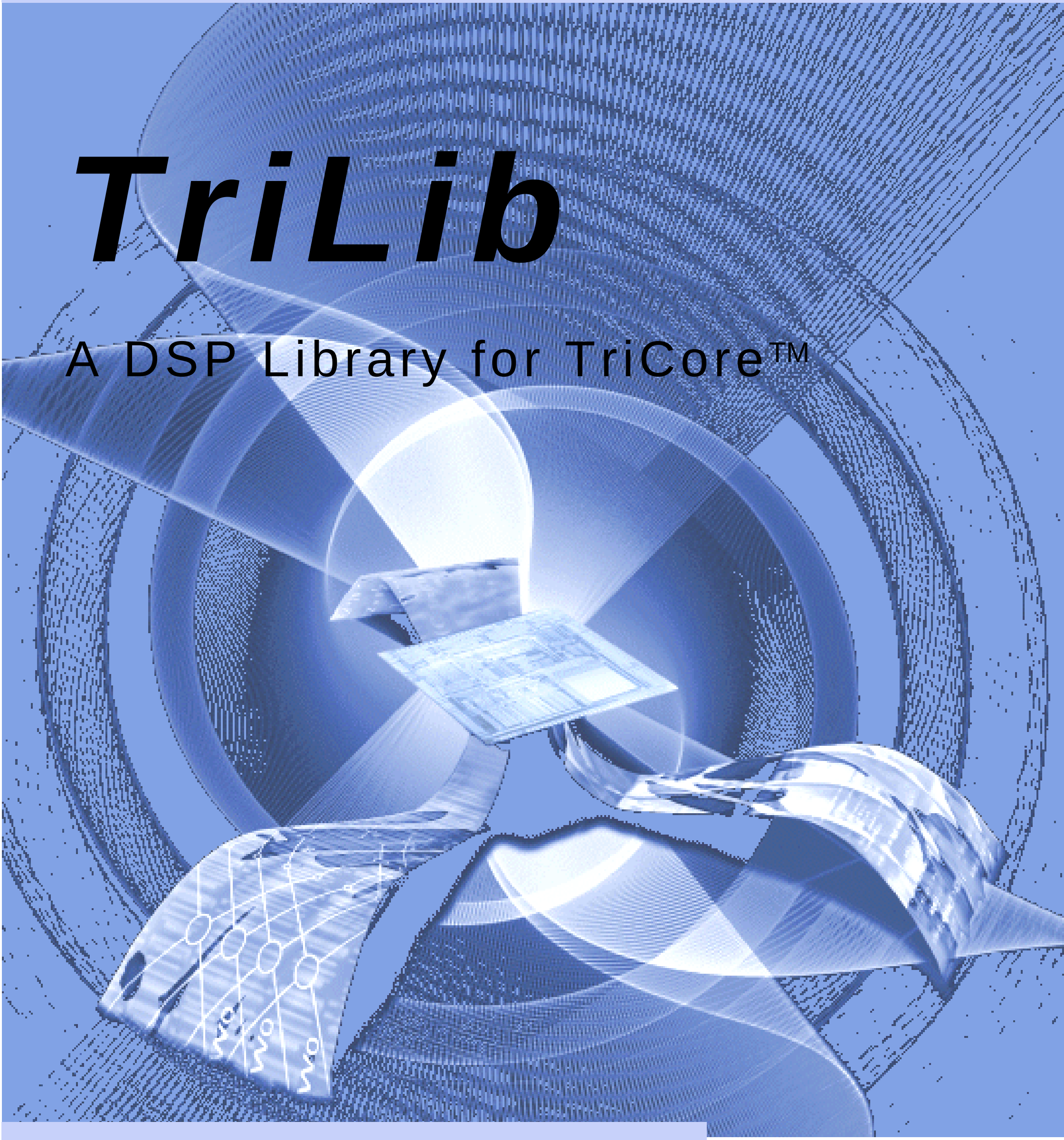
User's Manual, V 1.2, Jan. 2001

TriLib

A DSP Library for TriCore™



IP Cores

Never stop thinking.

Edition 2000-01

Published by
Infineon Technologies AG
81726 München, Germany

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

TriLib

A DSP Library for TriCore™

Never stop thinking.

Revision History:	**2000-01**	V 1.2
Previous Version:	-	V 1.1
Page	Subjects (major changes since last revision)	
	New functions (Mathematical, Statistical, FFT)	
Current Version	-	V 1.2
	All the functions are ported to GNU Compiler	
	New functions (Random number, Mixed Adaptive, Mixed FFT, Multirate FIR)	
Page 407	Applications	
	GUI on the host side to provide the visual control for two embedded target applications	
Page 425	FAQs	
Page 435	Appendix	
Page 459	Glossary	

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

trilib-support@infineon.com

Preface

This is the User Manual for TriLib-a DSP library for TriCore. TriCore is the first single-core 32-bit microcontroller-DSP architecture optimized for real-time embedded systems. The DSP core of TriCore is a fixed point one.

This manual describes the implementation of essential algorithms for general digital signal processing applications on the TriCore DSP. Characteristics of TriLib and the Installation and Build procedure are also described.

The source codes are C as well as C++ -callable and thus this library can be used as a library of basic functions for developing bigger applications on TriCore. The library serves as a user guide for TriCore programmers. It demonstrates how the processor's architecture can be exploited for achieving high performance. There are number of ways to implement an algorithm. The algorithms have been implemented with the primary aim of optimizing execution speed, i.e., minimize number of execution cycles.

The various functions and algorithms implemented and described about in the user manual are:

- Complex Arithmetic Functions
- Vector Arithmetic Functions
- Filters
– FIR
– IIR
– Adaptive FIR
- Transforms
– FFT
– DCT
- Mathematical Functions
- Matrix Operations
- Statistical Functions

The user manual describes each function in detail under the following heads:

Signature:

This gives the function interface.

Inputs:

Lists the inputs to the function.

Outputs:

Lists the output of the function.

Return:

Gives the return value of the function if any.

Description:

Gives a brief note on the implementation, the size of the inputs and the outputs, alignment requirements etc.

Pseudocode:

The implementation is expressed as a pseudocode using C conventions.

Techniques:

The techniques employed for optimization are listed here.

Assumptions:

Lists the assumptions made for an optimal implementation such as constraint on buffer size. The input output formats are also given here.

Memory Note:

A detailed sketch showing how the arrays are stored in memory, the nature of the buffers (linear/circular), the alignment requirements of the different buffers, the nature of the arithmetic performed on them (packed, simple). The diagrams give a great insight into the actual implementation.

Implementation Note:

Gives a very detailed note on the implementation. The codes in TriLib are optimized for speed. An optimized code is not very easy to understand. The implementation note is very helpful in overcoming this hurdle. For example, how techniques such as loop unrolling (if employed) help in optimization is described in detail.

Further, the path of an **Example** calling program, the **Cycle Count** and **Code Size** are given for each function.

Organization

Chapter 1, Introduction, gives a brief introduction of the TriLib and its features.

Chapter 2, Installation and Build, describes the TriLib content, how to install and build the TriLib.

Chapter 3, DSP Library Notations, describes the DSP Library data types, arguments, calling a function from the C code and the assembly code, and the implementation notes.

Chapter 4, Function Descriptions, describes the Complex arithmetic functions, Vector arithmetic functions, FIR filters, IIR filters, Adaptive filters, Fast Fourier Transforms, Discrete Cosine Transform, Mathematical functions, Matrix operations and Statistical functions. Each function is described with its signature, inputs, outputs, return, brief description, pseudocode, techniques used, assumptions made, memory note, implementation details, example, cycle count and code size.

Chapter 5, Applications, describes the applications such as Spectrum Analyzer, Sweep Oscillator and Equalizer using implemented TriLib functions.

Chapter 6, References, gives the list of related references.

Chapter 7, FAQs, gives Frequently Asked Questions about FIR, IIR and FFT.

Chapter 8, Appendix, gives the conventions for C and assembly code, file naming conventions, directory structure and porting for the Tasking, GHS and GNU compilers.

Chapter 9, Glossary, gives a brief explanation of the terminology used in the TriLib user manual in alphabetical order.

What's new?

- New functions have been added
- All functions are now supported on GNU compiler also
- Three Applications showing the use of functions from TriLib are added

- A powerful GUI on the host side is added to provide visual control to the embedded target application
- FAQs, Appendix and Glossary are added
- The GHS and Tasking compiler now have an extra implementation for C and C++ respectively thereby to give flexibility to the user to use anyone for their convenience
- TriLib Classes for the much bigger **TriApp** foundation classes called as **TFC (**TriCore application foundation classes**)** to enable developers to scale up their signal processing applications

Acknowledgements

The technical substance of this manual has been mainly developed by the Infineon's TriLib development team. These are designed, developed and tested over the hardware. We in advance would like to acknowledge users for their feedback and suggestions to improve this product. The development team would like to thank Dieter Stengl, Director for CMD TO S/W for all his support and encouragement. Rakesh Verma, Technical Manager, Wipro, for his support to the Wipro's development team and co-ordination with the Infineon team. Thomas Varghese, Arun Naik, Sreenivas, Mahesh for their valuable contribution in giving the feedback on user manual and active participation in some of the code reviews and also for their technical support. The team also recognizes the effort of Savitha for her patience in meticulously preparing, typesetting and reviewing the User Manual. We also would like to thank our marketing team for their comments and inputs.

Mark Nuchimowicz, Ramachandra, Rashmi, Preethi, Manoj, Ankur and Nagaraj

TriLib Development team - Infineon

Acronyms and Definitions

Acronyms and Definitions

Acronyms	**Definitions**
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DIF	Decimation-In-Frequency
DIT	Decimation-In-Time
DLMS	Delayed Least Mean Square
DSP	Digital Signal Processing

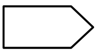
Acronyms and Definitions

Acronyms	Definitions
TriLib	DSP Library functions for TriCore
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
IIR	Infinite Impulse Response

Documentation/Symbol Conventions

The following is the list of documentation/symbol conventions used in this manual.

Documentation/Symbol Conventions

Documentation/ Symbol convention	Description
`Courier`	Pseudocode
`(*)`	Denotes Q format multiplication
Times-italic	File name
	Pointer
	Circular pointer

1 Introduction

1.1 Introduction to TriLib, a DSP Library for TriCore

The TriLib, a DSP Library for TriCore is C-callable, hand-coded assembly, general purpose signal processing routines. These routines are extensively used in real-time applications where speed is critical.

The TriLib includes more than 60 commonly used DSP routines. The throughput of the system using the TriLib routines is considerably better than those achieved using the equivalent code written in ANSI C language. The TriLib significantly helps in understanding the general purpose signal processing routines, its implementation on TriCore. It also reduces the DSP application development time. The TriLib also provides the source code. Few applications are also provided as part of TriLib to demonstrate the usage of functions.

The routines are broadly classified into the following functional categories:

- Complex Arithmetic
- Vector Arithmetic
- FIR Filters
- IIR Filters
- Adaptive Filters
- Fast Fourier Transforms
- Discrete Cosine Transform
- Mathematical functions
- Matrix operations
- Statistical functions

1.2 Features

- Covers the common DSP algorithms with Source codes
- Hand-coded and optimized assembly modules
- C/C++ callable functions on Tasking, GreenHills and GNU compilers
- Multi platform support - Win 95, Win 98, Win NT
- Bit-exact reference C codes for easy understanding and verification of the algorithms
- Assembly implementation tested for bit exactness against model C codes
- Workarounds implemented to take care of known Core errors
- Examples to demonstrate the usage of functions
- Example input test vectors and the output test data for verification

- Comprehensive Users manual covering many aspects of implementation
- Useful Applications built using the *TriLib* to demonstrate the product
- Powerful User friendly GUI interface for applications built using *TriLib*
- ***TriApp***-*TriLib* application foundation class for extending the *TriLib* functionality
- Supports the Object Oriented application development in C++ and Java
- User helpful Demoshield based setup and registration program

1.3 Future of the TriLib

The planned future releases will have the following improvements.

- Expansion of the library by adding more number of functions in the domains such as image processing, speech processing and the generic core routines of DSP.
- Upgrading the existing 16 bit functions to 32 bit

1.4 Support Information

Any suggestions for improvement, bug report if any, can be sent via e-mail to ***trilib-support@infineon.com***.

Visit ***www.infineon.com*** for update on TriLib releases.

2 Installation and Build

2.1 TriLib Content

The following table depicts the TriLib content with its directory structure.

Table 2-1 Directory Structure

Directory name	Contents	Files
TriLib	Directories which has all the files related to the TriLib	None
source	Directories Tasking, GreenHills and GNU	None
Tasking	Individual directories for each functional category. Each directory has respective assembly language implementation files of the library functions	*.asm
GreenHills	Individual directories for each functional category. Each directory has respective assembly language implementation files of the library functions	*.tri
GNU	Individual directories for each functional category. Each directory has respective assembly language implementation files of the library functions	*.S
include	Directories Tasking, GreenHills and GNU and common include file for 'C' of all the three compilers	TriLib.h
Tasking	Include files for assembly routine	*.inc for assembly
GreenHills	Include files for assembly routine	*.h for assembly
GNU	Include files for assembly routine	*.h for assembly
docs	User Manual Convention Manual readme.txt	*.fm, *.pdf *.doc *.txt
examples	Directories Tasking and GreenHills	None

Table 2-1 Directory Structure

Tasking	Individual directories for each functional category. Each directory has respective example ‘c’ and ’cpp’ functions to depict the usage of TriLib	*.c, *.cpp
GreenHills	Individual directories for each functional category. Each directory has respective example ‘cpp’ and ’c’ functions to depict the usage of TriLib	*.cpp, *.c
GNU	Individual directories for each functional category. Each directory has respective example ‘c’ functions to depict the usage of TriLib	*.c
refcode	Individual directories for each functional category. Each directory has respective reference ‘C’ code of the corresponding assembly implementation in source directory, which works on Tasking compiler	None
build	Build information	*.pjt, *.bld
testvectors	Test vectors for the different functions in their respective directories	*.dat

2.2 Installing TriLib

TriLib is distributed as a self extracting ZIP file. To install the TriLib on the system, unzip the ZIP file and run setup. This will install all the files in the respective directories.

The directory structure is as given in **“TriLib Content” on Page 17**

2.3 Building TriLib

Include the *TriLib.h* into your project and also include the same into the files that need to call the library function like:

```
#include “TriLib.h”
```

Set the include path in the tool that you are using for both the project as well as each of the files you have included (it is observed that sometimes you get errors if it is not set in the options of each individual files). Please refer the documentation of the Tasking, GreenHills and GNU for more details.

In case of Tasking, the *#define* part for **_TASKING** selection box should be checked and in case of GreenHills it should be typed manually as **_GHS,** otherwise it might give lot of compiler errors.

In both the compilers the **DSPEXT** has to be defined in the project options for both the assembly sources and the c files in the respective project options when the DSP extension for respective compilers (Tasking and GreenHills) have to be used.

For without DSP extension don't define DSPEXT for c compiler option. For assembler option set DSPEXT=0. GNU compiler doesn't support data types for DSP. So DSPEXT need not be defined or undefined in case of GNU compiler.

If the .cpp file is to be used, in case of Tasking or GreenHills compiler, the macro _cplusplus is to be defined under compiler options.

For setting the other CCD, such as H/W workarounds, use the assembler options.

Now include the respective source files for the required functionality into your project. Refer the functionality table, **Table 2-2**

Build the system and start using the library.

2.4 Source Files List

Table 2-2 Source files

Tasking	GreenHills	GNU
Complex Arithmetic functions		
CplxOp_16.asm *CplxOp_32.asm*	*CplxOp_16.tri* *CplxOp_32.tri*	*CplxOp_16.S* *CplxPhMag_16.S* *CplxOp_32.S* *CplxPhMag_32.S*
Vector Arithmetic functions		
VectOp_16.asm	*VectOp_16.tri* *VectOp1_16.tri*	*VectOp_16.S* *VectOp1_16.S*
FIR filters		
Fir_16.asm *FirBlk_16.asm* *Fir_4_16.asm* *FirBlk_4_16.asm*	*Fir_16.tri* *FirBlk_16.tri* *Fir_4_16.tri* *FirBlk_4_16.tri*	*Fir_16.S* *FirBlk_16.S* *Fir_4_16.S* *FirBlk_4_16.S*

Table 2-2 Source files

FirSym_16.asm *FirSymBlk_16.asm* *FirSym_4_16.asm* *FirSymBlk_4_16.asm* *FirDec_16.asm* *FirInter_16.asm*	*FirSym_16.tri* *FirSymBlk_16.tri* *FirSym_4_16.tri* *FirSymBlk_4_16.tri* *FirDec_16.tri* *FirInter_16.tri*	*FirSym_16.S* *FirSymBlk_16.S* *FirSym_4_16.S* *FirSymBlk_4_16.S* *FirDec_16.S* *FirInter_16.S*
IIR filters		
IirBiq_4_16.asm *IirBiqBlk_4_16.asm* *IirBiq_5_16.asm* *IirBiqBlk_5_16.asm*	*IirBiq_4_16.tri* *IirBiqBlk_4_16.tri* *IirBiq_5_16.tri* *IirBiqBlk_5_16.tri*	*IirBiq_4_16.S* *IirBiqBlk_4_16.S* *IirBiq_5_16.S* *IirBiqBlk_5_16.S*
Adaptive filters		
Dlms_4_16.asm *DlmsBlk_4_16.asm* *CplxDlms_4_16.asm* *CplxDlmsBlk_4_16.asm* *Dlms_2_16x32.asm* *DlmsBlk_2_16x32.asm*	*Dlms_4_16.tri* *DlmsBlk_4_16.tri* *CplxDlms_4_16.tri* *CplxDlmsBlk_4_16.tri* *Dlms_2_16x32.tri* *DlmsBlk_2_16x32.tri*	*Dlms_4_16.S* *DlmsBlk_4_16.S* *CplxDlms_4_16.S* *CplxDlmsBlk_4_16.S* *Dlms_2_16x32.S* *DlmsBlk_2_16x32.S*
FFT		
FFT_2_16.asm *FFT_2_32.asm* *FFT_2_16X32.asm*	*FFT_2_16.tri* *FFT_2_32.tri* *FFT_2_16X32.tri*	*FFT_2_16.S* *FFT_2_32.S* *FFT_2_16X32.S*
DCT		
DCT_2_8.asm	*DCT_2_8.tri*	*DCT_2_8.S*
Mathematical Functions		
Sine_32.asm *Cos_32.asm* *Arctan_32.asm* *Sqrt_32.asm* *Ln_32.asm* *AntiLn_16.asm* *Expn_16.asm* *XpowY_32.asm* *RandInit_16.asm* *Rand_16.asm*	*Sine_32.tri* *Cos_32.tri* *Arctan_32.tri* *Sqrt_32.tri* *Ln_32.tri* *AntiLn_16.tri* *Expn_16.tri* *XpowY_32.tri* *RandInit_16.tri* *Rand_16.tri*	*Sine_32.S* *Cos_32.S* *Arctan_32.S* *Sqrt_32.S* *Ln_32.S* *AntiLn_16.S* *Expn_16.S* *XpowY_32.S* *RandInit_16.S* *Rand_16.S*
Matrix Functions		

Table 2-2 Source files

MatAdd_16.asm *MatSub_16.asm* *MatMult_16.asm* *MatTrans_16.asm*	*MatAdd_16.tri* *MatSub_16.tri* *MatMult_16.tri* *MatTrans_16.tri*	*MatAdd_16.S* *MatSub_16.S* *MatMult_16.S* *MatTrans_16.S*
Statistical Functions		
ACorr_16.asm *Conv_16.asm* *Avg_16.asm*	*ACorr_16.tri* *Conv_16.tri* *Avg_16.tri*	*ACorr_16.S* *Conv_16.S* *Avg_16.S*

3 DSP Library Notations

3.1 TriLib Data Types

The TriLib handles the following fractional data types.

Table 3-1 TriLib Data Types

1Q15 (DataS)	1Q15 operand is represented by a short data type (frac16/_sfract) that is predefined as DataS in *TriLib.h* header file.
1Q31 (DataL)	1Q31 operand is represented by a long data type (frac32/_fract) that is predefined as DataL in *TriLib.h* header file.
CplxS	Complex data type contains the two 1Q15 data arranged in Re-Im format.
CplxL	Complex data type contains the two 1Q31 data arranged in Re-Im format.

3.2 Calling a DSP Library Function from C Code

After installing the TriLib, do the following to include a TriLib function in the source code.

1. Include the *TriLib.h* include file
2. Include the source file that contains required DSP function into the project along with the other source files
3. Include TriConv.inc (Tasking) or TriConv.h (GreenHills)
4. Set the include paths to point the location of the *TriLib.h*
5. Set the Compiler Conditional Directives (CCDs) for selection of DSP extension
6. Set the Compiler Conditional Directives (CCDs) to generate code with workarounds for the H/W bugs
7. Build the system

3.3 Calling a DSP Library Function from Assembly Code

The TriLib functions are written to be used from C. Calling the functions from assembly language source code is possible as long as the calling function conforms to the TriCore calling conventions. Refer TriCore Calling Conventions manual for more details.

3.4 TriLib Example Implementation

The examples of how to use the TriLib functions are implemented and are placed in *examples* subdirectory. This subdirectory contains a subdirectory for set of functions.

3.5 TriLib Implementation - A Technical Note

3.5.1 Memory Issues

The TriLib is implemented with the known alignment constraints for the TriCore memory addressing architecture. The following information gives the alignment and sizes conditions in order to work properly.

Halfword alignment for `ld.d` and `st.d` is only allowed when the source or destination address is located in on-chip memory. For external memory accesses over TriCore's peripherals bus, doubleword access must be word aligned (TriCore Architecture Manual p.13).

The size and length of a circular buffer have the following restrictions (TriCore Architecture Manual p.13).

- The start of the buffer start must be aligned to a 64-bit boundary.
- The length of the buffer must be a multiple of the data size, where the data size is determined from the instruction being used to access the buffer.

Different alignment requirements for `ld.d` and `st.d` for internal and external memories impose different alignment of data in functions that use those instructions. In some cases (for example filter delay-buffer defined as circular-buffer) halfword aligned accesses to the data is required which prohibit the location of such data structures in external memory.

For example Fir_4_16() function, the delay-buffer of the filter is defined as circular-buffer.

In this case, when located in internal memory the buffer must have doubleword alignment (circular-buffer). After each call to the function the start position of the delay-buffer is shifted (with circular update) by halfword. The delay-buffer cannot be located in external memory because the load from the delay-buffer is executed by `ld.d` instruction and word alignment is no more satisfied.

3.5.2 Optimization Approach

Extended parallelism of the processor architecture increases the speed of the algorithms execution, but at the same time imposes some constraints on the size of Input-Buffers. So for example Fir_4_16() FIR filter executes at maximal possible speed on the TriCore but the size must be multiple of 4.

In the implementation of the algorithms following optimizations are applied:

- Packed arithmetic

- Mixed packed /simple arithmetic
- Simple arithmetic

From the point of view of size of the algorithm (Vector length, Filter length) two cases can be identified:

- Constraint on the dimension of vector, order of filter
- Arbitrary size

Best performance can be achieved with some constrains on the size in which case fully packed arithmetic is used in the kernel loop. Arbitrary size (not for all algorithms) can be achieved by using

- Simple arithmetic in the kernel loop
- Mixed packed/simple arithmetic, partitioning of the algorithm size so that the kernel loop uses packed arithmetic with conditional post processing to achieve arbitrary size

To achieve maximal performance and flexibility some functions have several implementations optimized for specific target requirements.

Following implementations can be recognized:

- On sample, optimized for single sample processing
- On block, optimized for block processing
- Best performance with restriction on size
- Arbitrary size, trade-off between performance and flexibility

For example FIR filter is implemented as

Table 3-2 FIR Filter Implementations

Fir_16()	Sample processing, trade-off on performance, arbitrary size
Fir_4_16()	Sample processing, best performance, size restriction
FirBlk_16()	Block processing, trade-off on performance, arbitrary size
FirBlk_4_16()	Block processing, best performance, size restriction

The SIMD instructions are exploited in the FFT by the special arrangement of the Real and Imaginary parts of the complex number. The Real:Imag format is the conventional method of storing the complex number x+jy. In this case two complex numbers x_0+jy_0 and x_1+jy_1 is arranged as x_0x_1 and $j(y_0y_1)$.

3.5.3 Options in Library Configurations

Set of Conditional Compile Directives (CCD) on the C language level and assembly level define the configuration of the TriLib.

3.5.3.1 Compiler

Compiler selection is based on two CCD

Table 3-3 Compiler Selection

_Tasking	CCD on the C level for selecting the Tasking compiler
_GHS	CCD on the Cpp level for selecting the GHS compiler
COR1	Hardware workaround for TriCore ver1.2
COR14	Hardware workaround for TriCore ver1.2
CPU5	Hardware workaround for TriCore ver1.3

In the current implementation of the TriLib this setting is only evaluated in TriLib.h header file which is common to all the compilers.

All the library functions and examples have dedicated implementations for each compiler and are not influenced by this setting.

3.5.3.2 DSP Extensions

To improve the DSP functionality on the C language level Tasking compiler supports three additional special DSP specific intrinsic data types to perform fixed point arithmetic. Refer to the tools documentation for details.

Table 3-4 Tasking Special Data Types

_sfract	16 bits: 1 sign bit + 15 mantissa bits
_fract	32 bits: 1 sign bit + 31 mantissa bits
_accum	64 bits: 1 sign bit + 17 integral bits + 46 mantissa bits

To efficiently implement a circular buffer in the C language additional qualifier _circ is defined by Tasking. This can be used in conjunction with the other data types.

GHS compiler, extended support of DSP functionality is implemented by defining C++ classes.

Table 3-5 GHS Special Data Types

frac16	16 bits: 1 sign bit + 15 mantissa bits
frac32	32 bits: 1 sign bit + 31 mantissa bits
frac64	64 bits: 1 sign bit + 17 integral bits + 46 mantissa bits

Circular buffer pointer is implemented in GHS C++ compiler as a templatized class.

To make the library portable, TriLib function arguments use following predefined data types.

Table 3-6 Data Types

DataS	16-bit operands
DataL	32-bit operands
cptrDataS	circular-pointer to DataS circular-buffer
cptrDataL	circular-pointer to DataL circular-buffer

Depending on the compiler used and the setting of _DSPEXT CCD following assignments are used (implemented in TriLib.h)

Table 3-7 DSPEXT CCD Assignments

	DSPEXT=FALSE	DSPEXT=TRUE	
	Tasking, GHS, GNU	Tasking	GHS
DataS	short	_sfract	frac16
DataL	int	_fract	frac32
CptrDataS	struct (TriLib.h)	_sfract _circ*	circptr<frac16>

DSPEXT CCD has effect on the C/C++ level as well on the assembly implementations of the TriLib function.

3.5.4 Workarounds of known Behavioral Deviations

The instruction set of TriCore is defined in different syntax for the GreenHills and Tasking Tool sets. There are different deviations in each of the compilers. Particularly the GreenHills doesn't support some instructions in its Multi 2000 ver 2.0 and also there are behavioral changes in the ver 2.0.2. This can be potential risk in the development for people to migrate from one compiler to other. To give some instances of the known deviations.

Conditional move instruction (`cmov,cmovn`) is not supported in GHS ver 2.0 in this case select (`sel,seln`) instructions has to be used.

The data memory addressing is bit complicated in GHS, there are special syntax to do the same for instance syntaxes such as `%sdaoff` etc., are used. Refer the GHS documentation for more details.

The `jz` has problems in GHS ver 2.0 so in order to workaround this, usage of `jeq` is encouraged, The instruction `jz` works on GHS ver 2.0.2. The Sine/Cosine functions use `jz` instruction and will run on ver 2.0.2.

3.5.5 Testing Methodology

The TriLib is tested on GHS, Tasking simulator and TriCore TC10GP TriBoard ver2.4.

The Hardware workarounds have to be enabled only if the TriLib is intended to run on TC10GP (TriCore ver1.2, ver1.3) processor hardware.

4 Function Descriptions

Each function is described with its signature, inputs, outputs, return, brief description, pseudocode, techniques used, assumptions made, memory note, how it is implemented, example, cycle count and code size.

Functions are classified into the following categories.

- Complex Arithmetic functions
- Vector functions
- FIR filters
- IIR filters
- Adaptive filters
- Fast Fourier Transforms
- Discrete Cosine Transform
- Mathematical functions
- Matrix operations
- Statistical functions

4.1 Conventions

4.1.1 Argument Conventions

The following conventions have been followed while describing the arguments for each individual function.

Table 4-1 Argument Conventions

Argument	Convention
X,Y	Input data or input data vector
R	Output data
nX, nY, nR	The size of vectors X, Y, and R respectively. In functions where nX = nY = nR, only nX has been used
H	Filter coefficient vector (filter routines only)
nH	The size of vector H. Usually not defined explicitly
DataS	Data type definition equating a short, a 16-bit value representing a 1Q15 number
DataL	Data type definition equating a long, a 32-bit value representing a 1Q31 number
DataD	Reserved for 64-bit value

Table 4-1 Argument Conventions

Argument	Convention
cptrDataS	Circular pointer structure
CplxS	Data type definition equating a short, a 16-bit value representing a 1Q15 complex number
CplxL	Data type definition equating a long, a 32-bit value representing a 1Q31 complex number
aR	Pointer to Output-Buffer

4.1.2 Register Naming Conventions

The following register naming conventions have been followed.

Table 4-2 Register Naming Conventions

Argument	Convention
a	Address register or data type prefix
ca	Circular buffer address register pair

4.2 Complex Arithmetic Functions

4.2.1 Complex Numbers

A complex number z is an ordered pair (x,y) of real numbers x and y, written as

z= (x,y)

where x is called the real part and y the imaginary part of z.

4.2.2 Complex Number Representation

A complex number can be represented in different ways, such as

Rectangular form	$: C = R + iI$	[4.1]
Trigonometric form	$: C = M[\cos(\phi) + j\sin(\phi)]$	[4.2]
Exponential form	$: C = Me^{i\phi}$	[4.3]
Magnitude and angle form	$: C = M\angle\phi$	[4.4]

In the complex functions implementation, the rectangular form is considered.

4.2.3 Complex Plane

The geometrical representation of complex numbers as points in the plane is of great importance. Choose two perpendicular coordinate axis in the Cartesian coordinate system. The horizontal x-axis is called the real axis, and the vertical y-axis is called the imaginary axis. Plot a given complex number z=(x,y) = x + iy as the point P with coordinates (x, y). The xy-plane in which the complex numbers are represented in this way is called the Complex Plane. This is also called as the Argand diagram after the French mathematician Jean Robert Argand.

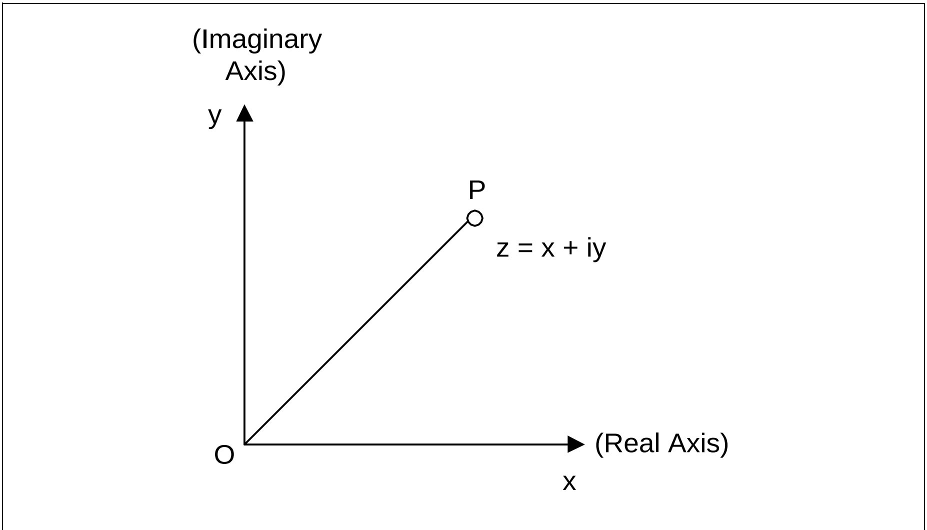


Figure 4-1 The Complex Plane (Argand Diagram)

4.2.4 Complex Arithmetic

Addition

if z_1 and z_2 are two complex numbers given by $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$,

$$z_1 + z_2 = (x_1 + iy_1) + (x_2 + iy_2) = (x_1 + x_2) + i(y_1 + y_2) \qquad [4.5]$$

Subtraction

if z_1 and z_2 are two complex numbers given by $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$,

$$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2) \qquad [4.6]$$

Multiplication

if z_1 and z_2 are two complex numbers given by $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$,

$$z_1.z_2 = (x_1 + iy_1).(x_2 + iy_2) = x_1x_2 + ix_1y_2 + iy_1x_2 + i^2\, y_1y_2$$

$$= (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1) \qquad [4.7]$$

Conjugate

The complex conjugate, $\bar{z}$ of a complex number z = x+iy is given by

$$\bar{z} = x - iy \tag{4.8}$$

and is obtained by geometrically reflecting the point z in the real axis.

Magnitude

The magnitude of a complex number z=x+iy is given by

$$|z| = \sqrt{x^2 + y^2} \tag{4.9}$$

Geometrically, |z| is the distance of the point z from the origin.
$|z_1-z_2|$ is the distance between z_1 and z_2.

Phase

The phase of complex number z=x+iy is given by

$$\text{phase} = \tan^{-1}(y/x) \tag{4.10}$$

Shift

Shifting of a complex number is indicated by the shift value. Left shifting is done if the shift value is positive and right shifting is done if shift value is negative.

$$\begin{aligned} Z_r &= x \gg \mathrm{abs}(\mathrm{shiftval}),\ \mathrm{if}(\mathrm{shiftval} < 0) \\ &\quad \mathrm{else}(x \ll \mathrm{shiftval}) \\ Z_i &= y \gg \mathrm{abs}(\mathrm{shiftval}),\ \mathrm{if}(\mathrm{shiftval} < 0) \\ &\quad \mathrm{else}(y \ll \mathrm{shiftval}) \end{aligned} \tag{4.11}$$

4.2.5 Complex Number Schematic

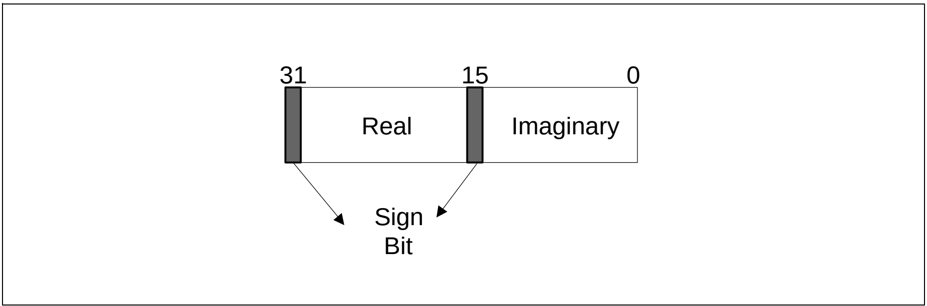


Figure 4-2 16-bit Complex number representation

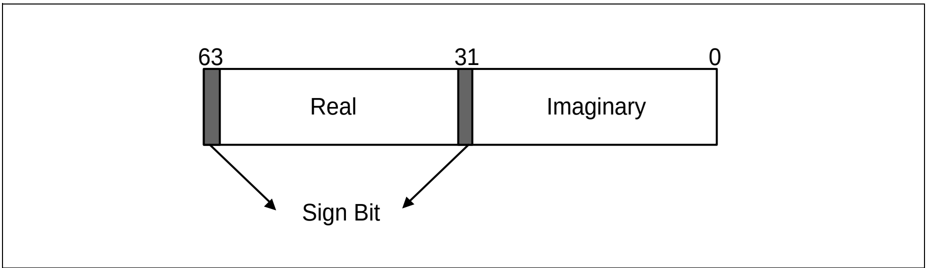


Figure 4-3 32-bit Complex number representation

4.2.6 Complex Data Structure

Table 4-3 Complex Data Structure

<table>
<tr><th>Tasking</th><th>GHS</th><th>ANSI C/GNU</th></tr>
<tr><td colspan="3">16 bit</td></tr>
<tr><td><pre>typedef struct
{
   _sfract imag;
   _sfract real;
} CplxS;</pre></td><td><pre>typedef struct
{
   frac16 imag;
   frac16 real;
} CplxS;</pre></td><td><pre>typedef struct
{
   short imag;
   short real;
} CplxS;</pre></td></tr>
<tr><td colspan="3">32 bits</td></tr>
<tr><td><pre>typedef struct
{
   _fract imag;
   _fract real;
} CplxL;</pre></td><td><pre>typedef struct
{
   frac32 imag;
   frac32 real;
} CplxL;</pre></td><td><pre>typedef struct
{
   long imag;
   long real;
} CplxL;</pre></td></tr>
</table>

4.2.7 Descriptions

The following complex arithmetic functions for 16 bit and 32 bit are described.

- Addition (with and without saturation)
- Subtraction (with and without saturation)
- Multiplication (with and without saturation)
- Conjugate
- Magnitude
- Phase
- Shift

CplxAdd_16 — Complex Number Addition for 16 bits

Signature

```
CplxS CplxAdd_16(CplxS X,
                 CplxS Y
                 );
```

Inputs

X : 16 bit Complex input value

Y : 16 bit Complex input value

Output None

Return The sum of two complex numbers as a 16 bit complex number

Description This function computes the sum of two 16 bit complex numbers. Wraps around the result in case of overflow.
The algorithm is as follows

$$\begin{aligned} R_r &= x_r + y_r \\ R_i &= x_i + y_i \end{aligned} \qquad [4.12]$$

Pseudo code

```
{
    R.real = X.real + Y.real;
                        //add the real part
    R.imag = X.imag + Y.imag;
                        //add the imaginary part
    return R;
}
```

Techniques None

Assumptions

- Input and output has a real and an imaginary part packed as 16 bit data to make a 32 bit complex data

CplxAdd_16 Complex Number Addition for 16 bits (cont'd)

Memory Note

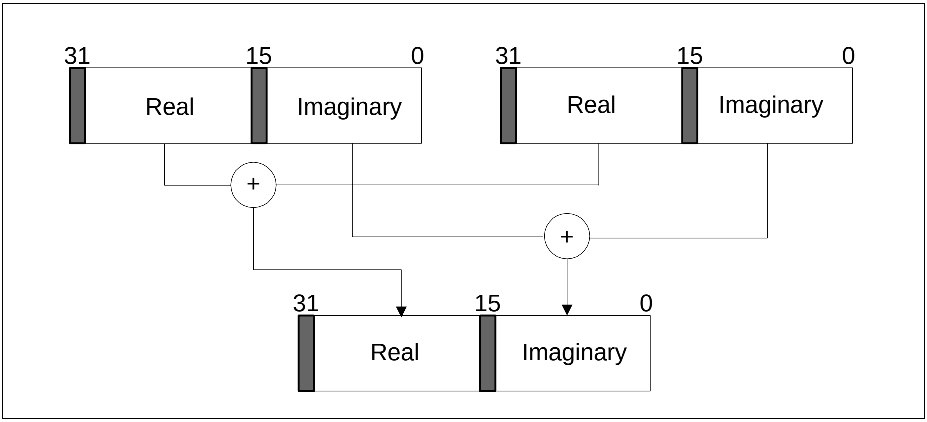


Figure 4-4 Complex Number addition for 16 bits

Example	*Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp* *Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c* *Trilib\Example\GNU\CplxArith\expCplx.c*
Cycle Count	1+2
Code Size	6 bytes

CplxAdds_16 Complex Number Addition for 16 bits with saturation

Signature

```
CplxS CplxAdds_16(CplxS X,
                  CplxS Y
                  );
```

Inputs

X : 16 bit Complex input value

Y : 16 bit Complex input value

Output None

Return The sum of two complex numbers as a 16 bit saturated complex number

Description This function computes the sum of two 16 bit complex numbers. In case of overflow, this saturates the result to 0x7FFF for positive values and 0x8000 for negative values. This is applicable for both real and imaginary part of the complex number. The algorithm is as follows

$$R_r = x_r + y_r$$
$$R_i = x_i + y_i \qquad [4.13]$$

Pseudo code

```
{
   R.real = (frac16 sat)(X.real + Y.real);
                      //add the real part
   R.imag = (frac16 sat)(X.imag + Y.imag);
                      //add the imaginary part
   return R;
}
```

Techniques None

Assumptions

- Input and output has a real and an imaginary part packed as 16 bit data to make a 32 bit complex data

CplxAdds_16 **Complex Number Addition for 16 bits with saturation** (cont'd)

Memory Note

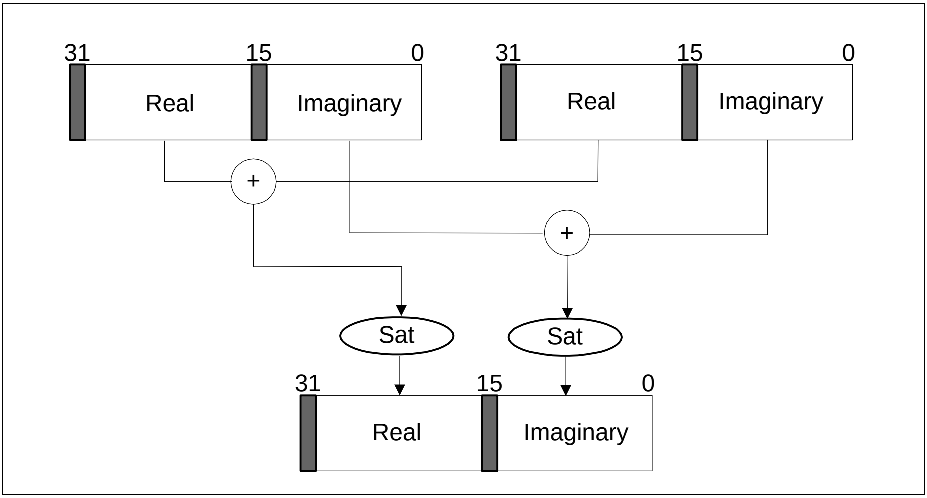


Figure 4-5 Complex number addition for 16 bits with saturation

Example	*Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp* *Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c* *Trilib\Example\GNU\CplxArith\expCplx.c*
Cycle Count	1+2
Code Size	6 bytes

CplxSub_16 Complex Number Subtraction for 16 bits

Signature

```
CplxS CplxSub_16(CplxS X,
                 CplxS Y
                 );
```

Inputs

X : 16 bit Complex input value

Y : 16 bit Complex input value

Output None

Return The difference of two complex numbers as a 16 bit complex number

Description This function finds the difference of two 16 bit complex numbers. Wraps around the result in case of underflow. The algorithm is as follows.

$$\begin{aligned} R_r &= x_r - y_r \\ R_i &= x_i - y_i \end{aligned} \qquad [4.14]$$

Pseudo code

```
{
   R.real = X.real - Y.real;
                  //subtract the real part
   R.imag = X.imag - Y.imag;
                  //subtract the imaginary part
   return R;
}
```

Techniques None

Assumptions

- Input and output has a real and an imaginary part packed as 16 bit data to make a 32 bit complex data

CplxSub_16 Complex Number Subtraction for 16 bits (cont'd)

Memory Note

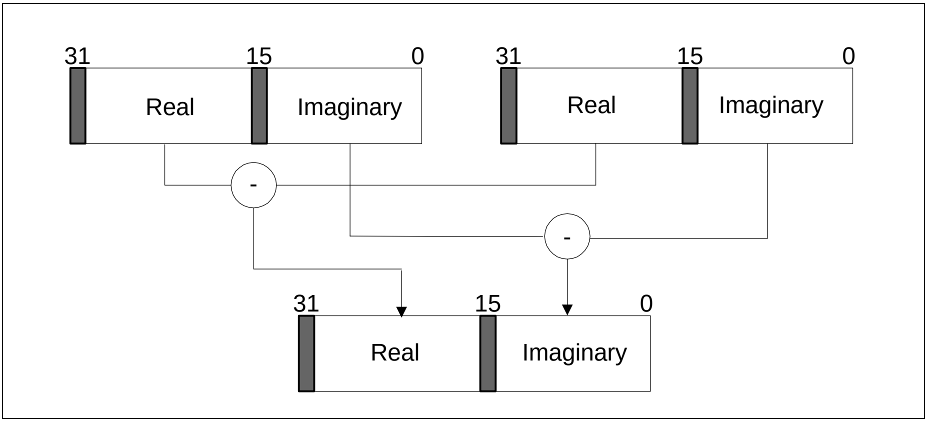


Figure 4-6 Complex number subtraction for 16 bits

Example *Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp*

Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c

Trilib\Example\GNU\CplxArith\expCplx.c

Cycle Count 1+2

Code Size 6 bytes

CplxSubs_16 — Complex Number Subtraction for 16 bits with saturation

Signature

```
CplxS CplxSubs_16(CplxS X,
                  CplxS Y
                  );
```

Inputs

X	:	16 bit Complex input value
Y	:	16 bit Complex input value

Output None

Return The difference of two complex numbers as a 16 bit saturated complex number

Description This function finds the difference of two 16 bit complex numbers. In case of overflow, this saturates the result to 0x7FFF for positive values and 0x8000 for negative values. The algorithm is as follows.

$$\begin{aligned} R_r &= x_r - y_r \\ R_i &= x_i - y_i \end{aligned} \qquad [4.15]$$

Pseudo code

```
{
   R.real = (frac16 sat)(X.real - Y.real);
                  //subtract the real part
   R.imag = (frac16 sat)(X.imag - Y.imag);
                  //subtract the imaginary part
   return R;
}
```

Techniques None

Assumptions

- Input and output has a real and an imaginary part packed as 16 bit data to make a 32 bit complex data

CplxSubs_16 — Complex Number Subtraction for 16 bits with saturation (cont'd)

Memory Note

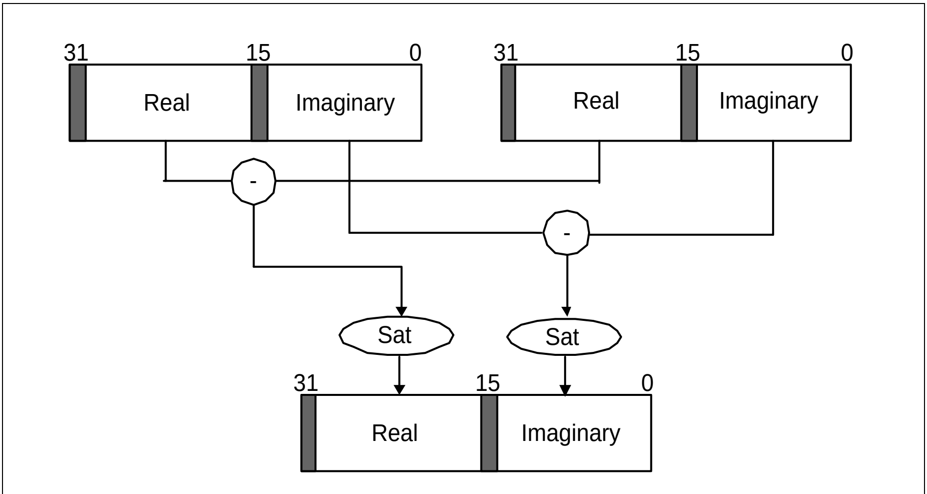


Figure 4-7 Complex number subtraction for 16 bits with saturation

Example	*Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp* *Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c* *Trilib\Example\GNU\CplxArith\expCplx.c*
Cycle Count	1+2
Code Size	6 bytes

CplxMul_16 Complex Number Multiplication for 16 bits

Signature

```
void CplxMul_16(CplxS X,
                CplxS Y,
                CplxL *R
               );
```

Inputs

X : 16 bit Complex input value

Y : 16 bit Complex input value

Output

R : The pointer to the product of two complex numbers as a 64 bit complex number

Return

None

Description

This function computes the product of the two 16 bit complex numbers. Wraps around the result in case of overflow.
The complex multiplication is computed as follows.

$$R_r = x_r \times y_r - x_i \times y_i$$

$$R_i = x_i \times y_r + x_r \times y_i$$

Pseudo code

```
{
   R->real = X.real*Y.real - Y.imag*X.imag;
   R->imag = X.real*Y.imag + Y.real*X.imag;
}
```

Techniques

None

Assumptions

- Input is in 1Q15 format
- Input and output has a real and an imaginary part packed as 16 bit data in 1Q15 format to make a 32 bit complex data

CplxMul_16 Complex Number Multiplication for 16 bits (cont'd)

Memory Note

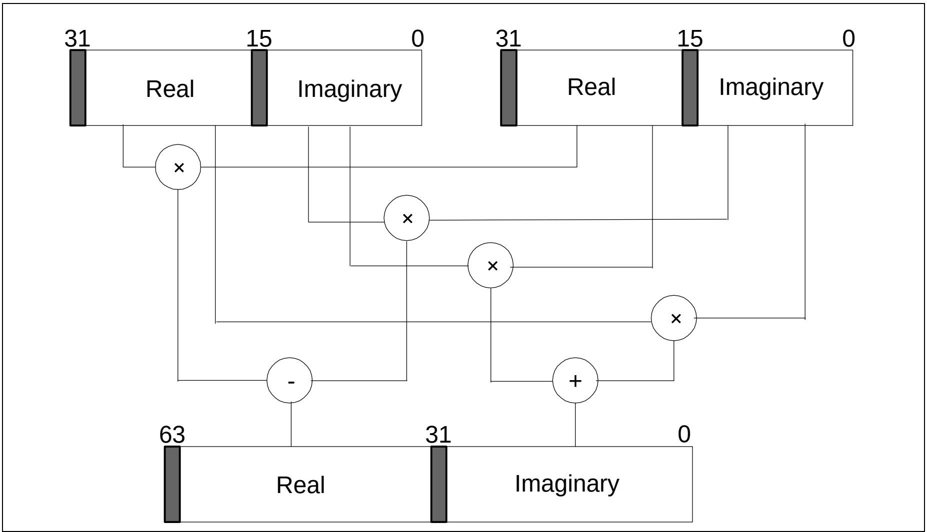


Figure 4-8 Complex number multiplication for 16 bits

Example	*Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp* *Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c* *Trilib\Example\GNU\CplxArith\expCplx.c*
Cycle Count	6+2
Code Size	30 bytes

CplxMuls_16 Complex Number Multiplication for 16 bits with Saturation

Signature
CplxS CplxMuls_16(CplxS X,
CplxS Y
);

Inputs

X	:	16 bit Complex input value
Y	:	16 bit Complex input value

Output None

Return The product of two complex numbers as a 32 bit saturated complex number

Description This function computes the product of the two 16 bit complex numbers. In case of overflow, the result is saturated to 0x7FFF for positive overflow and 0x8000 for negative underflow.
The complex multiplication is computed as follows.

$$R_r = x_r \times y_r - x_i \times y_i$$

$$R_i = x_i \times y_r + x_r \times y_i$$

Pseudo code

```
{
   R0.real = (frac32 sat)(X.real*Y.real - Y.imag*X.imag);
   R0.imag = (frac32 sat)(X.real*Y.imag + Y.real*X.imag);
   R0.real = (rnd)R0.real;
                         //rounding
   R0.imag = (rnd)R0.imag;
                         //rounding
   R.real = (frac16 sat)R0.real;
                         //load lower 16 bits
   R.imag = (frac16 sat)R0.imag;
                         //load lower 16 bits

   return R;
}
```

Techniques None

CplxMuls_16 Complex Number Multiplication for 16 bits with Saturation (cont'd)

Assumptions

- Inputs are in 1Q15 format
- Input and output has a real and an imaginary part packed as 16 bit data in 1Q15 format to make a 32 bit complex data

Memory Note

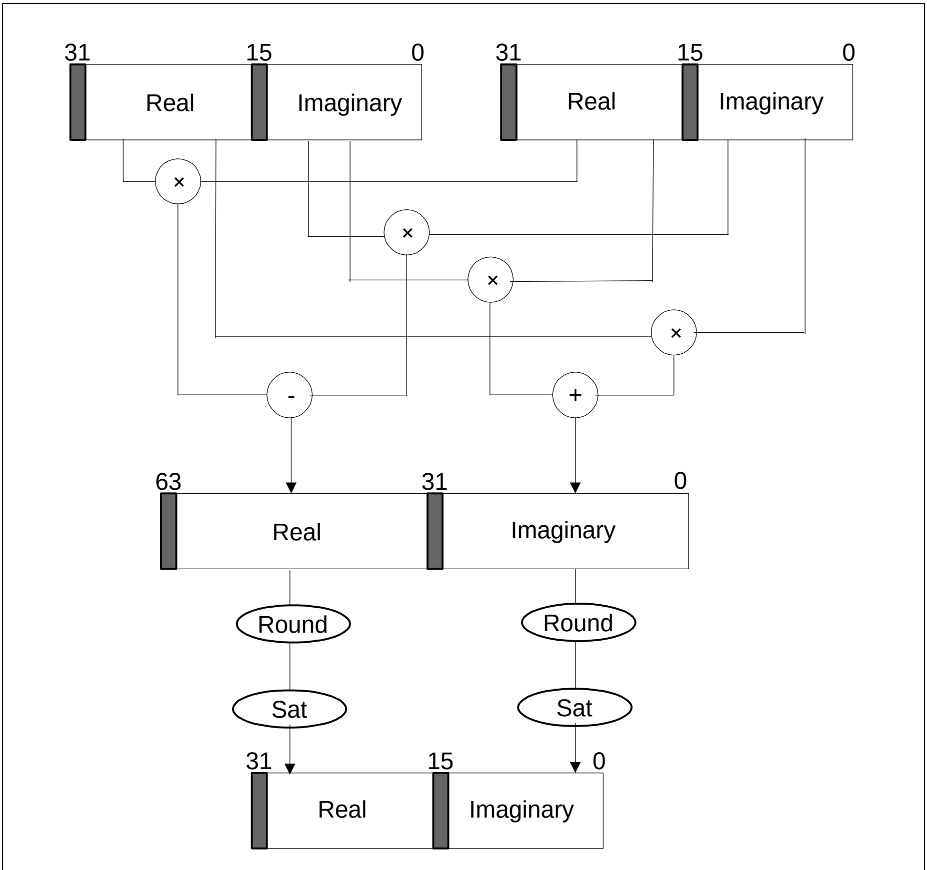


Figure 4-9 Complex number multiplication for 16 bits with saturation

CplxMuls_16 Complex Number Multiplication for 16 bits with Saturation (cont'd)

Example
Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp
Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c
Trilib\Example\GNU\CplxArith\expCplx.c

Cycle Count 9+2

Code Size 34 bytes

CplxConj_16 Complex Number Conjugate for 16 bits

Signature CplxS CplxConj_16(CplxS X);

Inputs X : 16 bit Complex input value

Output None

Return The conjugate of the complex number as a 16 bit complex number

Description This function finds the conjugate of a 16 bit complex number. Conjugate of a complex number is given by

$$\bar{R} = \overline{(x + iy)} = x - iy \qquad [4.16]$$

Pseudo code

```
{
   R.real = X.real;
   R.imag = 0.0 - X.imag;
                   //negate the imaginary part
   return R;
}
```

Techniques None

Assumptions

- Input and output has a real and an imaginary part packed as 16 bit data to make a 32 bit complex data

Memory Note

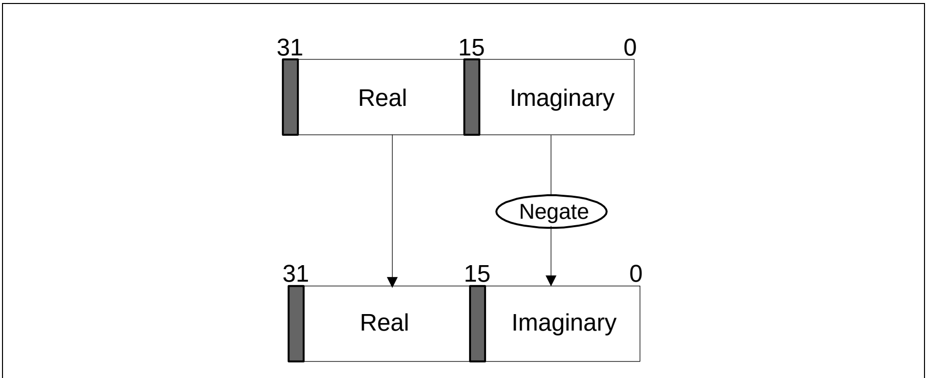


Figure 4-10 Complex number conjugate for 16 bits

CplxConj_16 Complex Number Conjugate for 16 bits (cont'd)

Example

Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp

Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c

Trilib\Example\GNU\CplxArith\expCplx.c

Cycle Count 3+2

Code Size 12 bytes

CplxMag_16 Magnitude of a Complex Number for 16 bits

Signature DataL CplxMag_16(CplxS X);

Inputs X : 16 bit Complex input value

Output None

Return Magnitude of the complex number as 32 bit integer or fract

Description This function finds the magnitude of a complex number. The algorithm is as follows

$$|R| = \sqrt{x^2 + y^2} \qquad [4.17]$$

Pseudo code

```
{
   int indx;
   frac32 sat tempX;
   frac32 sat tempY;
   frac32 sat temp;

   frac32 sqrttab[15] = {0.999999999999, 0.7071067811865, 0.5,
                             0.3535533905933, 0.25, 0.1767766952966,
                         0.125, 0.088388347648 32, 0.0625, 0.04419417382416,
                             0.03125, 0.02209708691208, 0.015625,
                             0.01104854345604, 0.0078125};

   //Scale down the input by 2
   X.real >>= 1;
   X.imag >>= 1;

   //Power = real^2 + imag^2
   tempX = (X.real * X.real);
   tempY = (X.imag * X.imag);
   tempX += tempY;
```

CplxMag_16 Magnitude of a Complex Number for 16 bits (cont'd)

```
    if (tempX == 0)
    {
       return tempX;
    }
    //Mag = sqrt(power);
    indx = exp1(tempX);//calculate the leading zero
    tempX = norm(tempX,indx);
                       //normalise
    tempY = tempX >> 1;//y = x/2
    tempY -= 0.5;      //y = x/2 - 0.5
    tempX = tempY + 0.9999999999999999;
                       //sqrt(x) = y + 1
    temp = (tempY * tempY);
                       // y^2
    tempX -= temp >> 1;//sqrt(x) = (y + 1) - 0.5*y^2
    temp =(temp*tempY);//y^3
    tempX += temp >> 1;//sqrt(x) = (y + 1) - 0.5*y^2 + 0.5*y^3
    temp = (temp * tempY);
                       //y^4
    tempX -= temp * 0.625;
                      //sqrt(x) = (y + 1) - 0.5*y^2 + 0.5*y^3 - 0.625*y^4
    temp = (temp * tempY);
                       //y^5

    tempX = tempX + (0.875 * temp);
                       //sqrt(x) = (y + 1) - 0.5*y^2 + 0.5*y^3
                       //           - 0.625*y^4 +0.875*y^5
    temp = tempX << 15;
    if (temp >= 0.5)
    {
        tempX >>= 16;
        tempX <<= 16;
        tempX += 0.0000305178125;
    }
    else
    {
       tempX >>=16;
       tempX <<=16;
    }
    tempX = tempX * sqrttab[indx];

    return tempX;
}
```

CplxMag_16 **Magnitude of a Complex Number for 16 bits** (cont'd)

Techniques None

Assumptions None

Memory Note None

Implementation

The real and imaginary parts of a complex number x+iy are scaled down by two to avoid overflow.
The computation of power(x^2+y^2) is done by a dual MAC instruction.
If the power is zero, then the whole computation is not done to save cycles. Power(x^2+y^2) is normalized and the exponent is used as the scale factor in the square root operation. The square root is computed using the taylor approximation series.
The taylor series for square root is as follows:
Let Z = x^2+y^2
R = (Z + 1)/2

$$\mathrm{sqrt}(Z) = R + 1 - 0.5R^2 + 0.5R^3 - 0.625R^4 - 0.875R^5 \qquad [4.18]$$

The final result sqrt(Z) is again rescaled using the scale factor as index of the square root table to give the magnitude.

Example

Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp

Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c

Trilib\Example\GNU\CplxArith\expCplxMag.c

Cycle Count

7+2	(Best)
7+42+2	(Worst)

Code Size

118 bytes

140 bytes (Data)

CplxPhase_16 Phase of a Complex Number for 16 bits

Signature DataL CplxPhase_16 (CplxS X);

Inputs X : 16 bit Complex input value

Output None

Return The phase of the input complex number as a 32 bit integer or fract

Description This function computes the phase of a complex number. The algorithm is as follows.

$$\text{Phase} = \tan^{-1}(y/x) \qquad [4.19]$$

Pseudo code

```
{
   int indx;
   int flag;
   frac32 sat tempX;
   frac32 sat tempY;
   frac32 sat temp;

   //Scale down the input by 2
   X.real >>= 1;
   X.imag >>= 1;

   //Power = real^2 + imag^2
   //Taking absolute value of input complex number
   if (X.real < 0)
   {
      tempX = -X.real;
   }
   else
   {
      tempX = X.real;
   }
```

CplxPhase_16 Phase of a Complex Number for 16 bits (cont'd)

```
if (X.imag < 0)
{
   tempY = -X.imag;
}
else
{
   tempY = X.imag;
}

//Phase = arctan(imag/real)
if (tempX <= tempY)
{
   flag = 1;
   temp = tempX/tempY;
}
else
{
   flag = 0;
   temp = tempY/tempX;
}
indx = exp1(temp); //calculate the leading zero
temp = norm(temp,indx);
                   //normalise
//Polynomial calculation
tempX = K5 * temp + K4;
tempX = tempX * temp + K3;
tempX = tempX * temp + K2;
tempX = tempX * temp + K1;
tempX = tempX * temp;
temp = tempX << 15;
```

CplxPhase_16 Phase of a Complex Number for 16 bits (cont'd)

```
    //if imag > real
    if (flag == 1)
    {
        tempX = 0.5 - tempX;
    }
    //third quadrant X = X - 180 deg
    if (X.real < 0 && X.imag < 0)
    {
       tempX = tempX - 0.9999999999999;
    }
    //second quadrant X = 180 - X deg
    else if (X.real < 0 && X.imag >= 0)
    {
       tempX = 0.9999999999999 - tempX;
    }
    //fourth quadrant X = - X
    else if (X.real >= 0 && X.imag < 0)
    {
       tempX = -tempX;
    }
    //Rounding
    if (temp >= 0.5)
    {
       tempX >>= 16;
       tempX <<= 16;
       tempX += 0.0000305178125;
    }
    else
    {
       tempX >>=16;
       tempX <<=16;
    }
    return tempX;
}
```

Techniques None

Assumptions None

Memory Note None

CplxPhase_16 Phase of a Complex Number for 16 bits (cont'd)

Implementation

The phase in a complex plane is the arctan(y/x), where y/x=z.

By Taylor series,

phase = $\tan^{-1}(z)$ for Z<=1 [4.20]

or $0.5-\tan^{-1}(1/z)$ for z>1 [4.21]

If $y \le x$, the flag is set to indicate that **Equation [4.20]** to be computed, otherwise **Equation [4.21]** is computed.

After calculating y/x, the results are normalized. Then the arctan is calculated by using the Taylor approximation series is a polynomial expansion. This is as follows:

$$\begin{aligned}\arctan(z) &= 0.318253z + 0.003314z^2 - 0.130908z^3 \\ &\quad + 0.068542z^4 - 0.009159z^5\end{aligned} \qquad [4.22]$$

The final part of the processing extracts the sign of real and imaginary part and branches to appropriate quadrant.
I quadrant : phase = arctan(y/x) radian
II quadrant : phase = π -arctan(y/x) radian
III quadrant: phase = arctan(y/x)-π radian
IV quadrant: phase = arctan(y/x) radian

The output of the function is given in radians and has to be scaled. The output is as follows
$+\pi$ = 0x7fff or 0.99999999
$-\pi$ = 0x8000 or -1.0
$\pi/2$ is approximately equal to 0.5
$-\pi/2$ is approximately equal to -0.5

Example

Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp
Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c
Trilib\Example\GNU\CplxArith\expCplxPh.c

CplxPhase_16 Phase of a Complex Number for 16 bits (cont'd)

Cycle Count	52+2	(Best)
	62+2	(Worst)
Code Size	180 bytes	
	20 bytes (Data)	

CplxShift_16 Complex Number Shift for 16 bits

Signature CplxS CplxShift_16(CplxS X,
int shiftVal
);

Inputs X : 16 bit Complex input value
shiftVal : shift value as a signed integer

Output None

Return Output value after the real and imaginary parts are shifted

Description This function performs shifting of a 16 bit complex number indicated by the *shiftVal*. Left shifting is done if the *shiftVal* is positive and Right shifting is done if *shiftVal* is negative.
The algorithm is as follows.

$$\begin{aligned} R_r &= x_r \gg \mathrm{abs}(\mathrm{shiftVal}),\ \mathrm{if}(\mathrm{shiftVal} < 0) \\ &\quad \mathrm{else}(x_r \ll \mathrm{shiftVal}) \\ R_i &= x_i \gg \mathrm{abs}(\mathrm{shiftVal}),\ \mathrm{if}(\mathrm{shiftVal} < 0) \\ &\quad \mathrm{else}(x_i \ll \mathrm{shiftVal}) \end{aligned} \qquad [4.23]$$

Pseudo code

```
{
   real.real = X.real << shiftVal;
   real.imag = X.imag << shiftVal;

   return real;
}
```

Techniques None

Assumptions None

CplxShift_16 Complex Number Shift for 16 bits (cont'd)

Memory Note

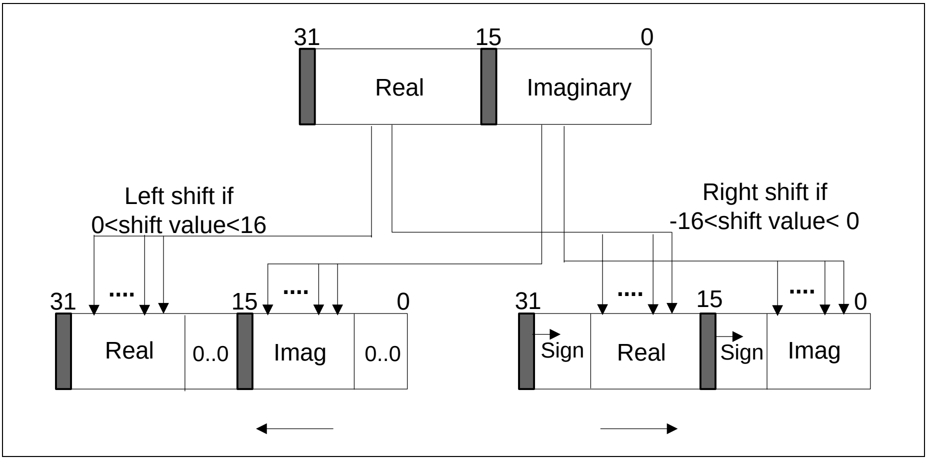


Figure 4-11 Complex number shift for 16 bits

Example *Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp*
Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c
Trilib\Example\GNU\CplxArith\expCplx.c

Cycle Count 1+2

Code Size 6 bytes

CplxAdd_32 Complex Number Addition for 32 bits

Signature

```
void CplxAdd_32(CplxL *X,
                CplxL *Y,
                CplxL *R
                );
```

Inputs

X : 32 bit Complex input value

Y : 32 bit Complex input value

Output

R : The sum of two complex numbers as a 32 bit complex number.

Return None

Description This function computes the sum of two 32 bit complex numbers. Wraps around the result in case of overflow.
The algorithm is as follows

$$R_r = x_r + y_r$$
$$R_i = x_i + y_i \qquad [4.24]$$

Pseudo code

```
{
   R->real = X->real + Y->real;
   R->imag = X->imag + Y->imag;
}
```

Techniques None

Assumptions

- Inputs are in 1Q31 format
- Input and output has a real and an imaginary part packed as 32 bit data in 1Q31 format to make a 64 bit complex data
- Inputs are doubleword aligned

CplxAdd_32 Complex Number Addition for 32 bits (cont'd)

Memory Note

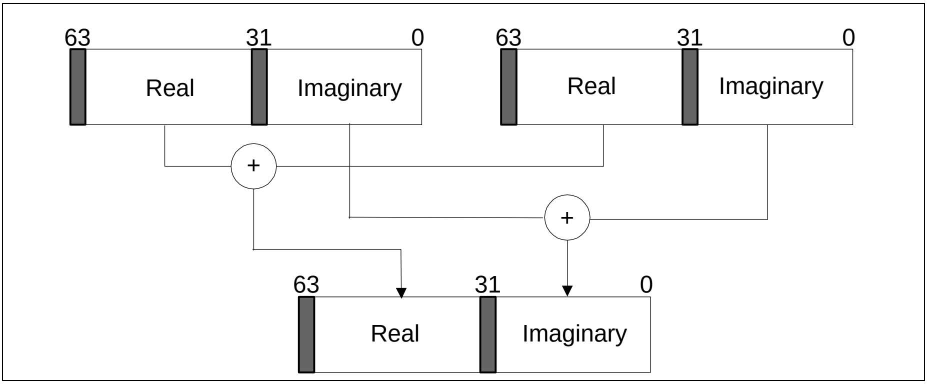


Figure 4-12 Complex number addition for 32 bits

Example *Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp*
Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c
Trilib\Example\GNU\CplxArith\expCplx.c

Cycle Count 4+2

Code Size 22 bytes

CplxAdds_32 Complex Number Addition for 32 bits with saturation

Signature

```
void CplxAdds_32(CplxL     *X,
                 CplxL     *Y,
                 CplxL_Sat *R
                 );
```

Inputs

X : 32 bit Complex input value

Y : 32 bit Complex input value

Output

R : The sum of two complex numbers as a 32 bit saturated complex number.

Return

None

Description

This function computes the sum of two 32 bit complex numbers. In case of underflow, this saturates the result to 0x7FFFFFFF for positive values and 0x80000000 for negative values.
Wraps around the result in case of overflow.

The algorithm is as follows

$$R_r = x_r + y_r$$
$$R_i = x_i + y_i \qquad [4.25]$$

Pseudo code

```
{
   R->real = (frac32 sat)(X->real + Y->real);
   R->imag = (frac32 sat)(X->imag + Y->imag);
}
```

Techniques

None

Assumptions

- Inputs are in 1Q31 format
- Input and output has a real and an imaginary part packed as 32 bit data in 1Q31 format to make a 64 bit complex data
- Inputs are doubleword aligned

CplxAdds_32 Complex Number Addition for 32 bits with saturation (cont'd)

Memory Note

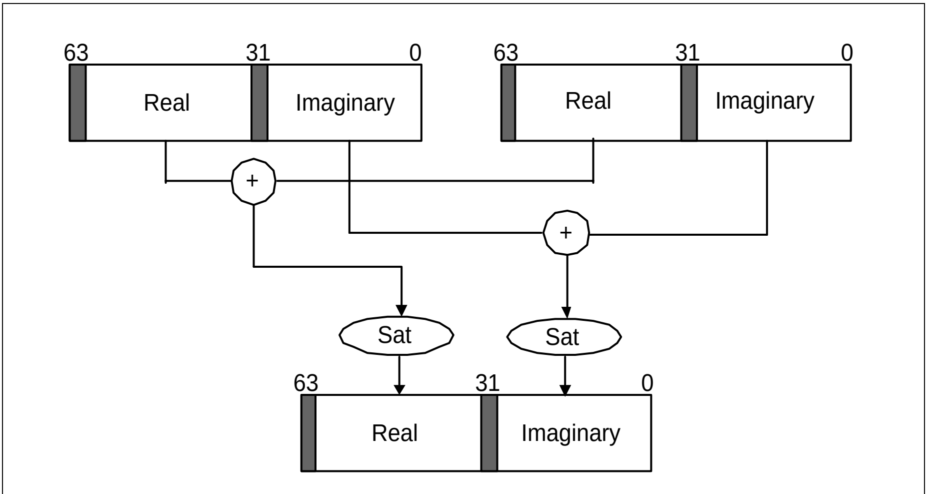


Figure 4-13 Complex number addition for 32 bits with saturation

Example	*Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp* *Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c* *Trilib\Example\GNU\CplxArith\expCplx.c*
Cycle Count	4+2
Code Size	22 bytes

CplxSub_32 Complex Number Subtraction for 32 bits

Signature

```
void CplxSub_32(CplxL *X,
                CplxL *Y,
                CplxL *R
               );
```

Inputs	X	:	32 bit Complex input value
	Y	:	32 bit Complex input value
Output	R	:	The difference of two complex numbers as a 32 bit complex number
Return	None		

Description

This function computes the difference of two 32 bit complex numbers. Wraps around the result in case of overflow.
The algorithm is as follows.

$$R_r = x_r - y_r$$
$$R_i = x_r - y_i \qquad [4.26]$$

Pseudo code

```
{
    R->real = X->real - Y->real;
    R->imag = X->imag - Y->imag;
}
```

Techniques None

Assumptions

- Inputs are in 1Q31 format
- Input and output has a real and an imaginary part packed as 32 bit data in 1Q31 format to make a 64 bit complex data
- Inputs are doubleword aligned

CplxSub_32 Complex Number Subtraction for 32 bits (cont'd)

Memory Note

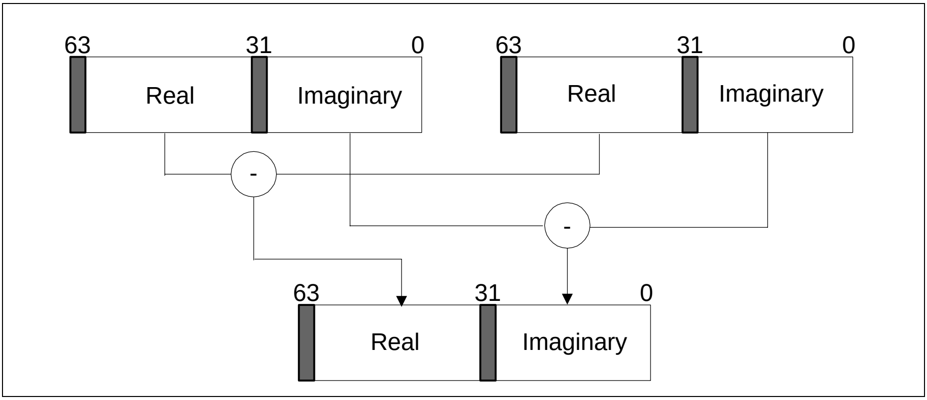


Figure 4-14 Complex number subtraction for 32 bits

Example	*Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp* *Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c* *Trilib\Example\GNU\CplxArith\expCplx.c*
Cycle Count	4+2
Code Size	22 bytes

CplxSubs_32 — Complex Number Subtraction for 32 bits with saturation

Signature

```
void CplxSubs_32(CplxL      *X,
                 CplxL      *Y,
                 CplxL_Sat  *R
                 );
```

Inputs

X : 32 bit Complex input value

Y : 32 bit Complex input value

Output

R : The difference of two complex numbers as a 32 bit saturated complex number

Return

None

Description

This function computes the difference of two 32 bit complex numbers. In case of underflow, this saturates the result to 0x7FFFFFFF for positive values and 0x80000000 for negative values. The algorithm is as follows.

$$
\begin{aligned}
R_r &= x_r - y_r \\
R_i &= x_r - y_i
\end{aligned}
\qquad [4.27]
$$

Pseudo code

```
{
   R->real = (frac32 sat)(X->real - Y->real);
   R->imag = (frac32 sat)(X->imag - Y->imag);
}
```

Techniques

None

Assumptions

- Inputs are in 1Q31 format
- Input and output has a real and an imaginary part packed as 32 bit data in 1Q31 format to make a 64 bit complex data
- Inputs are doubleword aligned

CplxSubs_32 — Complex Number Subtraction for 32 bits with saturation (cont'd)

Memory Note

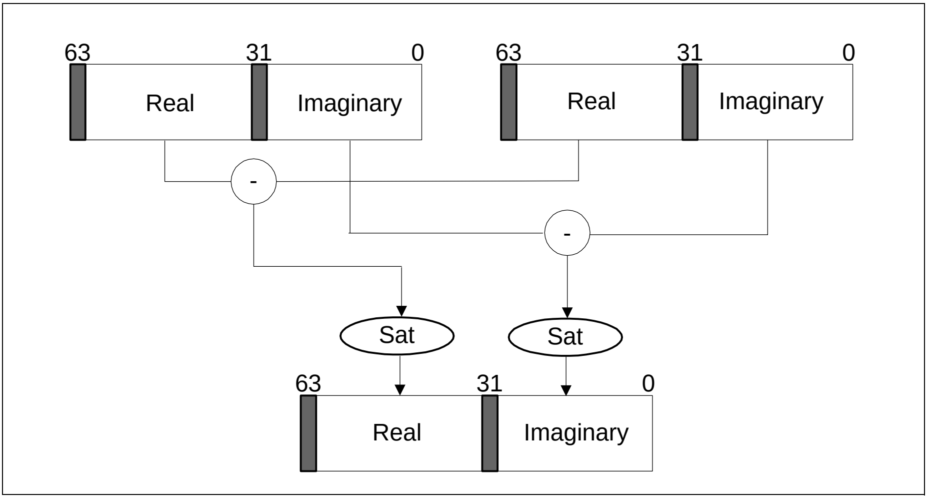


Figure 4-15 Complex number subtraction for 32 bits with saturation

Example	*Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp* *Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c* *Trilib\Example\GNU\CplxArith\expCplx.c*
Cycle Count	4+2
Code Size	22 bytes

CplxMul_32 Complex Number Multiplication for 32 bits

Signature

```
void CplxMul_32(CplxL *X,
                CplxL *Y,
                CplxL *R
                );
```

Inputs	X	:	32 bit Complex input value
	Y	:	32 bit Complex input value
Output	R	:	The product of two complex numbers as a 32 bit complex number
Return	None		

Description

This function computes the product of the two 32 bit complex numbers. Wraps around the result in case of overflow.

The complex multiplication is computed as follows.

$$R_r = x_r \times y_r - x_i \times y_i$$

$$R_i = x_i \times y_r + x_r \times y_i$$

Pseudo code

```
{
   frac64 real;
   frac64 ima;

   real = (frac64)((X->real * Y->real) - (X->imag * Y->imag));
                   //real part
   ima = (frac64)((X->real * Y->imag) + (X->imag * Y->real));
                   //imaginary part

   R->real = (frac32)real;
   R->imag = (frac32)ima;
}
```

Techniques None

Assumptions

- Inputs are in 1Q31 format
- Input and output has a real and an imaginary part packed as 32 bit data in 1Q31 format to make a 64 bit complex data
- Inputs are doubleword aligned

CplxMul_32 Complex Number Multiplication for 32 bits (cont'd)

Memory Note

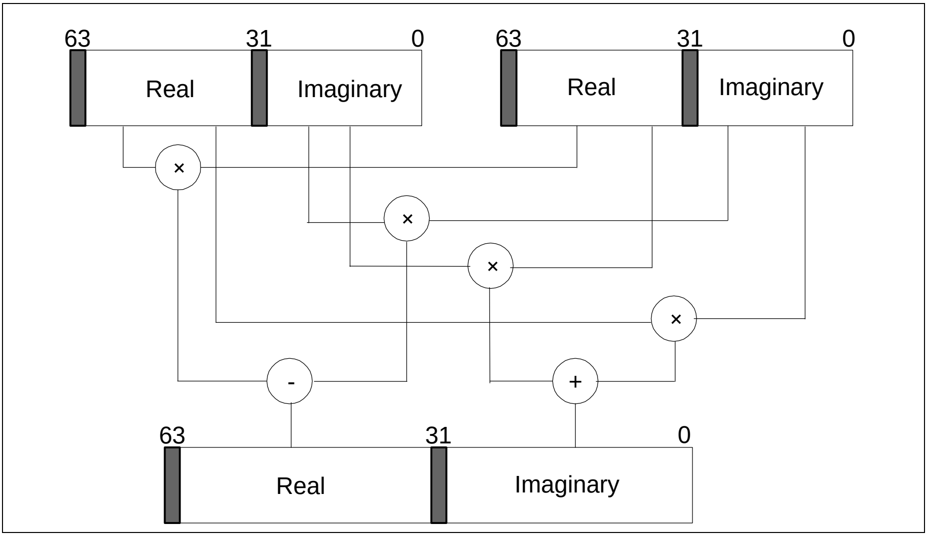


Figure 4-16 Complex number multiplication for 32 bits

Example *Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp*
Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c
Trilib\Example\GNU\CplxArith\expCplx.c

Cycle Count 13+2

Code Size 38 bytes

CplxMuls_32 Complex Number Multiplication for 32 bits with Saturation

Signature

```
void CplxMuls_32(CplxL      *X,
                 CplxL      *Y,
                 CplxL_Sat *R
                 );
```

Inputs	X	:	32 bit Complex input value
	Y	:	32 bit Complex input value
Output	R	:	The product of two complex numbers as a 32 bit complex number
Return	None		

Description

This function computes the product of the two 32 bit complex numbers. In case of overflow, the result is saturated to 0x7FFFFFFF for positive overflow and 0x80000000 for negative underflow.

The complex multiplication is computed as follows.

$$R_r = x_r \times y_r - x_i \times y_i$$

$$R_i = x_i \times y_r + x_r \times y_i$$

Pseudo code

```
{
   frac64  real;
   frac64  ima;

   real = (frac64)((X->real * Y->real) - (X->imag * Y->imag));
                   //real part
   ima = (frac64)((X->real * Y->imag) + (X->imag * Y->real));
                   //imaginary part

   R->real = (frac32 sat)real;
   R->imag = (frac32 sat)ima;
}
```

Techniques None

CplxMuls_32 Complex Number Multiplication for 32 bits with Saturation (cont'd)

Assumptions

- Inputs are in 1Q31 format
- Input and output has a real and an imaginary part packed as 32 bit data in 1Q31 format to make a 64 bit complex data
- Inputs are doubleword aligned

Memory Note

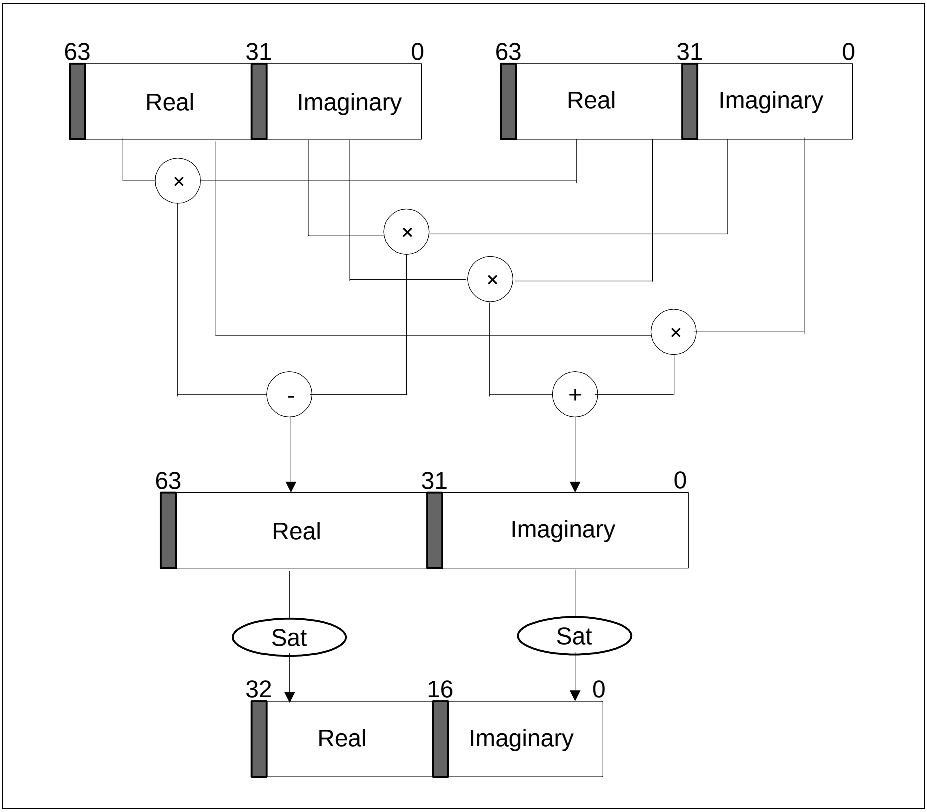


Figure 4-17 Complex number multiplication for 32 bits with saturation

CplxMuls_32 — Complex Number Multiplication for 32 bits with Saturation (cont'd)

Example

Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp

Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c

Trilib\Example\GNU\CplxArith\expCplx.c

Cycle Count 13+2

Code Size 38 bytes

CplxConj_32 Complex Number Conjugate for 32 bits

Signature void CplxConj_32(CplxL *X,
CplxL *R
);

Inputs X : 32 bit Complex input value

Output R : The conjugate of the complex number

Return None

Description This function finds the conjugate of a 32 bit complex number. Conjugate of a complex number is given by

$$\bar{R} = \overline{(x + iy)} = x - iy \qquad [4.28]$$

Pseudo code

```
{
   R->imag = 0.0 - X->imag;
   R->real = X->real;
}
```

Techniques None

Assumptions

- Input is in 1Q31 format
- Input and output has a real and an imaginary part packed as 32 bit data in 1Q31 format to make a 32 bit complex data
- Inputs are doubleword aligned

CplxConj_32 Complex Number Conjugate for 32 bits (cont'd)

Memory Note

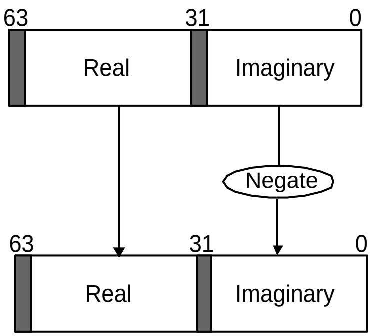


Figure 4-18 Complex number conjugate for 32 bits

Example	*Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp* *Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c* *Trilib\Example\GNU\CplxArith\expCplx.c*
Cycle Count	2+2
Code Size	14 bytes

CplxMag_32 Magnitude of a Complex Number for 32 bits

Signature DataL CplxMag_32(CplxL X);

Inputs X : 32 bit Complex input value

Output None

Return The magnitude of the complex number as a 32 bit integer or fract

Description This function finds the magnitude of a 32 bit complex number.

The algorithm is as follows

$$|R| = \sqrt{x^2 + y^2} \qquad [4.29]$$

Pseudo code

```
{
   int indx;
   frac32 sat tempX;
   frac32 sat tempY;
   frac32 sat temp;
   frac32 sat sqrttab[15] = {0.999999999999, 0.7071067811865, 0.5,
                     0.3535533905933, 0.25, 0.1767766952966, 0.125,
                        0.08838834764832, 0.0625, 0.04419417382416,
                         0.03125, 0.02209708691208, 0.015625,
                         0.01104854345604, 0.0078125};
   //Scale down the input by 2
   X->real >>= 1;
   X->imag >>= 1;

   //Power = real^2 + imag^2
   tempX = (X->real * X->real);
   tempY = (X->imag * X->imag);
   tempX += tempY;

   //Mag = sqrt(power);
   indx = exp1(tempX);//calculate the leading zero
   tempX = norm(tempX,indx);
                     //normalise
   tempY = tempX >> 1;//y = x/2
   tempY -= 0.5;      //y = x/2 - 0.5
   tempX = tempY + 0.9999999999999999;
                     //sqrt(x) = y + 1
```

CplxMag_32 Magnitude of a Complex Number for 32 bits (cont'd)

```
    temp = (tempY * tempY);
                       //y^2
    tempX -= temp >> 1;//sqrt(x) = (y + 1) - 0.5*y^2
    temp= (temp*tempY);//y^3
    tempX += temp >> 1;//sqrt(x) = (y + 1) - 0.5*y^2 + 0.5*y^3
    temp = (temp * tempY);
                       //y^4
    tempX -= temp * 0.625;
                     //sqrt(x) = (y + 1) - 0.5*y^2 + 0.5*y^3 - 0.625*y^4
    temp = (temp * tempY);
                       //y^5
    tempX = tempX + (0.875 * temp);
                       //sqrt(x) = (y + 1) - 0.5*y^2 + 0.5*y^3
                       //           - 0.625*y^4 +0.875*y^5
    tempX = tempX * sqrttab[indx];
    return tempX;
}
```

Techniques None

Assumptions
- Inputs are doubleword aligned

Memory Note None

CplxMag_32 — Magnitude of a Complex Number for 32 bits (cont'd)

Implementation

The real and imaginary parts of a complex number x+iy are scaled down by two to avoid overflow.

MAC is used to square the imaginary part and dual MAC is used to square the real part. Add these to give the power(x^2+y^2).

If the power is zero, then the whole computation is not done to save cycles. Power(x^2+y^2) is normalized and the exponent is used as the scale factor in the square root operation. The square root is computed using the taylor approximation series.

The taylor series for square root is as follows:
Let $Z = x^2+y^2$
$R = (Z + 1)/2$

$$\mathrm{sqrt}(Z) = R + 1 - 0.5R^2 + 0.5R^3 - 0.625R^4 - 0.875R^5 \qquad [4.30]$$

The final result sqrt(Z) is again rescaled using the scale factor as index of the square root table to give the magnitude.

Example

Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp

Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c

Trilib\Example\GNU\CplxArith\expCplxMag.c

Cycle Count

52	(Best)
62	(Worst)

Code Size

126 bytes

140 bytes (Data)

CplxPhase_32 Phase of a Complex Number for 32 bits

Signature DataL CplxPhase_32(CplxL *X);

Inputs X : 32 bit Complex input value

Output None

Return The phase of a complex number as a 32 bit integer or fract

Description This function computes the phase of a complex number. The algorithm is as follows.

$$\text{Phase} = \tan^{-1}(y/x) \quad [4.31]$$

Pseudo code

```
{
   int indx;
   int flag;
   frac32 sat tempX;
   frac32 sat tempY;
   frac32 sat temp;

   //Scale down the input by 2
   X->real >>= 1;
   X->imag >>= 1;

   //Power = real^2 + imag^2
   if (X->real < 0)
   {
      tempX = -X->real;
   }
   else
   {
      tempX = X->real;
   }
   if (X->imag < 0)
   {
      tempY = -X->imag;
   }
   else
   {
      tempY = X->imag;
   }
```

CplxPhase_32 Phase of a Complex Number for 32 bits (cont'd)

```
    //Phase = arctan(imag/real)
    if (tempX <= tempY)
    {
       flag = 1;
       temp = tempX/tempY;
    }
    else
    {
       flag = 0;
       temp = tempY/tempX;
    }

    indx = exp1(temp); //calculate the leading zero
    temp = norm(temp,indx);
                      //normalise
    tempX = K5 * temp + K4;
    tempX = tempX * temp + K3;
    tempX = tempX * temp + K2;
    tempX = tempX * temp + K1;
    tempX = tempX * temp;

    if (flag == 1)
    {
       tempX = 0.5 - tempX;
    }

    if (X->real < 0 && X->imag < 0)
    {
       tempX = tempX - 0.9999999999999;
    }
    else if (X->real < 0 && X->imag >= 0)
    {
       tempX = 0.9999999999999 - tempX;
    }
    else if (X->real >= 0 && X->imag < 0)
    {
       tempX = -tempX;
    }

    return tempX;
}
```

CplxPhase_32 Phase of a Complex Number for 32 bits (cont'd)

Techniques None

Assumptions
- Inputs are doubleword aligned

Memory Note None

Implementation The phase in a complex plane is the arctan(y/x), where y/x=z.

By Taylor series,

phase = $\tan^{-1}$(z) for Z<=1 [4.32]

or 0.5-$\tan^{-1}$(1/z) for z>1. [4.33]

If $y \le x$, the flag is set to indicate that **Equation [4.32]** to be computed, otherwise **Equation [4.33]** is computed.

After calculating y/x, the results are normalized. Then the arctan is calculated by using the Taylor approximation series is a polynomial expansion. This is as follows:

$$\begin{aligned} \arctan(z) &= 0.318253z + 0.003314z^2 - 0.130908z^3 \\ &\quad + 0.068542z^4 - 0.009159z^5 \end{aligned} \quad [4.34]$$

The final part of the processing extracts the sign of real and imaginary part and branches to appropriate quadrant.
I quadrant : phase = arctan(y/x) radian
II quadrant : phase = π -arctan(y/x) radian
III quadrant: phase = arctan(y/x)-π radian
IV quadrant: phase = arctan(y/x) radian

The output of the function is given in radians and has to be scaled. The output is as follows
$+\pi$ = 0x7fffffff or 0.99999999
$-\pi$ = 0x80000000 or -1.0
$\pi/2$ is approximately equal to 0.5
$-\pi/2$ is approximately equal to -0.5

CplxPhase_32 — Phase of a Complex Number for 32 bits (cont'd)

Example	*Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp* *Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c* *Trilib\Example\GNU\CplxArith\expCplxPh.c*	
Cycle Count	7 7+44	(Best) (Worst)
Code Size	180 bytes 20 bytes (Data)	

CplxShift_32 Complex Number Shift for 32 bits

Signature
```
void CplxShift_32(CplxL *X,
                  CplxL *R,
                  int    shiftVal
                  );
```

Inputs

X : 32 bit Complex input value

shiftVal : shift value as a signed integer

Output

R : Output value after the real and imaginary parts are shifted

Return

None

Description

This function performs shifting of a 32 bit complex number indicated by the *shiftVal*. Left shifting is done if the *shiftVal* is positive and Right shifting is done if *shiftVal* is negative.

The algorithm is as follows.

$$
\begin{aligned}
R_r &= x_r \gg \mathrm{abs}(\mathrm{shiftVal}),\ \mathrm{if}(\mathrm{shiftVal} < 0) \\
&\quad \mathrm{else}(x_r \ll \mathrm{shiftVal}) \\
R_i &= x_i \gg \mathrm{abs}(\mathrm{shiftVal}),\ \mathrm{if}(\mathrm{shiftVal} < 0) \\
&\quad \mathrm{else}(x_i \ll \mathrm{shiftVal})
\end{aligned}
\qquad [4.35]
$$

Pseudo code
```
{
   if (Y < 0)
   {
      R->real = X->real >> Y;
      R->imag = X->imag >> Y;
   }
   else if (Y > 0)
   {
      R->real = X->real << Y;
      R->imag = X->imag << Y;
   }
   else
   {
      R->real = X->real;
      R->imag = X->imag;
   }
}
```

Techniques

None

CplxShift_32 Complex Number Shift for 32 bits (cont'd)

Assumptions
- Inputs are doubleword aligned

Memory Note

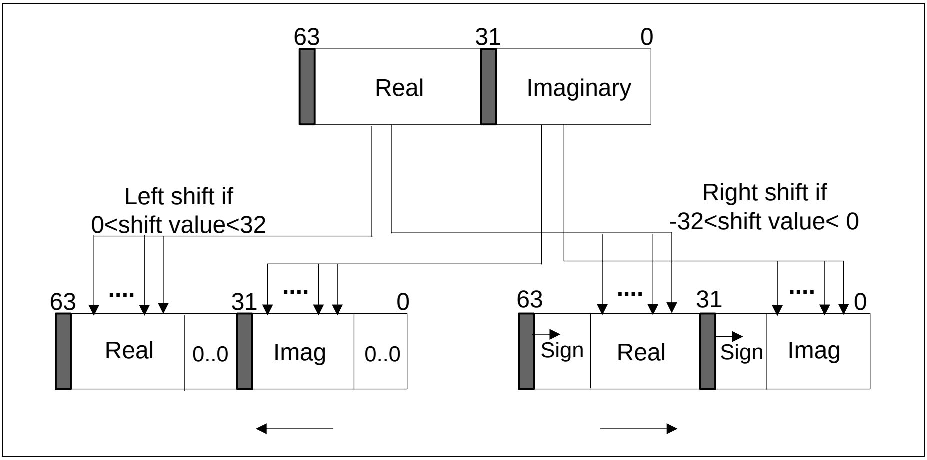


Figure 4-19 Complex number shift for 32 bits

Example *Trilib\Example\Tasking\CplxArith\expCplx.c, expCplx.cpp*
Trilib\Example\GreenHills\CplxArith\expCplx.cpp, expCplx.c
Trilib\Example\GNU\CplxArith\expCplx.c

Cycle Count 3+2

Code Size 18 bytes

4.3 Vector Arithmetic Functions

A vector is a quantity that has both magnitude and direction. Many physical quantities are vectors, e.g., force, velocity and momentum. In order to compare vectors and to operate on them mathematically, it is necessary to have some reference system that determines scale and direction, such as Cartesian coordinates. A vector is frequently symbolized by its components with respect to the coordinate axis. The concept of a vector can be extended to three or more dimensions.

4.3.1 Descriptions

The following vector arithmetic functions are described.

- Vector addition with saturation
- Vector subtraction with saturation
- Vector Dot product
- Maximum element by index
- Minimum element by index
- Maximum element by value
- Minimum element by value

VecAdd — Vector Operation - Addition of two vectors

Signature

```
int VecAdd(DataS      *X,
           DataS      * Y,
           DataS_Sat *R,
           int        nX
           );
```

Inputs

X : Pointer to first vector components

Y : Pointer to second vector components

nX : Dimension of vector

Output

R : Pointer to the sum of two vectors

Return

None

Description

This function finds the sum of two vectors.

If x and y are two vectors given by $x = [x_0, x_1, \ldots. x_{N-1}]^T$ and $y = [y_0, y_1, \ldots, y_{N-1}]^T$, their sum is given by

$$R_i = x_i + y_i \quad (i = 0,1,\ldots, N-1) \qquad [4.36]$$

Pseudo code

```
{
   int i;
   for (i = 0;i < nX;i++)
   {
      R[i] = X[i] + Y[i];
                        //Add
   }
}
```

Techniques

None

Assumptions

- The input vectors have the same dimension

VecAdd Vector Operation - Addition of two vectors (cont'd)

Memory Note

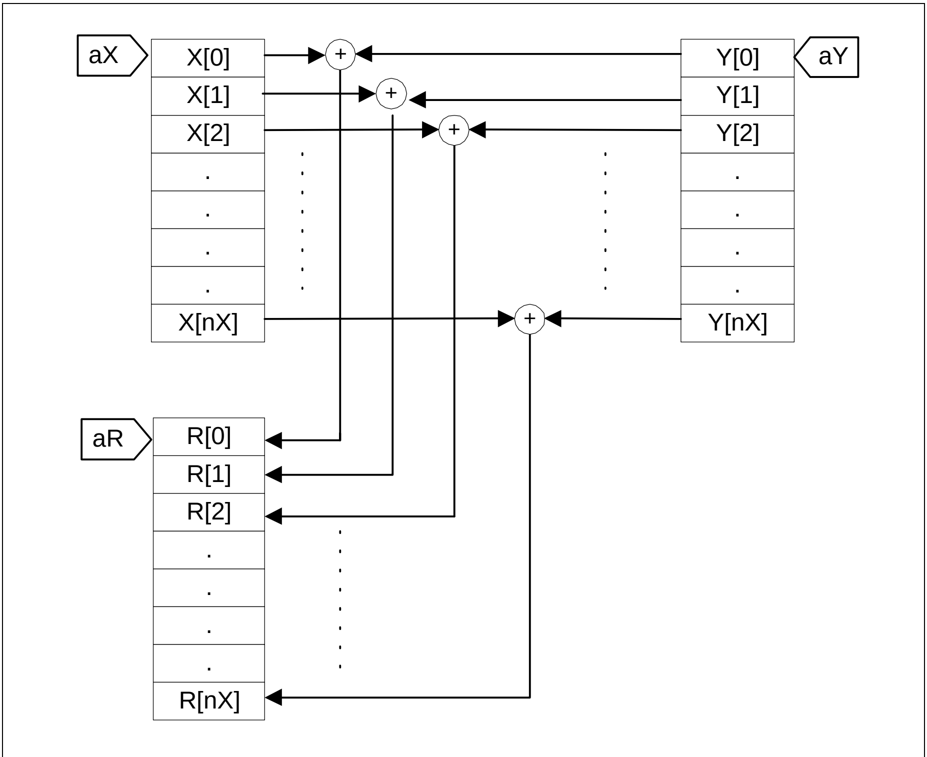


Figure 4-20 Vector Addition

VecAdd — Vector Operation - Addition of two vectors (cont'd)

Implementation

The Vector Add function adds with saturation the peer elements of two arrays and stores the result in the resultant array. It uses the packed Load/Store instruction to load 4 words of data simultaneously. It adds the 4 elements in one go and stores it into the result array. This is applicable for all the arrays with sizes equal to the multiples of 4 words. In case if the size is of odd or not the multiple of 4 words, it checks the remaining elements and correspondingly takes respective paths to execute the addition separately from the remaining words which is left out.

Example

Trilib\Example\Tasking\Vectors\expVect.c, expVect.cpp

Trilib\Example\GreenHills\Vectors\expVect.cpp, expVect.c

Trilib\Example\GNU\Vectors\expVect.c

Cycle Count

$\left\lceil 7 + 5 \times \frac{nX}{4} \right\rceil + 4 + 2$ (Best)

$\left\lceil 7 + 5 \times \frac{nX}{4} \right\rceil + 8 + 2$ (Worst)

Code Size

84 bytes

VecSub — Vector Operation - Difference of two vectors

Signature

```
int VecSub(DataS      *X,
           DataS      *Y,
           DataS_Sat  *R,
           int        nX
           );
```

Inputs

X : Pointer to first vector components

Y : Pointer to second vector components

nX : Dimension of vector

Output

R : Pointer to difference of two vectors

Return

None

Description

This function finds the difference of two vectors.

If x and y are two vectors given by $x = [x_0, x_1,....x_{N-1}]^T$ and $y = [y_0, y_1,...,y_{N-1}]^T$, their sum is given by

$$R_i = x_i - y_i \quad (i = 0,1,..., N-1) \qquad [4.37]$$

Pseudo code

```
{
   int i;
   for (i = 0;i < nX;i++)
   {
      R[i] = X[i] - Y[i];
                          //Subtract
   }
}
```

Techniques

None

Assumptions

- The input vectors have the same dimension

VecSub **Vector Operation - Difference of two vectors** (cont'd)

Memory Note

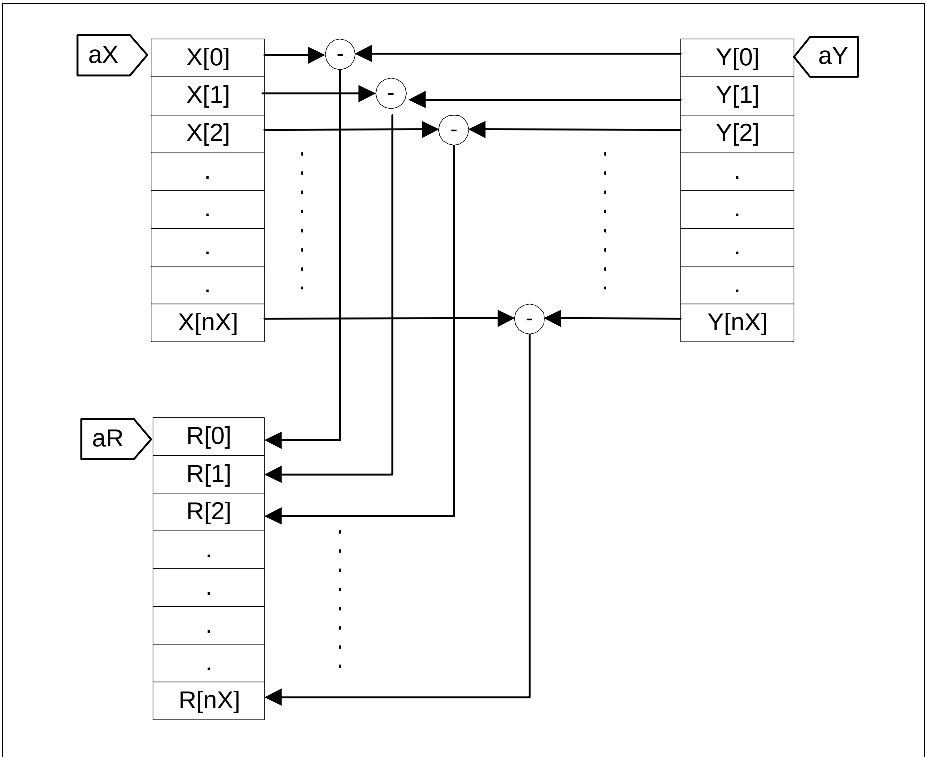


Figure 4-21 Vector Subtraction

VecSub **Vector Operation - Difference of two vectors** (cont'd)

Implementation

The Vector Subtract function subtracts with saturation the X array data by the corresponding peer element of Y array and stores the result in the resultant array. It uses the packed Load/Store instruction to load 4 words of data simultaneously. It adds the 4 elements in one go and stores it into the result array. This is applicable for all the arrays with sizes equal to the multiples of 4 words. In case if the size is of odd or not the multiple of 4 words, it checks the remaining elements and correspondingly takes respective paths to execute the subtraction separately from the remaining words which is left out.

Example

Trilib\Example\Tasking\Vectors\expVect.c, expVect.cpp

Trilib\Example\GreenHills\Vectors\expVect.cpp, expVect.c

Trilib\Example\GNU\Vectors\expVect.c

Cycle Count

$$\left\lceil 7 + 5 \times \frac{nX}{4} \right\rceil + 4 + 2$$ (Best)

$$\left\lceil 7 + 5 \times \frac{nX}{4} \right\rceil + 8 + 2$$ (Worst)

Code Size

84 bytes

VecDotPro — Vector Operation - Dot Product of two vectors

Signature

```
DataL VecDotPro(DataS   *X,
                DataS   *Y,
                int     nX
                );
```

Inputs

X : Pointer to first vector components

Y : Pointer to second vector components

nX : Dimension of vectors

Output

None

Return

Dot product of the two vectors (48-bit output value converted to 32-bit with saturation)

Description

If x and y are two vectors of dimension N, their dot product is given by

$$x \cdot y = \sum_{i=0}^{N-1} x_i \cdot y_i = x_0 \cdot y_0 + x_1 \cdot y_1 + \ldots + x_{N-1} \cdot y_{N-1} \quad [4.38]$$

Pseudo code

```
{
   int i;
   frac64 product = 0;

   for(i = 0;i < nX;i++)
   {
      product += (frac64) X[i](*)Y[i];
   }                    //calculating the dot product
   return(frac32 sat)product;
                        //Format the result to 32-bit saturated value
}
```

Techniques

- Use of MAC instructions
- Intermediate results stored in a 64 bit register (16 guard bits)
- Dot product output is converted to 16 bit with saturation
- Instruction ordering provided for zero overhead Load/Store

Assumptions

- The input vectors have the same dimension

VecDotPro **Vector Operation - Dot Product of two vectors** (cont'd)

Memory Note

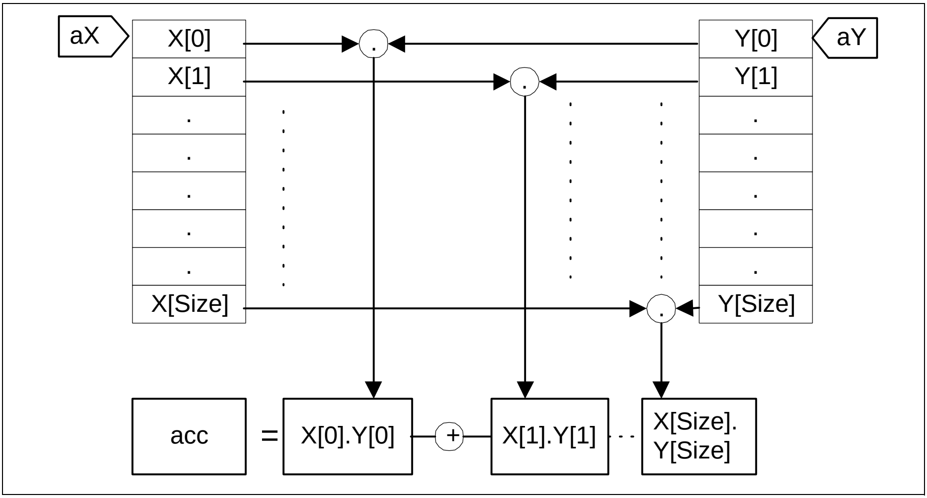


Figure 4-22 Dot product of two vectors

Implementation The Vector Dot Product function multiplies and accumulates the X array data by the corresponding peer element of Y array. It uses the `madd.q` instruction to do the multiply and accumulate the input data, the final result which is in 17Q47 format in a 64 bit register is converted to a 32 bit result and is saturated.

Example *Trilib\Example\Tasking\Vectors\expVect.c, expVect.cpp*

Trilib\Example\GreenHills\Vectors\expVect.cpp, expVect.c

Trilib\Example\GNU\Vectors\expVect.c

Cycle Count $5 + 2 \times [nX - 1] + 5$

Code Size 52 bytes

VecMaxIdx — Vector Operation - Maximum Element by Index of a vector

Signature

```
int VecMaxIdx(DataS   *X,
              int     nX
              );
```

Inputs

X	:	Pointer to the vector components
nX	:	Dimension of vector

Output None

Return The maximum element by index of the input vector

Description This function calculates the maximum element by index of a vector. The input vector components are 16 bit real values.

Pseudo code

```
{
   frac16 element = -1.0;
   int i;

   for (i = 0;i < nX;i++)
   {
      if (element < X[i])
      {
          element = X[i];
      }
   }
   i = 0;
   while (element != X[i])
   {
      i++;
   }

   return i;
}
```

Techniques None

Assumptions

- Inputs are in 1Q15 format

VecMaxIdx — Vector Operation - Maximum Element by Index of a vector (cont'd)

Memory Note

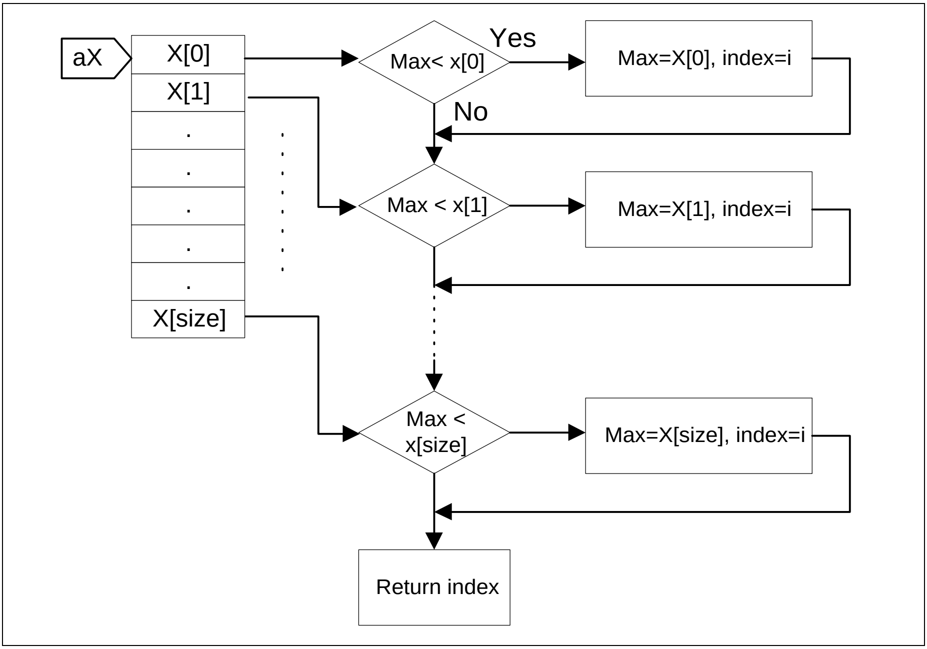


Figure 4-23 Maximum element by index

VecMaxIdx **Vector Operation - Maximum Element by Index of a vector** (cont'd)

Implementation

The Vector Maximum by Index function uses the `max.h` and `eq.h` instructions to optimally find the maximum value in the array. The `max.h` instruction checks the two 32 bit registers and returns the bigger 2 words among them into another register thereby does two comparison and movement of data in one go. Similarly the `eq.h` checks if the value is equal among the two registers, this is used here to find the greater value between the two words of a same 32 bit register finally, which is found to be in the maximum pair register after the computation of maximum element. Since the `max.h` does two comparisons, the loop count is reduced by half.
The final part of the function is to calculate the index of the maximum element, this is done by initializing a index variable and is kept on incrementing until the maximum element found matches with one of the array's element, odd array size is separately taken care.

Example

Trilib\Example\Tasking\Vectors\expVect.c, expVect.cpp
Trilib\Example\GreenHills\Vectors\expVect.cpp, expVect.c
Trilib\Example\GNU\Vectors\expVect.c

Cycle Count

$$4 + \left[2 \times \frac{nX}{4} + 1\right] + 3 + \left(2 \times \frac{1}{2}\right) + 2 \qquad \text{(Best)}$$

$$4 + \left[2 \times \frac{nX}{4} + 1\right] + 3 + \left(2 \times \frac{nX}{2}\right) + 2 \qquad \text{(Worst)}$$

Code Size

92 bytes

VecMinIdx — Vector Operation - Minimum Element by index of a vector

Signature

```
int VecMinIdx(DataS   *X,
              int      nX
              );
```

Inputs

X : Pointer to vector components

nX : Dimension of vector

Output None

Return The minimum element by index of the input vector

Description This function calculates the minimum element by index of a vector. The input vector components are 16 bit real values and are halfword aligned.

Pseudo code

```
{
   frac16 element = 0.99999999999999;
   int i;

   for (i = 0;i < nX;i++)
   {
      if (element > X[i])
      {
          element = X[i];
      }
   }
   i = 0;
   while (element != X[i])
   {
      i++;
   }

   return i;
}
```

Techniques None

Assumptions None

VecMinIdx — Vector Operation - Minimum Element by index of a vector (cont'd)

Memory Note

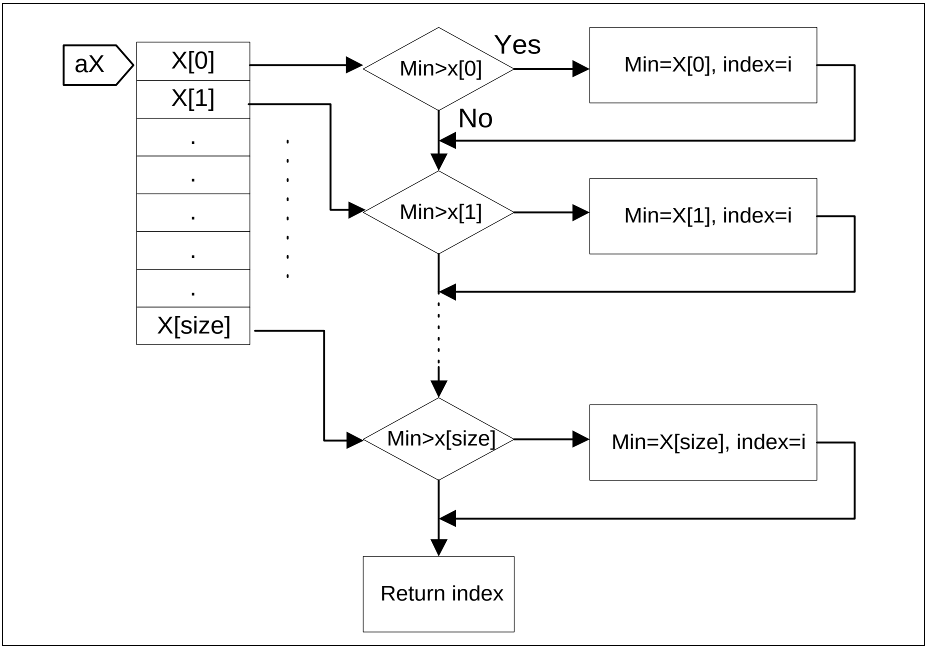


Figure 4-24 Minimum element by index

VecMinIdx **Vector Operation - Minimum Element by index of a vector** (cont'd)

Implementation

The Vector Minimum by Index function uses the `min.h` and `eq.h` instructions to optimally find the minimum value in the array. The `min.h` instruction checks the two 32 bit registers and returns the smaller 2 words among them into another register thereby does two comparison and movement of data in one go. Similarly the `eq.h` checks if the value is equal among the two registers, this is used here to find the smaller value between the two words of a same 32 bit register finally, which is found to be in the minimum pair register after the computation of minimum element. Since the `min.h` does two comparisons, the loop count is reduced by half.
The final part of the function is to calculate the index of the minimum element, this is done by initializing a index variable and is kept on incrementing until the minimum element found matches with one of the array's element, odd array size is separately taken care.

Example

Trilib\Example\Tasking\Vectors\expVect.c, expVect.cpp
Trilib\Example\GreenHills\Vectors\expVect.cpp, expVect.c
Trilib\Example\GNU\Vectors\expVect.c

Cycle Count

$$4 + \left[2 \times \frac{nX}{4} + 1\right] + 3 + \left(2 \times \frac{1}{2}\right) + 2 \qquad \text{(Best)}$$

$$4 + \left[2 \times \frac{nX}{4} + 1\right] + 3 + \left(2 \times \frac{nX}{2}\right) + 2 \qquad \text{(Worst)}$$

Code Size

98 bytes

VecMaxVal — Vector Operation - Maximum Element by value of a vector

Signature	int VecMaxVal(DataS *X, int nX);
Inputs	X : Pointer to vector components nX : Dimension of vector
Output	None
Return	The maximum element by value of the input vector
Description	This function calculates the maximum element by value of a vector. The input vector components are 16 bit real values and are halfword aligned.

Pseudo code

```
{
   frac16 element = -1.0;
   int i;

   for (i = 0;i < nX ;i++)
   {
      if (element < X[i])
      {
         element = X[i];
      }
   }
   return element;
}
```

Techniques	None
Assumptions	None

VecMaxVal **Vector Operation - Maximum Element by value of a vector** (cont'd)

Memory Note

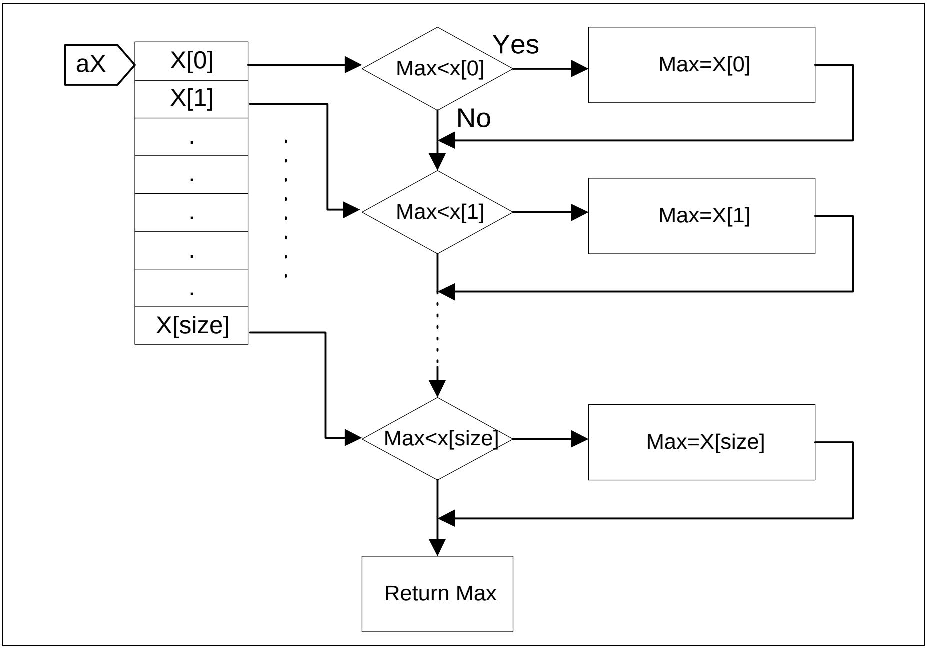


Figure 4-25 Maximum element by value

VecMaxVal **Vector Operation - Maximum Element by value of a vector** (cont'd)

Implementation The Vector Maximum by value function uses the `max.h` and `eq.h` instructions to optimally find the maximum value in the array. The `max.h` instruction checks the two 32 bit registers and returns the bigger 2 words among them into another register thereby does two comparison and movement of data in one go. Similarly the `eq.h` checks if the value is equal among the two registers, this is used here to find the greater value between the two words of a same 32 bit register finally, which is found to be in the maximum pair register after the computation of maximum element. Since the `max.h` does two comparisons, the loop count is reduced by half. It returns the maximum value among the two in the maximum element register.

Example *Trilib\Example\Tasking\Vectors\expVect.c, expVect.cpp*
Trilib\Example\GreenHills\Vectors\expVect.cpp, expVect.c
Trilib\Example\GNU\Vectors\expVect.c

Cycle Count

$$3 + \left[2 \times \frac{nX}{4} + 1\right] + 5$$ (Best)

$$3 + \left[2 \times \frac{nX}{4} + 1\right] + 7$$ (Worst)

Code Size 56 bytes

VecMinVal	**Vector Operation - Minimum Element by value of a vector**
Signature	int VecMinVal(DataS *X, int nX);
Inputs	X : Pointer to vector components nX : Dimension of vector
Output	None
Return	The minimum element by value of the input vector
Description	This function calculates the minimum element by value of a vector. The input vector components are 16 bit real values and are halfword aligned.

Pseudo code

```
{
   frac16 element = 0.999999999;
   int i;

   for (i = 0;i < nX;i++)
   {
      if (element > X[i])
      {
         element = X[i];
      }
   }
   return element;
}
```

Techniques	None
Assumptions	None

VecMinVal **Vector Operation - Minimum Element by value of a vector** (cont'd)

Memory Note

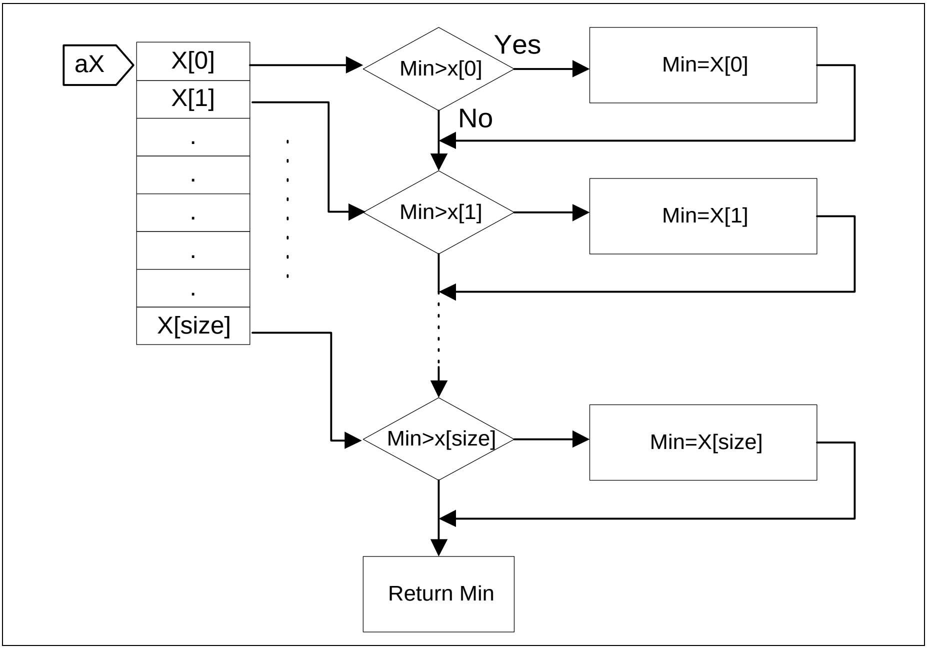


Figure 4-26 Minimum element by value

VecMinVal — Vector Operation - Minimum Element by value of a vector (cont'd)

Implementation

The Vector Minimum by value function uses the `min.h` and `eq.h` instructions to optimally find the minimum value in the array. The `min.h` instruction checks the two 32 bit registers and returns the smaller 2 words among them into another register thereby does two comparison and movement of data in one go. Similarly the `eq.h` checks if the value is equal among the two registers, this is used here to find the smaller value between the two words of a same 32 bit register finally, which is found to be in the minimum pair register after the computation of minimum element. Since the `min.h` does two comparisons, the loop count is reduced by half. It returns the minimum value among the two in the minimum element register.

Example

Trilib\Example\Tasking\Vectors\expVect.c, expVect.cpp
Trilib\Example\GreenHills\Vectors\expVect.cpp, expVect.c
Trilib\Example\GNU\Vectors\expVect.c

Cycle Count

$$3 + \left[2 \times \frac{nX}{4} + 1\right] + 5$$ (Best)

$$3 + \left[2 \times \frac{nX}{4} + 1\right] + 7$$ (Worst)

Code Size

56 bytes

4.4 FIR Filters

4.4.1 Normal FIR

The FIR (Finite Impulse Response) filter, as its name suggests, will always have a finite duration of non-zero output values for given finite duration of non-zero input values. FIR filters use only current and past input samples, and none of the filter's previous output samples, to obtain a current output sample value.

For causal FIR systems, the system function has only zeros (except for poles at z=0). The FIR filter can be realized in transversal, cascade and lattice forms. The implemented structure is of transversal type, which is realized by a tapped delay line. In case of FIR, delay line stores the past input values. The input x(n) for the current calculation will become x(n-1) for the next calculation. The output from each tap is summed to generate the filter output. For a general nH tap FIR filter, the difference equation is

$$R(n) = \sum_{i=0}^{nH-1} H_i \cdot X(n-i) \qquad [4.39]$$

where,

X(n)	:	the filter input for nth sample
R(n)	:	output of the filter for nth sample
H_i	:	filter coefficients
nH	:	filter order

The filter coefficients, which decide the scaling of current and past input samples stored in the delay line, define the filter response.

The transfer function of the filter in Z-transform is

$$H[z] = \frac{R[z]}{X[z]} = \sum_{i=0}^{nH-1} H_i \cdot Z^{-i} \qquad [4.40]$$

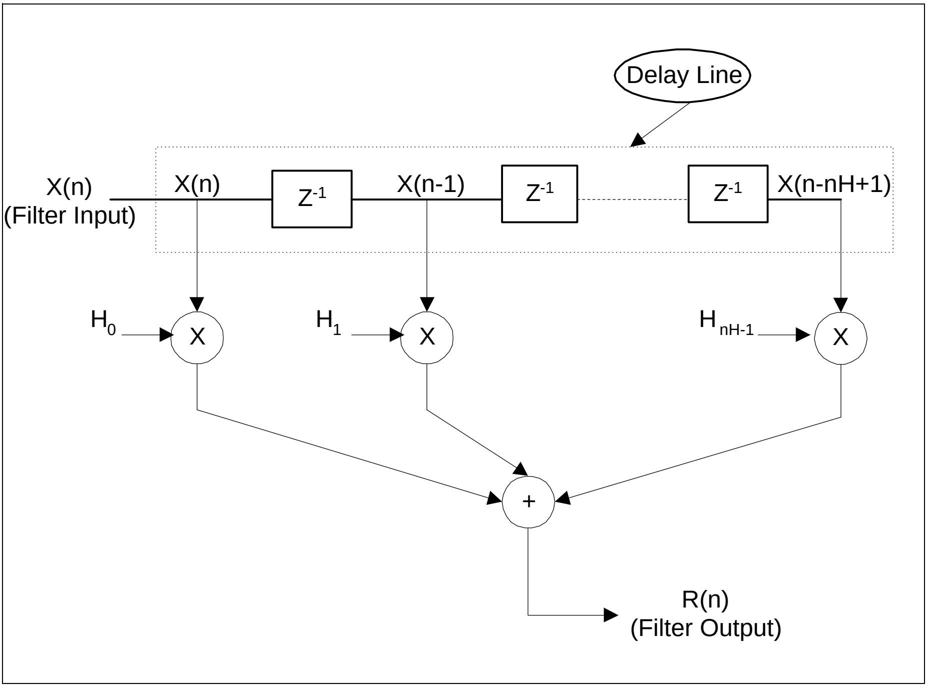


Figure 4-27 Block Diagram of the FIR Filter

4.4.1.1 Descriptions

The following Normal FIR filter functions are described.

- Normal, Arbitrary number of coefficients, Sample processing
- Normal, Arbitrary number of coefficients, Block processing
- Normal, coefficients - multiple of 4, Sample processing
- Normal, coefficients - multiple of 4, Block processing

Fir_16 — FIR Filter, Normal, Arbitrary number of coefficients, Sample processing

Signature

```
DataS Fir_16(DataS      X,
             DataS      *H,
             cptrDataS  *DLY
             );
```

Inputs

X	:	Real input value
H	:	Pointer to Coeff-Buffer of size nH
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct

Output

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer

Return

R	:	Output value of the filter (48-bit value converted to 16-bit with saturation)

Description

The implementation of FIR filter uses transversal structure (direct form). A single input is processed at a time and output for every sample is returned. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients given by the user is arbitrary. Circular buffer addressing mode is used for delay line. Coefficient buffer is halfword aligned. Delay line buffer is doubleword aligned.

Fir_16 FIR Filter, Normal, Arbitrary number of coefficients, Sample processing (cont'd)

Pseudo code

```
{
   frac64 acc;          //Filter Result
   int j,k=0;
   frac16circ *aDLY = &DLY;
                        //ptr to Circ-ptr of Delay-Buffer

   *DLY = X;            //Store input value in Delay-Buffer at
                        //the position of the oldest value
   acc = 0.0;
   if(nH%2 == 0)        //even coefficients
   {
   //'n' in the comments refers current instant
   //The index i,j of X(i),H(j)(in the comments) are valid
   //for first loop iteration
   //For each next loop i,j should be decremented and
   //incremented by 2 respectively.

      for(j=0; j<nH/2; j++)
      {
         acc = acc + (frac64)(*(H+k) * (*(DLY+k)) +
                      (*(H+k+1))* (*(DLY+k+1)));
                      //acc += X(n)*H(0) + X(n-1)*H(1)
         k=k+2;
      }

   }
   else                 //odd coefficients
   {
   //'n' in the comments refers current instant
   //The index i,j of X(i),H(j)(in the comments) are valid
   //for first loop iteration.
   //For each next loop i,j should be decremented and
   //incremented by 1 respectively.
```

Fir_16 **FIR Filter, Normal, Arbitrary number of coefficients, Sample processing** (cont'd)

```
        for(j=0; j<nH; j++)
        {
           acc = acc + (frac64)(*(H+k) * (*(DLY+k)));
                        //acc += X(n)*H(0)
           k++;
        }

     }

     DLY--;             //Set DLY.index to the oldest value
                        //in Delay-Buffer
     aDLY=&DLY;         //store updated delay
     R = (frac16 sat)acc;
                        //Format the filter output from 48-bit
                        //to 16-bit saturated value

     return R;          //Filter output returned
  }
```

Techniques

- Loop unrolling, two taps/loop if coefficients are even, else one tap/loop
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Use of dual MAC instruction for even coefficients and MAC instruction for odd coefficients
- Intermediate results stored in 64 bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer

Fir_16 FIR Filter, Normal, Arbitrary number of coefficients, Sample processing (cont'd)

Memory Note

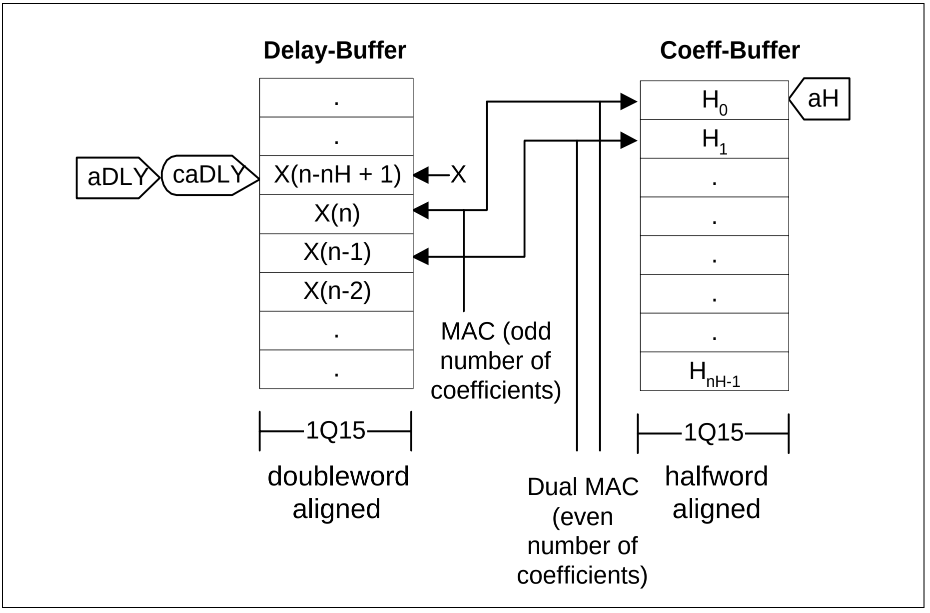


Figure 4-28 Fir_16

Fir_16 — FIR Filter, Normal, Arbitrary number of coefficients, Sample processing (cont'd)

Implementation

The FIR filter implemented structure is of transversal type, which is realized by a tapped delay line.

The FIR filter routine processes one sample at a time and returns the output of that sample. The input for which the output is to be calculated is sent as an argument to the function.

Implementation is different for even and odd coefficients.

Even number of coefficients:

TriCore's load word instruction loads the two delay line values and two coefficients in one cycle. Dual MAC instruction performs a pair of multiplications and additions according to the equation

$$acc = acc + X(n) \cdot H_0 + X(n-1) \cdot H_1 \qquad [4.41]$$

By using a dual MAC in the tap loop, the loop count is brought down by a factor of two. Here two taps are used during a single pass and loop is unrolled for efficient pointer update of delay line. Thus loop is executed (nH/2-1) times.

Odd number of coefficients:

TriCore's load halfword instruction loads one delay line value and one coefficient in one cycle. MAC instruction performs one multiplication and one addition according to the equation

$$acc = acc + X(n) \cdot H_0 \qquad [4.42]$$

By using a MAC in the tap loop, the loop count remains nH. Only one tap is used during a single pass and loop is unrolled for efficient pointer update of delay line. Thus loop is executed (nH-1) times.

Fir_16 **FIR Filter, Normal, Arbitrary number of coefficients, Sample processing** (cont'd)

The filter output R(n) is 16-bit saturated equivalent of acc when the tap loop is executed fully.

For delay line, circular addressing mode is used which helps in efficient delay update. The size of the circular Delay-Buffer is equal to the filter order, i.e., the number of coefficients. Circular buffer needs doubleword alignment. There is no restriction on the number of coefficients.

Delay pointer in the memory note shows updated pointer after tap loop is over. This points to the oldest value in the delay-buffer which is replaced by new input value.

Example

Trilib\Example\Tasking\Filters\FIR\expFir_16.c, expFir_16.cpp
Trilib\Example\GreenHills\Filters\FIR\expFir_16.cpp, expFir_16.c
Trilib\Example\GNU\Filters\FIR\expFir_16.c

Cycle Count

With DSP Extensions

For even number of coefficients

Pre-kernel	:	10
Kernel	:	$\left\lceil \frac{nH}{2} - 1 \right\rceil \times 2 + 2$
Post-kernel	:	2+2

For odd number of coefficients

Pre-kernel	:	8
Kernel	:	$[nH - 1] \times 2 + 2$
Post-kernel	:	2+2

Fir_16 **FIR Filter, Normal, Arbitrary number of coefficients, Sample processing** (cont'd)

Without DSP Extensions

For even number of coefficients

Pre-kernel	:	10
Kernel	:	same as With DSP Extensions
Post-kernel	:	3+2

For odd number of coefficients

Pre-kernel	:	8
Kernel	:	same as With DSP Extensions
Post-kernel	:	3+2

Code Size 110 bytes

FirBlk_16 — FIR Filter, Normal, Arbitrary number of coefficients, Block processing

Signature

```
void FirBlk_16(DataS      *X,
               DataS      *R,
               cptrDataS  H,
               cptrDataS  *DLY,
               int        nX
               );
```

Inputs

X	:	Pointer to Input-Buffer
R	:	Pointer to Output-Buffer
H	:	Circular pointer of Coeff-Buffer of size nH
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct
nX	:	Size of Input-Buffer

Outputs

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer
R(nX)	:	Output-Buffer

Return

None

Description

The implementation of FIR filter uses transversal structure (direct form). The block of inputs are processed at a time and output for every sample is stored in the output array. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients given by user is arbitrary. Circular buffer addressing mode is used for coefficients and delay line. Both coefficient buffer and delay line buffer are doubleword aligned. The input buffer and the output buffer are halfword aligned.

FirBlk_16 FIR Filter, Normal, Arbitrary number of coefficients, Block processing (cont'd)

Pseudo code

```
{
   frac64 acc;          //Filter Result
   int j,i,k;
   frac16circ *aDLY=&DLY;
                        //ptr to Circ-ptr of Delay-Buffer

   for(i=0; i<nX; i++)
   {
      *DLY = *X;        //Store input value in Delay-Buffer at
                        //the position of the oldest value
      acc = 0.0;
      if(nH%2 == 0)
      {
      // 'n' in the comments refers current instant
      //The index i,j of X(i),H(j)(in the comments) are
      //valid for first loop iteration.
      //For each next loop i,j should be decremented
      //and incremented by 2 respectively.

         for(j=0; j<nH/2; j++)
         {
            acc = acc + (frac64)(*(H+k) * (*(DLY+k)) +
                           (*(H+k+1)) * (*(DLY+k+1)));
                        //acc += X(n)*H(0) + X(n-1)*H(1)
            k=k+2;
         }
      }
      else
      {
      // 'n' in the comments refers current instant
      //The index i,j of X(i),H(j)(in the comments) are
      //valid for first loop iteration.
      //For each next loop i,j should be decremented and
      //incremented by 1 respectively.
```

FirBlk_16 FIR Filter, Normal, Arbitrary number of coefficients, Block processing (cont'd)

```
            for(j=0; j<nH; j++)
            {
               acc = acc + (frac64)(*(H+k) * (*(DLY+k)));
                         //acc += X(n)*H(0)
               k=k+1;
            }
         }
         DLY--;          //Set DLY.index to the oldest value
                         //in Delay-Buffer
         aDLY=&DLY;      // store updated delay
         *R++ = (frac16 sat)acc;
                         //Format the filter output from 48-bit
                         //to 16-bit saturated value
      }//end of indata loop
   }
```

Techniques

- Loop unrolling, two taps/loop if coefficients are even else one tap/loop
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Coefficient buffer implemented as circular buffer
- Use of dual MAC instruction for even number of coefficients and MAC instructions for odd number of coefficients
- Intermediate results stored in 64 bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer

FirBlk_16 — FIR Filter, Normal, Arbitrary number of coefficients, Block processing (cont'd)

Memory Note

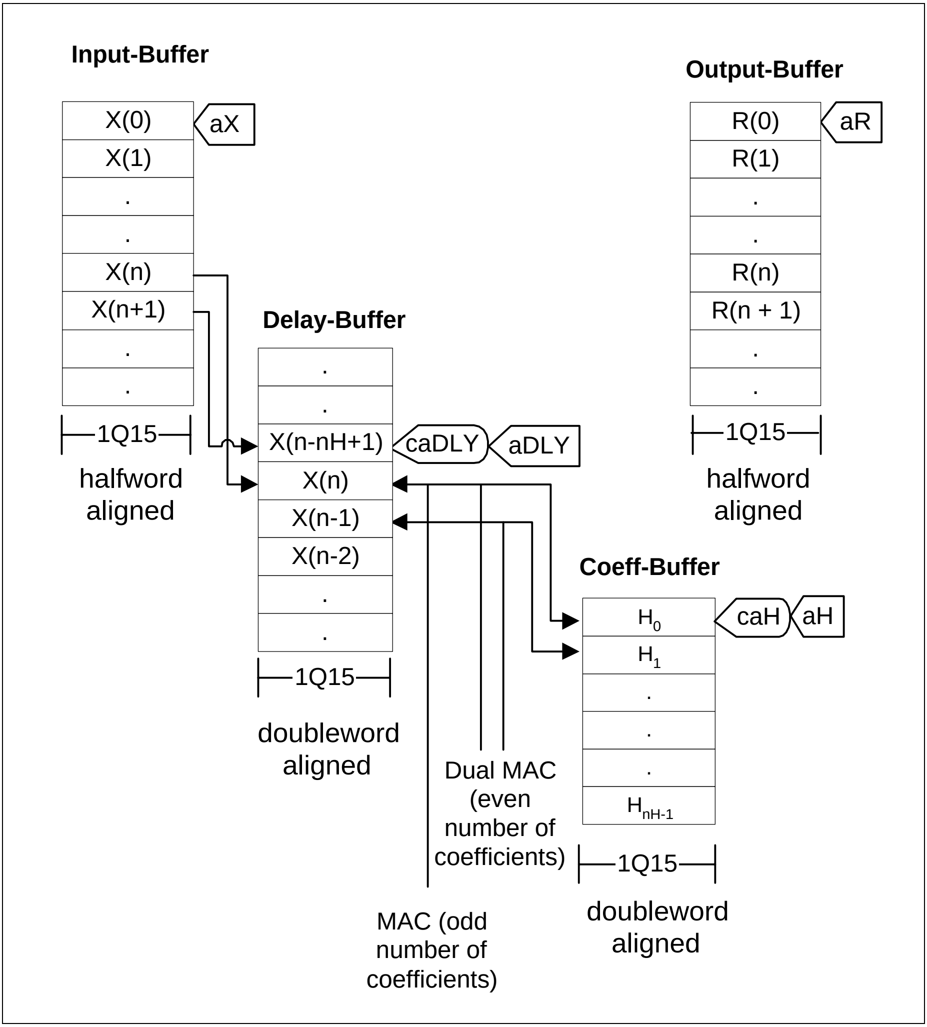


Figure 4-29 FirBlk_16

FirBlk_16 — FIR Filter, Normal, Arbitrary number of coefficients, Block processing (cont'd)

Implementation

This FIR filter routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Implementation details are same as Fir_16, except that the Coeff-Buffer is also circular and needs doubleword alignment. The size of the Coeff-Buffer is equal to the filter order, i.e., the number of coefficients. Because of circular addressing used for Coeff-Buffer, at the end of the tap loop coeff-pointer always points to H_0, i.e., first coefficient which is needed for next instant. An additional loop is needed to calculate the output for every sample in the buffer. Hence, this loop is repeated as many times as the size of the input buffer.

Example

Trilib\Example\Tasking\Filters\FIR\expFirBlk_16.c, expFirBlk_16.cpp
Trilib\Example\GreenHills\Filters\FIR\expFirBlk_16.cpp, expFirBlk_16.c
Trilib\Example\GNU\Filters\FIR\expFirBlk_16.c

Cycle Count

With DSP Extensions

For even number of coefficients

Pre-loop : 9

Loop :
$$nX \times \left\{5 + \left[\left(\frac{nH}{2} - 1\right) \times 2 + 1\right] + 3\right\}$$
$$+3$$

Post-loop : 1+2

For odd number of coefficients

Pre-loop : 6

Loop :
$$nX \times \{5 + [(nH - 1) \times 2 + 1] + 3\}$$
$$+3$$

Post-loop : 1+2

FirBlk_16 **FIR Filter, Normal, Arbitrary number of coefficients, Block processing** (cont'd)

Without DSP Extensions

For even number of coefficients

Pre-loop	:	11
Loop	:	same as With DSP Extensions
Post-Loop	:	1+2

For odd number of coefficients

Pre-loop	:	8
Loop	:	same as With DSP Extensions
Post-loop	:	1+2

Code Size 178 bytes

Fir_4_16 FIR Filter, Normal, Coefficients - multiple of four, Sample processing

Signature

```
DataS Fir_4_16(DataS       X,
               DataS       *H,
               cptrDataS   *DLY
               );
```

Inputs

X	:	Real input value
H	:	Pointer to Coeff-Buffer of size nH
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct

Output

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer

Return

R	:	Output value of the filter (48-bit value converted to 16-bit with saturation)

Description

The implementation of FIR filter uses transversal structure (direct form). The single input is processed at a time and output for every sample is returned. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients given by the user is multiple of four. Optimal implementation requires filter order to be multiple of four. Circular buffer addressing mode is used for delay line. Delay line buffer is doubleword aligned and it should be in internal memory. Coefficient-Buffer should be word aligned if it is in the external memory.

Fir_4_16 FIR Filter, Normal, Coefficients - multiple of four, Sample processing (cont'd)

Pseudo code

```
{
   frac64 acc;          //Filter Result
   int j,k;
   frac16circ *aDLY=&DLY;
                        //ptr to Circ-ptr of Delay-Buffer

   *DLY = X;            //Store input value in Delay-Buffer at
                        //the position of the oldest value
   acc = 0.0;
   //'n' in the comments refers to current instant
   //The index i,j of X(i),H(j)(in the comments) are valid
   //for first loop iteration
   //For each next loop i,j should be decremented and
   //incremented by 4 respectively.

   for(j=0; j<nH/4; j++)
   {
     acc = acc + (frac64)(*(H+k)*(*(DLY+k)) + (*(H+k+1)) * (*(DLY+k+1)));
                        //acc += X(n)*H(0) + X(n-1)*H(1)
      acc = acc + (frac64)(*(H+k+2) * (*(DLY+k+2))+
                             (*(H+k+3)) * (*(DLY+k+3)));
                        //acc += X(n-2)*H(2) + X(n-3)*H(3)
      k=k+4;
   }

   DLY--;               //Set DLY.index to the oldest value
                        //in Delay-Buffer
   aDLY=&DLY;           //store updated delay

   R = (frac16 sat)acc;
                        //Format the filter output from 48-bit
                        //to 16-bit saturated value
   return R;            //Filter output returned
}
```

Techniques

- Loop unrolling, four taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Use of dual MAC instructions
- Intermediate results stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Fir_4_16 — FIR Filter, Normal, Coefficients - multiple of four, Sample processing (cont'd)

Assumptions

- Filter size must be multiple of 4 and minimum filter order is eight
- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer
- Delay-Buffer is in internal memory

Memory Note

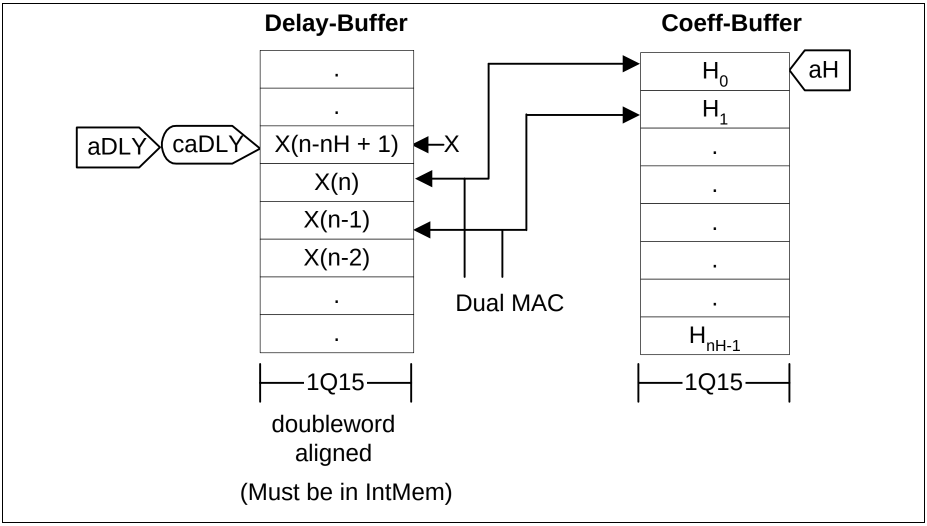


Figure 4-30 Fir_4_16

Fir_4_16 FIR Filter, Normal, Coefficients - multiple of four, Sample processing (cont'd)

Implementation

The FIR filter implemented structure is of transversal type, which is realized by a tapped delay line.

The FIR filter routine processes one sample at a time and returns the output of that sample. The input for which the output is to be calculated is sent as an argument to the function.

TriCore's load doubleword instruction loads four delay line values and four coefficients in one cycle. Each dual MAC instruction performs a pair of multiplications and additions according to the equation

$$acc = acc + X(n) \cdot H_0 + X(n-1) \cdot H_1 \qquad [4.43]$$

Thus by using two dual MACs in the tap loop, the loop count is brought down by a factor of four. Here four taps are used during a single pass and loop is unrolled for efficient pointer update of delay line. Thus loop is executed (nH/4-1) times. The filter output R(n) is 16-bit saturated equivalent of acc when the tap loop is fully executed.

To support load doubleword instruction, coeff-buffer should be word aligned if it is in the external memory and halfword aligned if it is in the internal memory. For delay line, circular addressing mode is used which helps in efficient delay update. The size of the circular Delay buffer is equal to the filter order, i.e., the number of coefficients. Circular buffer needs doubleword alignment and to use load doubleword instruction, size of the buffer should be multiple of eight bytes. This implies that the coefficients should be multiple of four.

Delay pointer in the memory note shows updated pointer after tap loop is over. This points to the oldest value in the Delay-Buffer which is replaced by new input value.

Note: To Use load doubleword instruction for the delay line the Delay-Buffer should be in internal memory only.

Fir_4_16 — FIR Filter, Normal, Coefficients - multiple of four, Sample processing (cont'd)

Example

Trilib\Example\Tasking\Filters\FIR\expFir_4_16.c, expFir_4_16.cpp
Trilib\Example\GreenHills\Filters\FIR\expFir_4_16.cpp, expFir_4_16.c
Trilib\Example\GNU\Filters\FIR\expFir_4_16.c

Cycle Count

With DSP Extensions

Pre-kernel : 7

Kernel : $\left[\frac{nH}{4} - 1\right] \times 2 + 2$

if nH > 8

$\left[\frac{nH}{4} - 1\right] \times 2 + 1$

if nH = 8

Post-kernel : 3+2

Without DSP Extensions

Pre-kernel : 7

Kernel : same as With DSP Extensions

Post-kernel : 4+2

Code Size

80 bytes

FirBlk_4_16 — FIR Filter, Normal, Coefficients - multiple of four, Block processing

Signature

```
void FirBlk_4_16(DataS      *X,
                 DataS      *R,
                 cptrDataS   H,
                 cptrDataS  *DLY,
                 int         nX
                 );
```

Inputs

X	:	Pointer to Input-Buffer
R	:	Pointer to Output-Buffer
H	:	Circular pointer of Coeff-Buffer of size nH
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct
nX	:	Size of Input-Buffer

Output

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer
R(nX)	:	Output-Buffer

Return

None

Description

The implementation of FIR filter uses transversal structure (direct form). The block of inputs are processed at a time and output for every sample is stored in the output array. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients given by user is multiple of four. Optimal implementation requires filter order to be multiple of four. Circular buffer addressing mode is used for coefficients and delay line. Both coefficient buffer and delay line buffer are doubleword aligned. Input and output buffer are halfword aligned.

FirBlk_4_16 FIR Filter, Normal, Coefficients - multiple of four, Block processing (cont'd)

Pseudo code

```
{
   frac64 acc;          //Filter Result
   int j,i,k;
   frac16circ *aDLY=&DLY;
                        //Ptr to Circ-ptr of Delay-Buffer
   frac16circ *H;       //Circ-ptr of Coeff-Buffer

   for(i=0; i<nX; i++)
   {
      *DLY = *X;        //Store input value in Delay-Buffer at
                        //the position of the oldest value
      acc = 0.0;
      //'n' in the comments refers to current instant
      //The index i,j of X(i),H(j)(in the comments) are
      //valid for first loop iteration
      //For each next loop i,j should be decremented
      //and incremented by 4 resp.

      for(j=0; j<nH/4; j++)
      {
          acc = acc + (frac64)(*(H+k) * (*(DLY+k)) +
                           (*(H+k+1)) * (*(DLY+k+1)));
                        //acc += X(n)*H(0) + X(n-1)*H(1)
          acc = acc + (frac64)(*(H+k+2) * (*(DLY+k+2)) +
                           (*(H+k+3)) * (*(DLY+k+3)));
                        //acc += X(n-2)*H(2) + X(n-3)*H(3)
          k=k+4;
      }

     DLY--;            //Set DLY.index to the oldest value in Delay-Buffer
      aDLY = &DLY;      //store updated delay
      *R++ = (frac16 sat)acc;
                        //Format the filter output from 48-bit
                        //to 16-bit saturated value
   }
}
```

FirBlk_4_16 **FIR Filter, Normal, Coefficients - multiple of four, Block processing** (cont'd)

Techniques

- Loop unrolling, four taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Coefficient buffer implemented as circular buffer
- Use of dual MAC instructions
- Intermediate results stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Filter order is a multiple of four and minimum filter order is eight
- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer
- Delay-Buffer is in internal memory

FirBlk_4_16 FIR Filter, Normal, Coefficients - multiple of four, Block processing (cont'd)

Memory Note

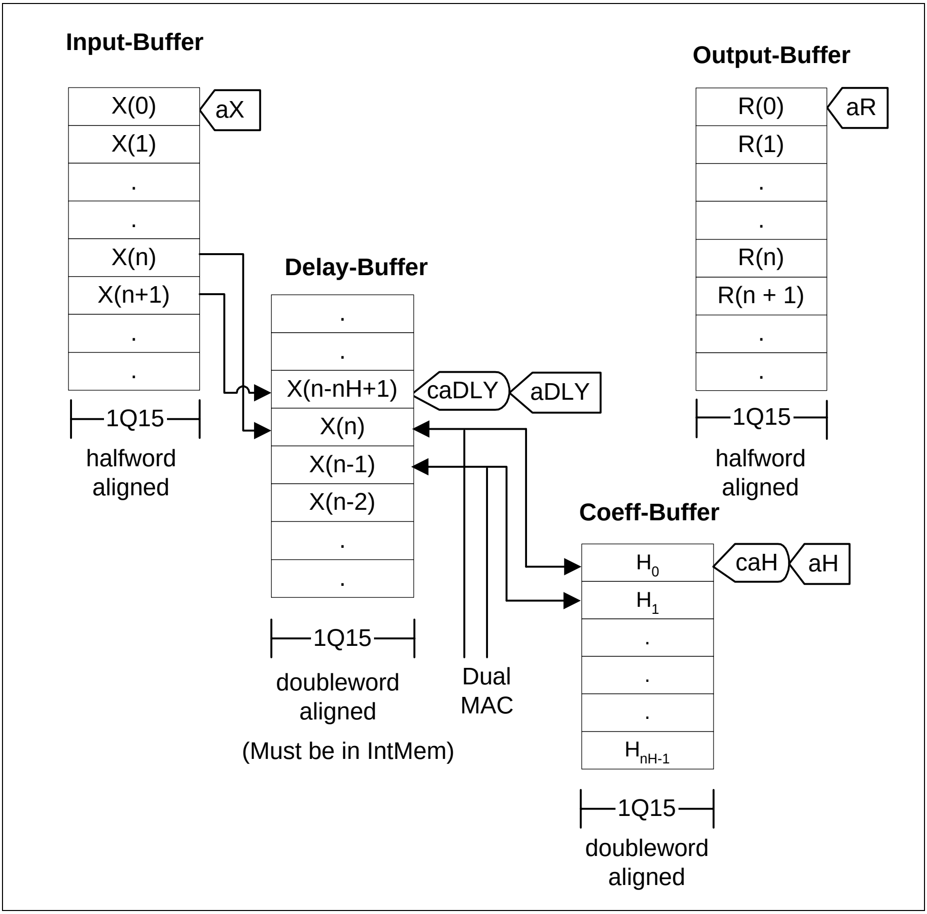


Figure 4-31 Fir_Blk_4_16

FirBlk_4_16 — FIR Filter, Normal, Coefficients - multiple of four, Block processing (cont'd)

Implementation

This FIR filter routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Implementation details are same as Fir_4_16, except that the Coeff-Buffer is also circular and needs doubleword alignment. The size of the Coeff-Buffer is equal to the filter order, i.e., the number of coefficients. Because of circular addressing used for Coeff-Buffer, at the end of the tap loop coeff-pointer always points to H_0, i.e., first coefficient which is needed for next instant. An additional loop is needed to calculate the output for every sample in the buffer. Hence, this loop is repeated as many times as the size of the input buffer.

Note: To Use load doubleword instruction for the delay line the Delay-Buffer should be in internal memory only.

Example

Trilib\Example\Tasking\Filters\FIR\expFirBlk_4_16.c, expFirBlk_4_16.cpp
Trilib\Example\GreenHills\Filters\FIR\expFirBlk_4_16.cpp, expFirBlk_4_16.c
Trilib\Example\GNU\Filters\FIR\expFirBlk_4_16.c

Cycle Count

With DSP Extensions

Pre-loop : 5

Loop :

$$nX \times \left\{5 + \left[2 \times \left(\frac{nH}{4} - 1\right) + 1\right] + 4\right\} + 3$$

Post-loop : 1+2

Without DSP Extensions

Pre-loop : 7

FirBlk_4_16 **FIR Filter, Normal, Coefficients - multiple of four, Block processing** (cont'd)

	Loop	:	same as With DSP Extensions
	Post-loop	:	1+2
Code Size	104 bytes		

4.4.2 Symmetric FIR

FIR filters with symmetrical Finite Impulse Response are called Symmetrical FIR filters. Such filters find use in signal processing applications such as speech processing where linear phase response is required to avoid phase distortion.

4.4.2.1 Descriptions

The following Symmetric FIR filter functions are described.

- Symmetric, Arbitrary number of coefficients, Sample processing
- Symmetric, Arbitrary number of coefficients, Block processing
- Symmetric, coefficients - multiple of 4, Sample processing
- Symmetric, coefficients - multiple of 4, Block processing

FirSym_16 — FIR Filter, Symmetric, Arbitrary number of coefficients, Sample processing

Signature

```
DataS FirSym_16(DataS       X,
                DataS       *H,
                cptrDataS   *DLY
                );
```

Inputs

X	:	Real input value
H	:	Pointer to Coeff-Buffer of size nH/2
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct

Output

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer

Return

R	:	Output value of the filter (48-bit value converted to 16-bit with saturation)

Description

The implementation of FIR filter uses transversal structure (direct form). A single input is processed at a time and output for that sample is returned. The filter operates on 16-bit real input, 16-bit coefficients and returns 16-bit real output. The number of coefficients given by the user is arbitrary and half of the filter order. Circular buffer addressing mode is used for delay line. Delay line buffer is double word aligned. Coeff-Buffer is halfword aligned. The Delay-Buffer is twice the size of Coeff-Buffer.

FirSym_16 FIR Filter, Symmetric, Arbitrary number of coefficients, Sample processing (cont'd)

Pseudo code

```
{
   frac64 acc;         //Filter Result
   int j,k;
   frac16circ *aDLY=&DLY1;
                       //ptr to Circ-ptr of Delay-Buffer

   DLY2 = DLY1-1;      //Ptr to X(n-nH+1)
   aDLY=&DLY2;         //store index to the oldest value for next instant
   *DLY1 = X;          //Store input value in Delay-Buffer at
                   //the position of the oldest value for current instant
   acc = 0.0;

   //The index i,j,k of X1(i),X2(j),H(k)(in the comments)
   //are valid for first loop iteration.
   //For each next loop i,j,k should be decremented, incremented and
   //incremented by 1 respectively.
   //'n' in the comments refers to current instant

   for(j=0; j<nH/2; j++)
   {
      acc = acc + (frac64)(*(H+k) * (*(DLY1+k)));
                       //acc += X1(n) * H(0)
      acc = acc + (frac64)(*(H+k) * (*(DLY2-k)));
                       //acc += X2(n-nH+1) * H(0)
      k=k+1;
   }
   DLY1=*aDLY;         //Set DLY.index to the oldest value
                       //in Delay-Buffer for next instant

   R = (frac16 sat)acc;
                       //Format the filter output from 48-bit
                       //to 16-bit saturated value

   return R;           //Filter output is returned
}
```

FirSym_16 — FIR Filter, Symmetric, Arbitrary number of coefficients, Sample processing (cont'd)

Techniques

- Loop unrolling, two taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Use of MAC instructions
- Intermediate results stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer

Memory Note

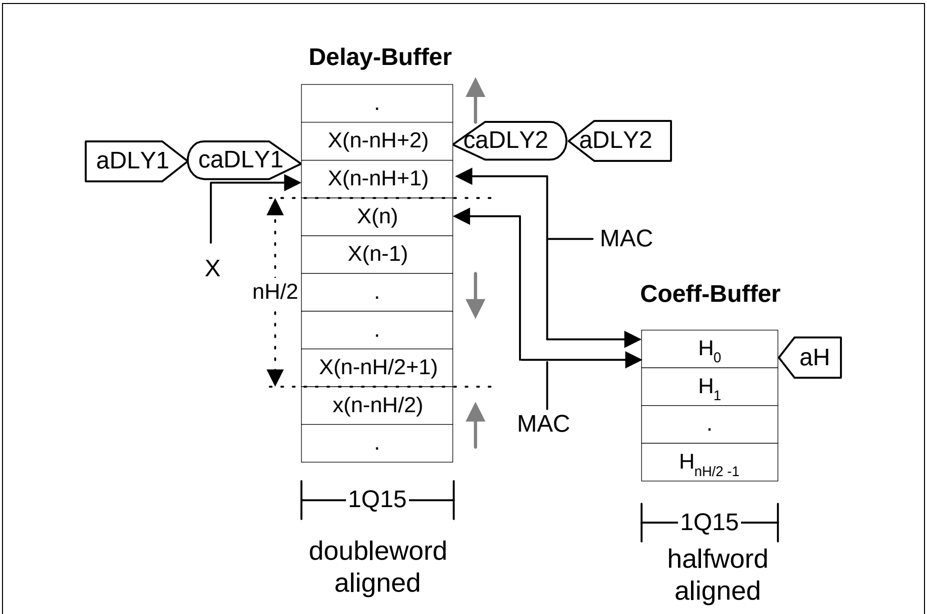


Figure 4-32 FirSym_16

FirSym_16 **FIR Filter, Symmetric, Arbitrary number of coefficients, Sample processing** (cont'd)

Implementation

The FIR filter implemented structure is of transversal type, which is realized by a tapped delay line.

The FIR filter routine processes one sample at a time and returns the output of that sample. The input for which the output is to be calculated is sent as an argument to the function.

TriCore's load halfword instruction loads the one delay line value and one coefficient in one cycle each. For delay line, circular addressing mode is used. Two pointers are initialized for circular delay line, one points to X(n), which is incremented and the other points to X(n-nH+1), which is decremented to access all the delay line values. Each pointer accesses nH/2 values.

In a symmetric FIR filter, X(n) and X(n-nH+1) get multiplied with the same coefficient H_0. This fact can be made use of to reduce the number of loads for coefficients. So, for the first pass in tap loop, one delay line pointer loads X(n) and the other pointer loads X(n-nH+1) by using load halfword instruction.

MAC instruction performs multiplication and addition. Two MACs are used in the tap loop, which for the first pass perform

$$\begin{aligned} acc &= acc + X(n) \cdot H_0 \\ acc &= acc + X(n - nH + 1) \cdot H_0 \end{aligned} \qquad [4.44]$$

Here two taps are used during a single pass and loop is unrolled to save cycle. Thus loop is executed (nH/2-1) times. The filter output R(n) is 16-bit saturated equivalent of acc when the tap loop is fully executed.

As Delay-Buffer is circular, the delay line update is done efficiently. The size of the circular Delay-Buffer is equal to the filter order, i.e., twice the number of given coefficients. Circular buffer needs doubleword alignment and to use load halfword instruction, size of the buffer should be multiple of two bytes. There is no restriction on the number of coefficients.

FirSym_16 **FIR Filter, Symmetric, Arbitrary number of coefficients, Sample processing** (cont'd)

Delay pointers in the memory note show updated pointers for the next iteration. caDLY1 points to the oldest value in the Delay-Buffer which is replaced by new input value.

Example

Trilib\Example\Tasking\Filters\FIR\expFirSym_16.c, expFirSym_16.cpp
Trilib\Example\GreenHills\Filters\FIR\expFirSym_16.cpp, expFirSym_16.c
Trilib\Example\GNU\Filters\FIR\expFirSym_16.c

Cycle Count

With DSP Extensions

Pre-kernel	:	9
Kernel	:	$\left[\frac{nH}{2} - 1\right] \times 3 + 2$
Post-kernel	:	4+2

Without DSP Extensions

Pre-kernel	:	9
Kernel	:	same as With DSP Extensions
Post-kernel	:	5+2

Code Size 88 bytes

FirSymBlk_16 — FIR Filter, Symmetric, Arbitrary number of coefficients, Block processing

Signature

```
void FirSymBlk_16(DataS      *X,
                  DataS      *R,
                  DataS      *H,
                  cptrDataS  *DLY,
                  int         nX
                  );
```

Inputs

X	:	Pointer to Input-Buffer of size nX
R	:	Pointer to Output-Buffer of size nX
H	:	Pointer to Coeff-Buffer of size nH/2
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct
nX	:	Number of input samples

Outputs

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer
R(nX)	:	Output-Buffer

Return None

Description

The implementation of FIR filter uses transversal structure (direct form). A block of inputs are processed at a time and output for every sample is stored in the output array. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients given by the user is arbitrary and half of the filter order. Circular buffer addressing mode is used for delay line. Delay line buffer is doubleword aligned. Coefficient, Input and output buffer are halfword aligned. The Delay-Buffer is twice the size of Coeff-Buffer.

FirSymBlk_16 FIR Filter, Symmetric, Arbitrary number of coefficients, Block processing (cont'd)

Pseudo code

```
{
   frac64 acc;         //Filter Result
   int i,j,k;
   frac16circ *aDLY=&DLY1;
                       //ptr to Circ-ptr of Delay-Buffer
   frac16 *H0;         //Ptr to Coeff-Buffer

   H0 = H;             //store coeff-buffer ptr
   DLY2 = DLY1-1;      //Ptr to X(n-nH+1)
   aDLY = &DLY2;       //store index to the oldest value of next instant
   *DLY1 = X;          //Store input value in Delay-Buffer at
                    //the position of the oldest value of current instant
   for(i=0; i<nX; i++)
   {
      acc = 0.0;
      k=0;

      //The index i,j,k of X1(i),X2(j),H(k)(in the comments)
      //are valid for first loop iteration.
     // For each next loop i,j,k should be decremented, incremented and
      //incremented by 1 respectively.
      //'n' in the comments refers to current instant

      for(j=0; j<nH/2; j++)
      {
         acc = acc + (frac64)(*(H+k) * (*(DLY1+k)));
                       //acc += X1(n) * H(0)
         acc = acc + (frac64)(*(H+k) * (*(DLY2-k)));
                       //acc += X2(n-nH+1) * H(0)
         k=k+1;
      }
     DLY1 = *aDLY;   //Set DLY.index to the oldest value in Delay-Buffer
      H = H0;          //initialize coeff-ptr

      *R++ = (frac16 sat)acc;
                       //Format the filter output from 48-bit
                       //to 16-bit saturated value
   }
}
```

FirSymBlk_16 **FIR Filter, Symmetric, Arbitrary number of coefficients, Block processing** (cont'd)

Techniques

- Loop unrolling, two taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Use of MAC instructions
- Intermediate results stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer

FirSymBlk_16 FIR Filter, Symmetric, Arbitrary number of coefficients, Block processing (cont'd)

Memory Note

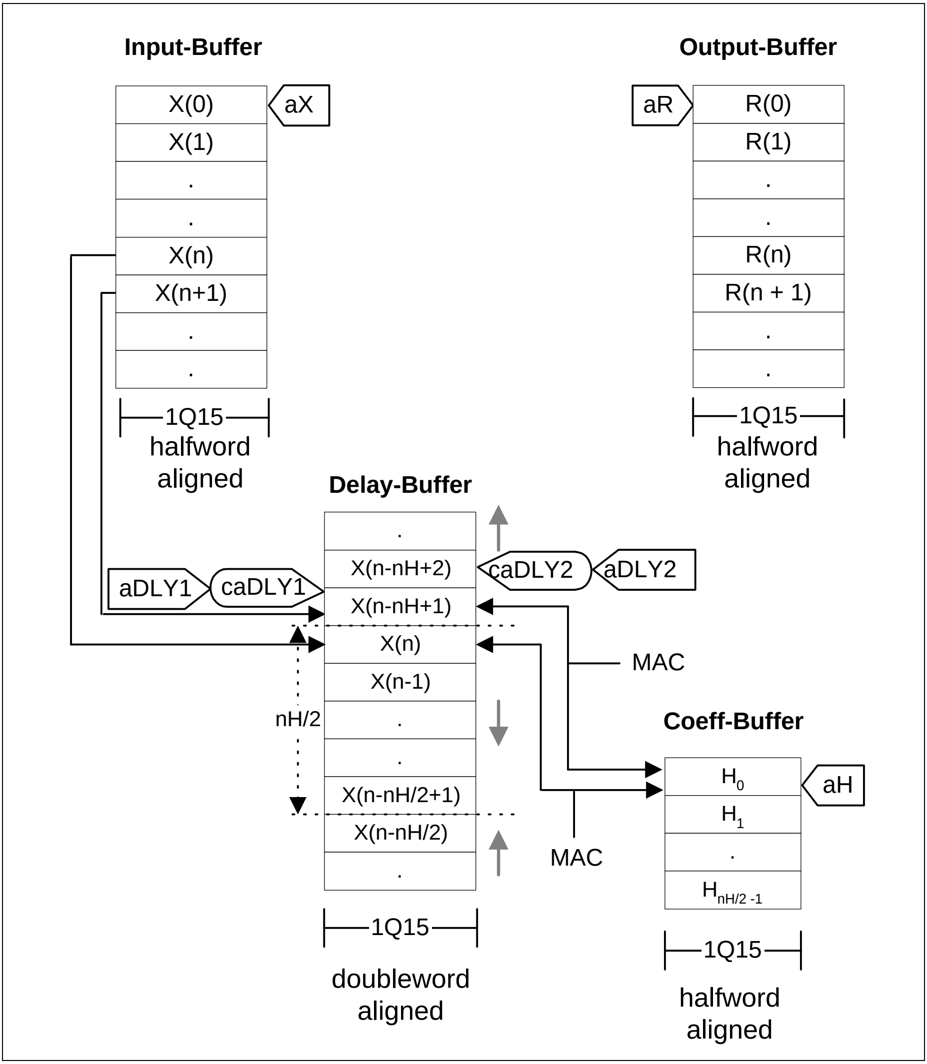


Figure 4-33 FirSymBlk_16

FirSymBlk_16 — FIR Filter, Symmetric, Arbitrary number of coefficients, Block processing (cont'd)

Implementation

This symmetric FIR filter routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Implementation details are same as FirSym_16, except that the Coeff-Buffer pointer is stored for next iteration and an additional loop is needed to calculate the output for every sample in the buffer. Hence, this loop is repeated as many times as the size of the input buffer.

Example

Trilib\Example\Tasking\Filters\FIR\expFirSymBlk_16.c, expFirSymBlk_16.cpp
Trilib\Example\GreenHills\Filters\FIR \expFirSymBlk_16.cpp, expFirSymBlk_16.c
Trilib\Example\GNU\Filters\FIR\expFirSymBlk_16.c

Cycle Count

Pre-loop : 4

Loop : $nX \times \left\{8 + \left[3 \times \left(\frac{nH}{2} - 1\right) + 1\right] + 5\right\}$

$+3$

Post-loop : 0+2

Code Size

112 bytes

FirSym_4_16 FIR Filter, Symmetric, Coefficients - multiple of four, Sample processing

Signature

```
DataS FirSym_4_16(DataS      X,
                  DataS      *H,
                  cptrDataS  *DLY
                  );
```

Inputs

X	:	Real input value
H	:	Pointer to Coeff-Buffer of size nH/2
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct

Output

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer

Return

R	:	Output value of the filter (48-bit value converted to 16-bit with saturation)

Description

The implementation of FIR filter uses transversal structure (direct form). A single input is processed at a time and output for that sample is returned. The filter operates on 16-bit real input, 16-bit coefficients and returns 16-bit real output. The filter order should be a multiple of four. Therefore number of coefficients given by the user should be even and half of the filter order. Optimal implementation requires filter order to be multiple of four. Circular buffer addressing mode is used for delay line. Delay line buffer is double word aligned. Coefficient buffer is halfword aligned. The Delay-Buffer is twice the size of Coeff-Buffer.

FirSym_4_16 FIR Filter, Symmetric, Coefficients - multiple of four, Sample processing (cont'd)

Pseudo code

```
{
   frac64 acc;          //Filter Result
   int j,k;
   frac16circ  *aDLY=&DLY1;
                        //ptr to Circ-ptr of Delay-Buffer
   DLY2 = DLY1-1;
   aDLY=&DLY2;         //store index to the oldest value for next instant
   DLY2 = DLY2-1;       //Ptr to X(n-nH+2)
   *DLY1 = X;           //Store input value in Delay-Buffer at
                        //the position of the oldest value
   acc = 0.0;

   //The index i,j,k of X1(i),X2(j),H(k)(in the comments)
   //are valid for first loop iteration.
   //For each next loop i,j,k should be decremented,incremented and
   //incremented by 2 resp.
   //'n' in the comments refers to current instant
   for(j=0; j<nH/2; j++)
   {
      acc = acc + (frac64)(*(H+k) * (*(DLY1+k)) +
                          (*(H+k+1)) * (*(DLY1+k+1)));
                        //acc += X1(n) * H(0) + X1(n-1) * H(1)
      acc = acc + (frac64)(*(H+k) * (*(DLY2-k)) + (*(H+k+1)) *
                           (*(DLY2-k-1)));
                       //acc += X2(n-nH+1) * H(0) +  X2(n-nH+2) * H(1) ||
      k=k+2;
   }
   DLY1=*aDLY;          //Set DLY.index to the oldest value
                        //in Delay-Buffer for next instant

   R = (frac16 sat)acc;
                        //Format the filter output from 48-bit
                        //to 16-bit saturated value

   return R;            //Filter output is returned
}
```

FirSym_4_16 **FIR Filter, Symmetric, Coefficients - multiple of four, Sample processing** (cont'd)

Techniques

- Loop unrolling, four taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Use of dual MAC instructions
- Intermediate results stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Filter order is a multiple of four
- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer

Memory Note

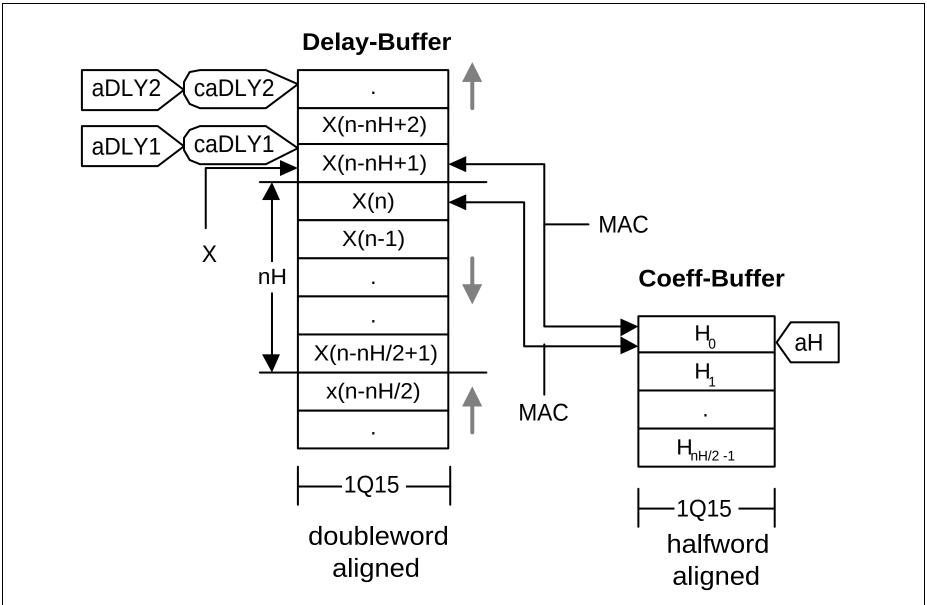


Figure 4-34 FirSym_4_16

FirSym_4_16 — FIR Filter, Symmetric, Coefficients - multiple of four, Sample processing (cont'd)

Implementation

The FIR filter implemented structure is of transversal type, which is realized as a tapped delay line.

The FIR filter routine processes one sample at a time and returns the output of that sample. The input for which the output is to be calculated is sent as an argument to the function.

TriCore's load word instruction loads the two delay line values and two coefficients in one cycle. For delay line, circular addressing mode is used. Two pointers are initialized for circular delay line, one points to X(n), which is incremented and the other points to X(n-nH+2), which is decremented to access all the delay line values. Each pointer accesses nH/2 values.

In a symmetric FIR filter, X(n) and X(n-nH+1) get multiplied with the same coefficient H_0. This fact can be made use of to reduce the number of loads for coefficients. So, for the first pass in tap loop, one delay line pointer loads X(n), X(n-1) and the other pointer loads X(n-nH+1), X(n-nH+2) by using load word instruction.

Dual MAC instruction performs a pair of multiplication and additions. Two dual MACs are used in the tap loop, which for the first pass perform

$$\begin{aligned} acc &= acc + X(n) \cdot H_0 + X(n-1) \cdot H_1 \\ acc &= acc + X(n-nH+1) \cdot H_0 + X(n-nH+2) \cdot H_1 \end{aligned} \quad [4.45]$$

Here four taps are used during a single pass and loop is unrolled to save cycle. Thus loop is executed (nH/4-1) times. The filter output R(n) is 16-bit saturated equivalent of acc when the tap loop is executed fully.

FirSym_4_16 — FIR Filter, Symmetric, Coefficients - multiple of four, Sample processing (cont'd)

As Delay-Buffer is circular, the delay line update is done efficiently. The size of the circular Delay-Buffer is equal to the filter order, i.e., twice the number of given coefficients. Circular buffer needs doubleword alignment and to use load word instruction, size of the buffer should be multiple of four bytes. The number of coefficients given should be even, which means the filter order is a multiple of four.

Delay pointers in the memory note show updated pointers for the next iteration. caDLY1 points to the oldest value in the Delay-Buffer which is replaced by new input value.

Example

Trilib\Example\Tasking\Filters\FIR\expFirSym_4_16.c, expFirSym_4_16.cpp
Trilib\Example\GreenHills\Filters\FIR\expFirSym_4_16.cpp , expFirSym_4_16.c
Trilib\Example\GNU\Filters\FIR\expFirSym_4_16.c

Cycle Count

With DSP Extensions

Pre-kernel	:	10
Kernel	:	$\left[\frac{nH}{4} - 1\right] \times 3 + 2$ if nH > 8
		$\left[\frac{nH}{4} - 1\right] \times 3 + 1$ if nH = 8
Post-Kernel	:	4+2

Without DSP Extensions

Pre-kernel	:	10
Kernel	:	same as With DSP Extensions

FirSym_4_16 **FIR Filter, Symmetric, Coefficients - multiple of four, Sample processing** (cont'd)

Post-kernel	:	5+2

Code Size 92 bytes

FirSymBlk_4_16 FIR Filter, Symmetric, Coefficients - multiple of 4, Block processing

Signature

```
void FirSymBlk_4_16(DataS      *X,
                    DataS      *R,
                    DataS      *H,
                    cptrDataS  *DLY,
                    int        nX
                    );
```

Inputs

X	:	Pointer to Input-Buffer
R	:	Pointer to Output-Buffer
H	:	Pointer to Coeff-Buffer of size nH/2
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct
nX	:	Size of Input-Buffer

Output

DLY	:	Updated circular buffer with index set to the oldest value of the filter Delay-Buffer
R	:	Output-Buffer

Return None

Description

The implementation of FIR filter uses transversal structure (direct form). A block of inputs are processed at a time and output for every sample is stored in the output array. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The filter order should be a multiple of four. Therefore the number of coefficients given by the user should be even and half of the filter order. Optimal implementation requires filter order to be multiple of four. Circular buffer addressing mode is used for delay line. Delay line buffer is doubleword aligned. Input, output and coefficient buffer are halfword aligned. The Delay-Buffer is twice the size of Coeff-Buffer.

FirSymBlk_4_16 FIR Filter, Symmetric, Coefficients - multiple of 4, Block processing (cont'd)

Pseudo code

```
{
   frac64 acc;          //Filter Result
   int i,j,k;
   frac16circ  *aDLY=&DLY1;
                        //ptr to Circ-ptr of Delay-Buffer
   frac16 *H0;          //Ptr to Coeff-Buffer
   H0 = H;
   DLY2 = DLY1-1;
   aDLY = &DLY2;       //store index to the oldest value for next instant
   DLY2 = DLY2-1;       //Ptr to X(n-nH+2)
   *DLY1 = X;           //Store input value in Delay-Buffer at
                        //the position of the oldest value
   for(i=0; i<nX; i++)
   {
      acc = 0.0;
      k=0;
      //The index i,j,k of X1(i),X2(j),H(k)(in the comments)
      //are valid for first loop iteration.
     //For each next loop i,j,k should be decremented, incremented and
      //incremented by 2 respectively.
      //'n' in the comments refers to current instant

      for(j=0; j<nH/2; j++)
      {
         acc = acc + (frac64)(*(H+k) * (*(DLY1+k)) +
                        (*(H+k+1)) * (*(DLY1+k+1)));
                        //acc += X1(n) * H(0) + X1(n-1) * H(1)
         acc = acc + (frac64)(*(H+k) * (*(DLY2-k)) +
                            (*(H+k+1)) * (*(DLY2-k-1)));
                       //acc += X2(n-nH+1) * H(0) +  X2(n-nH+2) * H(1) ||
         k=k+2;
      }
     DLY1 = *aDLY;   //Set DLY.index to the oldest value in Delay-Buffer
      H = H0;

      *R++ = (frac16 sat)acc;
                        //Format the filter output from 48-bit
                        //to 16-bit saturated value
   }

}
```

FirSymBlk_4_16 FIR Filter, Symmetric, Coefficients - multiple of 4, Block processing (cont'd)

Techniques

- Loop unrolling, four taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Use of dual MAC instructions
- Intermediate results stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer

FirSymBlk_4_16 FIR Filter, Symmetric, Coefficients - multiple of 4, Block processing (cont'd)

Memory Note

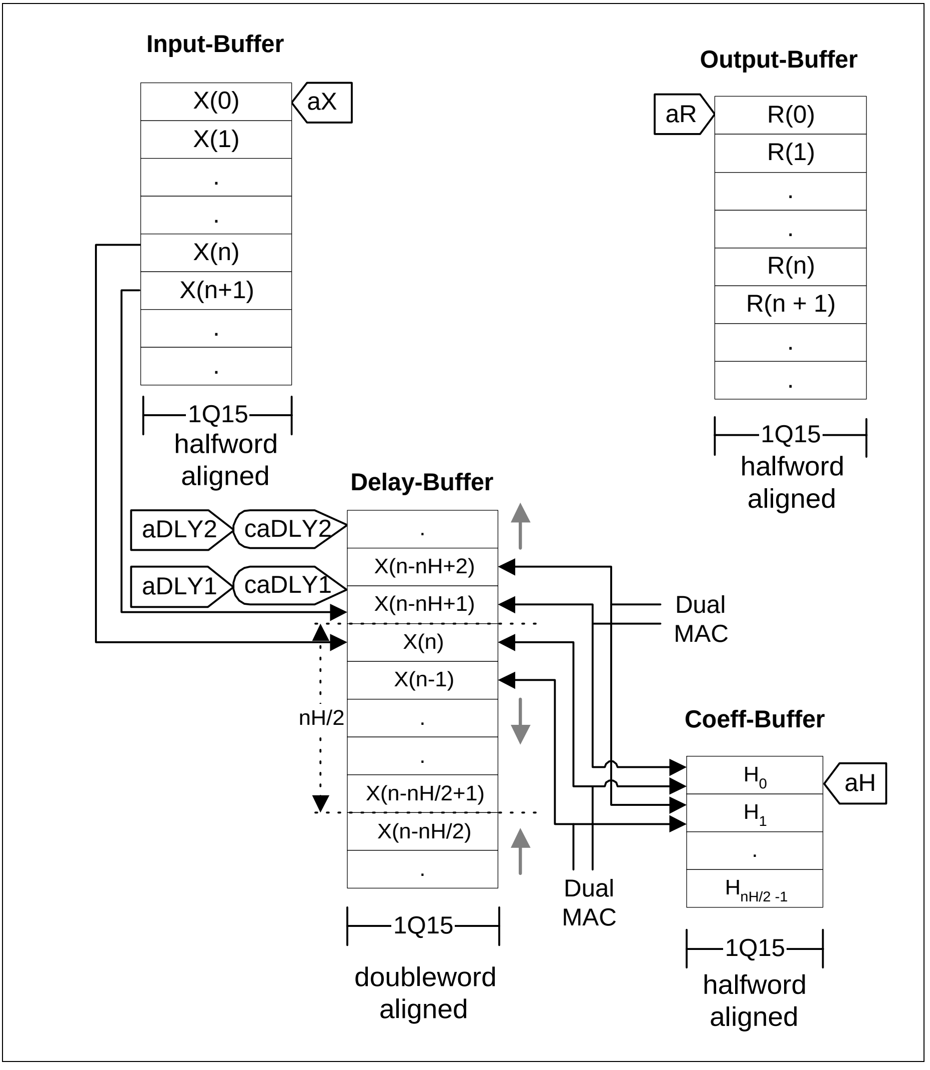


Figure 4-35 FirSymBlk_4_16

FirSymBlk_4_16 **FIR Filter, Symmetric, Coefficients - multiple of 4, Block processing** (cont'd)

Implementation

This symmetric FIR filter routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Implementation details are same as FirSym_4_16, except that the Coeff-Buffer pointer is stored for next iteration and an additional loop is needed to calculate the output for every sample in the buffer. Hence, this loop is repeated as many times as the size of the input buffer.

Example

Trilib\Example\Tasking\Filters\FIR\expFirSymBlk_4_16.c, expFirSymBlk_4_16.cpp
Trilib\Example\GreenHills\Filters\FIR\ expFirSymBlk_4_16.cpp, expFirSymBlk_4_16.c
Trilib\Example\GNU\Filters\FIR\expFirSymBlk_4_16.c

Cycle Count

Pre-kernel	:	4
Kernel	:	$nX \times \left\{9 + \left[3 \times \left(\frac{nH}{4} - 1\right) + 1\right] + 5\right\}$ $+ 1+2$
Post-kernel	:	0+2

Code Size 116 bytes

4.4.3 Multirate Filters

Discrete time systems with unequal sampling rates at various parts of the system are called Multirate Systems. For sampling rate alterations, the basic sampling rate alteration devices are invariably employed together with lowpass digital filters. Filters having different sampling rates at input and output of filter are called Multirate Filters. The two types of multirate filtering processes are Decimation filtering and Interpolation filtering.

4.4.3.1 Decimating Filters

Decimation is equivalent to down sampling a discrete-time signal. It is used to eliminate redundant data, allowing more information to be stored, processed or transmitted in the same amount of data.

Decimator or down sampler reduces the sampling rate by a factor of integer M.

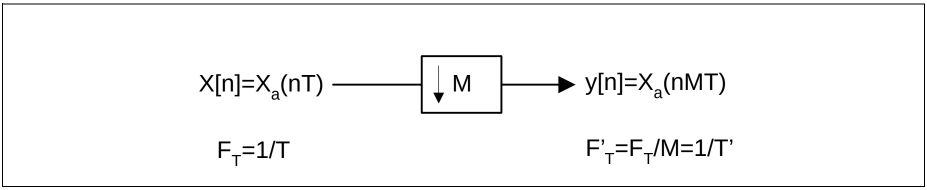


Figure 4-36 Decimation/down Sampling Illustration

The sampling rate of a critically sampled discrete time signal with a spectrum occupying the full Nyquist range cannot be reduced any further since such a reduction will introduce aliasing. Hence the bandwidth of a critically sampled signal must first be reduced by lowpass filtering before its sampling rate is reduced by a down sampler. The decimation algorithm can be implemented using FIR or IIR filter structure. But generally, FIR is used. The overall system comprising of a lowpass filter followed by a down sampler ahead of a lowpass FIR filter is called decimator or decimating FIR. Such a filter would give an output for every M^{th} input.

The decimating FIR filter is given by

$$y(m) = \sum_{K=0}^{N-1} h(K)x(Mm-K) \qquad [4.46]$$

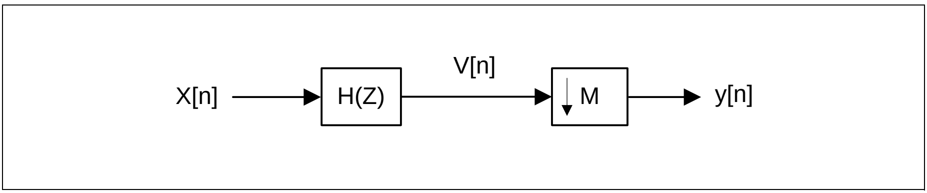


Figure 4-37 Decimation Filter Block Diagram

4.4.3.2 Interpolating FIR Filters

Interpolation increases the sample rate of a signal inserting zeros between the samples of input data. In practice, the zero-valued samples inserted by the up sampler are replaced with appropriate non-zero values using some type of interpolation process in order that the new higher rate sequence be useful. This interpolation can be done by digital lowpass filtering.

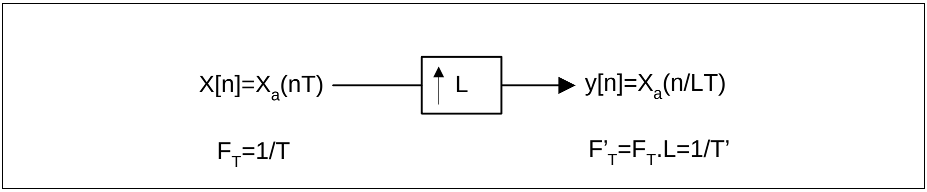


Figure 4-38 Interpolation/Down Sampling Illustration

The system comprising of up sampler followed by FIR lowpass filter which is used to remove the unwanted images in the spectra of up sampled signal is called Interpolating FIR filter.

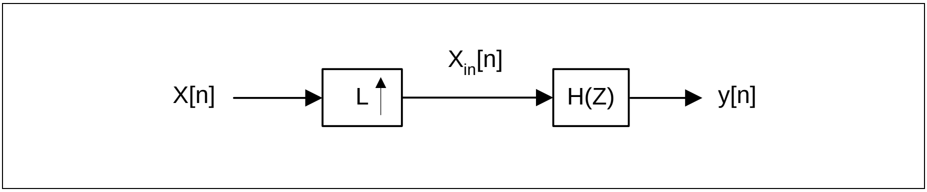


Figure 4-39 Interpolation Filter Block Diagram

The rate expander inserts If-1 zero valued samples after each input sample. The resulting samples $X_{in}[n]$ are lowpass filtered to produce output y(n), a smooth and anti imaged version of $X_{in}[n]$. The transfer function of interpolator H(k) incorporates a gain of 1/If because the If-1 zeros inserted by the rate expander cause the energy of each input to be spread over If output samples. The lowpass filter of interpolator uses a direct form FIR filter structure for computational efficiency. Output of an FIR filter is given by

$$y[n] = \sum_{k=0}^{N-1} h(k)X_{in}[n-k] \tag{4.47}$$

where,

N-1	:	the number of filter coefficients (taps)
$X_{in}[n-k]$	:	the rate expanded version of the input X[n]

X[n] is related to X_{in}[n-k] by

$$X_{in}[n-k] = \begin{cases} X((n-k)/If) & \text{for } (n\text{-}k)=0, \pm If, \pm 2If\ldots \\ 0 & \text{Otherwise} \end{cases}$$

4.4.3.3 Description

The following Multirate FIR filters are described.

- Decimation FIR
- Interpolation FIR

FirDec_16 Decimation FIR Filter

Signature

```
void FirDec_16(DataS       *X,
               DataS       *R,
               cptrDataS    H,
               cptrDataS   *DLY,
               int          nX,
               int          Df
               );
```

Inputs

X	:	Pointer to Input-Buffer
R	:	Pointer to Output-Buffer
H	:	Circular pointer of Coeff-Buffer of size nH
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH Without DSP Extension - Pointer to Circ-Struct
(nH)	:	Transferred as a part of Circular Pointer data type in a DLY parameter
nX	:	Size of Input-Buffer
Df	:	Decimation length

Outputs

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer
R(nX)	:	Output-Buffer

Return None

Description

The implementation of Decimation FIR filter uses transversal structure (direct form). A block of inputs are processed at a time. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. Number of coefficients is arbitrary. If nX/Df is not an integer, the trailing samples are lost. Circular buffer addressing mode is used for coefficients and delay line. Both coefficient buffer and Delay-Buffer are doubleword aligned. Input and output buffers are halfword aligned.

FirDec_16 Decimation FIR Filter

Pseudo code

```
{
   frac64 acc;        //Filter result
   int j,i,k;
   frac16circ *adly=&DLY;
                      //Ptr to Circ-ptr of Delay-Buffer
   //macro
   macro FirDec EV_Coef, EV_Coef_Odd_Df
   {
      if EV_Coef==TRUE
      {
         //FIR filtering
         for(i=0; i<nX; i++)
         {
            *DLY = *X++;
                      //Store input value in Delay-Buffer at
                      //the position of the oldest value
            acc = 0.0;
            // 'n' in the comments refers current instant
            //The index i,j of X(i),H(j)(in the comments) are
            //valid for first loop iteration.
            //For each next loop i,j should be decremented
            //and incremented by 2 respectively.
            for(j=0; j<nH/2; j++)
            {
               acc = acc + (frac64)(*(H+k) * (*(DLY+k)) +
                      (*(H+k+1)) * (*(DLY+k+1)));
                      //acc += X(n)*H(0) + X(n-1)*H(1)
               k=k+2;
            }
            DLY--;
            //(Df-1) values loaded into delay buffer before next output
            //calculation
            if (EV_Coef_Odd_Df==TRUE)
            {
               for(i=0;i<(Df-1)/2;i++)
               {
                  *DLY-- = *X++;
                  *DLY-- = *X++;
               }
            }
            else
            {
```

FirDec_16 Decimation FIR Filter

```
                for(i=0;i<Df-1;i++)
                {
                   *DLY-- = *X++;
                }
    else
    {
    // 'n' in the comments refers to current instant
    //The index i,j of X(i),H(j)(in the comments) are
    //valid for first loop iteration.
    //For each next loop i,j should be decremented and
    //incremented by 1 respectively.
       for(j=0; j<nH; j++)
       {
          acc = acc + (frac64)(*(H+k) * (*(DLY+k)));
                   //acc += X(n)*H(0)
          k=k+1;
       }
       DLY--;
       //(Df-1) values loaded into delay buffer before next output
       //calculation
       for(i=0;i<Df-1;i++)
       {
          *DLY-- = *X++;
       }
    }
 }//End of Macro

 FirDec_16:
 {
    nR = nX/Df;
    if (nH%2 == 0)
    {
       if (Df%2 != 0)
       {
           FirDec TRUE, TRUE;
       }
       FirDec TRUE, FALSE;
    }
    else
    {
       FirDec FALSE, FALSE;
    }
 }
}
```

FirDec_16 Decimation FIR Filter

Techniques

- Loop unrolling, two taps/loop if coefficients are even else one tap/loop
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Coefficient buffer implemented as circular buffer
- Intermediate results stored in 64-bit register
- Instruction ordering for zero overhead Load/Store

Assumptions

- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer

FirDec_16 Decimation FIR Filter

Memory Note

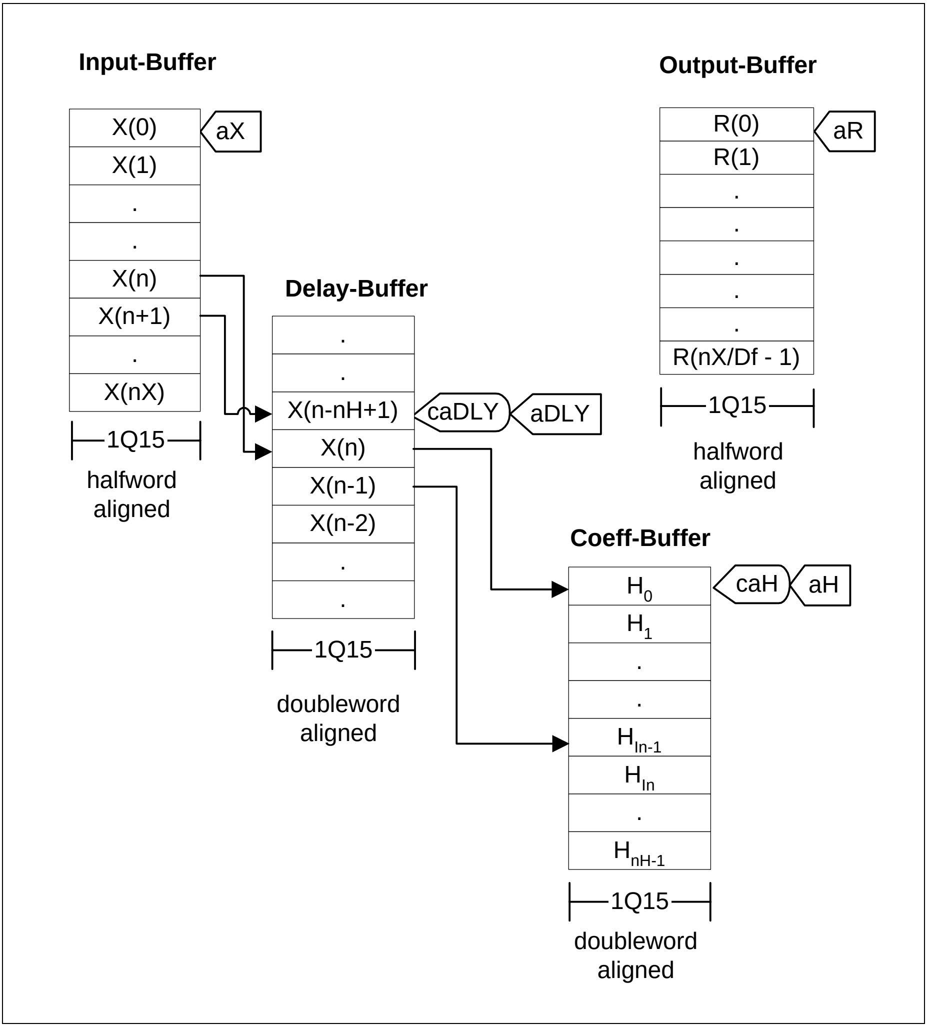


Figure 4-40 FirDec_16

FirDec_16 Decimation FIR Filter

Implementation

Decimation FIR filter is implemented with Transversal structure which is realized by a tapped delay line. This Decimation FIR filter routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Both Coeff-Buffer and data buffer are circular and need doubleword alignment. The size of Coeff-Buffer and Delay-Buffer are equal to filter order, i.e., the number of coefficients. The size of output buffer is nX/Df as there will be an output only for every Dfth input. A macro is used for performing the decimating FIR filtering. The macro is called with two arguments, EV_Coef, EV_Coef_Odd_Df.
If the number of coefficients is even (EV_Coef = TRUE) TriCore's load word instruction loads the two delay line values and two coefficients in one cycle. Dual MAC instruction performs a pair of multiplications and additions according to the equation

$$\mathrm{acc} = \mathrm{acc} + X(n) \cdot H_0 + X(n-1) \cdot H_1 \qquad [4.48]$$

By using a dual MAC in the tap loop, the loop count is brought down by a factor of two. Here two taps are used during a single pass and loop is unrolled for efficient pointer update of delay line. Thus loop is executed (nH/2-1) times.

In case of odd number of coefficients TriCore's load halfword instruction loads one delay line value and one coefficient in one cycle. MAC instruction performs one multiplication and one addition according to the equation

$$\mathrm{acc} = \mathrm{acc} + X(n) \cdot H_0 \qquad [4.49]$$

By using a MAC in the tap loop, the loop count remains nH. Only one tap is used during a single pass and loop is unrolled for efficient pointer update of delay line. Thus loop is executed (nH-1) times.

For decimation, after each FIR output calculation the delay line has to be updated by (Df-1) inputs for which output will not be calculated.

FirDec_16 Decimation FIR Filter

If the number of coefficients is even and Df is odd, (EV_Coef_Odd_Df = TRUE) then the updation of delay line can be done using TriCore's load word instructions thereby reducing the loop count for the decimation loop by a factor of two else the load halfword instruction is used and the loop is executed (Df-1) times.

Thus the implementation is most optimal for the case of even coefficient and odd Df.

Example

Trilib\Example\Tasking\Filters\FIR\expFirDec_16.c, expFirDec_16.cpp
Trilib\Example\GreenHills\Filters\FIR\expFirDec_16.cpp, expFirDec_16.c
Trilib\Example\GNU\Filters\FIR\expFirDec_16.c

Cycle Count

For Macro FirDec

Mcall (TRUE,TRUE)

Pre-loop : 3

Loop :

$$\frac{nX}{Df} \times \left[5 + \left(\frac{nH}{2} - 1\right)2 + 5 + ((Df - 1)/2)3 + 3\right] + 2$$

Post-loop : 2

Mcall (TRUE,FALSE)

Pre-loop : 3

Loop :

$$\frac{nX}{Df} \times \left[5 + \left(\frac{nH}{2} - 1\right)2 + 5 + Df(2) + 3\right] + 2$$

Post-loop : 2

Mcall (TRUE,FALSE)

Pre-loop : 2

FirDec_16 Decimation FIR Filter

Loop : $\frac{nX}{Df} \times [5 + (nH - 1)2 + 5 + Df(2)$
$+3] + 2$

Post-loop : 2

where integer part of nX/Df is considered. The number of cycles taken by the Loop should be reduced by nX/Df if either the tap loop or the decimation loop gets executed only once. If both get executed only once then the total reduction in number of cycles taken by the loop is 2(nX/Df) for all the cases.

For FirDec_16

With DSP Extensions

Even nH and odd Df

$31 + \mathrm{Mcall}(\mathrm{TRUE}, \mathrm{TRUE}) + 2 + 2$

Even nH and even Df

$27 + \mathrm{Mcall}(\mathrm{TRUE}, \mathrm{FALSE}) + 2 + 2$

Odd nH

$28 + \mathrm{Mcall}(\mathrm{FASLE}, \mathrm{FALSE}) + 2 + 2$

where Mcall (X,Y) is the number of cycles taken by the macro when the arguments passed to it are X and Y.

Without DSP Extensions

Even nH and odd Df

$$33 + \mathrm{Mcall}(\mathrm{TRUE}, \mathrm{TRUE}) + 2 + 2$$

Even nH and even Df

$$29 + \mathrm{Mcall}(\mathrm{TRUE}, \mathrm{FALSE}) + 2 + 2$$

Odd nH

$$30 + \mathrm{Mcall}(\mathrm{FALSE}, \mathrm{FALSE}) + 2 + 2$$

where Mcall (X,Y) is the number of cycles taken by the macro when the arguments passed to it are X and Y.

Code Size 308 bytes

FirInter_16 Interpolation FIR Filter

Signature

```
void FirInter_16(DataS      *X,
                 DataS      *R,
                 cptrDataS  H,
                 cptrDataS  *DLY,
                 int        nX,
                 int        If
                 );
```

Inputs

X	:	Pointer to Input-Buffer
R	:	Pointer to Output-Buffer
H	:	Circular pointer of Coeff-Buffer of size nH
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH Without DSP Extension - Pointer to Circ-Struct
(nH)	:	Transferred as a part of Circular Pointer data type in a DLY parameter
nX	:	Size of Input-Buffer
If	:	Interpolation length

Outputs

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer
R(nX)	:	Output-Buffer

Return None

Description

The implementation of Interpolation FIR filter uses transversal structure (direct form). The block of inputs are processed at a time and output for every sample is stored in the output array. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients given by user are arbitrary, but nX/If must be an integer. Circular buffer addressing mode is used for coefficients and delay line. Both coefficient buffer and delay line buffer are doubleword aligned. Input and output buffer are halfword aligned.

FirInter_16 Interpolation FIR Filter (cont'd)

Pseudo code

```
{
   frac64 acc;        //Filter result
   int i,j,k,l;
   frac16 circ*aDLY=DLY
                      //Ptr to Circ-Ptr of Delay-Buffer
   if ((nH/If)%2 == 0)
   {
      for (i=0;i<nX;i++)
      {
         *DLY=*X      //store input value in Delay-Buffer at the
                      //position of the oldest value
         acc = 0.0;
         l = 0;
         for (j=0;j<If;j++)
         {
         // 'n' in the comments refers current instant
         //The index i,j of X(i),H(j)(in the comments) are
         //valid for first loop iteration.
         //For each next loop i,j should be decremented and
         //incremented by 1 respectively.

            for (k=0;k<nH/2If;k++)
            {
               m = 0;
               acc = acc + (frac64)(*(H+l+m)*(*DLY+k)) + (*(H+l+m+1)*
                     (*(DLY+k+1)));
                     //acc = X(n)*H(0)+X(n-1)*H(If)
               m = m + If;
               k = k + 2;
            }//(nH/2If) loop
            l++;
            *R++ = (frac16 sat)acc;
                   //format the filter output from 48-bit to 16-bit
                    //saturated value
          }//(If) loop
            DLY--;
       }//nX loop
    }//If
    else
    {
```

FirInter_16 Interpolation FIR Filter (cont'd)

```
       for (i=0;i<nX;i++)
       {
          *DLY=*X        //store input value in Delay-Buffer at the
                      //position of the oldest value
          acc = 0.0;
          l = 0;
          for (j=0;j<If;j++)
          {
          // 'n' in the comments refers current instant
          //The index i,j of X(i),H(j)(in the comments) are
          //valid for first loop iteration.
          //For each next loop i,j should be decremented and
          //incremented by 1 respectively.
             for (k=0;k<nH/If;k++)
             {
                m = 0;
                acc = acc + (frac64)(*(H+l+m)*(*DLY+k))
                      //acc = X(n)*H(0)+X(n-1)*H(If)
                m = m + If;
                k = k + 1;
             }//(nH/If) loop
             l++;
             *R++ = (frac16 sat)acc;
                    //format the filter output from 48-bit to 16-bit
                      //saturated value
           }//(If) loop
           DLY--;

        }//nX loop
        aDLY = DLY;    //store updated delay
     }//else loop
}
```

Techniques

- Loop unrolling, one tap/loop if (nH/If) is odd and two taps/loop if even
- Use of packed data Load/Store
- Delay line implemented as circular buffer
- Coefficient buffer implemented as circular buffer
- Intermediate results stored in 64-bit register
- Instruction ordering for zero overhead Load/Store

FirInter_16 Interpolation FIR Filter (cont'd)

Assumptions

- Inputs, outputs, coefficients and delay line are in 1Q15 format
- Filter order nH is not explicitly sent as an argument, instead it is sent through the argument DLY as a size of circ-Delay-Buffer
- The size of circ-Delay-Buffer is nH/If and it should be integer

FirInter_16 Interpolation FIR Filter (cont'd)

Memory Note

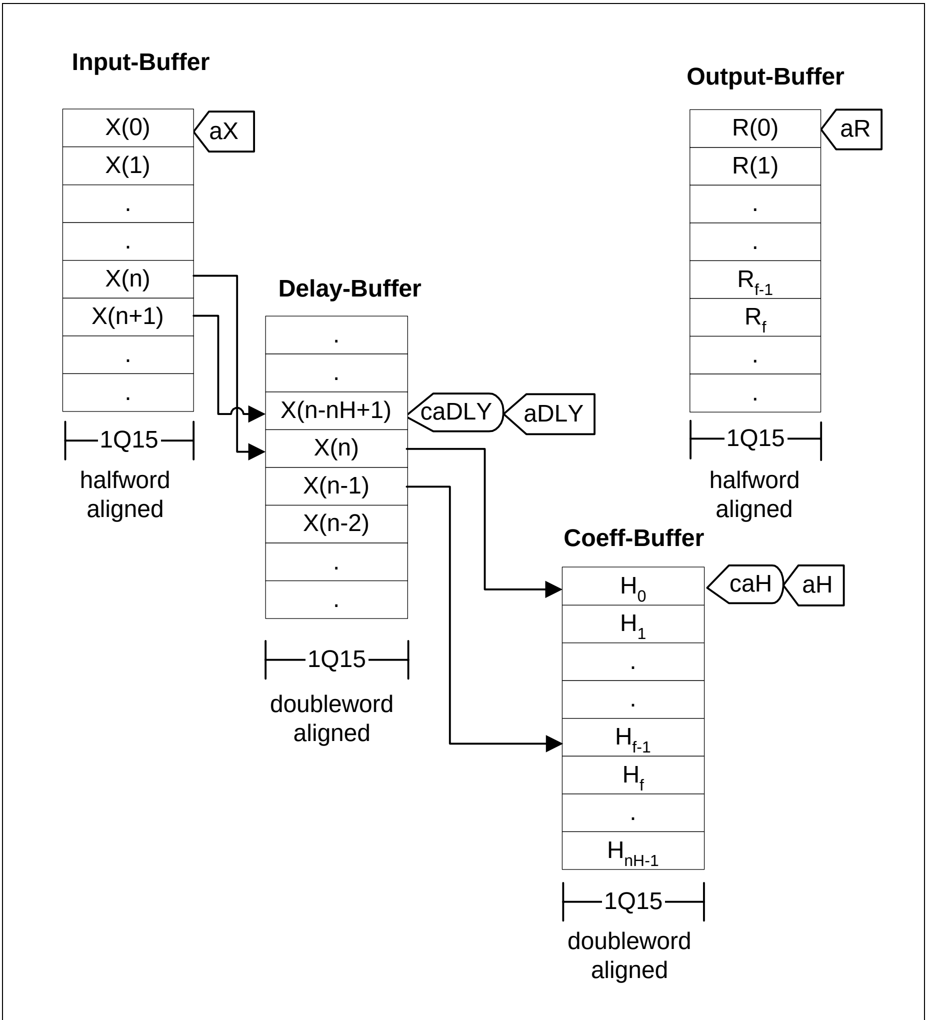


Figure 4-41 FirInter_16

FirInter_16 — Interpolation FIR Filter (cont'd)

Implementation

Interpolation FIR filter implemented structure is transversal type which is realized by a tapped delay line. This interpolation FIR filter routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.
In Interpolation FIR both Coeff-Buffer and data-buffer are circular and needs doubleword alignment. The size of Coeff-Buffer is equal to filter order, i.e., the number of coefficients.

Implementation is different for even and odd coefficients.

Even number of coefficients:

TriCore's load word instruction loads the two delay line values and two coefficients in one cycle. Dual MAC instruction performs a pair of multiplications and additions according to the equation

$$acc = acc + X(n) \cdot H_0 + X(n-1) \cdot H_{If} \qquad [4.50]$$

By using a dual MAC in the tap loop, the loop count is brought down by a factor of two. This tap loop which is innermost loop, is executed (nX/2If-1) times. Delay pointer is incremented once every cycle, so that successive data are multiplied. Coefficient pointer after each product and accumulation is incremented by If. This is done to make the routine efficient on the multiplication by zero in data samples are avoided by incrementing the coefficients pointer by If.

Odd number of coefficients:

TriCore's load halfword instruction loads one delay line value and one coefficients in one cycle. MAC instruction performs one multiplication and one addition according to the equation

$$acc = acc + X(n) \cdot H_0 \qquad [4.51]$$

FirInter_16 — Interpolation FIR Filter (cont'd)

This tap loop which is innermost loop turns (nX/If-1) times. Delay pointer is incremented once every cycle, so that successive data are multiplied. Coefficient pointer after each product and accumulation is incremented by If. This is done to make the routine efficient, as the multiplication by zeros in data samples are avoided by incrementing the coefficients pointer by If.
In data loop runs nX times. Delay pointer points to the oldest data and coefficient pointer to beginning of Coeff-Buffer.
Interpolation loop runs If times. Delay pointer points to the new data which is loaded and coefficient pointer points to one more than what it has pointed during last iteration.

Example

Trilib\Example\Tasking\Filters\FIR\expFirInter_16.c, expFirInter_16.cpp
Trilib\Example\GreenHills\Filters\FIR\expFirInter_16.cpp, expFirInter_16.c
Trilib\Example\GNU\Filters\FIR\expFirInter_16.c

Cycle Count

With DSP Extensions

For even number of coefficients

$$12 + nX \times \left[3 + If \times \left\{11 + \left(\left(\frac{nH}{2 \times If}\right) - 1\right) \times (5) + 1\right\} + 2 + 2\right]$$

$$+1+2+1+2$$

For odd number of coefficients

$$7 + nX \times \left[3 + If \times \left\{9 + \left(\frac{nH}{If} - 1\right) \times (3) + 1\right\} + 2 + 2\right]$$

$$+1+2+1+2$$

Without DSP Extensions

For even number of coefficients

$$14 + nX \times \left[3 + If \times \left\{11 + \left(\left(\frac{nH}{2 \times If}\right) - 1\right) \times (5) + 1\right\} + 2 + 2\right]$$

$$+1+2+1+2$$

FirInter_16 **Interpolation FIR Filter** (cont'd)

For odd number of coefficients

$$9 + nX \times \left[3 + If \times \left\{ 9 + \left(\frac{nH}{If} - 1 \right) \times (3) + 1 \right\} + 2 + 2 \right]$$

$$+1+2+1+2$$

Code Size 142 bytes

4.5 IIR Filters

Infinite Impulse Response (IIR) filters have infinite duration of non-zero output values for a given finite duration of non-zero impulse input. Infinite duration of output is due to the feedback used in IIR filters.

Recursive structures of IIR filters make them computationally efficient but because of feedback not all IIR structures are realizable (stable). The transfer function for the direct form of the biquad (second order) IIR filter is given by

$$H[z] = \frac{R[z]}{X[z]} = \frac{H_0 + H_1 \cdot z^{-1} + H_2 \cdot z^{-2}}{1 - (H_3 \cdot z^{-1}) - (H_4 \cdot z^{-2})} \qquad [4.52]$$

where H_3, H_4 correspond to the poles and H_0, H_1, H_2 correspond to the zeroes of the filter.

The equivalent difference equation is

$$\begin{aligned} R(n) &= H_0 \cdot X(n) + H_1 \cdot X(n-1) + H_2 \cdot X(n-2) \\ &\quad + H_3 \cdot R(n-1) + H_4 \cdot R(n-2) \end{aligned} \qquad [4.53]$$

where, X(n) is the n^{th} input and R(n) is the corresponding output.

The direct form is not commonly used in IIR filter design. In the case of a linear shift-invariant system, the overall input-output relationship of a cascade is independent of the order in which systems are cascaded. This property suggests a second direct form realization. Therefore, another form called Canonical form (also called direct form II) which uses half the number of delay stages and thereby less memory, is used for the implementation. All the IIR filters in this DSP Library have been implemented in this form.

The block diagram for a biquad (second order) filter in canonical form is as follows.

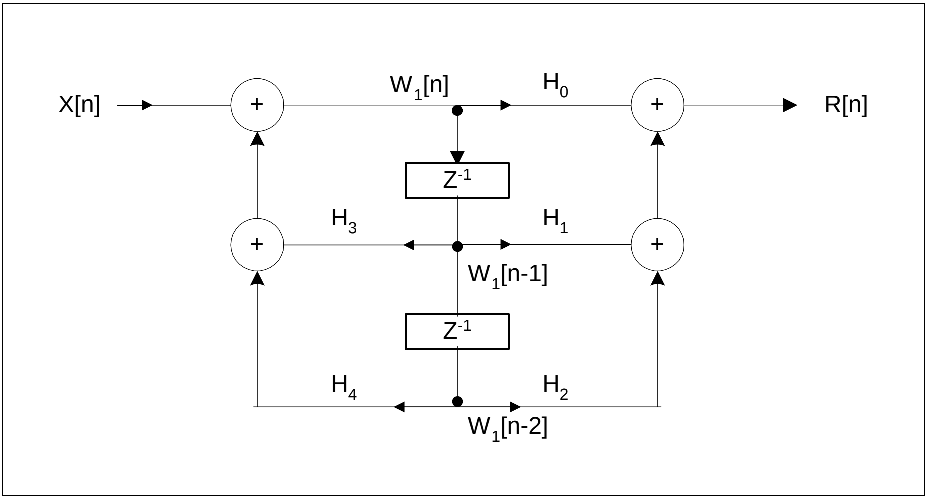


Figure 4-42 Canonical Form (Direct Form II) Second-order Section

Equation [4.52] can be broken into two parts in terms of zeroes and poles of transfer function as

$$
\begin{aligned}
W(n) &= X(n) + H_3 \cdot W(n-1) + H_4 \cdot W(n-2) \\
R(n) &= H_0 \cdot W(n) + H_1 \cdot W(n-1) + H_2 \cdot W(n-2)
\end{aligned}
\qquad [4.54]
$$

From the figure, it is clear that the first part of this equation corresponds to poles and the second corresponds to zeros. All the implementations of IIR filters use this equation.

The term W(n), called as the delay line, refers to the intermediate values. Any higher order IIR filter can be constructed by cascading several biquad stages together. A cascaded realization of a fourth order system using direct form II realization of each biquad subsystem would be as shown in the following diagram.

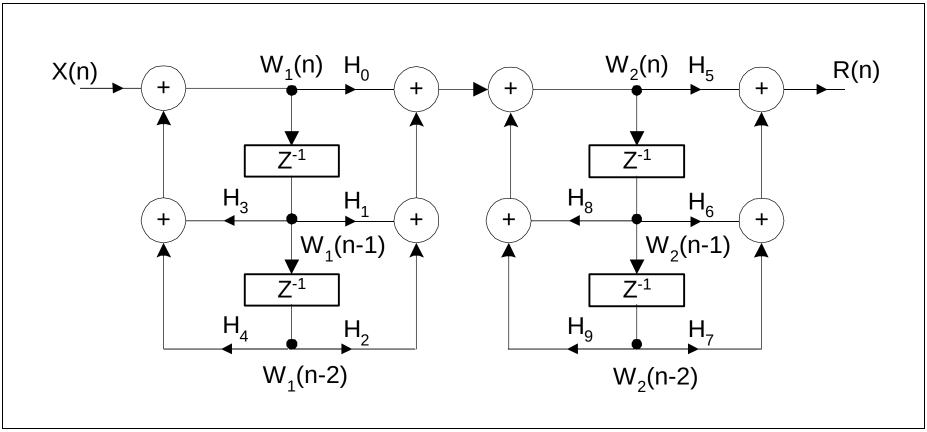


Figure 4-43 Cascaded Biquad IIR Filter

A Comparison between FIR and IIR filters:

- IIR filters are computationally efficient than FIR filters i.e., IIR filters require less memory and fewer instruction when compared to FIR to implement a specific transfer function.
- The number of necessary multiplications are least in IIR while it is most in FIR.
- IIR filters are made up of poles and zeroes. The poles give IIR filter an ability to realize transfer functions that FIR filters cannot do.
- IIR filters are not necessarily stable, because of their recursive nature it is designer's task to ensure stability, while FIR filters are guaranteed to be stable.
- IIR filters can simulate prototype analog filter while FIR filters cannot.
- Probability of overflow errors is quite high in IIR filters in comparison to FIR filters.
- FIR filters are linear phase as long as $H(z) = H(z^{-1})$ but all stable, realizable IIR filters are not linear phase except for the special cases where all poles of the transfer function lie on the unit circle.

4.5.1 Descriptions

The following IIR filter functions are described.

- Coefficients - multiple of four, Sample processing
- Coefficients - multiple of four, Block processing
- Coefficients - multiple of five, Sample processing
- Coefficients - multiple of five, Block processing

IirBiq_4_16 — IIR Filter, Coefficients - multiple of four, Sample processing

Signature

```
DataS IirBiq_4_16(DataS     X,
                  DataS     *H,
                  DataS     *DLY,
                  int       nBiq
                  );
```

Inputs	X	: Real input value
	H	: Pointer to Coeff-Buffer
	DLY	: Pointer to Delay-Buffer
	nBiq	: Number of Biquads
Output	DLY[2*nBiq]	: Updated delay line is an implicit output - $W_i(n)$ and $W_i(n-1)$ are stored as $W_i(n-1)$ and $W_i(n-2)$ for next sample computation
Return	R	: Output value of the filter (48-bit output value converted to 16-bit with saturation).

Description

The IIR filter is implemented as a cascade of direct form II Biquads. If number of biquads is 'n', the filter order is 2*n. A single sample is processed at a time and output for that sample is returned. The filter operates on 16-bit real input, 16-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary, while the number of coefficients is 4*(number of Biquads). Length of delay line is 2*(number of Biquads). In internal memory Coeff-Buffer can be halfword/ word aligned but in external memory it has to be halfword and *not* word aligned. This ensures that after the scale value is read and the pointer incremented, the starting address of the coefficients is word aligned. Delay-Buffer can be halfword aligned in both internal and external memory.

IirBiq_4_16 IIR Filter, Coefficients - multiple of four, Sample processing (cont'd)

Pseudo code

```
{
    frac16 *W;          //Ptr to Delay-Buffer
    frac64 W64;
    frac64 acc;         //Filter result
    int i,j;
    InScale = *H;       //InScale value is read

    W =DLY;
    H++;                //Ptr to Coefficients
    acc =(frac64) (X * InScale);
                      //Input scaled by InScale and stored in 19Q45 format
    //Biquad loop
    //'n' (in the comments) refers to the current instant
    //Indices i and j of H(i) and W_j in the comments are valid only for
    //the first iteration
    //For subsequent iterations they have to be incremented by 4
    //and 1 respectively

    for(i=0;i<nBiq;i++)
    {
       //W64 in 19Q45
       W64 = acc + ( *(H+2) * (*W) + *(H+3) * (*(W+1)) );
                     //W_1(n) = X(n) + H(3) * W_1(n-1) + H(4) * W_1(n-2)
       //acc in 19Q45
       acc = W64 +(frac64) ( (*H) * (*W)   + (*(H+1)) * (*(W+1)) );
                        //acc = acc + H(1) * W_1(n-1) + H(2) * W_1(n-2)
       *(W+1) = *W;     //Update the  Delay line

       *W =((_frac16 _sat)W64);
                        //Format the delay line value to 16-bit(1Q15)
                        //saturated and store the updated value in memory

       W = W + 2;       //Ptr to W_2(n-1)
       H = H + 4;       //Ptr to H(5)
    }

    R = (frac16 sat)acc;
                        //Format the Filter output to 16-bit (1Q15)
                        //saturated value

    return R;           //Filter Output returned
}
```

IirBiq_4_16 **IIR Filter, Coefficients - multiple of four, Sample processing** (cont'd)

Techniques

- Use of packed data Load/Store
- Use of dual MAC instructions
- Intermediate results stored in a 64-bit register (16 guard bits)
- Filter output converted to 16-bit with saturation
- Instruction ordering provided for zero overhead Load/Store

Assumptions

- Input and output are in 1Q15 format
- Coefficients are in 2Q14 format

Memory Note

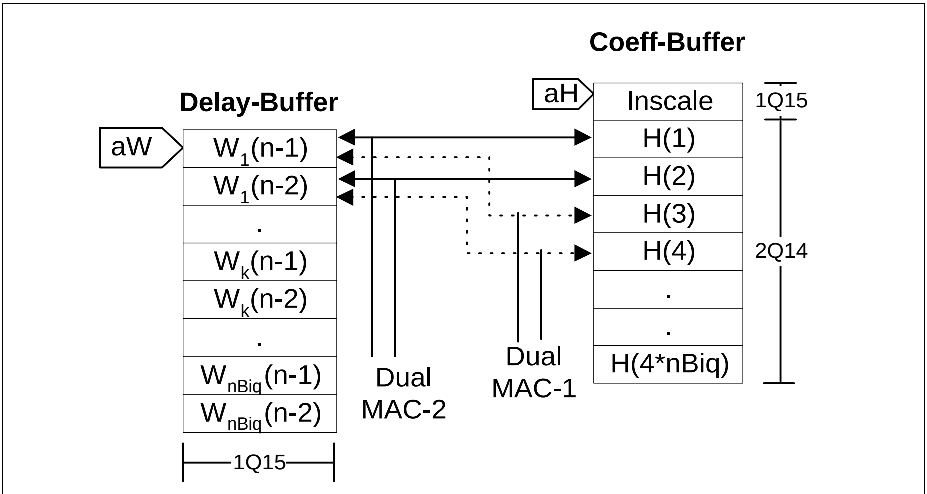


Figure 4-44 IirBiq_4_16

IirBiq_4_16 — IIR Filter, Coefficients - multiple of four, Sample processing (cont'd)

Implementation

The IIR filter implemented as a cascade of biquads has two delay elements per biquad and five coefficients per biquad. In this implementation, the fifth coefficient which scales the current delay line value of the biquad (H_0) is taken to be one. The input is scaled by a constant value, Inscale. Hence, only four coefficients per biquad are considered. The k^{th} biquad uses the coefficients H(4k-3), H(4k-2), H(4k-1) and H(4k), k = 1,2,...nBiq.

This IIR filter routine processes one sample at a time and returns the output for that sample. The input for which the output is to be calculated is sent as an argument to the function.

TriCore's load doubleword instruction loads the four coefficients used in a biquad in one cycle. Load word instruction loads the corresponding two delay line values (W_k(n-1),W_k(n-2)). A dual MAC instruction performs a pair of multiplications and additions to generate the new delay line value for that biquad in one cycle according to the equation

$$\begin{aligned} W_k(n) &= R_{k-1}(n) + H(4k-1) \times W_k(n-1) \\ &\quad + H(4K) \times W_k(n-2) \end{aligned} \qquad [4.55]$$

where, R_0(n) = X(n).

A second Dual MAC instruction uses this delay line value and performs another pair of multiplication and additions to generate the output for that biquad in one cycle according to the equation

$$\begin{aligned} R_k[n] &= W_k(n) + H(4k-3) \times W_k(n-1) \\ &\quad + H(4K-2) \times W_k(n-2) \end{aligned} \qquad [4.56]$$

where, R_{nBiq}(n) = R(n).

W_k(n) and W_k(n-1) of the current sample become W_k(n-1) and W_k(n-2) for the next sample computation. The Delay line is updated accordingly in memory.

IirBiq_4_16 — IIR Filter, Coefficients - multiple of four, Sample processing (cont'd)

Hence a loop executed as many times as there are biquad stages will generate the filter output, with each pass through it yielding the output for that biquad stage.

Load doubleword instruction of TriCore requires word alignment in external memory. If external memory is used, since first value in the Coeff-Buffer is Inscale, followed by the coefficients used in each biquad stage, the address of the Coeff-Buffer should be halfword and *not* word aligned. That is, it should be a multiple of two bytes but not a multiple of four bytes. This ensures that once Inscale (16 bit value) is read and pointer is incremented, the address at which the coefficients begin would be a multiple of four bytes as required by the load double word instruction.

Example

Trilib\Example\Tasking\Filters\IIR\expIirBiq_4_16.c, expIirBiq_4_16.cpp
Trilib\Example\GreenHills\Filters\IIR\expIirBiq_4_16.cpp, expIirBiq_4_16.c
Trilib\Example\GNU\Filters\IIR\expIirBiq_4_16.c

Cycle Count

With DSP Extensions

Pre-kernel	:	5
Kernel	:	$[\mathrm{nBiq} \times 4] + 2$ if nBiq > 1
		$[\mathrm{nBiq} \times 4] + 1$ if nBiq = 1
Post-kernel	:	2+2

Without DSP Extensions

Pre-kernel	:	5
Kernel	:	same as With DSP Extensions

IirBiq_4_16 **IIR Filter, Coefficients - multiple of four, Sample processing** (cont'd)

Post-kernel	:	3+2

Code Size 78 bytes

IirBiqBlk_4_16 IIR Filter, Coefficients - multiple of four, Block processing

Signature

```
void IirBiqBlk_4_16(DataS   *X,
                    DataS   *R,
                    DataS   *H,
                    DataS   *DLY,
                    int     nBiq,
                    int     nX
                    );
```

Inputs

X	:	Pointer to Input-Buffer
R	:	Pointer to Output-Buffer
H	:	Pointer to Coeff-Buffer
DLY	:	Pointer to Delay-Buffer
nBiq	:	Number of Biquads
nX	:	Size of Input-Buffer

Output

DLY[nW]	:	Updated Delay-Buffer values
R[nX]	:	Output-Buffer

Return None

Description

The IIR filter is implemented as a cascade of direct form II Biquads. If number of biquads is 'n', the filter order is 2*n. A block of input is processed at a time and output for every sample is stored in the output buffer. The filter operates on 16-bit real input, 16-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary, while the number of coefficients is 4*(number of Biquads). Length of delay line is 2*(number of Biquads). Coeff-Buffer can be halfword/word aligned in internal memory, but in external memory it should be only halfword and not word aligned. This ensures that after Inscale value is read, the coefficient array is word aligned. Delay-Buffer can be halfword aligned in both internal and external memory.

IirBiqBlk_4_16 IIR Filter, Coefficients - multiple of four, Block processing (cont'd)

Pseudo code

```
{
   frac16 *W;          //Ptr to Delay-Buffer
   frac16 *H0;         //Ptr to InScale
   frac16 *H;          //H0+1 - Ptr to Coefficients
   frac64 W64;
   frac64 acc;         //Filter result
   int i,j;
   InScale = *H0;      //InScale value is read
   H0++;               //Ptr to coefficients

   // Loop for Input-Buffer
   for(j=0;j<nX;j++)
   {
      W =DLY;
      H=H0
      acc =(frac64) (*(X+j) * InScale);
                    //X(n)scaled by InScale and stored in 19Q45 format

      //Biquad loop
      //'n' refers to the current instant
      //Indices i and j of H(i) and W_j  in the comments are
      //valid only for the first iteration. For subsequent iterations
      //they have to be incremented by 4 and 1 respectively

      for(i=0;i<nBiq;i++)
      {
         //W64 in 19Q45
         W64 = acc + ( *(H+2) * (*W) + *(H+3) * (*(W+1)) );
                   //W_1(n) = X(n) + H(3) * W_1(n-1) + H(4) * W_1(n-2)
         //acc in 19Q45
         acc = W64 +(frac64) ( (*H) * (*W)   + (*(H+1)) * (*(W+1)) );
                    //acc = W64 + H(1) * W_1(n-1) + H(2) * W_1(n-2)

         *(W+1) = *W; //Update the Delay line
         *W =((_frac16 _sat)W64);
                    //Format the delay line value to 16-bit(1Q15)
                    //saturated and store the updated value in memory
         W = W + 2;    //Ptr to W_2(n-1)
         H = H + 4;    //Ptr to H(5)
      }
```

IirBiqBlk_4_16 IIR Filter, Coefficients - multiple of four, Block processing (cont'd)

```
        (R+j) =((_frac16 _sat)acc);
                        //Format the Filter output to 16-bit (1Q15)
                        //saturated value and store in output buffer
    }
}
```

Techniques

- Use of packed data Load/Store
- Use of dual MAC instructions
- Intermediate results stored in a 64-bit register (16 guard bits)
- Filter output converted to 16-bit with saturation
- Instruction ordering provided for zero overhead Load/Store

Assumptions

- Input and output are in 1Q15 format
- Coefficients are in 2Q14 format

IirBiqBlk_4_16 IIR Filter, Coefficients - multiple of four, Block processing (cont'd)

Memory Note

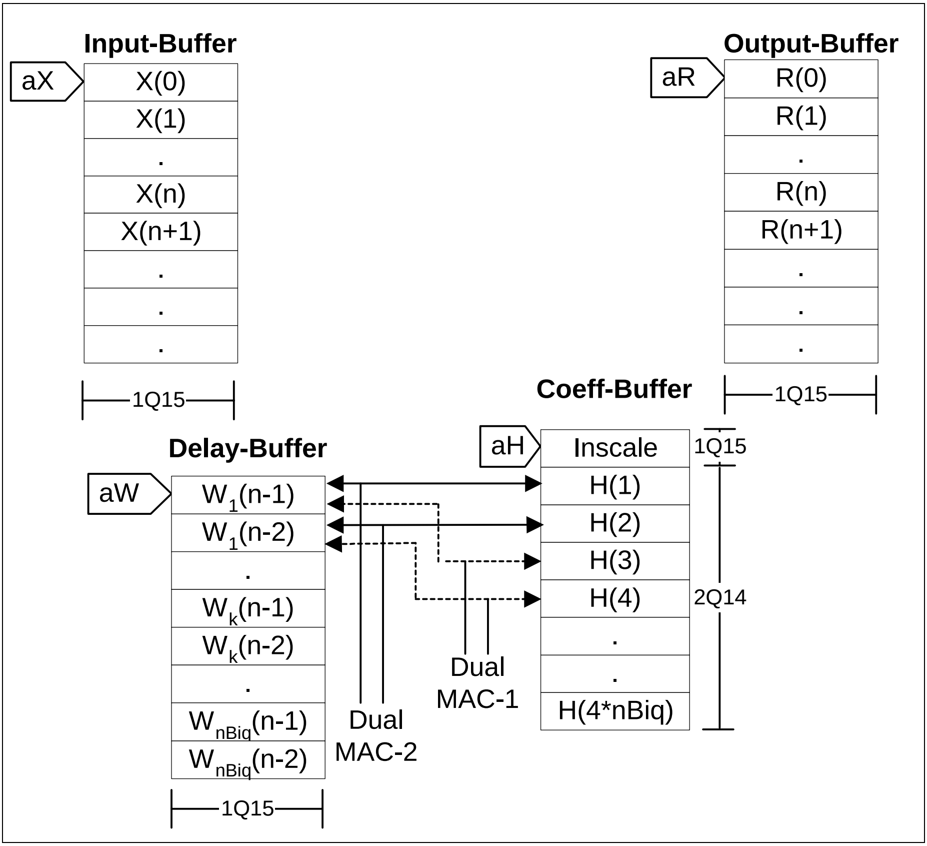


Figure 4-45 IirBiqBlk_4_16

IirBiqBlk_4_16 — IIR Filter, Coefficients - multiple of four, Block processing (cont'd)

Implementation

This IIR filter routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Implementation details are same as that of IirBiq_4_16. The difference is than an additional loop is needed to calculate the output for every sample in the buffer. Hence, this loop is repeated as many times as the size of the input buffer.

Example

Trilib\Example\Tasking\Filters\IIR\expIirBiqBlk_4_16.c, expIirBiqBlk_4_16.cpp
Trilib\Example\GreenHills\Filters\IIR \expIirBiqBlk_4_16.cpp, expIirBiqBlk_4_16.c
Trilib\Example\GNU\Filters\IIR\expIirBiqBlk_4_16.c

Cycle Count

Pre-loop	:	1
Loop	:	$nX \times \{7 + [nBiq \times 4] + 4\} + 1 + 2$
Post-loop	:	0+2

Code Size

98 bytes

IirBiq_5_16 IIR Filter, Coefficients - multiple of five, Sample processing

Signature

```
DataS IirBiq_5_16(DataS    X,
                  DataS    *H,
                  DataS    *DLY,
                  int      nBiq
                  );
```

Inputs	X	:	Real input value
	H	:	Pointer to Coeff-Buffer
	DLY	:	Pointer to Delay-Buffer
	nBiq	:	Number of Biquads
Output	DLY[nW]	:	Updated delay line is an implicit output - $W_i(n)$ and $W_i(n-1)$ are stored as $W_i(n-1)$ and $W_i(n-2)$ for next sample computation
Return	R	:	Output value of the filter(48-bit output value converted to 16-bit with saturation).

Description

The IIR filter is implemented as a cascade of direct form II Biquads. If number of biquads is 'n', the filter order is 2*n. A single sample is processed at a time and output for that sample is returned. The filter operates on 16-bit real input, 16-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary, while the number of coefficients is 5*(number of Biquads). Length of delay line is 2*(number of Biquads). Coeff-Buffer and Delay-Buffer are halfword aligned in both internal and external memory.

IirBiq_5_16 IIR Filter, Coefficients - multiple of five, Sample processing (cont'd)

Pseudo code

```
{
   frac16 *W;          //Ptr to Delay-Buffer
   frac16 W16;
   frac64 W64;
   frac64 HW64;
   frac64 acc;         //Filter result
   int i,j;

   acc =(frac64) (X); //Input stored in 19Q45 format
   //Biquad loop.
   //'n' refers to the current instant
   //Indices i and j of H(i) and W_j  in the comments are valid only
   //for the first iteration. For subsequent iterations they
   // have to be incremented by 5 and 1 respectively
   //
   for(i=0;i<nBiq;i++)
   {
      //W64 in 19Q45
      W64 = acc + ( *(H+3) * (*W) + *(H+4) * (*(W+1)) );
                      //W_1(n) = acc + H(3) * W_1(n-1) + H(4) * W_1(n-2)
      W16 = (frac16 sat)W64;
                       //Format the delay line value W_1(n) to 16 bit
                       //value with saturation
      //HW64 in 19Q45
      HW64 = (frac64)(W16 * (*H));
                       //HW64 = H(0) * W_1(n)
      //acc in 19Q45
      acc = HW64 +(frac64) (*(H+1) * (*W)   + (*(H+2)) * (*(W+1)));
                 //acc = H(0) * W_1(n)+ H(1) * W_1(n-1) +  H(2) * W_1(n-2)
      *(W+1) = *W;     //update the delay line
      *W = W16;        //update the delay line
      W = W + 2;       //Ptr to W_2(n-1)
      H = H + 4;       //Ptr to H(5)
   }
   R =(frac16 sat)acc);
                       //Format the Filter output to 16-bit (1Q15)
                       //saturated value
}
```

IirBiq_5_16 **IIR Filter, Coefficients - multiple of five, Sample processing** (cont'd)

Techniques

- Use of packed data Load/Store
- Use of dual MAC instructions
- Intermediate results stored in a 64-bit register (16 guard bits)
- Filter output converted to 16-bit with saturation
- Instruction ordering provided for zero overhead Load/Store

Assumptions

- Inputs and outputs are in 1Q15 format
- Coefficients are in 2Q14 format

Memory Note

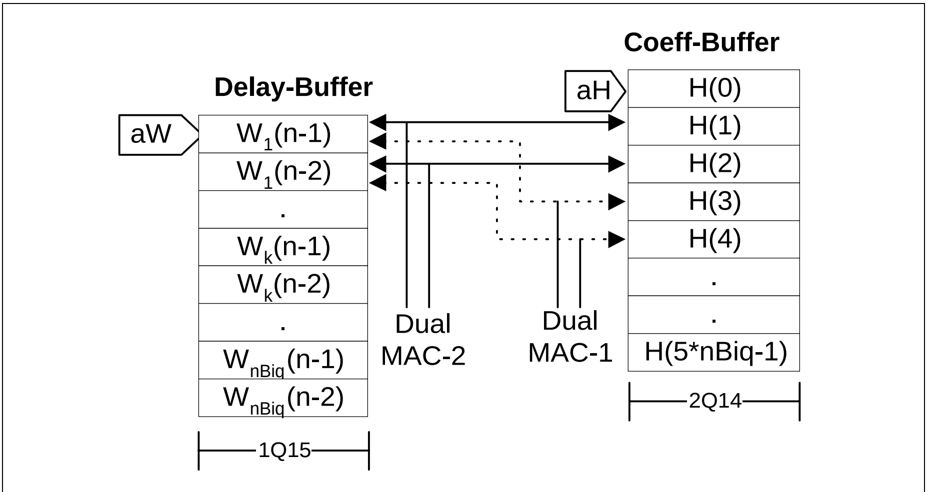


Figure 4-46 IirBiq_5_16

IirBiq_5_16 — IIR Filter, Coefficients - multiple of five, Sample processing (cont'd)

Implementation

In this implementation, there are five coefficients per biquad. The k[th] biquad uses the coefficients H(5k-5), H(5k-4), H(5k-3), H(5k-2) and H(5k-1), k=1,2,.....nBiq.

To perform two multiplication in one cycle using dual MAC, the values should be packed in one register. Hence, H(5k-4), H(5k-3) and H(5k-2), H(5k-1) are loaded in one cycle each using load word instruction. H(5k-5) is loaded separately using load halfword instruction.

The first dual MAC instruction performs a pair of multiplications and additions to generate the new delay line value for that biquad in one cycle according to the equation

$$\begin{aligned} W_k(n) &= R_{k-1}(n) + H(5k-2) \times W_k(n-1) \\ &\quad + H(5K-1) \times W_k(n-2) \end{aligned} \quad [4.57]$$

where, $R_0(n) = X(n)$.

This delay line value is multiplied by H(5k-5).

The second dual MAC uses the above result and performs another pair of multiplication and additions to generate the output for that biquad according to the equation

$$\begin{aligned} R_k[n] &= H(5k-5) \times W_k(n) + H(5k-4) \times W_k(n-1) \\ &\quad + H(5K-3) \times W_k(n-2) \end{aligned} \quad [4.58]$$

where, $R_{nBiq}(n) = R(n)$.

$W_k(n)$ and $W_k(n-1)$ of the current sample become $W_k(n-1)$ and $W_k(n-2)$ for the next sample computation. The Delay line is updated accordingly in memory.

Hence a loop executed as many times as there are biquad stages will generate the filter output, with each pass through it yielding the output for that biquad stage.

IirBiq_5_16 — IIR Filter, Coefficients - multiple of five, Sample processing (cont'd)

Example

Trilib\Example\Tasking\Filters\IIR\expIirBiq_5_16.c, expIirBiq_5_16.cpp
Trilib\Example\GreenHills\Filters\IIR\expIirBiq_5_16.cpp, expIirBiq_5_16.c
Trilib\Example\GNU\Filters\IIR\expIirBiq_5_16.c

Cycle Count

With DSP Extensions

Pre-kernel	:	4
Kernel	:	$[nBiq \times 7] + 2$ if nBiq > 1
		$[nBiq \times 7] + 1$ if nBiq = 1
Post-kernel	:	2+2

Without DSP Extensions

Pre-kernel	:	4
Kernel	:	same as With DSP Extensions
Post-kernel	:	3+2

Code Size 92 bytes

IirBiqBlk_5_16 — IIR Filter, Coefficients - multiple of five, Block processing

Signature

```
void IirBiqBlk_5_16(DataS   *X,
                    DataS   *R,
                    DataS   *H,
                    DataS   *DLY,
                    int     nBiq,
                    int     nX
                    );
```

Inputs

X	:	Pointer to Input-Buffer
R	:	Pointer to Output-Buffer
H	:	Pointer to Coeff-Buffer
DLY	:	Pointer to Delay-Buffer
nBiq	:	Number of Biquads
nX	:	Size of Input-Buffer

Output

DLY[nW]	:	Updated Delay-Buffer values
R[nX]	:	Output-Buffer

Return None

Description

The IIR filter is implemented as a cascade of direct form II Biquads. A block of input is processed at a time and output for every sample is stored in the output buffer. The filter operates on 16-bit real input, 16-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary, while the number of coefficients is 5*(number of Biquads). Length of delay line is 2*(number of biquads). Both Coeff-Buffer and Delay-Buffer are halfword aligned.

IirBiqBlk_5_16 IIR Filter, Coefficients - multiple of five, Block processing (cont'd)

Pseudo code

```
{
   frac16 *W;          //Ptr to Delay-Buffer
   frac16 *H0;         //Ptr to Coeff-Buffer
   frac16 W16;
   frac64 W64;
   frac64 HW64;
   frac64 acc;         //Filter result
   int i,j;

   //Loop for Input-Buffer
   for(j=0;j<nX;j++)
   {
      W =DLY;
      H=H0;            //Ptr to coefficients initialized
      acc =(frac64) *(X+j);
                       //X(n) stored in 19Q45 format
      //Biquad loop
      //'n' refers to the current instant
      //Indices i and j of H(i) and W_j  in the comments are valid
      //only for the first iteration. For subsequent iterations
      //they have to be incremented by 5 and 1 respectively
      for(i=0;i<nBiq;i++)
      {
         //W64 in 19Q45
         W64 = acc + ( *(H+3) * (*W) + (*(H+4)) * (*(W+1)) );
                      //W_1(n) = acc + H(3) * W_1(n-1) + H(4) * W_1(n-2)
         W16 = (frac16 sat)W64;
                       //Format the delay line value W_1(n) to 16 bit
                       //value with saturation
         //HW64 in 19Q45
         HW64 = (frac64)(W16 * (*H));
                       // HW64 = H(0) * W_1(n)
         //acc in 19Q45
        acc = HW64 +(frac64) ( (*(H+1) * (*W)   + (*(H+2)) * (*(W+1)) );
                  //acc = H(0) * W_1(n)+ H(1) * W_1(n-1) +  H(2) * W_1(n-2)
         *(W+1) = *W; //update the delay line
         *W = W16;    //update the delay line
         W = W + 2;   //Ptr to W_2(n-1)
         H = H + 4;   //Ptr to H(5)
      }
```

IirBiqBlk_5_16 IIR Filter, Coefficients - multiple of five, Block processing (cont'd)

```
        *(R+j) =((_frac16 _sat)acc);
                        //Format the Filter output to 16-bit (1Q15)
                        //saturated value and store in output buffer

    }
}
```

Techniques

- Use of packed data Load/Store.
- Use of dual MAC instructions.
- Intermediate results stored in a 64-bit register(16 guard bits)
- Filter output converted to 16-bit with saturation
- Instruction ordering provided for zero overhead Load/Store

Assumptions

- Input and output are in 1Q15 format
- Coefficients are in 2Q14 format

IirBiqBlk_5_16 IIR Filter, Coefficients - multiple of five, Block processing (cont'd)

Memory Note

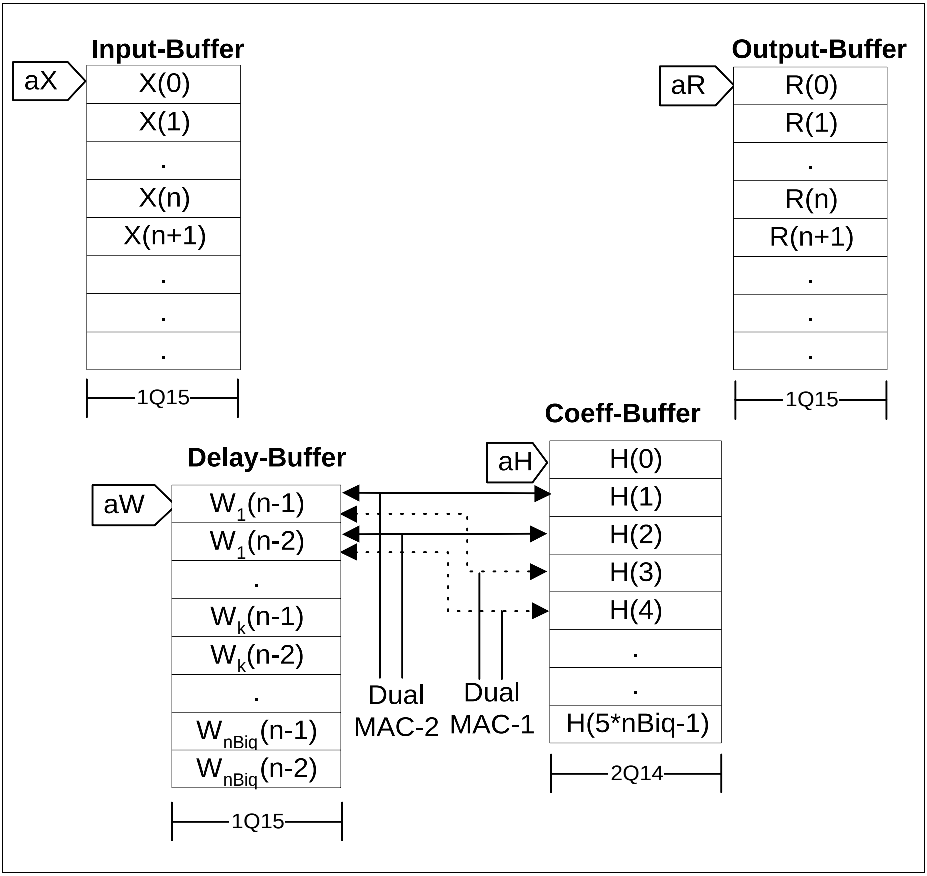


Figure 4-47 IirBiqBlk_5_16

IirBiqBlk_5_16 — IIR Filter, Coefficients - multiple of five, Block processing (cont'd)

Implementation

This IIR filter routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Implementation details are same as that of IirBiq_5_16. The difference is that an additional loop is needed to calculate the output for every sample in the buffer. Hence, this loop is repeated as many times as the size of the input buffer.

Example

Trilib\Example\Tasking\Filters\IIR\expIirBiqBlk_5_16.c, expIirBiqBlk_5_16.cpp
Trilib\Example\GreenHills\Filters\IIR \expIirBiqBlk_5_16.cpp, expIirBiqBlk_5_16.c
Trilib\Example\GNU\Filters\IIR\expIirBiqBlk_5_16.c

Cycle Count

Pre-loop	:	1
Loop	:	$nX \times \{6 + [nBiq \times 7] + 4\} + 1 + 2$
Post-loop	:	0+2

Code Size

112 bytes

4.6 Adaptive Digital Filters

An adaptive filter adapts to changes in its input signals automatically.

Conventional linear filters are those with fixed coefficients.These can extract signals where the signal and noise occupy fixed and separate frequency bands. Adaptive filters are useful when there is a spectral overlap between the signal and noise or if the band occupied by the noise is unknown or varies with time. In an adaptive filter, the filter characteristics are variable and they adapt to changes in signal characteristics. The coefficients of these filters vary and cannot be specified in advance.

The self-adjusting nature of adaptive filters is largely used in applications like telephone echo cancelling, radar signal processing, equalization of communication channels etc.

Adaptive filters with the LMS (Least Mean Square) algorithm are the most popular kind. The basic concept of an LMS adaptive filter is as follows.

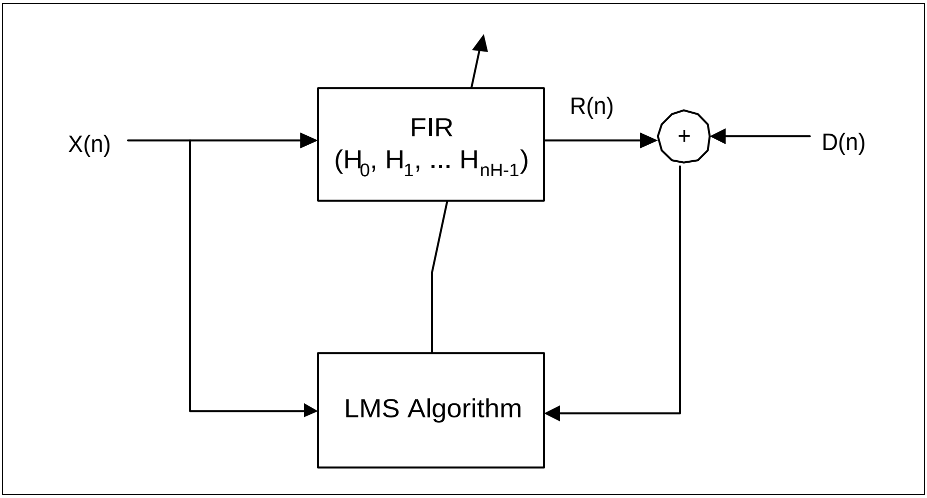


Figure 4-48 Adaptive filter with LMS algorithm

The filter part is an N-tap filter with coefficients H_0, H_1,..., H_{nH-1}, whose input signal is X(n) and output is R(n). The difference between the actual output R(n) and a desired output D(n), gives an error signal

$$\mathrm{Err}(n) = D(n) - R(n) \qquad [4.59]$$

The algorithm uses the input signal X(n) and the error signal Err(n) to adjust the filter coefficients H_0, H_1,..., H_{nH-1}, such that the difference, Err(n) is minimized on a criterion. The LMS algorithm uses the minimum mean square error criterion

$$\min_{H0,\ H1,\ldots,\ HnH-1} E(Err^2(n)) \qquad [4.60]$$

Where E denotes statistical expectation.The algorithm of a delayed LMS adaptive filter is mathematically expressed as follows.

$$\begin{aligned} R(n) &= H_{n-1}(0) \times X(n) + H_{n-1}(1) \times X(n-1) + H_{n-2}(2) \times X(n-2) + \ldots \\ &\quad + H_{n-1}(nH-1) \times X(n-nH+1) \end{aligned} \qquad [4.61]$$

$$H_n(k) = H_{n-1}(k) + X(n-k) \times \mu \times Err_{n-1} \qquad [4.62]$$

$$Err_n = D(n) - R(n) \qquad [4.63]$$

where μ >0 is a constant called step-size. Note that the filter coefficients are time varying. $H_n(i)$ denotes the value of the i-th coefficient at time n. The algorithm has three stages.

1. The filter output R(n) is produced.
2. The error value from previous iteration is read and coefficients are updated.
3. The expected value is read, error is calculated and stored in memory.

Step-size μ controls the convergence of the filter coefficients to the optimal (or stationary) state. The larger the μ value, faster the convergence of the adaptation. On the other hand, a large value of μ also leads to a large variation of $H_n(i)$ (a bad accuracy) and thus a large variation of the output error (a large residual error). Therefore, the choice of μ is always a trade-off between fast convergence and high accuracy. μ must not be larger than a certain threshold. Otherwise, the LMS algorithm diverges.

4.6.1 Delayed LMS algorithm for an adaptive real FIR

Delayed LMS algorithm for an adaptive real FIR filter can be represented by the following mathematical equation.

$$R(n) = \sum_{K=0}^{nH-1} H_{n-1}(k) \times X(n-k) \qquad [4.64]$$

$$H_n(k) = H_{n-1}(k) + X(n-k) \times U \times Err_{n-1} \qquad [4.65]$$

$$Err_n = D(n) - R(n) \qquad [4.66]$$

where,

R(n)	:	output sample of the filter at index n
X(n)	:	input sample of the filter at index n
D(n)	:	expected output sample of the filter at index n
$H_n(0), H_n(1)$,..	:	filter coefficients at index n
nH	:	filter order (number of coefficients)
Err_n	:	error value at index n which will be used to update coefficients at index n+1

4.6.2 Delayed LMS algorithm for an adaptive Complex FIR

Delayed LMS algorithm for an adaptive Complex FIR filter can be represented by the following mathematical equations.

$$Rr(n) = \sum_{K=0}^{nH-1} [Hr_{n-1}(k) \times Xr(n-k) - Hi_{n-1}(k) \times Xi(n-k)] \qquad [4.67]$$

$$Ri(n) = \sum_{K=0}^{nH-1} [Hr_{n-1}(k) \times Xi(n-k) + Hi_{n-1}(k) \times Xr(n-k)] \qquad [4.68]$$

$$\begin{aligned} Hr_n(k) &= Hr_{n-1}(k) \\ &+ U \times (Xr(n-k) \times Errr_{n-1} - Xi(n-k) \times Erri_{n-1}) \end{aligned} \qquad [4.69]$$

$$\begin{aligned} Hi_n(k) &= Hi_{n-1}(k) \\ &+ U \times (Xr(n-k) \times Erri_{n-1} + Xi(n-k) \times Errr_{n-1}) \end{aligned} \qquad [4.70]$$

$$Errr_n = Dr(n) - Rr(n) \qquad [4.71]$$

$$\mathrm{Erri}_n = \mathrm{Di}(n) - \mathrm{Ri}(n) \quad [4.72]$$

where,

Rr(n)	:	Real output sample of the filter at index n
Ri(n)	:	Imag output sample of the filter at index n
Xr(n)	:	Real input sample of the filter at index n
Xi(n)	:	Imag input sample of the filter at index n
Dr(n)	:	Real desired output sample of the filter at index n
Di(n)	:	Imag desired output sample of the filter at index n
$Hr_n(0),Hr_n(1),..$	:	filter coefficients (real) at index n
$Hi_n(0),Hi_n(1),..$	:	filter coefficients (imag) at index n
nH	:	filter order (number of coefficients)
Err_n	:	error value at index n which will be used to update coefficients at index n+1

4.6.3 Descriptions

The following are adaptive FIR filter functions with 16 bit input and 16 bit coefficients.

- Real, Coefficients - multiple of four, Sample processing
- Real, Coefficients - multiple of four, Block processing
- Complex, Coefficients - multiple of four, Sample processing
- Complex, Coefficients - multiple of four, Block processing

The following are mixed adaptive FIR filter functions with 16 bit input and 32 bit coefficients.

- Real, Coefficients - multiple of two, Sample Processing
- Real, Coefficients - multiple of two, Block Processing

Dlms_4_16 Adaptive FIR Filter, Coefficients - multiple of four, Sample Processing

Signature

```
DataS Dlms_4_16(DataS      X,
                DataS      *H,
                cptrDataS  *DLY,
                DataS      D,
                DataS      *Err,
                DataS      U
                );
```

Inputs

X	:	Real Input Value
H	:	Pointer to Coeff-Buffer
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct
D	:	Real expected value
Err	:	Pointer to Error value
U	:	Step size

Output

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer
H(nH)	:	Modified Coeff-Buffer

Return

R	:	Output value of the filter (48-bit output value converted to 16-bit with saturation)

Description

Delayed LMS algorithm implemented for adaptive FIR filter, FIR filter transversal structure (direct form), Single sample processing, 16-bit fractional input, coefficients and output data format, Optimal implementation, requires filter order to be multiple of four.

Dlms_4_16 Adaptive FIR Filter, Coefficients - multiple of four, Sample Processing (cont'd)

Pseudo code

```
{
   frac64 acc;        //filter result
   frac16 circ *aDLY = &DLY;
                      //ptr to Circ-ptr of Delay-Buffer
   int j;
   //Error value multiplied by step size
   uerr = (frac16 rnd)(*Err * U);
   //store input value in Delay-Buffer at the position
   //of the oldest value
   *DLY = X;
   acc = 0;
   k = 0;
   //tap loop
  //The index i and j of H_n-1(i) and X(j) in the comments are valid only
   //for the first iteration.For each next iteration it has to be
   //incremented and decremented by 4 respectively.
   for (j=0; j<nH/4; j++)
   {
      acc = acc + (frac64)[(*(H+k) * (*(DLY + k))
                           +(*(H+k+1)) * (*(DLY+k+1))];
                      //acc = acc + X(n)* H_n-1(0) + X(n-1) * H_n-1(1)
      acc = acc + (frac64)[(*(H+k+2) * (*(DLY+k+2))+
                          (*(H+k+3)) * (*(DLY+k+3));
                      //acc = X(n-2) * (H_n-1(2) + X(n-3) * H_n-1(3)
      //coefficient update
      *(H+k) = (frac16 sat rnd)((*(H+k)) + uerr * (*(DLY+k)));
      *(H+k+1) = (frac16 sat rnd)((*(H+k+1)) + uerr * (*(DLY+k+1)));
      *(H+k+2) = (frac16 sat rnd))(*(H+k+2) + uerr * (*(DLY+k+2)));
      *(H+k+3) = (frac16 sat rnd)((*(H+k+3)) + uerr * (*(DLY+k+3)));

      k = k + 4;
    }
    //Set DLY.index to the oldest value in Delay-Buffer
    DLY--;
    aDLY = *DLY;

    //format the filter output from 48-bit to 16-bit saturated value
    R = (frac16 sat)acc;
    //calculate error for the current output
    *Err = D - R;
    return R;
}
```

Dlms_4_16 — Adaptive FIR Filter, Coefficients - multiple of four, Sample Processing (cont'd)

Techniques

- Loop unrolling, four taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular-buffer
- Use of dual MAC instructions
- Intermediate result stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Filter size must be multiple of four
- Inputs, outputs, coefficients are in 1Q15 format
- Delay-Buffer is in Internal Memory

Memory Note

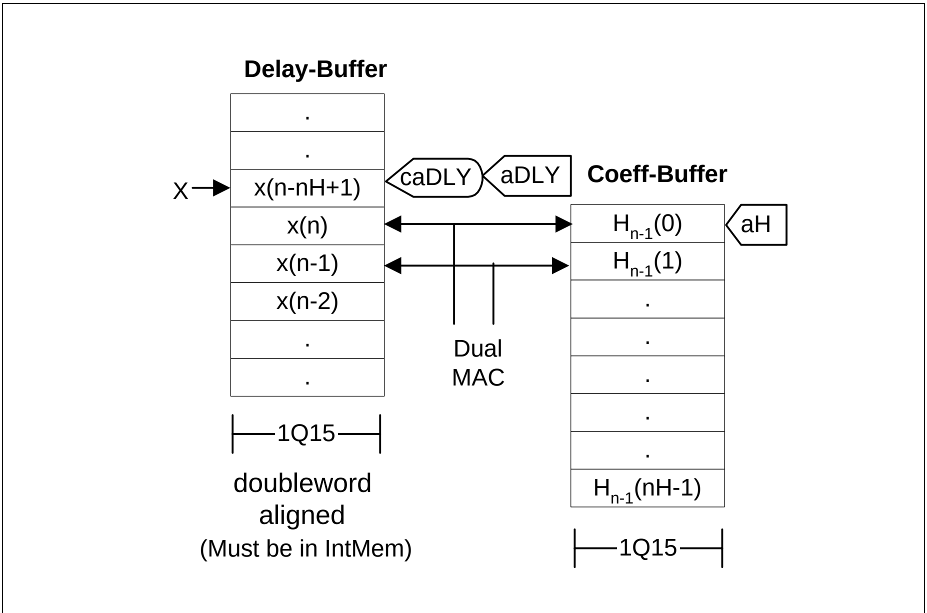


Figure 4-49 Dlms_4_16

Dlms_4_16 Adaptive FIR Filter, Coefficients - multiple of four, Sample Processing (cont'd)

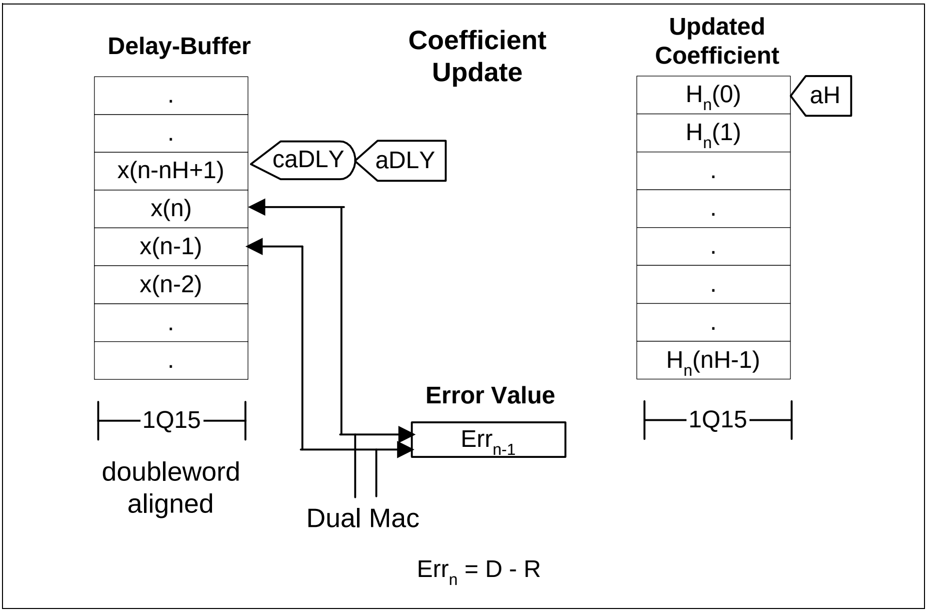


Figure 4-50 Dlms_4_16 Coefficient update

Dlms_4_16 Adaptive FIR Filter, Coefficients - multiple of four, Sample Processing (cont'd)

Implementation

LMS algorithm has been used to realize an adaptive FIR filter. The implemented filter is a Delayed LMS adaptive filter. That is, the updation of coefficients in the current instant is done using the error in the previous output.

The FIR filter is implemented using transversal structure and is realized as a tapped delay line.

This routine processes one sample at a time and returns output of that sample. The input for which the output is to be calculated is sent as an argument to the function.

TriCore's load doubleword instruction loads four delay line values and four coefficients in one cycle. Dual MAC instruction performs a pair of multiplications and additions according to the equation

$$\mathrm{acc} = \mathrm{acc} + X(n-k) \cdot H_{n-1}(k) + X(n-(k-1)) \cdot H_{n-1}(k+1) \tag{4.73}$$

where, k=0,1,...., nH-1.

The coefficient is updated using error from the previous output, i.e., err_{n-1}. As $H_{n-1}(0)$ and $H_{n-1}(1)$ are packed in one register, one dual MAC instruction can be used to update both the coefficients in one cycle. TriCore provides a dual MAC instruction which performs packed multiplication and addition with rounding and saturation. Hence the two coefficients are updated at a time and packed in one register according to the equation

$$\begin{aligned} H_n(k) &= H_{n-1}(k) + X(n-k) \cdot Err_{n-1} \\ H_n(k+1) &= H_{n-1}(k+1) + X(n-(k-1)) \cdot Err_{n-1} \end{aligned} \tag{4.74}$$

where, k=0,1,...,nH-1.

Dlms_4_16 Adaptive FIR Filter, Coefficients - multiple of four, Sample Processing (cont'd)

Thus by using four dual MAC operations, four coefficients are used and updated on a single pass through the loop. This brings down the loop count by a factor of four. For the sake of optimization one set of four dual MACs are performed outside the loop. Hence loop is unrolled. This implies it is executed (nH/4-1) times. For delay line, circular addressing mode is used which helps in efficient delay update. The size of the circular delay buffer is equal to the filter order, i.e., the number of coefficients. Circular buffer needs doubleword alignment and to use load doubleword instruction, size of the buffer should be multiple of eight bytes. This implies that the coefficients should be multiple of four.

Note: To use load doubleword instruction for delay line, the delay-buffer should be in internal memory only.

Example

Trilib\Example\Tasking\Filters\Adaptive\expDlms_4_16.c, expDlms_4_16.cpp
Trilib\Example\GreenHills\Filters\Adaptive \expDlms_4_16.cpp, expDlms_4_16.c
Trilib\Example\GNU\Filters\Adaptive\expDlms_4_16.c

Cycle Count

With DSP Extensions

Pre-kernel : 12

Kernel : $\left[\frac{nH}{4} - 1\right] \times 4 + 2$

if TapLoopCount > 1

$\left[\frac{nH}{4} - 1\right] \times 4 + 1$

if TapLoopCount = 1

Post-kernel : 4+2

Dlms_4_16 **Adaptive FIR Filter, Coefficients - multiple of four, Sample Processing** (cont'd)

	Without DSP Extensions		
	Pre-kernel	:	12
	Kernel	:	same as With DSP Extensions
	Post-kernel	:	5+2
Code Size	130 bytes		

DlmsBlk_4_16 Adaptive FIR Filter, Coefficients - multiple of four, Block Processing

Signature

```
void DlmsBlk_4_16(DataS       *X,
                  DataS       *R,
                  cptrDataS    H,
                  cptrDataS   *DLY,
                  int          nX,
                  DataS       *D,
                  DataS       *Err,
                  DataS        U
                  );
```

Inputs

X	:	Pointer to Input-Buffer
R	:	Pointer to Output-Buffer
H	:	With DSP Extension - circular pointer of Coeff-Buffer of size nH Without DSP Extension - circ-Struct. Whose members are base address, size and index
DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct
D	:	Pointer to Desired-Output-Buffer
Err	:	Pointer to Error value
U	:	Step size

Output

DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer
H(nH)	:	Modified Coeff-Buffer
R(nX)	:	Output-Buffer

Return None

Description

Delayed LMS algorithm implemented for adaptive FIR filter, FIR filter transversal structure (direct form), Block processing, 16-bit fractional input, coefficients and output data format, Optimal implementation, requires filter order to be multiple of four.

DlmsBlk_4_16 Adaptive FIR Filter, Coefficients - multiple of four, Block Processing (cont'd)

Pseudo code

```
{
   frac64 acc;         //filter result
   frac16 circ *aDLY = &DLY;
                       //ptr to Circ-ptr of Delay-Buffer
   int i, j;
   //loop for input buffer
   for (i=0; i<nX; i++)
   {
      //Error value multiplied by step size
      uerr = (frac16 rnd)(*Err * U);
      //store input value in Delay-Buffer at the position
      //of the oldest value
      *DLY = *X++;
      acc = 0;
      k = 0;
      //tap loop
      for (j=0; j<nH/4; j++)
      {
         acc = acc + (frac64)[(*(H+k) * (*(DLY + k))
                           +(*(H+k+1)) * (*(DLY+k+1))];
                       //acc = acc + X(n)* H_n-1(0) + X(n-1) * H_n-1(1)
         acc = acc + (frac64)[(*(H+k+2) * (*(DLY+k+2))+
                           (*(H+k+3)) * (*(DLY+k+3));
                       //acc = X(n-2) * (H_n-1(2) + X(n-3) * H_n-1(3)
         //coefficient update
         *(H+k) = (frac16 sat rnd)((*(H+k)) + uerr * (*(DLY+k)));
         *(H+k+1) = (frac16 sat rnd)((*(H+k+1)) + uerr * (*(DLY+k+1)));
         *(H+k+2) = (frac16 sat rnd))(*(H+k+2) + uerr * (*(DLY+k+2)));
         *(H+k+3) = (frac16 sat rnd)((*(H+k+3)) + uerr * (*(DLY+k+3)));
         k = k + 4;
       }
       //Set DLY.index to the oldest value in Delay-Buffer
       DLY--;
       aDLY = *DLY;
       //format the filter output from 48-bit to 16-bit saturated value
       //and store to Output-Buffer
       *R = (frac16 sat)acc;
       //calculate error for the current output
       *Err = *D++ - *R++;
   }
}
```

DlmsBlk_4_16 Adaptive FIR Filter, Coefficients - multiple of four, Block Processing (cont'd)

Techniques

- Loop unrolling, four taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular-buffer
- Use of dual MAC instructions
- Intermediate result stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Filter size is a multiple of four
- Inputs, outputs, coefficients are in 1Q15 format
- Delay-Buffer is in internal memory

Memory Note

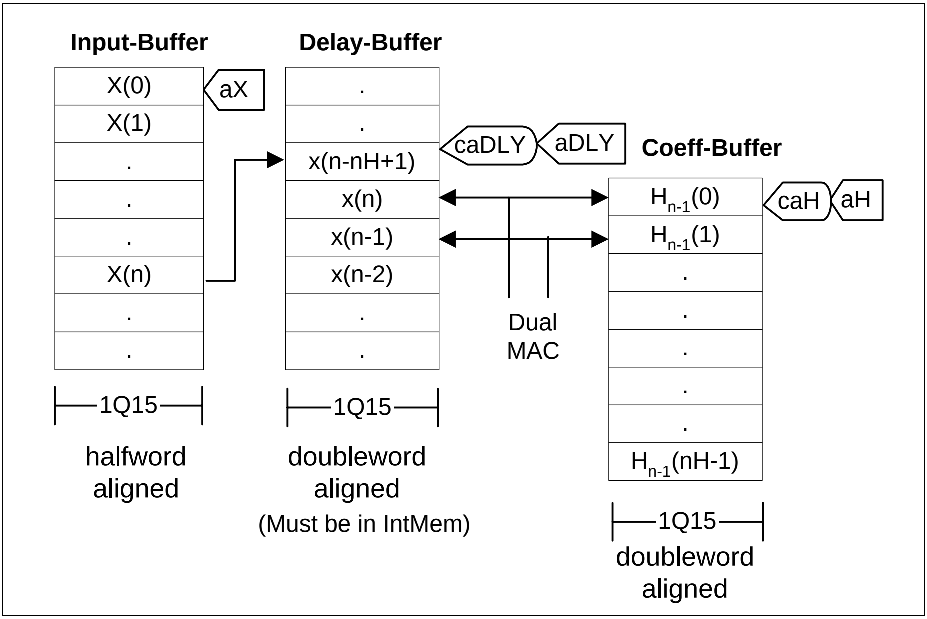


Figure 4-51 DlmsBlk_4_16

DlmsBlk_4_16 Adaptive FIR Filter, Coefficients - multiple of four, Block Processing (cont'd)

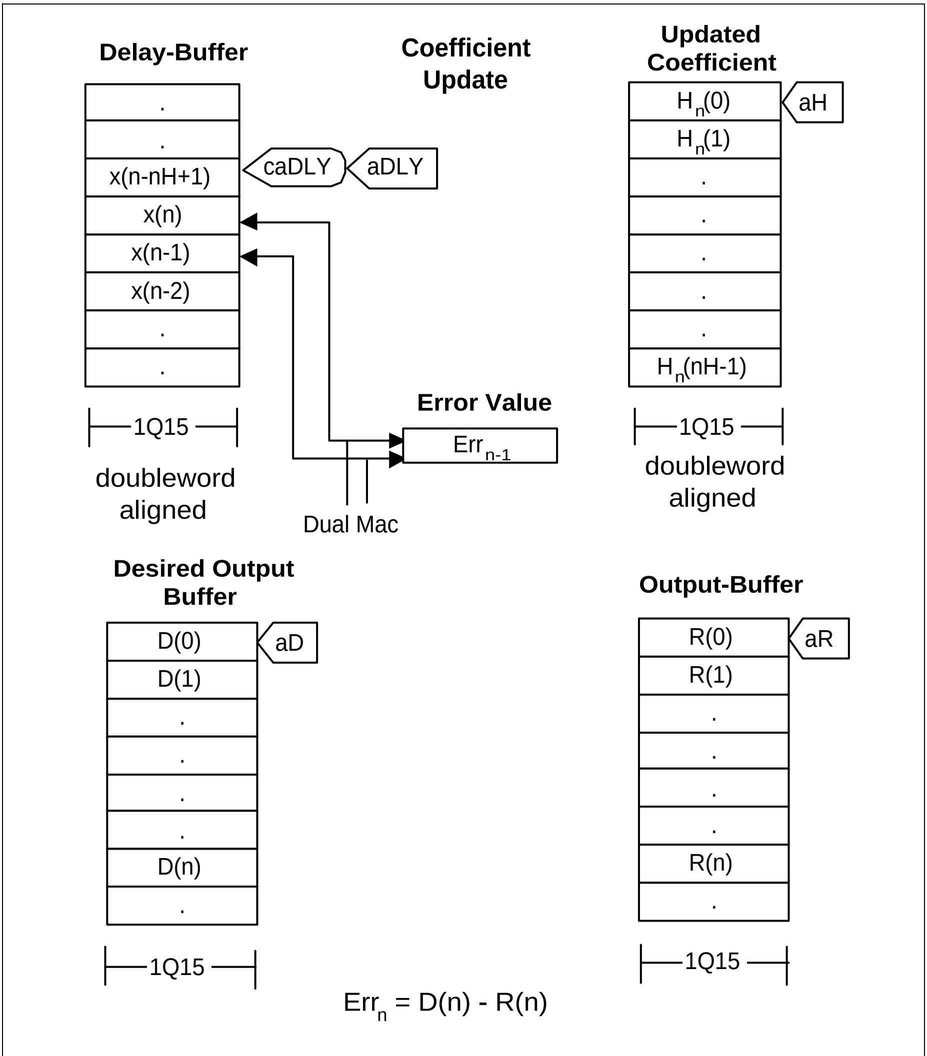


Figure 4-52 DlmsBlk_4_16 Coefficient update

DlmsBlk_4_16 — Adaptive FIR Filter, Coefficients - multiple of four, Block Processing (cont'd)

Implementation

This DLMS routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Implementation details are same as Dlms_4_16, except that the Coeff-Buffer is also circular and needs doubleword alignment. The advantage of using circular buffer for coefficients is efficient pointer update. In this implementation while exiting the tap loop, the first two coefficients are already loaded for the next input value. This helps in saving one cycle in the next sample processing.

Example

Trilib\Example\Tasking\Filters\Adaptive \expDlmsBlk_4_16.c, expDlmsBlk_4_16.cpp
Trilib\Example\GreenHills\Filters\Adaptive \expDlmsBlk_4_16.cpp, expDlmsBlk_4_16.c
Trilib\Example\GNU\Filters\Adaptive \expDlmsBlk_4_16.c

Cycle Count

With DSP Extensions

Pre-loop : 7

Loop : $nX \times \left\{8 + \left[\left(\frac{nH}{4} - 1\right) \times 4 + 6\right]\right\}$

$+1+2$

Post-loop : 1+2

Without DSP Extensions

Pre-loop : 8

Loop : same as With DSP Extensions

DlmsBlk_4_16 **Adaptive FIR Filter, Coefficients - multiple of four, Block Processing** (cont'd)

Post-loop : 1+2

Code Size 166 bytes

CplxDlms_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Sample Processing

Signature

```
DataL CplxDlms_4_16(CplxS      X,
                    DataS      * H,
                    cptrDataS  *DLYr,
                    cptrDataS  *DLYi,
                    CplxS      D,
                    CplxS      *Err,
                    DataS      U
                    );
```

Inputs	X	:	Complex input value
	H	:	Pointer to Cplx-Coeff-Buffer
	DLYr	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer (Real) Without DSP Extension - Pointer to Circ-Struct
	DLYi	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer (Imag) Without DSP Extension - Pointer to Circ-Struct
	D	:	Desired complex value
	Err	:	Pointer to complex Error value
	U	:	Step size
Output	DLYr	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer (Real)
	DLYi	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer (Imag)
	H(nH*2)	:	Modified Coeff-Buffer (Real and Imag)
Return	R	:	Output value of the filter (48-bit output value converted to 16-bit with saturation)

CplxDlms_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Sample Processing (cont'd)

Description

Delayed LMS algorithm implemented for adaptive Complex FIR filter, FIR filter transversal structure (direct form), Single sample processing, 16-bit fractional input, coefficients and output data format, Optimal implementation, requires filter order to be multiple of four.

Pseudo code

```
{
   frac64 accr,acci;  //Filter result
   int i,j,k;
   frac16circ  *aDLYr=&DLYr, *aDLYi=&DLYi;
                    //Ptr to circ-ptr of real and imaginary Delay-Buffer
   //Error value multiplied by step size
   uerrr = (frac16 rnd)(*Errr * U);
   uerri = (frac16 rnd)(*Erri * U);

   //Store input value in Delay-Buffer at the position of the
   //oldest value
   *DLYi = Xi         //Imag part of Input is stored in delay line(imag)
   *DLYr = Xr         //Real part of Input is stored in delay line(real)

   accr = 0.0;
   acci = 0.0;

   k=0;
   //tap loop
   for(j=0; j<nH/2; j++)
   {
     //Filter result
     //Imag
    acci += (frac64)(*(H+k) * (*(DLYi+k)) + (*(H+k+1) * (*(DLYi+k+1)));
                      //acci += Xi(n) * Hr_n-1(0) + Xi(n-1) * Hr_n-1(1)
   acci -= (frac64)(*(H+k+2) * (*(DLYr+k)) + (*(H+k+3) * (*(DLYr+k+1)));
                      //acci += Xr(n) * Hi_n-1(0) + Xr(n-1) * Hi_n-1(1)
     //Real
    accr += (frac64)(*(H+k) * (*(DLYr+k)) + (*(H+k+1) * (*(DLYr+k+1)));
                     //accr += Xr(n) * Hr_n-1(0) +  Xr(n-1) * Hr_n-1(1)
   accr -= (frac64)(*(H+k+2) *(*(DLYi+k)) + (*(H+k+3) * (*(DLYi+k+1)));
                     //accr -= Xi(n) * Hi_n-1(0) +  Xi(n-1) * Hi_n-1(1)
```

CplxDlms_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Sample Processing (cont'd)

```
          //Coefficient update
          //Real_i
          *(H+k) = (frac16 sat rnd)(*(H+k) + (uerrr * (*(DLYr+k)));
                   //Hr_n(0) = Hr_n-1(0) + Xr(n) * Errr_n-1
          *(H+k) = (frac16 sat rnd)(*(H+k) - (uerri * (*(DLYi+k)));
                   //Hr_n(0) -= Xi(n) * Erri_n-1
                   //Real_i+1
        *(H+k+1) = (frac16 sat rnd)(*(H+k+1) + (uerrr * (*DLYr+k+1)));
                   //Hr_n(1) = Hr_n-1(1) + Xr(n-1) * Errr_n-1
        *(H+k+1) = (frac16 sat rnd)(*(H+k+1) - (uerri * (*(DLYi+k+1)));
                   //Hr_n(1) -= Xi(n-1) * Erri_n-1

          //Imag_i
         *(H+k+2) = (frac16 sat rnd)(*(H+k+2) + (uerri * (*(DLYr+k)));
                   //Hi_n(0) = Hi_n-1(0) + Xr(n) * Erri_n-1
         *(H+k+2) = (frac16 sat rnd)(*(H+k+2) + (uerrr * (*(DLYi+k)));
                   //Hi_n(0) += Xi(n) * Errr_n-1
          //Imag_i+1
        *(H+k+3) = (frac16 sat rnd)(*(H+k+3) + (uerri * (*(DLYr+k+1)));
                   //Hi_n(1) = Hi_n-1(1) + Xr(n-1) * Erri_n-1
        *(H+k+3) = (frac16 sat rnd)(*(H+k+3) + (uerrr * (*(DLYi+k+1)));
                   //Hi_n(1) += Xi(n-1) * Errr_n-1

          k=k+4;
       }

    //Set DLYr.index and DLYi.index to the oldest value in Delay-Buffer
      *DLYr--;
      *DLYi--;
      aDLYr = &DLYr;
      aDLYi = &DLYi;

   //Format the real and imaginary parts of the filter output from
   //48-bit to 16-bit saturated values and pack them in the return
   //register (Rr : Ri)

      RLo = (frac16 sat)acci;
      RHi = (frac16 sat)accr;

      //Calculate error in current output
      *Err = D - R;

   }
}
```

CplxDlms_4_16 — Adaptive Complex Filter, Coefficients - multiple of four, Sample Processing (cont'd)

Techniques

- Loop unrolling, four taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular-buffer
- Use of dual MAC instructions
- Intermediate result stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Filter size is a multiple of four
- Inputs, outputs, coefficients are in 1Q15 format

CplxDlms_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Sample Processing (cont'd)

Memory Note

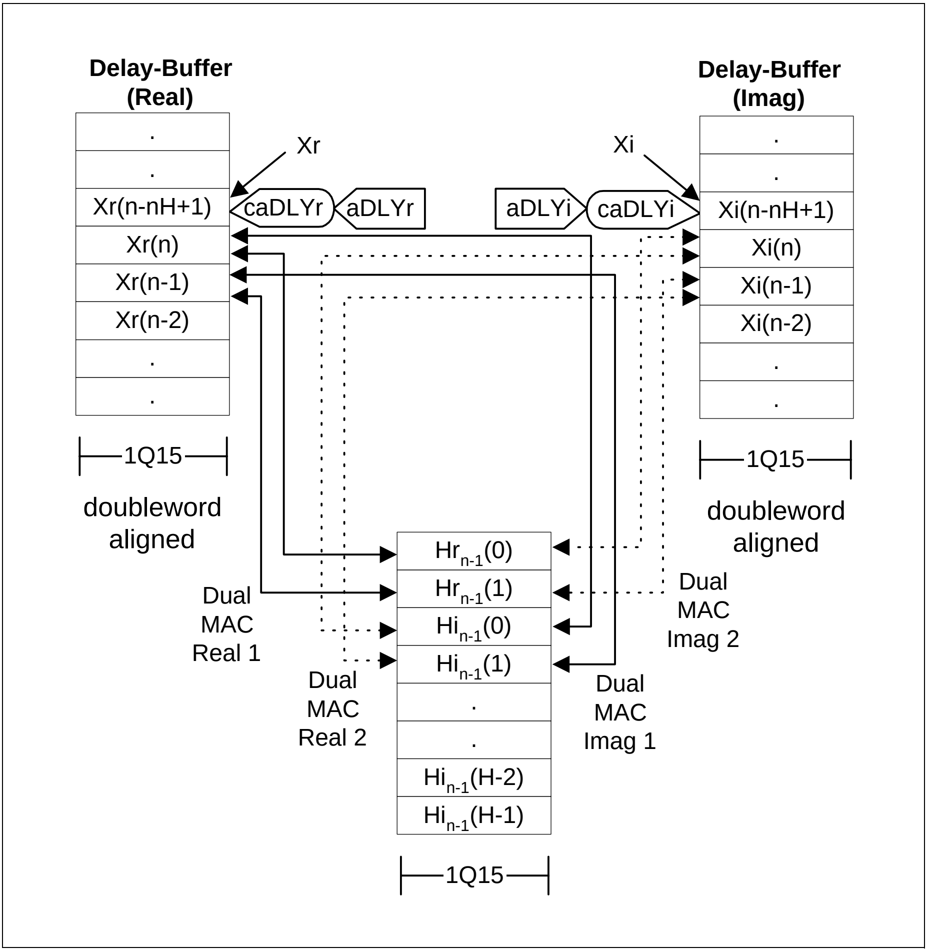


Figure 4-53 CplxDlms_4_16

CplxDlms_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Sample Processing (cont'd)

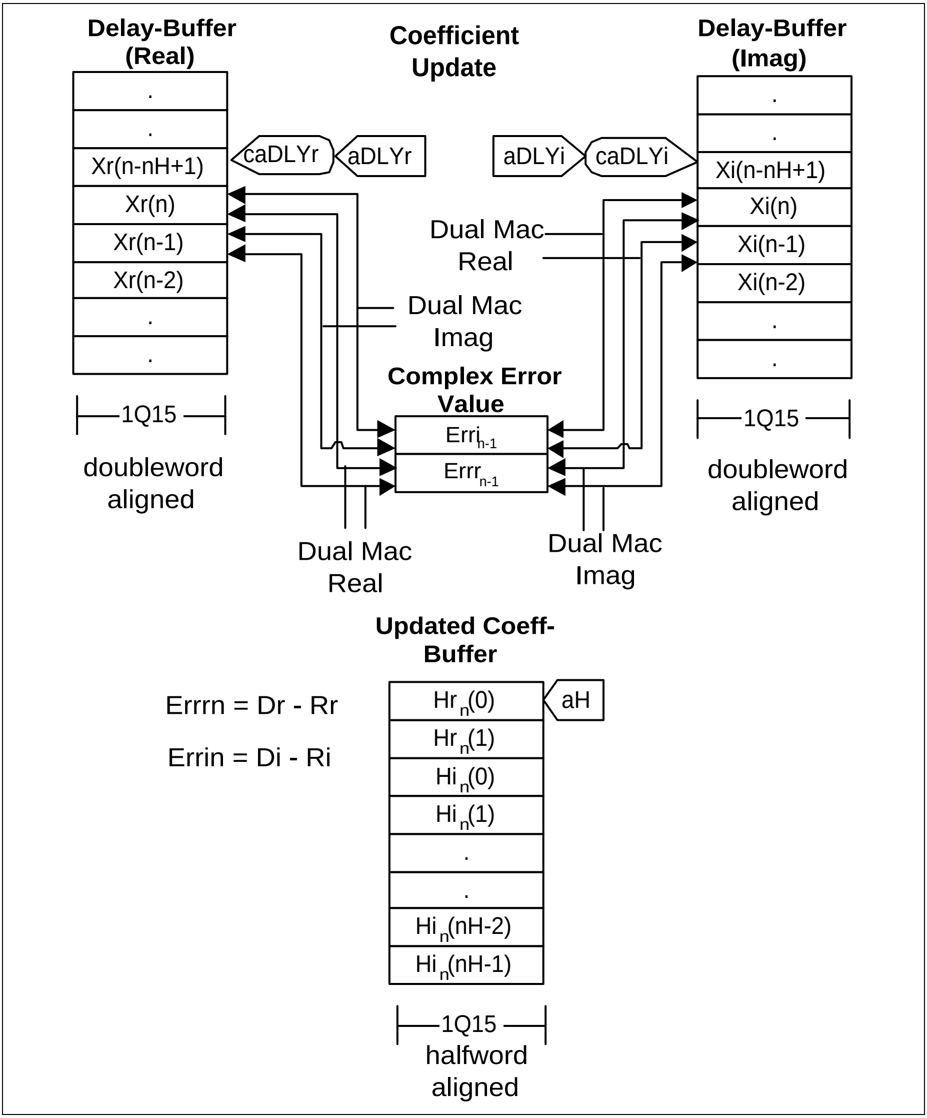


Figure 4-54 CplxDlms_4_16

CplxDlms_4_16 **Adaptive Complex Filter, Coefficients - multiple of four, Sample Processing** (cont'd)

Implementation

Delayed LMS has been implemented for realizing an adaptive complex FIR filter. Circular addressing mode is used for Delay-Buffer. As the filter is complex, two delay buffers are initialized, one for real part of input and the other for imaginary part of the input. The real and imaginary part of the input are separated and they replace the oldest value in the corresponding delay buffers.

To make use of the dual MAC feature of TriCore, coefficients are arranged in a special way as shown in the memory note. Real parts of a pair of coefficients are packed in a register using load word instruction. The corresponding imaginary parts are packed into another register.

A pair of real part of input and a pair of imaginary part of input are also packed in two registers in one cycle each by using the load word instruction.

The complex multiplication requires four multiplications (real - real, imaginary - imaginary, real - imaginary and imaginary-real). Four dual MACs are used which perform each of the above multiplications for a pair of inputs at a time and accumulate the result separately for real and imaginary parts. Hence the loop is executed nH/2 times. Similarly coefficient updation requires four more dual MACs with rounding and saturation. Loop unrolling is done for efficient update of delay line. Thus tap loop is executed (nH/2-1) times. The accumulated real and imaginary parts of the result are formatted to 16-bit saturated value and packed into the return register.

Example

Trilib\Example\Tasking\Filters\Adaptive \expCplxDlms_4_16.c, expCplxDlms_4_16.cpp
Trilib\Example\GreenHills\Filters\Adaptive \expCplxDlms_4_16.cpp, expCplxDlms_4_16.c
Trilib\Example\GNU\Filters\Adaptive \expCplxDlms_4_16.c

CplxDlms_4_16 — Adaptive Complex Filter, Coefficients - multiple of four, Sample Processing (cont'd)

Cycle Count

With DSP Extensions

Pre-kernel	:	14
Kernel	:	$\left\{\left[8 \times \left(\frac{nH}{2} - 1\right) + 1\right] + 1\right\}$
Post-kernel	:	13+2

Without DSP Extensions

Pre-kernel	:	3
Kernel	:	same as With DSP Extensions
Post-kernel	:	13+2

Code Size 206 bytes

CplxDlmsBlk_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Block Processing

Signature

```
void CplxDlmsBlk_4_16(CplxS       *X,
                      CplxS       *R,
                      DataS       *H,
                      cptrDataS   *DLYr,
                      cptrDataS   *DLYi,
                      int         nX,
                      CplxS       *D,
                      CplxS       *Err,
                      DataS       U
                      );
```

Inputs

X	:	Pointer to complex Input-Buffer
R	:	Pointer to complex Output-Buffer
H	:	Pointer to Cplx-Coeff-Buffer
DLYr	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer (Real) Without DSP Extension - Pointer to Circ-Struct
DLYi	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer (Imag) Without DSP Extension - Pointer to Circ-Struct
nX	:	Size of complex Input-Buffer
D	:	Pointer to complex Desired-Output-Buffer
Err	:	Pointer to complex Error value
U	:	Step size

Output

DLYr	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer (Real)
DLYi	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer (Imag)

CplxDlmsBlk_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Block Processing (cont'd)

H(nH*2)	:	Modified Coeff-Buffer (Real and Imag)
R(nX)	:	Complex Output-Buffer

Return None

Description Delayed LMS algorithm implemented for adaptive Complex FIR filter, FIR filter transversal structure (direct form), Block processing, 16-bit fractional input, coefficients and output data format, Optimal implementation, requires filter order to be multiple of four.

Pseudo code

```
{
   frac64 accr,acci;  //Filter result
   int i,j,k;
   frac16circ  *aDLYr=&DLYr, *aDLYi=&DLYi;
                    //Ptr to circ-ptr of real and imaginary Delay-Buffer
   for(i=0; i<nX; i++)
      {
         //Error value multiplied by step size
         uerrr = (frac16 rnd)(*Errr * U);
         uerri = (frac16 rnd)(*Erri * U);

         //Store input value in Delay-Buffer at the position of the
         //oldest value
         *DLYi = *X++;//Imag part of Input
         *DLYr = *X++;//Real part of Input

         accr = 0.0;
         acci = 0.0;

         k=0;
         //tap loop
```

CplxDlmsBlk_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Block Processing (cont'd)

```
for(j=0; j<nH/2; j++)
{
   //Filter result
   //Imag
   acci += (frac64)(*(H+k) * (*(DLYi+k))
                    + (*(H+k+1) * (*(DLYi+k+1)));
            //acci += Xi(n) * Hr_n-1(0) + Xi(n-1) * Hr_n-1(1)
   acci -= (frac64)(*(H+k+2) * (*(DLYr+k)) +
                    (*(H+k+3) * (*(DLYr+k+1)));
            //acci += Xr(n) * Hi_n-1(0) + Xr(n-1) * Hi_n-1(1)
   //Real
   accr += (frac64)(*(H+k) * (*(DLYr+k))
                    + (*(H+k+1) * (*(DLYr+k+1)));
            //accr += Xr(n) * Hr_n-1(0) +  Xr(n-1) * Hr_n-1(1)
   accr -= (frac64)(*(H+k+2) * (*(DLYi+k))
                    + (*(H+k+3) * (*(DLYi+k+1)));
            //accr -= Xi(n) * Hi_n-1(0) +  Xi(n-1) * Hi_n-1(1)
   //Coefficient update
   //Real_i
   *(H+k) = (frac16 sat rnd)(*(H+k) + (uerrr * (*(DLYr+k)));
             //Hr_n(0) = Hr_n-1(0) + Xr(n) * Errr_n-1
   *(H+k) = (frac16 sat rnd)(*(H+k) - (uerri * (*(DLYi+k)));
             //Hr_n(0) -= Xi(n) * Erri_n-1
             //Real_i+1
  *(H+k+1) = (frac16 sat rnd)(*(H+k+1) + (uerrr * (*DLYr+k+1)));
             //Hr_n(1) = Hr_n-1(1) + Xr(n-1) * Errr_n-1
 *(H+k+1) = (frac16 sat rnd)(*(H+k+1) - (uerri * (*(DLYi+k+1)));
             //Hr_n(1) -= Xi(n-1) * Erri_n-1

   //Imag_i
  *(H+k+2) = (frac16 sat rnd)(*(H+k+2) + (uerri * (*(DLYr+k)));
             //Hi_n(0) = Hi_n-1(0) + Xr(n) * Erri_n-1
  *(H+k+2) = (frac16 sat rnd)(*(H+k+2) + (uerrr * (*(DLYi+k)));
             //Hi_n(0) += Xi(n) * Errr_n-1
   //Imag_i+1
 *(H+k+3) = (frac16 sat rnd)(*(H+k+3) + (uerri * (*(DLYr+k+1)));
             //Hi_n(1) = Hi_n-1(1) + Xr(n-1) * Erri_n-1
 *(H+k+3) = (frac16 sat rnd)(*(H+k+3) + (uerrr * (*(DLYi+k+1)));
             //Hi_n(1) += Xi(n-1) * Errr_n-1

   k=k+4;
}
```

CplxDlmsBlk_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Block Processing (cont'd)

```
        //Set DLYr.index and DLYi.index to the oldest value in Delay-Buffer
          *DLYr--;
          *DLYi--;
          aDLYr = &DLYr;
          aDLYi = &DLYi;

          //Format the real and imaginary parts of the filter output
          //from 48 bit to 16-bit saturated values and store the
          //result to Output-Buffer
          *RLo = (frac16 sat)acci;
          *RHi = (frac16 sat)accr;
          R++;
          //Calculate error in current output
          *Err = *D++ - *R++;

        }//end of indata loop

  }//end of main
```

Techniques

- Loop unrolling, two taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular-buffer
- Use of dual MAC instructions
- Intermediate result stored in 64-bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Filter size is a multiple of four
- Inputs, outputs, coefficients are in 1Q15 format

CplxDlmsBlk_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Block Processing (cont'd)

Memory Note

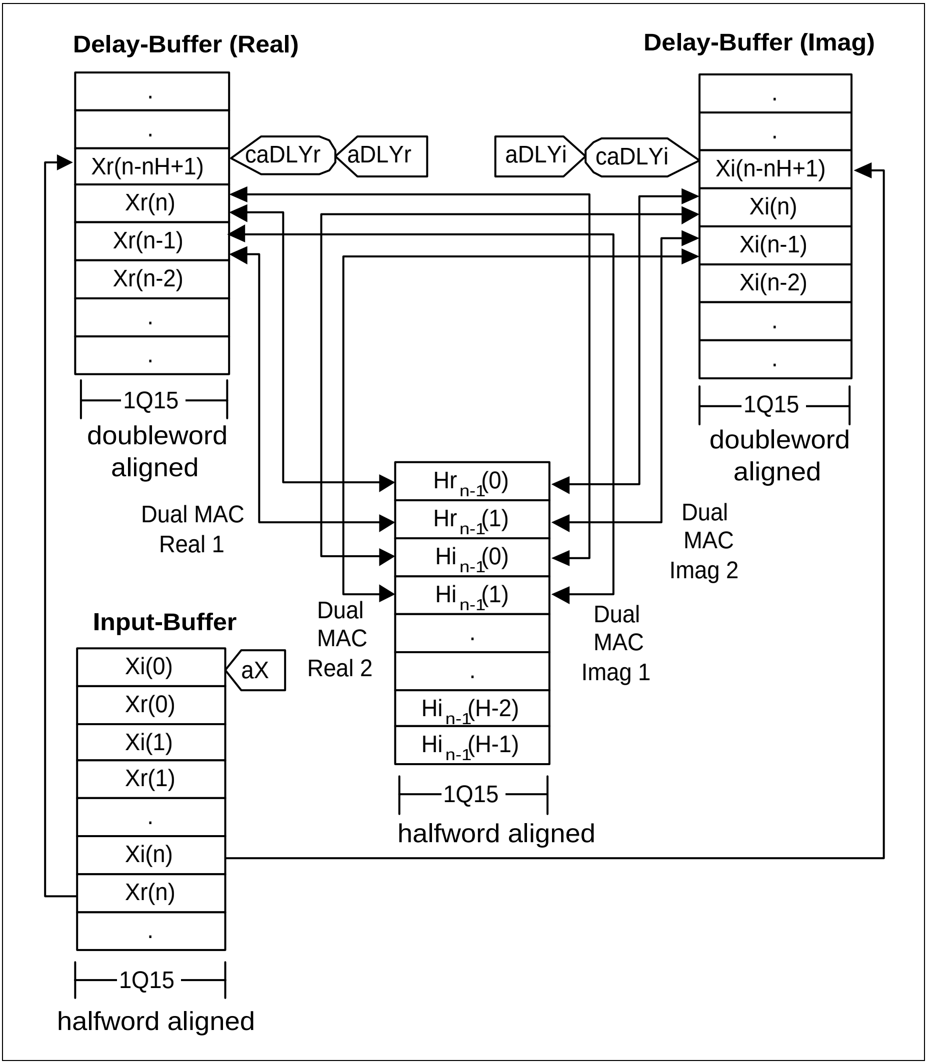


Figure 4-55 CplxDlmsBlk_4_16

CplxDlmsBlk_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Block Processing (cont'd)

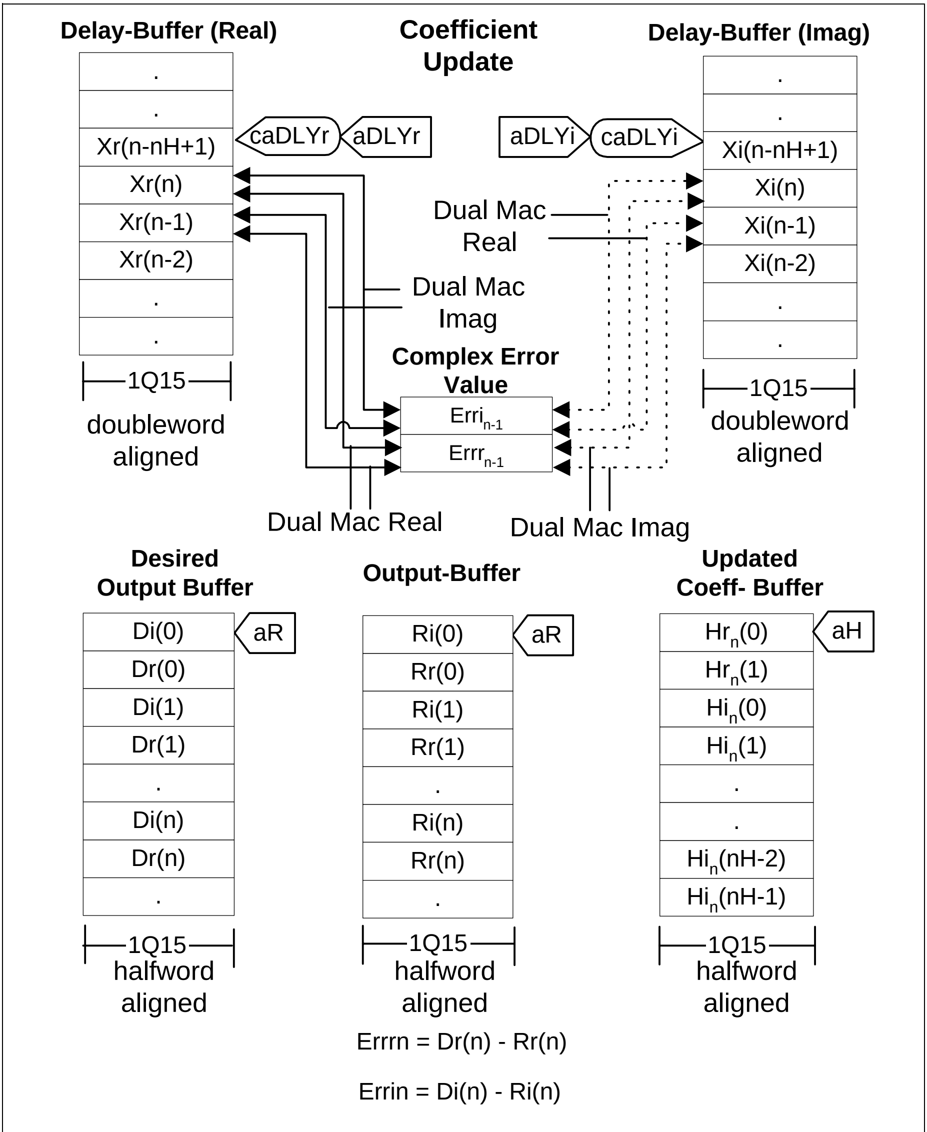


Figure 4-56 CplxDlmsBlk_4_16 Coefficient update

CplxDlmsBlk_4_16 Adaptive Complex Filter, Coefficients - multiple of four, Block Processing (cont'd)

Implementation

This DLMS routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Implementation details are same as CplxDlms_4_16. An additional loop is needed to calculate the output for every sample in the buffer. Hence, this loop is repeated as many times as the size of the input buffer.

Example

Trilib\Example\Tasking\Filters\Adaptive \expCplxDlmsBlk_4_16.c, expCplxDlmsBlk_4_16.cpp
Trilib\Example\GreenHills\Filters\Adaptive \expCplxDlmsBlk_4_16.cpp, expCplxDlmsBlk_4_16.c
Trilib\Example\GNU\Filters\Adaptive \expCplxDlmsBlk_4_16.c

Cycle Count

With DSP Extensions

Pre-loop : 9

Loop :

$$nX \times \left\{8 + \left[\left(\frac{nH}{2} - 1\right) \times 8 + 16\right]\right\}$$

$$+1+2$$

Post-loop : 3+2

Without DSP Extensions

Pre-loop : 9

Loop : same as With DSP Extensions

Post-loop : 3+2

Code Size

252 bytes

Dlms_2_16x32 Mixed Adaptive FIR Filter, Coefficients - multiple of two, Sample Processing

Signature

```
DataL Dlms_2_16x32(DataS      X,
                   DataL      *H,
                   cptrDataS  *DLY,
                   DataL      D,
                   DataL      *Err,
                   DataL      U
                   );
```

Inputs	X	: Real Input Value
	H	: Pointer to Coeff-Buffer
	DLY	: With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct
	(nH)	: Implicit filter order stored in Circ-Ptr DLY
	D	: Real expected value
	Err	: Pointer to Error value
	U	: Step size
Outputs	DLY	: Updated circular pointer with index set to the oldest value of the filter Delay-Buffer
	H(nH)	: Modified Coeff-Buffer
Return	R	: Output value of the filter (32-bit output)

Description

Delayed LMS algorithm implemented for mixed adaptive FIR filter, FIR filter transversal structure (direct form), Single sample processing, 16-bit fractional input, 32-bit coefficients and output data format, Optimal implementation, requires filter order to be multiple of two.

Dlms_2_16x32 Mixed Adaptive FIR Filter, Coefficients - multiple of two, Sample Processing (cont'd)

Pseudo code

```
{
   frac32 acc;        //filter result
   frac16 circ *aDLY = &DLY;
                      //ptr to Circ-ptr of Delay-Buffer
   int j;
   //Error value multiplied by step size
   uerr = (frac32)(*Err * U);
   //store input value in Delay-Buffer at the position
   //of the oldest value
   *DLY = X;
   acc = 0;
   k = 0;
   //tap loop
  //The index i and j of H_n-1(i) and X(j) in the comments are valid only
   //for the first iteration.For each next iteration it has to be
   //incremented and decremented by 2 respectively.
   for (j=0; j<nH/2; j++)
   {
     acc = acc + (frac32 sat)(*(H+k) * (*(DLY + k)));
                      //acc = acc + X(n)* H_n-1(0)
     acc = acc + (frac32 sat)(*(H+k+1) * (*(DLY+k+1)));
                      //acc = X(n-1) * (H_n-1(1)
      //coefficient update
      *(H+k) = (frac32 sat)((*(H+k)) + uerr * (*(DLY+k)));
      *(H+k+1) = (frac32 sat)((*(H+k+1)) + uerr * (*(DLY+k+1)));

      k = k + 2;
    }
    //Set DLY.index to the oldest value in Delay-Buffer
    DLY--;
    aDLY = *DLY;
    //filter output stored to output buffer
    R = acc;
    //calculate error for the current output
    *Err = D - R;
    return R;
}
```

Techniques

- Loop unrolling, two taps/loop
- Use of packed data Load/Store
- Delay line implemented as circular-buffer
- Instruction ordering for zero overhead Load/Store

Dlms_2_16x32 Mixed Adaptive FIR Filter, Coefficients - multiple of two, Sample Processing (cont'd)

Assumptions

- Filter order is a multiple of two
- Inputs in 1Q15 format, all other parameters in 1Q31 format

Memory Note

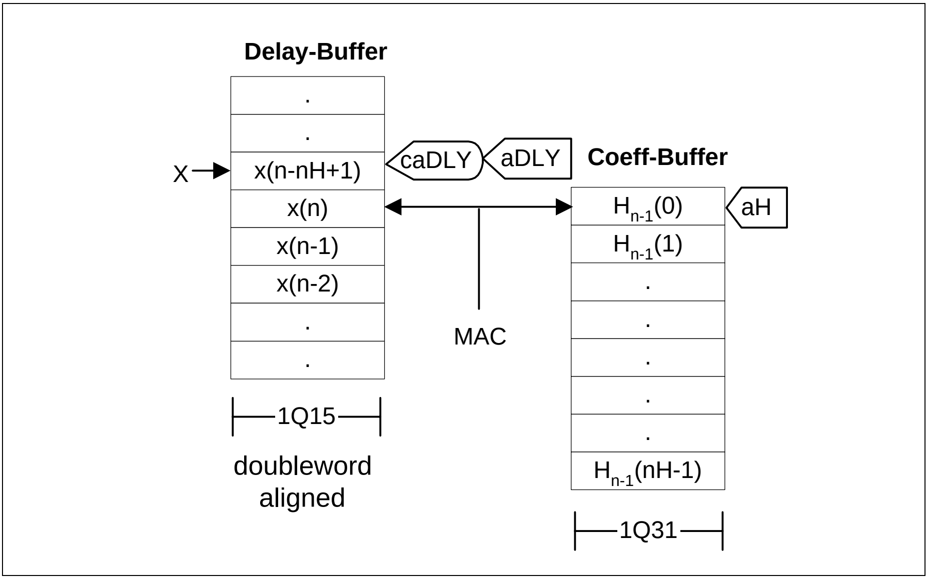


Figure 4-57 Dlms_2_16x32

Dlms_2_16x32 **Mixed Adaptive FIR Filter, Coefficients - multiple of two, Sample Processing** (cont'd)

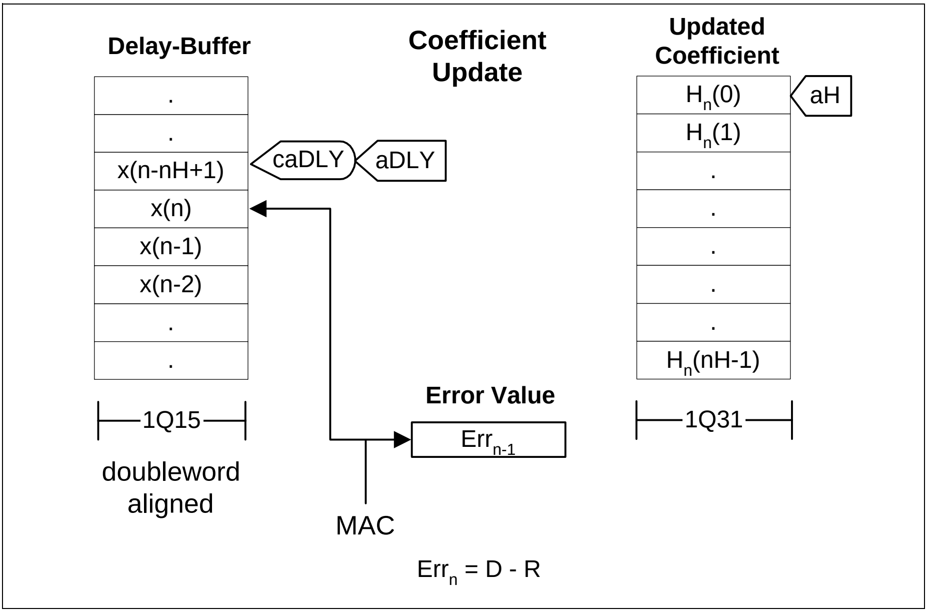


Figure 4-58 Dlms_2_16x32 Coefficient update

Dlms_2_16x32 Mixed Adaptive FIR Filter, Coefficients - multiple of two, Sample Processing (cont'd)

Implementation

LMS algorithm has been used to realize an adaptive FIR filter. The implemented filter is a Delayed LMS adaptive filter i.e., the updation of coefficients in the current instant is done using the error in the previous output.

The FIR filter is implemented using transversal structure and is realized as a tapped delay line.

This routine processes one sample at a time and returns output of that sample. The input for which the output is to be calculated is sent as an argument to the function.

TriCore's load word instruction loads two delay line values and two coefficients in one cycle each. MAC instruction performs a multiplication and an addition according to the equation

$$acc = acc + X(n-k) \cdot H_{n-1}(k) \qquad [4.75]$$

where, k=0,1,...., nH-1.

The coefficient is updated using error from the previous output, i.e., err_{n-1}. A MAC instruction updates a coefficient in one cycle according to the equation

$$H_n(k) = H_{n-1}(k) + X(n-k) \cdot Err_{n-1} \qquad [4.76]$$

where, k=0,1,...,nH-1.

By using four MACs two coefficients are used and updated in one pass through the loop. The loop is unrolled for efficient pointer update. Hence tap loop is executed (nH/2 - 1) times.

For delay line, circular addressing mode is used. The size of the circular delay buffer is equal to the filter order, i.e., the number of coefficients. Circular buffer needs doubleword alignment and to use load word instruction, size of the buffer should be multiple of four bytes. This implies that the coefficients should be multiple of two.

Dlms_2_16x32 **Mixed Adaptive FIR Filter, Coefficients - multiple of two, Sample Processing** (cont'd)

Example

Trilib\Example\Tasking\Filters\Adaptive\expDlms_2_16x32.c, expDlms_2_16x32.cpp
Trilib\Example\GreenHills\Filters\Adaptive\expDlms_2_16x32.cpp, expDlms_2_16x32.c
Trilib\Example\GNU\Filters\Adaptive\expDlms_2_16x32.c

Cycle Count

With DSP Extensions

Pre-kernel	:	12
Kernel	:	$\left\lceil \frac{nH}{2} - 1 \right\rceil \times 4 + 2$ if LoopCount > 1 $\left\lceil \frac{nH}{2} - 1 \right\rceil \times 4 + 1$ if LoopCount = 1
Post-kernel	:	4+2

Without DSP Extensions

Pre-kernel	:	12
Kernel	:	same as With DSP Extensions
Post-kernel	:	4+2

Code Size 108 bytes

DlmsBlk_2_16x32 Mixed Adaptive FIR Filter, Coefficients - multiple of two, Block Processing

Signature

```
void DlmsBlk_2_16x32(DataS      *X,
                     DataL      *R,
                     cptrDataL   H,
                     cptrDataS  *DLY,
                     int         nX,
                     DataL      *D,
                     DataL      *Err,
                     DataL       U
                    );
```

Inputs	X	:	Pointer to Input-Buffer
	R	:	Pointer to Output-Buffer
	H	:	With DSP Extension - circular pointer of Coeff-Buffer of size nH Without DSP Extension - circ-Struct. Whose members are base address, size and index
	DLY	:	With DSP Extension - Pointer to circular pointer of Delay-Buffer of size nH, where nH is the filter order Without DSP Extension - Pointer to Circ-Struct
	(nH)	:	Implicit filter order stored in Circ-Pointer DLY
	D	:	Pointer to Desired-Output-Buffer
	Err	:	Pointer to Error value
	U	:	Step size
Output	DLY	:	Updated circular pointer with index set to the oldest value of the filter Delay-Buffer
	H(nH)	:	Modified Coeff-Buffer
	R(nX)	:	Output-Buffer
Return	None		

DlmsBlk_2_16x32 Mixed Adaptive FIR Filter, Coefficients - multiple of two, Block Processing (cont'd)

Description

Delayed LMS algorithm implemented for mixed adaptive FIR filter, FIR filter transversal structure (direct form), Block processing, 16-bit fractional input, 32-bit coefficients and output data format, Optimal implementation, requires filter order to be multiple of two.

DlmsBlk_2_16x32 Mixed Adaptive FIR Filter, Coefficients - multiple of two, Block Processing (cont'd)

Pseudo code

```
{
   frac32 acc;        //filter result
   frac16 circ *aDLY = &DLY;
                      //ptr to Circ-ptr of Delay-Buffer
   int i, j;
   //loop for input buffer
   for (i=0; i<nX; i++)
   {
      //Error value multiplied by step size
      uerr = (frac32 rnd)(*Err * U);
      //store input value in Delay-Buffer at the position
      //of the oldest value
      *DLY = *X++;
      acc = 0;
      k = 0;
      //tap loop
      for (j=0; j<nH/4; j++)
      {
         acc = acc + (frac32 sat)(*(H+k) * (*(DLY + k)));
                      //acc = acc + X(n)* H_n-1(0)
         acc = acc + (frac32 sat)(*(H+k+1) * (*(DLY+k+1)));
                      //acc = X(n-1) * (H_n-1(1)

         //coefficient update
         *(H+k) = (frac32 sat)((*(H+k)) + uerr * (*(DLY+k)));
         *(H+k+1) = (frac32 sat)((*(H+k+1)) + uerr * (*(DLY+k+1)));
         k = k + 2;
        }
        //Set DLY.index to the oldest value in Delay-Buffer
        DLY--;
        aDLY = *DLY;
        //filter output stored to output buffer
        *R = acc;
        //calculate error for the current output
        *Err = *D++ - *R++;
   }
}
```

DlmsBlk_2_16x32 Mixed Adaptive FIR Filter, Coefficients - multiple of two, Block Processing (cont'd)

Techniques
- Loop unrolling, two taps/loop
- Use of packed data Load/Store
- Delay line and coefficient array implemented as circular-buffer
- Instruction ordering for zero overhead Load/Store

Assumptions
- Filter size is a multiple of two
- Inputs in 1Q15, all other parameters in 1Q31 format

Memory Note

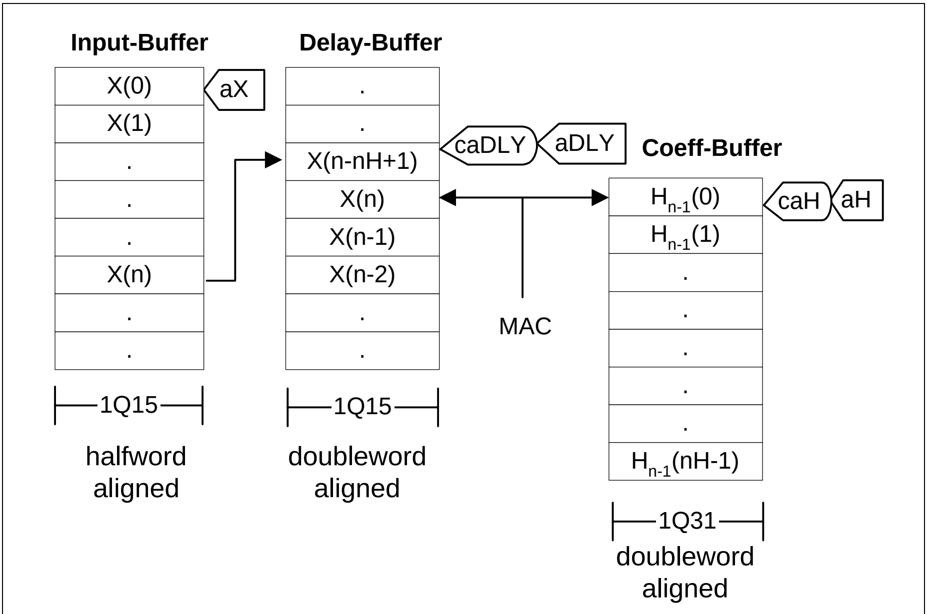


Figure 4-59 DlmsBlk_2_16x32

DlmsBlk_2_16x32 Mixed Adaptive FIR Filter, Coefficients - multiple of two, Block Processing (cont'd)

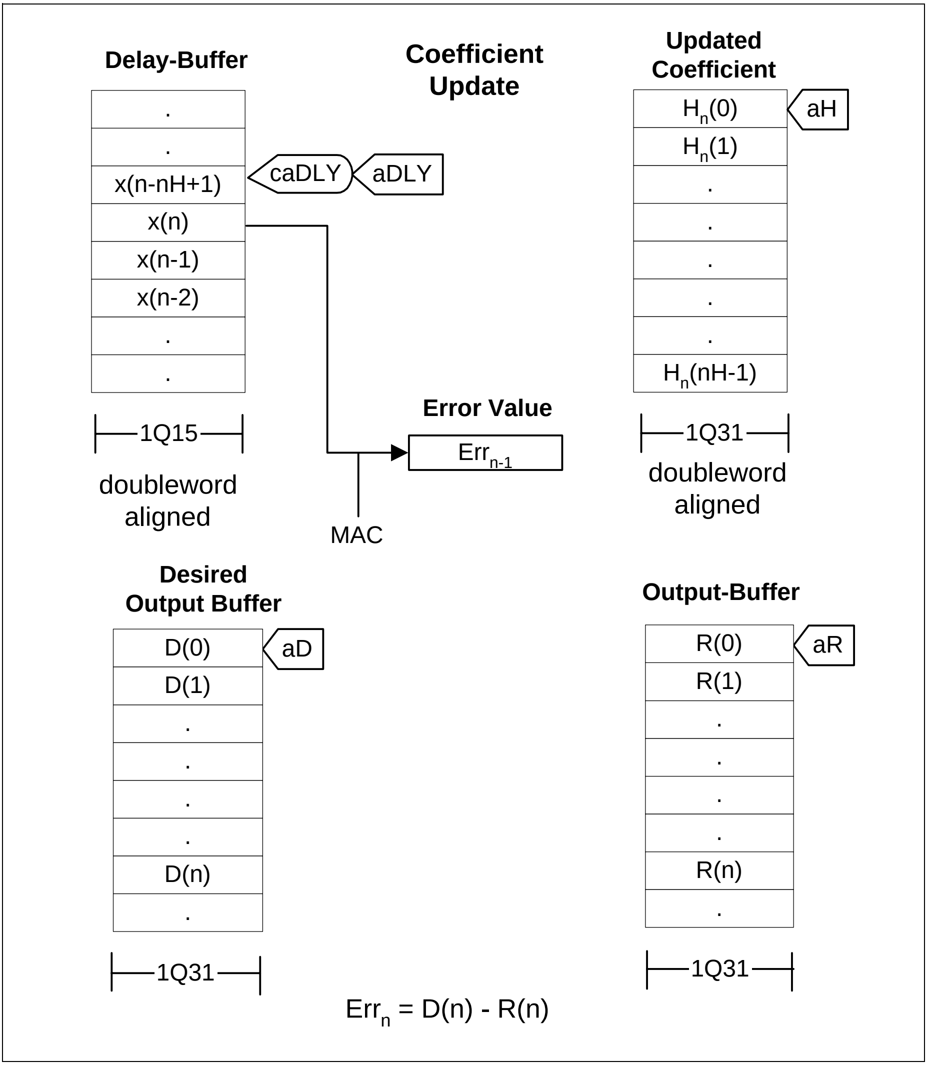


Figure 4-60 DlmsBlk_2_16x32 Coefficient update

DlmsBlk_2_16x32 **Mixed Adaptive FIR Filter, Coefficients - multiple of two, Block Processing** (cont'd)

Implementation

This DLMS routine processes a block of input values at a time. The pointer to the input buffer is sent as an argument to the function. The output is stored in output buffer, the starting address of which is also sent as an argument to the function.

Implementation details are same as Dlms_4_16, except that the Coeff-Buffer is also circular and needs doubleword alignment. The advantage of using circular buffer for coefficients is efficient pointer update. In this implementation while exiting the tap loop, the first two coefficients are already loaded for the next input value. This helps in saving one cycle in the next sample processing.

Example

Trilib\Example\Tasking\Filters\Adaptive \expDlmsBlk_2_16x32.c, expDlmsBlk_2_16x32.cpp
Trilib\Example\GreenHills\Filters\Adaptive \expDlmsBlk_2_16x32.cpp, expDlmsBlk_2_16x32.c
Trilib\Example\GNU\Filters\Adaptive \expDlmsBlk_2_16x32.c

Cycle Count

With DSP Extensions

Pre-loop : 7

Loop (for input data) : $nX \times \left[9 + \left[\left(\frac{nH}{2} - 1\right) \times 4 + 6\right]\right]$ $+1+2$

Post-loop : 1+2

Without DSP Extensions

Pre-loop : 8

Loop : same as With DSP Extensions

Post-loop : 1+2

Code Size

136 bytes

4.7 Fast Fourier Transforms

Spectrum (Spectral) analysis is a very important methodology in Digital Signal Processing. Many applications have a requirement of spectrum analysis. The spectrum analysis is a process of determining the correct frequency domain representation of the sequence. The analysis gives rise to the frequency content of the sampled waveform such as bandwidth and centre frequency.

One of the method of doing the spectrum analysis in Digital Signal Processing is by employing the Discrete Fourier Transform (DFT).

The DFT is used to analyze, manipulate and synthesize signals in ways not possible with continuous (analog) signal processing. It is a mathematical procedure that helps in determining the harmonic, frequency content of a discrete signal sequence. DFTs origin is from a continuous fourier transform which is given by

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt \quad [4.77]$$

where x(t) is continuous time varying signal and X(f) is the fourier transform of the same. The DFT is given by

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} \text{ [exponential form]} \quad [4.78]$$

where the DFT coefficients used in the DFT Kernel, W, is

$$W_N = e^{-j2\pi/N} \quad [4.79]$$

$$X(k) = \sum_{n=0}^{N-1} x(n)[\cos(2\pi nk/N) - j\sin(2\pi nk/N)] \quad [4.80]$$

X(k) is the k^{th} DFT output component for k=0,1,2,....,N-1

x(n) is the sequence of discrete sample for n=0,1,2,...,N-1

j is imaginary unit $\sqrt{-1}$

N is the number of samples of the input sequence (and number of frequency points of DFT output).

While the DFT is used to convert the signal from time domain to frequency domain. The complementary function for DFT is the IDFT, which is used to convert a signal from frequency to time domain. The IDFT is given by

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)e^{j2\pi nk/N} \text{ [exponential form]} \quad [4.81]$$

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)[\cos(2\pi nk/N) + j\sin(2\pi nk/N)] \quad [4.82]$$

Notice the difference between DFT in **Equation [4.78]** and **Equation [4.80]**, the IDFT Kernel is the complex conjugate of the DFT and the output is scaled by N.

W_N^{nk}, the Kernel of the DFT and IDFT is called the Twiddle-Factor and is given by,

In exponential form,

$e^{-j2\pi nk/N}$ for DFT

$e^{j2\pi nk/N}$ for IDFT

In rectangular form,

$\cos(2\pi nk/N) - j\sin(2\pi nk/N)$ for DFT

$\cos(2\pi nk/N) + j\sin(2\pi nk/N)$ for IDFT

While calculating DFT, a complex summation of N complex multiplications is required for each of N output samples. N^2 complex multiplications and N(N-1) complex additions compute an N-point DFT. The processing time required by large number of calculation limits the usefulness of DFT. This drawback of DFT is overcome by a more efficient and fast algorithm called Fast Fourier Transform (FFT). The radix-2 FFT computes the DFT in $N*\log_2(N)$ complex operations instead of N^2 complex operations for that of the DFT. (where N is the transform length.)

The FFT has the following preconditions to operate at a faster rate.

- The radix-2 FFT works only on the sequences with lengths that are power of two.
- The FFT has a certain amount of overhead that is unavoidable, called bit reversed ordering. The output is scrambled for the ordered input or the input has to be arranged in a predefined order to get output properly arranged. This makes the straight DFT better suited for short length computation than FFT. The graph shows the algorithm complexity of both on a typical processor like pentium.

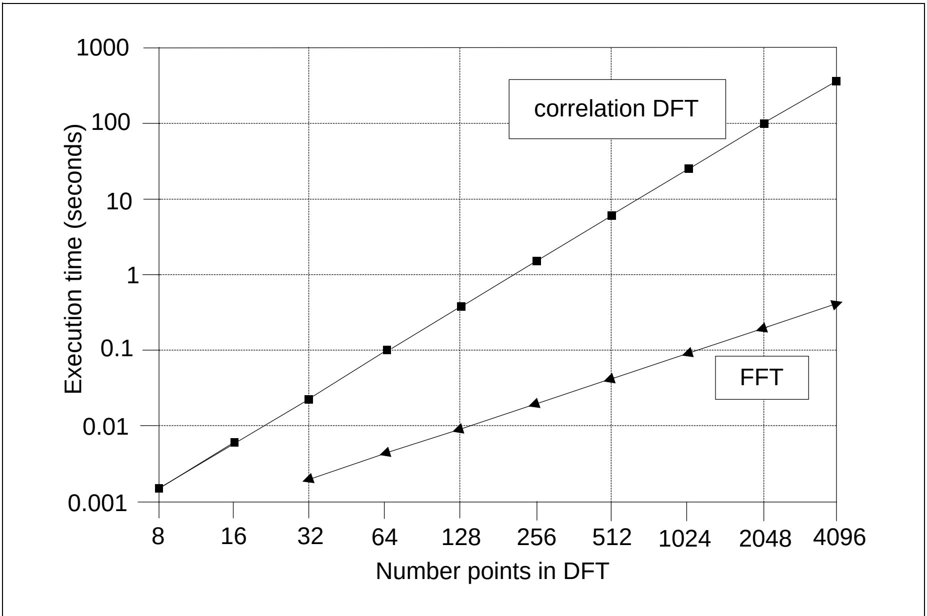


Figure 4-61 Complexity Graph

The Fourier transform plays an important role in a variety of signal processing applications. Anytime, if it is more comfortable to work with a signal in the frequency domain than in the original time or space domain, we need to compute Fourier transform.

Given N input samples of a signal x(n) = 0,1,..., (N-1), its Fourier transform is defined by

$$X(f) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi fn} \quad [4.83]$$

Since n is an integer, X(f) is periodic with the period 1. Therefore, we only consider X(f) in the basic interval $0 \le f \le 1$. In digital computation, X(f) is often evaluated at N uniformly spaced points f = k/N (k=0,1,.....,N-1). This leads to the Discrete Fourier Transform (DFT)

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} \quad (k=0,1,.....,N-1) \quad [4.84]$$

with $W_N = e^{-j2\pi/N}$. Direct computation of this length N, DFT takes N^2 complex multiplications and N(N-1) complex additions. FFT is an incredibly efficient algorithm for computing DFT. The main idea of FFT is to exploit the periodic and symmetric properties of the DFT Kernel W_N^{nk}. The resulting algorithm depends strongly on the transform length N. The basic Cooley-Tukey algorithm assumes that N is a power of two. Hence it is called radix-2 algorithm. Depending on how the input samples x(n) and the output data X(k) are grouped, either a decimation-in-time (DIT) or a decimation-in-frequency (DIF) algorithm is obtained. The technique used by Cooley and Tukey can also be applied to DFTs, where N is a power of r. The resulting algorithms are referred to as radix-r FFT. It turns out that radix-4, radix-8, and radix-16 are especially interesting. In cases where N cannot be represented in terms of powers of single number, mixed-radix algorithms must be used. For example for 28 point input, since 28 cannot be represented in terms of powers of 2 and 4 we use radix-7 and radix-4 FFT to get the frequency spectrum. The basic radix-2 decimation-in-frequency FFT algorithm is implemented.

4.7.1 Radix-2 Decimation-In-Time FFT Algorithm

The decimation-in-time (DIT) FFT divides the input (time) sequence into two groups, one of even samples and the other of odd samples. N/2-point DFTs are performed on these sub-sequences and their outputs are combined to form the N-point DFT.

First, x(n) the input sequence in the **Equation [4.84]** is divided into even and odd sub-sequences.

$$\begin{aligned} X(k) &= \sum_{n=0}^{\frac{N}{2}-1} x(2n)W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)W_N^{(2n+1)k} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(2n)W_N^{2nk} + W_N^{\ k} \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)W_N^{2nk} \end{aligned}$$

for k=0 to N-1 [4.85]

But, $W_N^{\ 2nk} = (e^{(-j2\pi)/N})^{2nk} = (e^{(-j2\pi)/(N/2)})^{nk} = W_{N/2}^{\ nk}$

By substituting the following in **Equation [4.85]**

$x_1(n)=x(2n)$

$x_2(n)=x(2n+1)$

Equation [4.85] becomes

$$\begin{aligned} X(k) &= \sum_{n=0}^{N/2-1} x_1(n)W_{N/2}^{nk} + W_N^{\ k} \sum_{n=0}^{N/2-1} x_2(n)W_{N/2}^{nk} \\ &= Y(k) + W_N^{\ k}Z(k) \end{aligned}$$

for k=0 to N-1 [4.86]

Equation [4.86] is the radix-2 DIT FFT equation. It consists of two N/2-point DFTs (Y(k) and Z(k)) performed on the subsequences of even and odd samples respectively of the input sequence, x(n). Multiples of W_N, the Twiddle-Factors are the coefficients in the FFT calculation.

Further,

$$W_N^{k+N/2} = (e^{-j2\pi/N})^k \times (e^{-j2\pi/N})^{N/2} = -W_N^k \tag{4.87}$$

Equation [4.86] can be expressed in two equations

$$X(k) = Y(k) + W_N^k Z(k) \tag{4.88}$$

$$X(k+N/2) = Y(k) - W_N^k Z(k) \tag{4.89}$$

for k=0 to N/2-1

The complete 8-point DIT FFT is illustrated in figure.

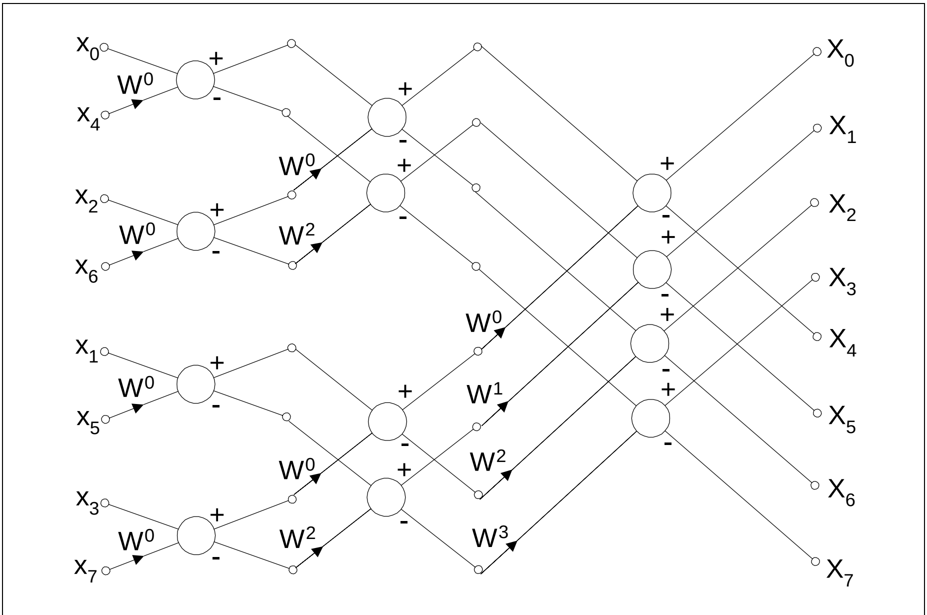


Figure 4-62 8-point DIT FFT

The complete 8-point DIF FFT is illustrated in figure.

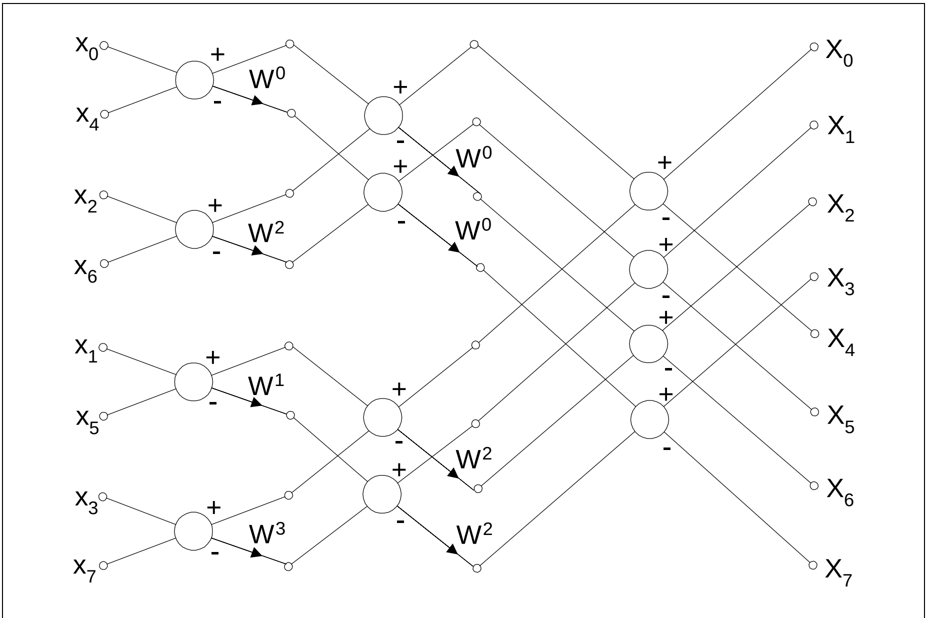


Figure 4-63 8-point DIF FFT

In the diagram, each pair of arrows represents a Butterfly. The whole of FFT is computed by different patterns of Butterflies. These are called groups and stages.

For 8-point FFT the first stage consists of four groups of one Butterfly each, second consists of two groups of two butterflies and third stage has one group of four Butterflies. Each Butterfly is represented as in diagram.

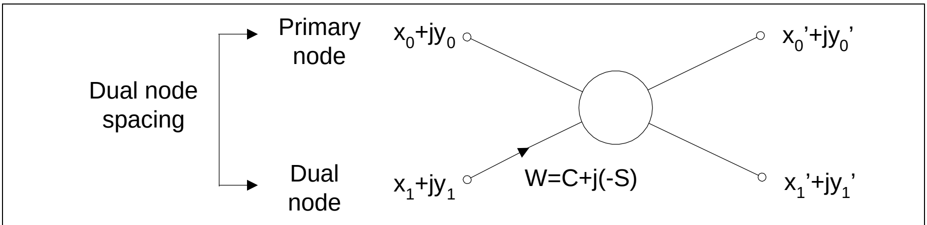


Figure 4-64 Radix-2 DIT Butterfly

The output is derived as follows

$$x_0' = x_0 + [(C)x_1 - (-S)y_1] \qquad [4.90]$$

$$y_0' = y_0 + [(C)y_1 + (-S)x_1] \qquad [4.91]$$

$$x_1' = x_0 - [(C)x_1 - (-S)y_1] \qquad [4.92]$$

$$y_1' = y_0 - [(C)y_1 + (-S)x_1] \qquad [4.93]$$

4.8 TriCore Implementation Note

4.8.1 Organization of FFT functions

The FFT radix-2 DIT function set consists of the following functions.

- Forward FFT
- Inverse FFT
- Forward Real FFT
- Inverse Real FFT

The above set of functions makes use of macros for efficient computation. The basic bit reversal module, Butterflies and the Spectrum split operations are implemented in form of macros.

The TriLib FFT implementation is one of the most optimal implementation which makes use of several optimization techniques. Further, it makes use of different optimization methods at instruction level. Secondly, it is organized as macros to save time during function calls and also overcome the conditional checks such as shift etc., which perhaps is done during assembling time itself as it is implemented as macros. Thirdly, the algorithmic optimization, where the first pass or the first stage Butterflies are computed outside the loop separately. This saves time as the first stage Butterflies need not be multiplied by Twiddle-Factors.

4.8.2 16 Bit Implementation Modules

The classical FFT takes the input and Twiddle-Factor in the form of 16 bit complex number representation as in **Figure 4-2**. For computational efficiency and to make use of the parallel architecture of TriCore, a more efficient form of complex representation is devised for internal operations of the FFT. The REAL:IMAG, REAL:IMAG pairs are converted to REAL:REAL, IMAG:IMAG representation before processing.

Twiddle-Factors for the computation of 16 bit FFT is done by a utility function called **FFT_TF_16()**.

The main modules of FFTs are:

FFT_2_16()	Forward FFT for 16 bit Complex input, radix-2 decimation-in-time implementation
IFFT_2_16()	Inverse FFT for 16 bit Complex input, radix-2 decimation-in-time implementation

FFTReal_2_16()	Forward FFT for 16 bit Real sequence input, radix-2 decimation-in-time implementation
IFFTReal_2_16()	Inverse Real FFT for 16 bit Complex sequence input, radix-2 decimation-in-time implementation to generate the two real output sequences

4.8.3 16 bit Implementation for Mixed FFT

The mixed 16 bit FFT is the combination of features of 32 bit and 16 bit FFT, while 16 bit is more efficient and 32 bit is more precise. The mixed FFT is a combination of both. It has better precision than 16 bit and better speed than 32 bit implementation.

Internally the mixed FFT uses 32 bit representation and the final stage output is converted to 16 bit precision using **ConvertBuf** macro.

Twiddle-Factors for the computation of mixed FFT is done by a utility function called **FFT_TF_16x32()**.

The main modules of Mixed FFTs are:

FFT_2_16x32()	Forward FFT for 16 bit Complex input, radix-2 decimation-in-time implementation. Internal processing will be 32 bits, output will be rounded to 16 bits
IFFT_2_16x32()	Inverse FFT for 16 bit Complex input, radix-2 decimation-in-time implementation. Internal processing will be 32 bits, output will be rounded to 16 bits
FFTReal_2_16x32()	Forward FFT for 16 bit Real sequence input, radix-2 decimation-in-time implementation. Internal processing will be 32 bits, output will be rounded to 16 bits
IFFTReal_2_16x32()	Inverse Real FFT for 16 bit Complex sequence input, radix-2 decimation-in-time implementation to generate the two real output sequences. Internal processing will be 32 bits, output will be rounded to 16 bits

4.8.4 32 Bit Implementation

The 32 bit implementation follows the straight forward approach in implementation. The first pass (stage) is done outside the stage loop for the optimization purpose like it is done in the 16 bit implementation. This is done by the **Firstpass** macro.

Subsequent passes (stages) uses the **Butterfly2** macro for the forward FFT and the **IButterfly2** macro for the inverse FFT. This is same as the 16 bit implementation, except that this doesn't need the special arrangement of the data.

Twiddle-Factors for FFT and IFFT are complex conjugate of each other, the Twiddle-Factors calculated for FFT are used for IFFT. The Butterfly calculation for IFFT is changed accordingly.

The Real FFT uses the Complex FFT functionality for computation and the final output is split to separate the real part from the complex result and is arranged as a real half in and imaginary half like Re[0], Re[1],...,Re[N/2-1], Im[0], Im[1],...,Im[N/2-1] in a continuous order.

Twiddle-Factors for the computation of FFT is done by a utility function called **FFT_TF_32()** as shown in the example.

The input for the 32 bit FFT, IFFT, RFFT, RIFFT are all in 1Q31 packed into a 64 bit data as shown in the **Figure 4-3** the input and the output is in normal order.

The main modules of FFTs are:

FFT_2_32()	Forward FFT for 32 bit Complex input, radix-2 decimation-in-time implementation
IFFT_2_32()	Inverse FFT for 32 bit Complex input, radix-2 decimation-in-time implementation
FFTReal_2_32()	Forward FFT for 32 bit Real sequence input, radix-2 decimation-in-time implementation
IFFTReal_2_32()	Inverse Real FFT for 32 bit Complex sequence input, radix-2 decimation-in-time implementation to generate the two real output sequences

4.8.5 Functional Implementation

The main functions tested in **Section 4.8.2** has a generic structure. It uses three nested loops. It computes the first pass outside the nested loops.

First Stage

The First stage is executed outside the nested loops. The advantage of having this has been already discussed in the **Section 4.8.1**. The First stage makes use of the

FirstPass macro. The idea to separate the first stage Butterfly outside the loop can be depicted as follows

$$x_0' = x_0 + [(C)x_1 - (-S)y_1] \quad [4.94]$$

$$y_0' = y_0 + [(C)y_1 + (-S)x_1] \quad [4.95]$$

$$x_1' = x_0 - [(C)x_1 - (-S)y_1] \quad [4.96]$$

$$y_1' = y_0 - [(C)y_1 + (-S)x_1] \quad [4.97]$$

In the first stage, there are N/2 groups, each containing a single Butterfly. Each Butterfly uses a Twiddle-Factor W^0, where

$$W^0 = e^{j0} = \cos(0) + j\sin(0) = 1 + j0 \quad [4.98]$$

All of the multiplications in the first stage are by a value of either 0 or 1 and therefore can be removed. The first stage Butterflies do not need multiplications. The Butterfly equations reduce to the following.

$$x_0' = x_0 + x_1 \quad [4.99]$$

$$y_0' = y_0 + y_1 \quad [4.100]$$

$$x_1' = x_0 - x_1 \quad [4.101]$$

$$y_1' = y_0 - y_1 \quad [4.102]$$

Because there is only one Butterfly per group in the first stage, the Butterfly loop (which would execute only once per group) and the group loop can be combined.

The **FirstPass** macro does the following operations.

- It copies the Input-Buffer elements in the bit reversal order to output array which is used for in-place processing.
- It calculates the first Butterfly.
- It converts the conventional complex notation REAL:IMAG, REAL:IMAG format to REAL:REAL, IMAG:IMAG format for efficient computation.

The following sections describe each of the loops.

Butterfly Loop

The inner most loop is the Butterfly loop in the FFT.

The **Butterfly** macro is used to perform the basic Butterfly operation with or without shifting. The Butterfly operation is as given below.

The Butterfly macro exploits the parallel architecture of the TriCore to achieve two parallel operations in one single operation. Therefore it can compute two Butterfly outputs in parallel.

$$x_0' = x_0 + [(C)x_1 - (-S)y_1] \tag{4.103}$$

$$y_0' = y_0 + [(C)y_1 + (-S)x_1] \tag{4.104}$$

$$x_1' = x_0 - [(C)x_1 - (-S)y_1] \tag{4.105}$$

$$y_1' = y_0 - [(C)y_1 + (-S)x_1] \tag{4.106}$$

The Butterfly macro involves two packed multiplications and two packed additional subtraction. The MAC operation can cause the output of Butterfly to grow by two bits from input to output. So the Butterfly also has a version with shift to take care of the conditions to avoid errors caused by bits growth.

The **Inverse Butterfly (IButterfly)** macro is used by the Inverse FFT functions to compute the Butterfly operation. In classical method the Twiddle-Factor is the complex conjugate of the forward FFT. For efficient computation, the Twiddle-Factor is computed by the same method as that of the forward FFT. But the computational mechanism is changed in case of Inverse Butterfly, so as to achieve the same output as that by using the complex conjugate. In contrast to the Forward Butterfly, inverse will compute using the following equations.

$$x_0' = x_0 + [(C)x_1 + (-S)y_1] \tag{4.107}$$

$$y_0' = y_0 + [(C)y_1 - (-S)x_1] \tag{4.108}$$

$$x_1' = x_0 - [(C)x_1 + (-S)y_1] \tag{4.109}$$

$$y_1' = y_0 - [(C)y_1 - (-S)x_1] \tag{4.110}$$

An example of bit growth and overflow is shown below.

Bit Growth:

Input to the Butterfly = 0000 1100 0000 0000
H#0C00

Output from Butterfly H#1800	= 0001 1000 0000 0000

Overflow:

Input to the Butterfly H#3000	= 0011 0000 0000 0000
Output from Butterfly H#C000	= 1100 0000 0000 0000

In overflow, the positive number H#3000 is multiplied by a positive number, resulting in H#C000, which is too large to represent as a positive, signed 16 bit number. H#C000 is erroneously interpreted as a negative number.

To avoid overflow errors there are methods for compensating the growth of bits.

Following are the standard methods of compensation for the bit growth error.

a) Scaling of Input data to the Butterfly
b) Scaling of the output data unconditionally using the block floating point fundamental method
c) Scaling of the output data conditionally using the block floating point fundamental method
d) Extra sign bits to protect the output data

The method depicted in (d) is the fastest and the most efficient method but unfortunately this has limited accuracy and is not suited for large FFTs.

Method (a) Input data scaling requires the extra shifting or scaling for all the input before passing to FFT for processing, this becomes overhead in using the FFT and the purpose is not served since it involves extra processing and also programming effort.

Method (b) is another way of compensating the bit growth, it unconditionally scales down the input to Butterfly by a factor of two so that the output never overflows. This adds extra time as the overhead and also the precision is lost in every iteration. The method adapted here is to shift the whole block of data one bit to the right and updating the block exponent.

Method adapted in the *TriLib* FFT implementation

The most optimal method (c), the conditional block floating point scales the input data only if the bit growth occurs. This shifting is done for the entire block with the updating of the block exponent if one or more output grows. The condition is checked before every stage of the loop begins and then it is branched to execute the nested loops with or without pre-shift depending upon the status of the Sticky Advance Overflow (SAV) flag of the Program Status Word (PSW).

Group Loop

The main objective of the group loop is to control the group of Butterfly. It sets the address pointers for each of the Butterflies for their respective Twiddle-Factor-Buffers and the input data buffers.

Stage Loop

The Stage Loop is the outer most loop of the FFTs nested loop. It controls the group count, the number of Butterflies for each of the group and most importantly it performs the conditional block floating point scaling on the stage calculation before it enters the Group Loop.

Post Processing

The Post processing is involved in case of 16 bits, Mixed 16 bits and all the Real FFT implementations.

In case of 16 bit implementation, **ToComplexSfm** is used to convert the REAL:REAL, IMAG:IMAG internal representation to REAL:IMAG format.

In case of mixed 16 bit implementation, the output buffer after the FFT has 32 bit precision it uses the **ConvertBuf** macro to make it 16 bit.

In Real Forward FFT implementation of all the types, the **Split** macro is used to separate the output of the two real sequences given as the input to the Real FFT.

4.8.6 Implementation of FFT to Process the Real Sequences of Data

Many applications have the real valued data to be processed. Though the data is real valued, one trivial approach is to use the Complex FFT by making the real portion of the complex sequence filled by the real values and the imaginary portion equated to zero.

However, this method is very inefficient. Following steps are followed to efficiently implement the Real FFT using the Complex FFT algorithm.

1. Input complex sequence x(n) has to be formed from the two N length real valued sequences x1(n), x2(n).

For n = 0, 1,..., N-1

x(n).real = x1(n) [4.111]

x(n).imag = x2(n) [4.112]

2. Compute the N-length Complex FFT on x(n).

$$X(k) = FFT[x(n)] \tag{4.113}$$

3. Perform the **Split** of the output spectrum. The Splitting of the spectrum is done by **Split** macro that implements the following equations.

$$X_1r(0) = X_r(0) \qquad X_1i(0) = 0 \tag{4.114}$$

$$X_2r(0) = Xi(0) \qquad X_2i(0) = 0 \tag{4.115}$$

$$X_1r(N/2) = X_r(N/2) \qquad X_1i(N/2) = 0 \tag{4.116}$$

$$X_2r(N/2) = Xi(N/2) \qquad X_2i(N/2) = 0 \tag{4.117}$$

For k = 1,..., N/2-1

$$X_1r(k) = 0.5 \times [Xr(k) + Xr(N-k)] \qquad X_1i(k) = 0.5 \times [Xi(k) + Xi(N-k)] \tag{4.118}$$

$$X_2r(k) = 0.5 \times [Xi(k) + Xi(N-k)] \qquad X_2i(k) = -0.5 \times [Xr(k) + Xr(N-k)] \tag{4.119}$$

$$X_1r(N-k) = X_1r(k) \qquad X_1i(N-k) = -X_1i(k) \tag{4.120}$$

$$X_2r(N-k) = X_2r(k) \qquad X_2i(N-k) = -X_2i(k) \tag{4.121}$$

Implementation of the Inverse Real FFT is done by forming the single complex sequence X(k) from two sequences X1(k) and X2(k). The **Unify** macro is used to perform this operation. The following equations are implemented in the **Unify** macro.

For k = 0,...,N-1

$$Xr(k) = X_1r(k) + X_2i(k) \qquad [4.122]$$

$$Xi(k) = X_1i(k) + X_2r(k) \qquad [4.123]$$

The unified complex sequence X(k) is used as the single sequence as input to the Inverse FFT.

$$x(n) = IDFT[X(k)] \qquad [4.124]$$

4.8.7 Design of Test Cases for the FFT functions

The test cases are designed using the math lab references. The characteristics of the FFT is used to simplify the design of test cases. The Complex FFT contains the real and imaginary components in the input data. By careful examination of the FFT equation it can be found that when the real component is a cosine term with or without the harmonics and the imaginary component is the sine term with same frequency and harmonics as that of the cosine term, the output of the FFT will have a peak in second position of the output array

Say, the input is given by the following equation

$$\sum_{n=0}^{N} \cos(2\pi nk) + i\sin(2\pi nk) \qquad [4.125]$$

where k=0,....,∞

The corresponding output will have only one peak as shown in the graphics below.

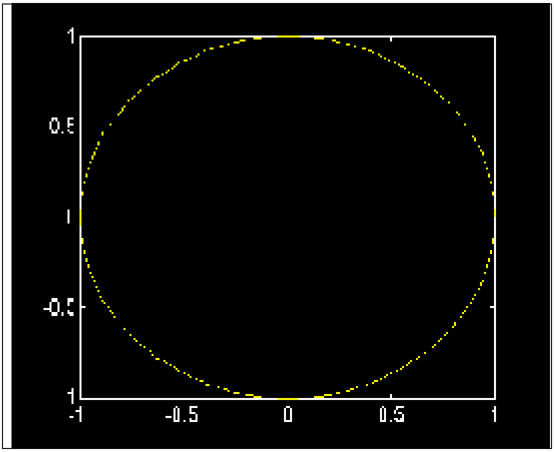


Figure 4-65 The plot of Equation [4.125] for a typical value of k given as input

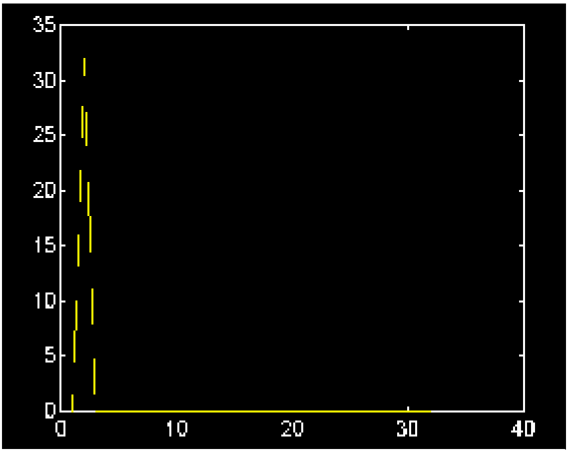


Figure 4-66 The output plot from the FFT contains only one peak

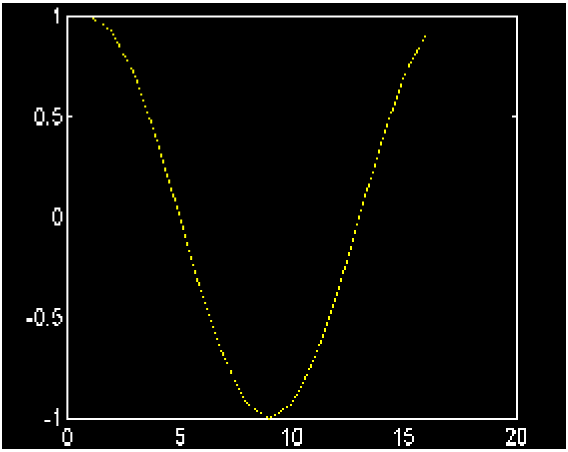


Figure 4-67 The Real cosine component for the Real FFT input

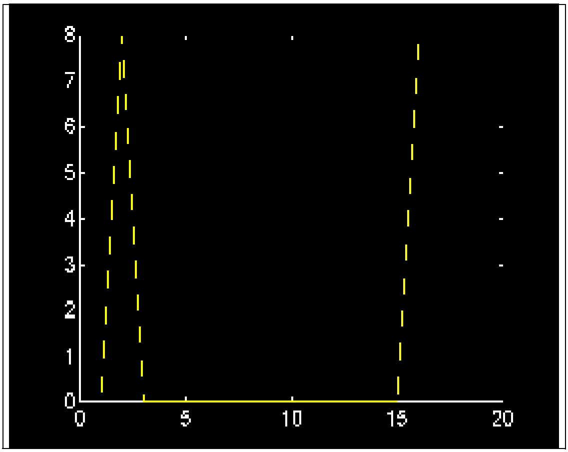


Figure 4-68 The output of the FFT contains two peaks for the input Figure 4-67

The presence of only cosine component and the sine component if equated to zero, the output should have two peaks in second and Nth position in the real part of the output array. This is the test used for the real FFT

The DC test is optional which gives rise to one peak in the first position of the output array.This can be used to verify the scaling factor of the FFT.

4.8.8 Using FFT functions

TriLib has three versions of FFT implementation 16 bit precision, 32 bit precision and 16 bit mixed precision.

16 bit implementation is most efficient.

32 bit implementation is most accurate.

16 bit mixed implementation is a compromise between speed of 16 bit and accuracy of 32 bit. It should be noted that mixed FFT is not efficient at all for FFTs at low points say, 8, 16.

FFTs are demonstrated by respective example main files such as

expCplx FFT_2_16() - demonstrates 16 bit FFT

expCplx FFT_2_32() - demonstrates 32 bit FFT

expCplx FFT_2_16X32() - demonstrates 16 bit mixed FFT and so the Real too.

The test data can be included into the above main functions such as

FFT_X.h - where X is points of FFT. e.g.,

FFT_8.h - 8 point Complex 16 bit data

FFT_16_32.h - 16 point Complex 32 bit data

RFFT_16.h - 16 point Real 16 bit data and so on.

Important Note:

- The 16 bit, 32 bit Real FFT and 16 bit Real, Complex FFT requires an output buffer to be 2N size
- The Real FFT functions of 16, 32 and 16 mixed versions modifies the contents of input buffer

4.8.9 Description

The following FFT functions for 16 bit, 32 bit and mixed are described.

- Complex Forward Radix-2 DIT FFT
- Complex Inverse Radix-2 DIT FFT
- Real Forward Radix-2 DIT FFT
- Real Inverse Radix-2 DIT FFT

Important Note on Cycle Count:

The actual cycle count depends upon the dynamic path followed while execution which depends on the input given. The actual cycle count should lie within the range given by higher and lower limit of cycle count.

I

FFT_2_16 Complex Forward Radix-2 DIT FFT for 16 bits

Signature

```
short FFT_2_16(CplxS   *R,
               CplxS   *X,
               CplxS   *TF,
               int     nX
               );
```

Inputs

X	:	Pointer to Input-Buffer of 16 bit complex value
TF	:	Pointer to Twiddle- Factor-Buffer of 16 bit complex value in predefined format
nX	:	Size of Input-Buffer (power of 2)

Output

R	:	Pointer to Output-Buffer of 16 bit complex value

Return

NF	:	Scaling factor used for normalization

Description

This function computes the Complex Forward Radix-2 decimation-in-time Fast fourier transform on the given input complex array. The detailed implementation is given in the **Section 4.8**.

FFT_2_16 Complex Forward Radix-2 DIT FFT for 16 bits (cont'd)

Pseudo code

```
{
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * y->real + k->real * y->imag);
           y'->real = x->real - (k->real * y->real - k->imag * y->imag);
           y'->imag = x->imag - (k->real * y->imag + k->imag * y->real);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q15 format
- Input and Output has real and imaginary part packed as 16 bit data to form 32 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order

FFT_2_16 **Complex Forward Radix-2 DIT FFT for 16 bits** (cont'd)

Memory Note

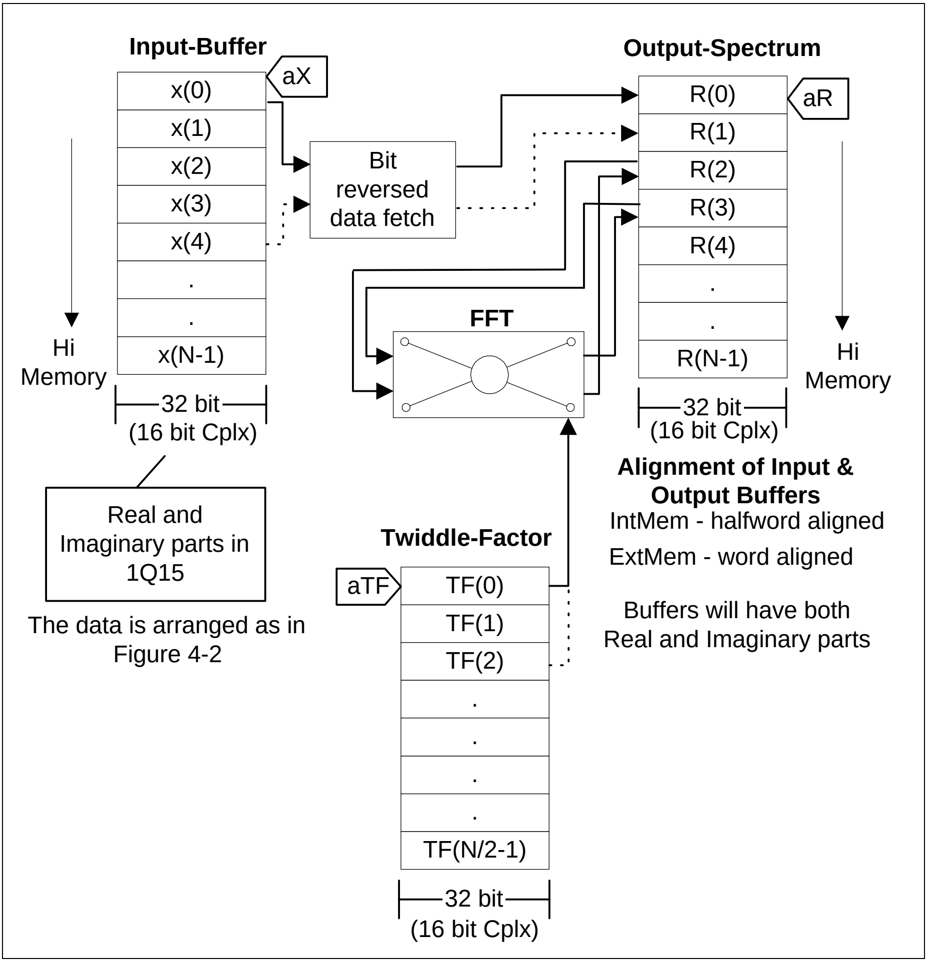


Figure 4-69 FFT_2_16

Implementation Refer Section 4.8.2

FFT_2_16 Complex Forward Radix-2 DIT FFT for 16 bits (cont'd)

Example

Trilib\Example\Tasking\Transforms\FFT\expCplxFFT_2_16.c, expCplxFFT_2_16.cpp
Trilib\Example\GreenHills\Transforms\FFT\expCplxFFT_2_16.cpp, expCplxFFT_2_16.c
Trilib\Example\GNU\Transforms\FFT\expCplxFFT_2_16.c

Cycle Count

Initialization : 7

First Pass Loop : $7 + 7 \times N/2 + 2$

Kernel : $10 \times (\mathrm{Log}_2 N - 1) + 2$
$+8 \times (N/2 - 1) + 2$
$+(13 \mathrm{or} 11)(\mathrm{Log}_2 N - 1) \times N/4 + 2$

- Stage Loop : $10 \times (\mathrm{Log}_2 N - 1) + 2$
- Group Loop : $8 \times (N/2 - 1) + 2$
- Butterfly : $(13 \mathrm{or} 11)(\mathrm{Log}_2 N - 1) \times N/4 + 2$

Post Processing : $6 + 4 \times N/2 + 4$

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	167	172	164
256	8350	8350	7453

Code Size

344 bytes

IFFT_2_16 Complex Inverse Radix-2 DIT IFFT for 16 bits

Signature

```
short IFFT_2_16(CplxS    *R,
                CplxS    *X,
                CplxS    *TF,
                int      nX
                );
```

Inputs	X	:	Pointer to Input-Buffer of 16 bit complex value
	TF	:	Pointer to Twiddle- Factor-Buffer of 16 bit complex number value in predefined format
	nX	:	Size of Input-Buffer (power of 2)
Output	R	:	Pointer to Output-Buffer of 16 bit complex value
Return	NF	:	Scaling factor used for normalization

Description

This function computes the Complex Inverse Radix-2 decimation-in-time Fast fourier transform on the given input complex array. The detailed implementation is given in the **Section 4.8**.

IFFT_2_16 Complex Inverse Radix-2 DIT IFFT for 16 bits (cont'd)

Pseudo code

```
{
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x’->real = x->real + (k->real * y->real - k->imag * y->imag);
           x’->imag = x->imag + (k->imag * y->real - k->imag * y->real);
           y’->real = x->real - (k->real * y->real - y->imag * k->imag);
           y’->imag = x->imag - (k->real * y->imag - y->real * k->imag);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q15 format
- Input and Output has real and imaginary part packed as 16 bit data to form 32 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order

IFFT_2_16 **Complex Inverse Radix-2 DIT IFFT for 16 bits** (cont'd)

Memory Note

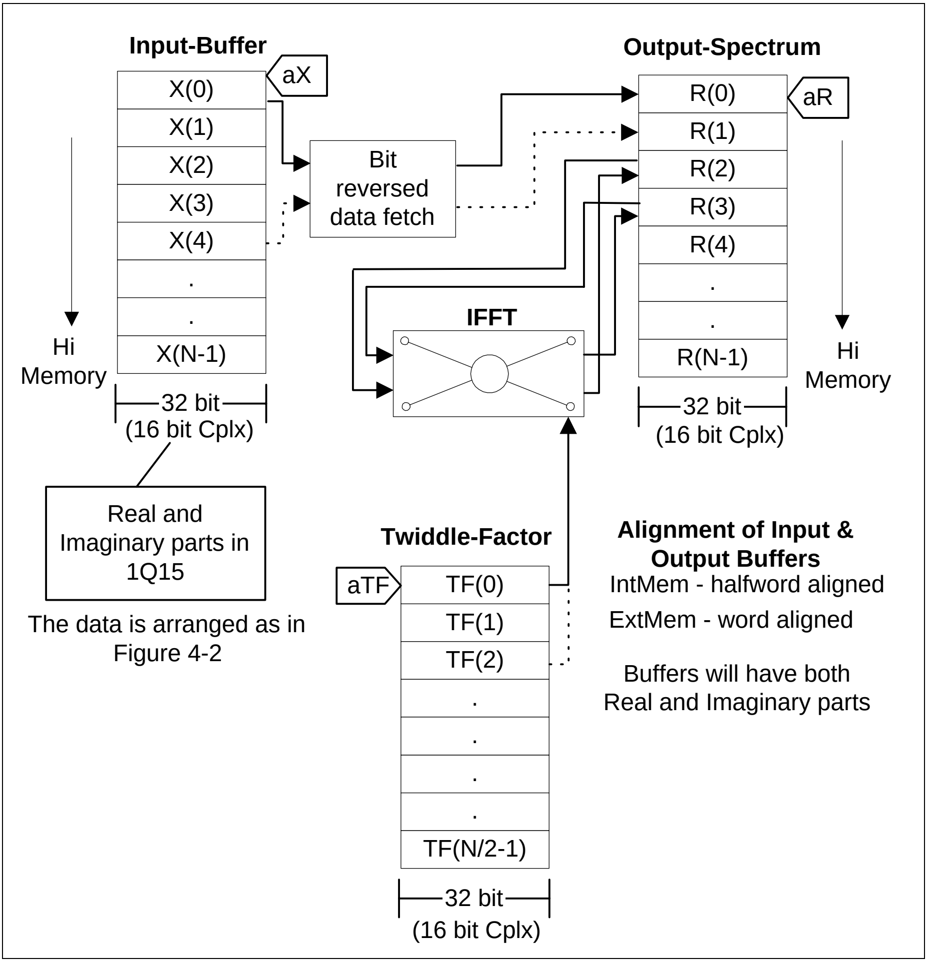


Figure 4-70 IFFT_2_16

Implementation Refer **Section 4.8.2**

IFFT_2_16 Complex Inverse Radix-2 DIT IFFT for 16 bits (cont'd)

Example

Trilib\Example\Tasking\Transforms\FFT \expCplxFFT_2_16.c, expCplxFFT_2_16.cpp
Trilib\Example\GreenHills\Transforms\FFT \expCplxFFT_2_16.cpp, expCplxFFT_2_16.c
Trilib\Example\GNU\Transforms\FFT\expCplxFFT_2_16.c

Cycle Count

Initialization : 7

First Pass Loop : $7 + 7 \times N/2 + 2$

Kernel : $10 \times (\mathrm{Log}_2 N - 1) + 2$
$+8 \times (N/2 - 1) + 2$
$+(13 \mathrm{or} 11)(\mathrm{Log}_2 N - 1) \times N/4 + 2$

- Stage Loop : $10 \times (\mathrm{Log}_2 N - 1) + 2$
- Group Loop : $8 \times (N/2 - 1) + 2$
- Butterfly : $(13 \mathrm{or} 11)(\mathrm{Log}_2 N - 1) \times N/4 + 2$

Post Processing : $6 + 4 \times N/2 + 4$

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	162	172	164
256	7581	8350	7453

Code Size 345 bytes

FFTReal_2_16 Real Forward Radix-2 DIT FFT for 16 bits

Signature

```
short FFTReal_2_16(CplxS   *R,
                   CplxS   *X,
                   CplxS   *TF,
                   int     nX
                   );
```

Inputs	X	:	Pointer to Input-Buffer of 16 bit complex value
	TF	:	Pointer to Twiddle- Factor-Buffer of 16 bit complex value in predefined format
	nX	:	Size of Input-Buffer (power of 2)
Output	R	:	Pointer to Output-Buffer of 16 bit complex value
Return	NF	:	Scaling factor used for normalization

Description

This function computes the Real Forward Radix-2 decimation-in-time Fast Fourier Transform on the given input complex array. The detailed implementation is given in the **Section 4.8**. The Real FFT is implemented by using the complex FFT and the output spectrum is split to separate the Real FFT results.

FFTReal_2_16 Real Forward Radix-2 DIT FFT for 16 bits (cont'd)

Pseudo code

```
{
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * y->real + k->imag * y->real);
           y'->real = x->real - (k->real * y->real - y->imag * k->imag);
           y'->imag = x->imag - (k->real * y->imag + y->real * k->imag);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
   Split Spectrum     // separate the real from the complex output
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q15 format
- Input and Output has real and imaginary part packed as 16 bit data to form 32 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order
- Input contains two real sequences, x1 and x2, each of length N. x1 is in real part and x2 is in imaginary part of input complex data
- The output spectra has two complex blocks, each of length N, wherein the first block is for x1 and subsequent block for x2

FFTReal_2_16 Real Forward Radix-2 DIT FFT for 16 bits (cont'd)

Memory Note

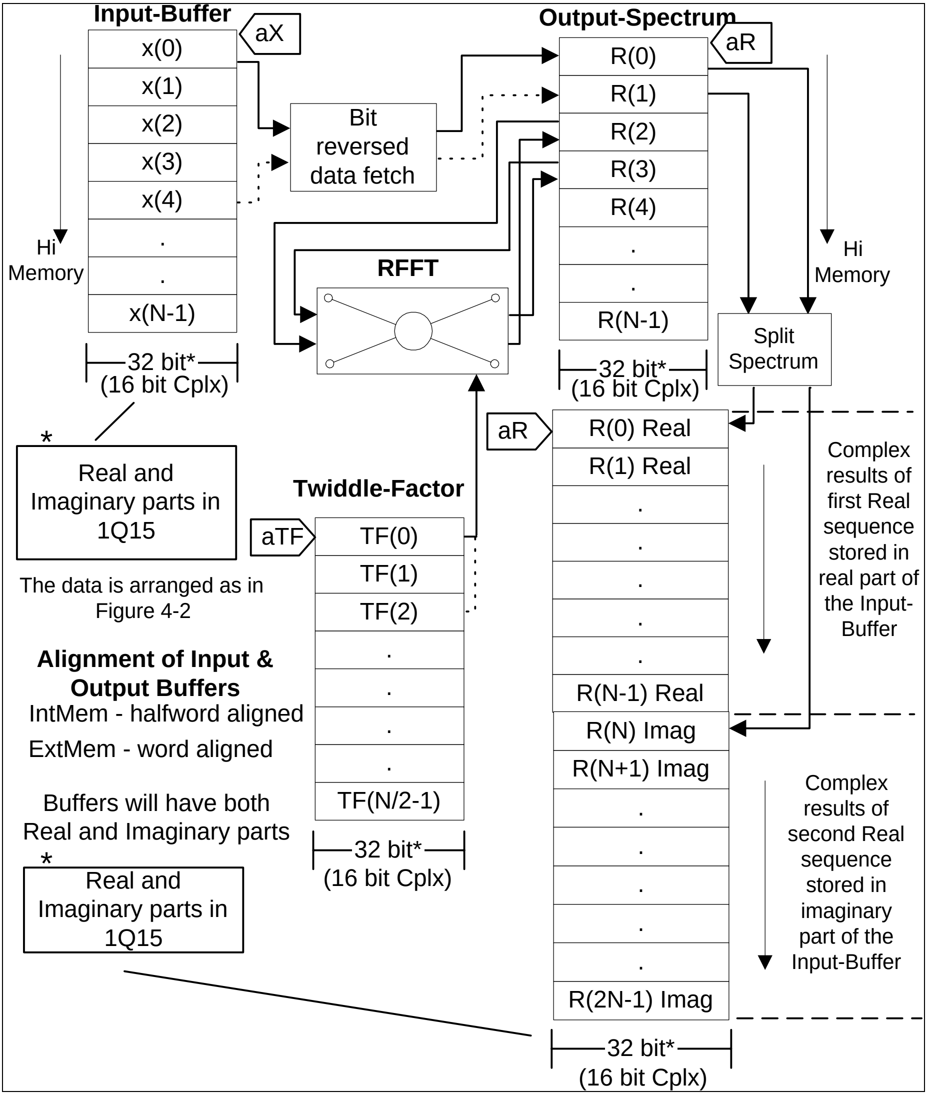


Figure 4-71 FFTReal_2_16

FFTReal_2_16 — Real Forward Radix-2 DIT FFT for 16 bits (cont'd)

Implementation

Refer Section 4.8.2

Example

Trilib\Example\Tasking\Transforms\FFT\expRealFFT_2_16.c, expRealFFT_2_16.cpp
Trilib\Example\GreenHills\Transforms\FFT\expRealFFT_2_16.cpp, expRealFFT_2_16.c
Trilib\Example\GNU\Transforms\FFT\expRealFFT_2_16.c

Cycle Count

Initialization : 7

First Pass Loop : $7 + 7 \times N/2 + 2$

Kernel : $10 \times (\mathrm{Log}_2 N - 1) + 2$
$+8 \times (N/2 - 1) + 2$
$+(13 or 11)(\mathrm{Log}_2 N - 1) \times N/4 + 2$

- Stage Loop : $10 \times (\mathrm{Log}_2 N - 1) + 2$
- Group Loop : $8 \times (N/2 - 1) + 2$
- Butterfly : $(13 or 11)(\mathrm{Log}_2 N - 1) \times N/4 + 2$

Post Processing : $6 + 4 \times N/2 + 4$

Split Spectrum : $14 + 11 \times (N/2 - 1) + 5$

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	219	224	216
256	9766	9766	8869

Code Size

678 bytes

IFFTReal_2_16 Real Inverse Radix-2 DIT IFFT for 16 bits

Signature

```
short IFFTReal_2_16(CplxS    *R,
                    CplxS    *X,
                    CplxS    *TF,
                    int       nX,
                    int       SFlg
                    );
```

Inputs

X	:	Pointer to Input-Buffer of 16 bit complex value
TF	:	Pointer to Twiddle-Factor-Buffer of 16 bit complex value in predefined format
nX	:	Size of Input-Buffer (power of 2)
SFlg	:	Indicates scale down the input by 2 if this flag is TRUE

Output

R	:	Pointer to Output-Buffer of 16 bit complex value

Return

NF	:	Scaling factor used for normalization

Description

This function computes the Real Inverse Radix-2 decimation-in-time Fast fourier transform on the given input complex array. The detailed implementation is given in the **Section 4.8**.The Real IFFT is implemented by using the complex IFFT and before processing the input is arranged to form a single valued complex sequence from two complex sequences.

IFFTReal_2_16 Real Inverse Radix-2 DIT IFFT for 16 bits (cont'd)

Pseudo code

```
{
  unify spectrum    //Forms a single valued complex sequence from two
                      sequences
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * k->real - k->imag * y->real);
           y'->real = x->real - (k->real * y->real - y->imag * k->imag);
           y'->imag = x->imag - (k->real * y->imag - y->real * k->imag);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q15 format
- Input and Output has real and imaginary part packed as 16 bit data to form 32 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order
- Input contains two complex blocks each of length N, wherein the first block is for x1 and subsequent block is for x2
- The output spectra contains two real sequences x1 and x2, each of length N. x1 is in real part and x2 is in imaginary part of output complex data

Caution

- The input array gets modified after processing

IFFTReal_2_16 **Real Inverse Radix-2 DIT IFFT for 16 bits** (cont'd)

Memory Note

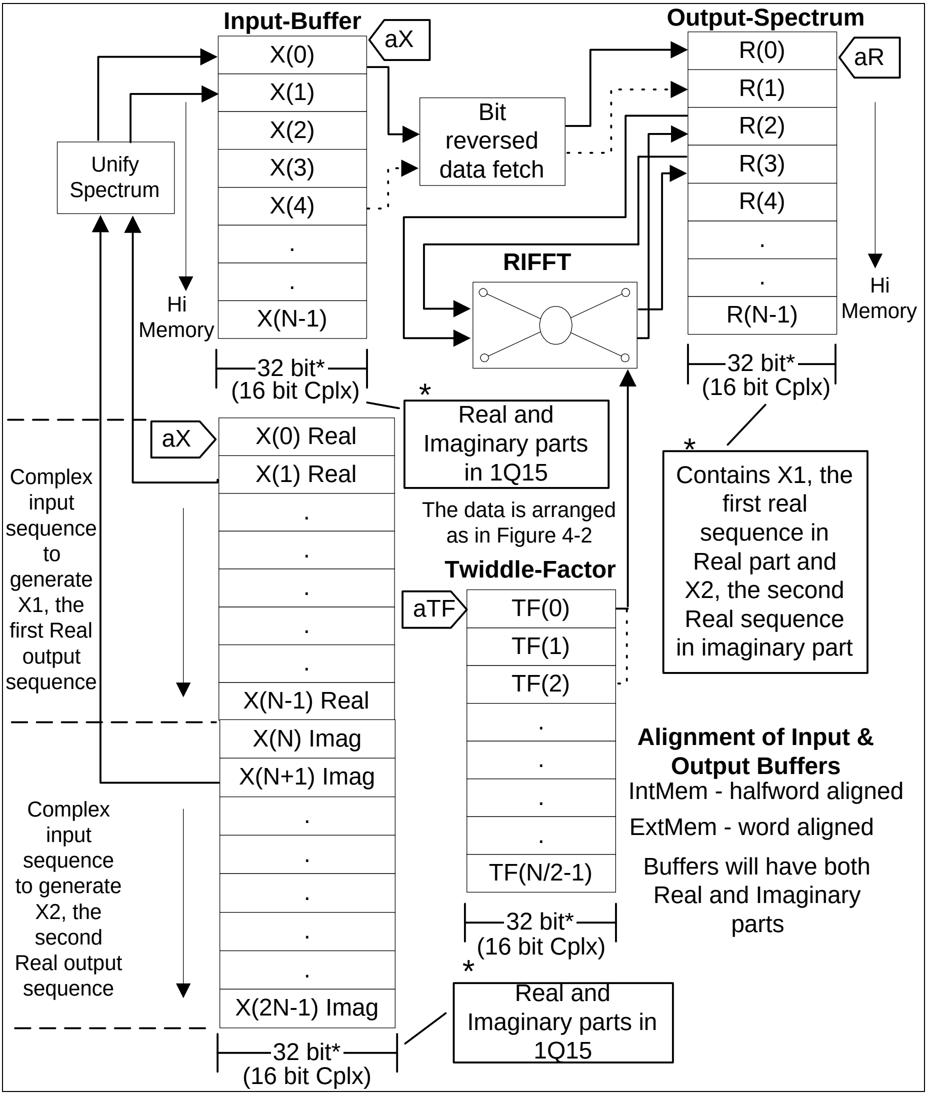


Figure 4-72 IFFTReal_2_16

IFFTReal_2_16 — Real Inverse Radix-2 DIT IFFT for 16 bits (cont'd)

Implementation Refer **Section 4.8.2**

Example

Trilib\Example\Tasking\Transforms\FFT
\expRealFFT_2_16.c, expRealFFT_2_16.cpp
Trilib\Example\GreenHills\Transforms\FFT
\expRealFFT_2_16.cpp, expRealFFT_2_16.c
Trilib\Example\GNU\Transforms\FFT
\expRealFFT_2_16.c

Cycle Count

Initialization : 6

Unify : $5 + (10 \times N/2) + 2$

First Pass Loop : $7 + 7 \times N/2$

Kernel : $10 \times (\mathrm{Log}_2 N - 1) + 2$
$+8 \times (N/2 - 1) + 2$
$+(13 or 11)(\mathrm{Log}_2 N - 1) \times N/4 + 2$

- Stage Loop : $10 \times (\mathrm{Log}_2 N - 1) + 2$
- Group Loop : $8 \times (N/2 - 1) + 2$
- Butterfly : $(13 or 11)(\mathrm{Log}_2 N - 1) \times N/4 + 2$

Post Processing : $6 + 4 \times N/2 + 4$

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	209	219	211
256	8868	9637	8740

Code Size 680 bytes

FFT_2_32 Complex Forward Radix-2 DIT FFT for 32 bits

Signature

```
short FFT_2_32(CplxL   *R,
               CplxL   *X,
               CplxL   *TF,
               int      nX
               );
```

Inputs

X	:	Pointer to Input-Buffer of 32 bit complex value
TF	:	Pointer to Twiddle- Factor-Buffer of 32 bit complex value in predefined format
nX	:	Size of Input-Buffer (power of 2)

Output

R	:	Pointer to Output-Buffer of 32 bit complex value

Return

NF	:	Scaling factor used for normalization

Description

This function computes the Complex Forward Radix-2 decimation-in-time Fast fourier transform on the given input complex array. The detailed implementation is given in the **Section 4.8.4**.

FFT_2_32 Complex Forward Radix-2 DIT FFT for 32 bits (cont'd)

Pseudo code

```
{
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * k->real + k->imag * y->real);
           y'->real = x->real - (k->real * y->real - y->imag * k->imag);
           y'->imag = x->imag - (k->real * y->imag + y->real * k->imag);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q31 format
- Input and Output has real and imaginary part packed as 32 bit data to form 64 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order

FFT_2_32 **Complex Forward Radix-2 DIT FFT for 32 bits** (cont'd)

Memory Note

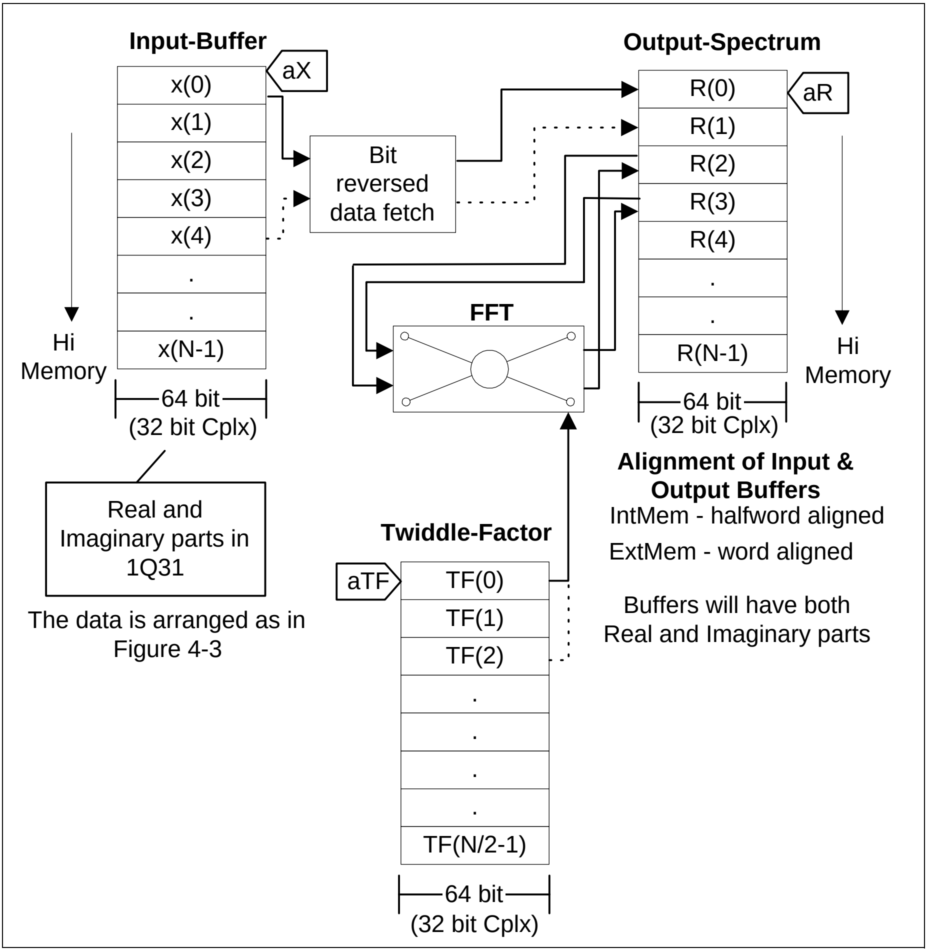


Figure 4-73 FFT_2_32

Implementation Refer **Section 4.8.4**

FFT_2_32 — Complex Forward Radix-2 DIT FFT for 32 bits (cont'd)

Example

Trilib\Example\Tasking\Transforms\FFT\expCplxFFT_2_32.c, expCplxFFT_2_32.cpp
Trilib\Example\GreenHills\Transforms\FFT\expCplxFFT_2_32.cpp, expCplxFFT_2_32.c
Trilib\Example\GNU\Transforms\FFT\expCplxFFT_2_32.c

Cycle Count

Initialization : 8

First Pass Loop : $7 + 9 \times N/2 + 2$

Kernel : $10 \times (\mathrm{Log}_2 N - 1) + 2$
$+7 \times (N/2 - 1) + 2$
$+(20 \mathrm{or} 18)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

- Stage Loop : $10 \times (\mathrm{Log}_2 N - 1) + 2$
- Group Loop : $7 \times (N/2 - 1) + 2$
- Butterfly : $(20 \mathrm{or} 18)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

Post Processing : 4

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	260	264	244
256	19803	20058	18267

Code Size

350 bytes

IFFT_2_32 Complex Inverse Radix-2 DIT IFFT for 32 bits

Signature

```
short IFFT_2_32(CplxL   *R,
                CplxL   *X,
                CplxL   *TF,
                int      nX
                );
```

Inputs

X	:	Pointer to Input-Buffer of 32 bit complex value
TF	:	Pointer to Twiddle- Factor-Buffer of 32 bit complex value in predefined format
nX	:	Size of Input-Buffer (power of 2)

Output

R	:	Pointer to Output-Buffer of 32 bit complex value

Return

NF	:	Scaling factor used for normalization

Description

This function computes the Complex Inverse Radix-2 decimation-in-time Fast fourier transform on the given input complex array. The detailed implementation is given in the **Section 4.8.4**.

IFFT_2_32 Complex Inverse Radix-2 DIT IFFT for 32 bits (cont'd)

Pseudo code

```
{
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * y->real - k->imag * y->real);
           y'->real = x->real - (k->real * y->real - y->imag * k->imag);
           y'->imag = x->imag - (k->real * y->imag - y->real * k->imag);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q31 format
- Input and Output has real and imaginary part packed as 32 bit data to form 64 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order

IFFT_2_32 **Complex Inverse Radix-2 DIT IFFT for 32 bits** (cont'd)

Memory Note

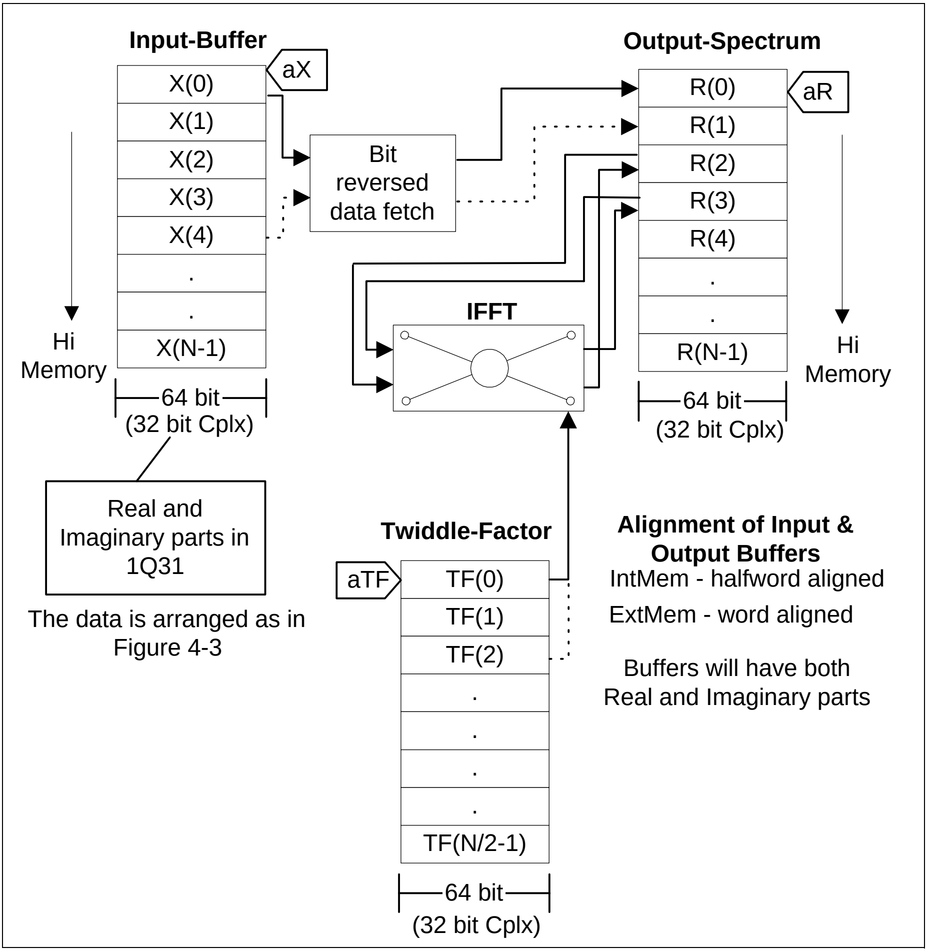


Figure 4-74 IFFT_2_32

Implementation Refer **Section 4.8.4**

IFFT_2_32 Complex Inverse Radix-2 DIT IFFT for 32 bits (cont'd)

Example

Trilib\Example\Tasking\Transforms\FFT\expCplxFFT_2_32.c, expCplxFFT_2_32.cpp
Trilib\Example\GreenHills\Transforms\FFT\expCplxFFT_2_32.cpp, expCplxFFT_2_32.c
Trilib\Example\GNU\Transforms\FFT\expCplxFFT_2_32.c

Cycle Count

Initialization : 8

First Pass Loop : $7 + 9 \times N/2 + 2$

Kernel : $10 \times (\mathrm{Log}_2 N - 1) + 2$
$+7 \times (N/2 - 1) + 2$
$+(20 \mathrm{or} 18)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

- Stage Loop : $10 \times (\mathrm{Log}_2 N - 1) + 2$
- Group Loop : $7 \times (N/2 - 1) + 2$
- Butterfly : $(20 \mathrm{or} 18)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

Post Processing : 4

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	244	264	244
256	18523	20058	18267

Code Size

352 bytes

FFTReal_2_32 Real Forward Radix-2 DIT FFT for 32 bits

Signature

```
short FFTReal_2_32(CplxL    *R,
                   CplxL    *X,
                   CplxL    *TF,
                   int      nX
                   );
```

Inputs	X	:	Pointer to Input-Buffer of 32 bit complex value
	TF	:	Pointer to Twiddle- Factor-Buffer of 32 bit complex value in predefined format
	nX	:	Size of Input-Buffer (power of 2)
Output	R	:	Pointer to Output-Buffer of 32 bit complex value
Return	NF	:	Scaling factor used for normalization

Description

This function computes the Real Forward Radix-2 decimation-in-time Fast fourier transform on the given input complex array. The detailed implementation is given in the **Section 4.8.4**.

FFTReal_2_32 Real Forward Radix-2 DIT FFT for 32 bits (cont'd)

Pseudo code

```
{
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * y->real + k->imag * y->real);
           y'->real = x->real - (k->real * y->real - y->imag * k->imag);
           y'->imag = x->imag - (k->real * y->imag + y->real * k->imag);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
   Split Spectrum     // separate the real from the complex output

}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q31 format
- Input and Output has real and imaginary part packed as 32 bit data to form 64 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order
- Input contains two real sequences, x1 and x2, each of length N. x1 is in real part and x2 is in imaginary part of input complex data
- The output spectra has two complex blocks, each of length N, wherein the first block is for x1 and subsequent block for x2

FFTReal_2_32 **Real Forward Radix-2 DIT FFT for 32 bits** (cont'd)

Memory Note

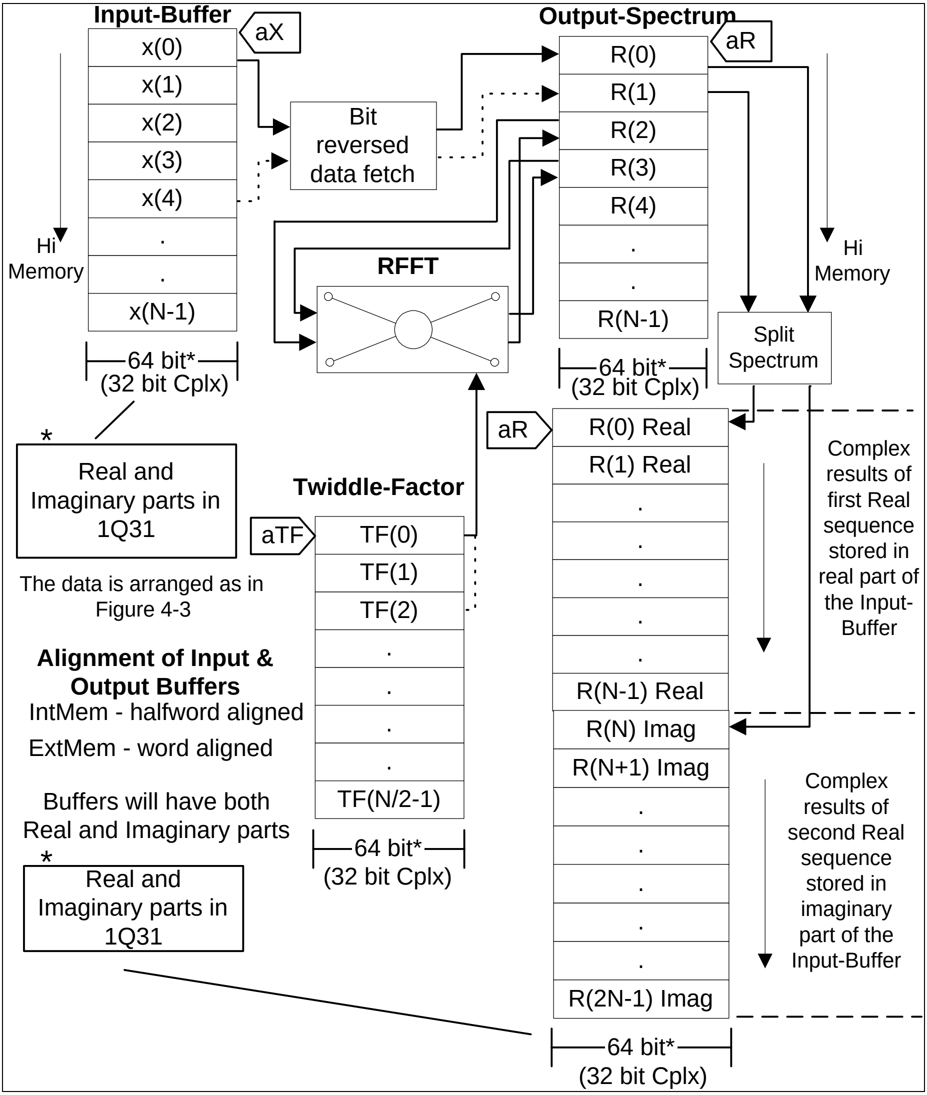


Figure 4-75 FFTReal_2_32

FFTReal_2_32 **Real Forward Radix-2 DIT FFT for 32 bits** (cont'd)

Implementation

Refer **Section 4.8.4**

Example

Trilib\Example\Tasking\Transforms\FFT \expRealFFT_2_32.c, expRealFFT_2_32.cpp
Trilib\Example\GreenHills\Transforms\FFT \expRealFFT_2_32.cpp, expRealFFT_2_32.c
Trilib\Example\GNU\Transforms\FFT \expRealFFT_2_32.c

Cycle Count

Initialization	:	8
First Pass Loop	:	$7 + 9 \times N/2 + 2$
Kernel	:	$10 \times (\text{Log}_2 N - 1) + 2$ $+7 \times (N/2 - 1) + 2$ $+(20 \text{or} 18)(\text{Log}_2 N - 1) \times N/2 + 2$
• Stage Loop	:	$10 \times (\text{Log}_2 N - 1) + 2$
• Group Loop	:	$7 \times (N/2 - 1) + 2$
• Butterfly	:	$(20 \text{or} 18)(\text{Log}_2 N - 1) \times N/2 + 2$
Post Processing	:	4
Split Spectrum	:	$13 + 8 \times (N/2 - 1) + 5$

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	302	306	286
256	20837	21092	19301

Code Size

784 bytes

IFFTReal_2_32 Real Inverse Radix-2 DIT IFFT for 32 bits

Signature

```
short IFFTReal_2_32(CplxL  *R,
                    CplxL  *X,
                    CplxL  *TF,
                    int    nX,
                    int    SFlg
                    );
```

Inputs	X	:	Pointer to Input-Buffer of 32 bit complex value
	TF	:	Pointer to Twiddle- Factor-Buffer of 32 bit complex value in predefined format
	nX	:	Size of Input-Buffer (power of 2)
	SFlg	:	Indicates scale down the input by 2 if this flag is TRUE
Output	R	:	Pointer to Output-Buffer of 32 bit complex value
Return	NF	:	Scaling factor used for normalization

Description

This function computes the Real Inverse Radix-2 decimation-in-time Fast fourier transform on the given input complex array. The detailed implementation is given in the **Section 4.8.4**. The Real IFFT is implemented by using the complex IFFT and before processing the input is arranged to form a single valued complex sequence from two complex sequences.

IFFTReal_2_32 Real Inverse Radix-2 DIT IFFT for 32 bits (cont'd)

Pseudo code

```
{
  unify spectrum    //Forms a single valued complex sequence from two
                      sequences
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * k->real - k->imag * y->real);
           y'->real = x->real - (k->real * y->real - y->imag * k->imag);
           y'->imag = x->imag - (k->real * y->imag - y->real * k->imag);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q31 format
- Input and Output has real and imaginary part packed as 32 bit data to form 64 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order
- Input contains two complex blocks each of length N, wherein the first block is for x1 and subsequent block is for x2
- The output spectra contains two real sequences x1 and x2, each of length N. x1 is in real part and x2 is in imaginary part of output complex data

Caution

- The input array gets modified after processing

IFFTReal_2_32 Real Inverse Radix-2 DIT IFFT for 32 bits (cont'd)

Memory Note

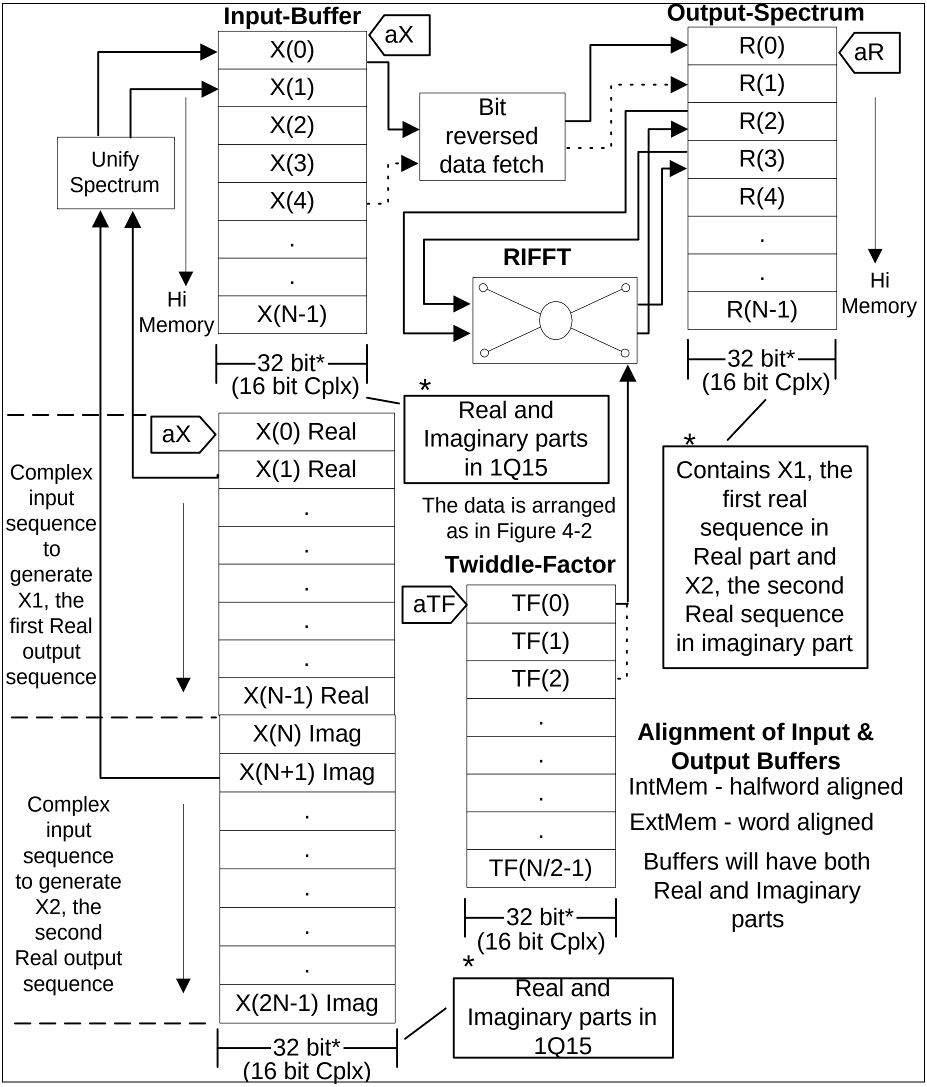


Figure 4-76 IFFTReal_2_32

IFFTReal_2_32 — Real Inverse Radix-2 DIT IFFT for 32 bits (cont'd)

Implementation Refer **Section 4.8.4**

Example

Trilib\Example\Tasking\Transforms\FFT \expRealFFT_2_32.c, expRealFFT_2_32.cpp
Trilib\Example\GreenHills\Transforms\FFT \expRealFFT_2_32.cpp, expRealFFT_2_32.c
Trilib\Example\GNU\Transforms\FFT \expRealFFT_2_32.c

Cycle Count

Initialization : 8

Unify : $4 + 4 \times N + 2$

First Pass Loop : $7 + 9 \times N/2 + 2$

Kernel : $10 \times (\mathrm{Log}_2 N - 1) + 2$
$+7 \times (N/2 - 1) + 2$
$+(20 or 18)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

- Stage Loop : $10 \times (\mathrm{Log}_2 N - 1) + 2$
- Group Loop : $7 \times (N/2 - 1) + 2$
- Butterfly : $(20 or 18)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

Post Processing : 4

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	298	302	282
256	20833	21088	19297

Code Size 816 bytes

FFT_2_16X32 Complex Forward Radix-2 DIT 16 bit mixed FFT

Signature

```
short FFT_2_16X32(CplxS    *R,
                  CplxS    *X,
                  CplxS    *TF,
                  int      nX
                  );
```

Inputs	X	: Pointer to Input-Buffer of 16 bit complex value
	TF	: Pointer to Twiddle- Factor-Buffer of 16 bit complex value in predefined format
	nX	: Size of Input-Buffer (power of 2)
Output	R	: Pointer to Output-Buffer of 16 bit complex value
Return	NF	: Scaling factor used for normalization

Description

This function computes the Complex Forward Radix-2 decimation-in-time Fast fourier transform on the given input complex array with better precision where it internally uses 32 bit for computation. The detailed implementation is given in the **Section 4.8**.

FFT_2_16X32 Complex Forward Radix-2 DIT 16 bit mixed FFT (cont'd)

Pseudo code

```
{
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * y->real + k->real * y->imag);
           y'->real = x->real - (k->real * y->real - k->imag * y->imag);
           y'->imag = x->imag - (k->real * y->imag + k->imag * y->real);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q15 format
- Input and Output has real and imaginary part packed as 16 bit data to form 32 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order

FFT_2_16X32 Complex Forward Radix-2 DIT 16 bit mixed FFT (cont'd)

Memory Note

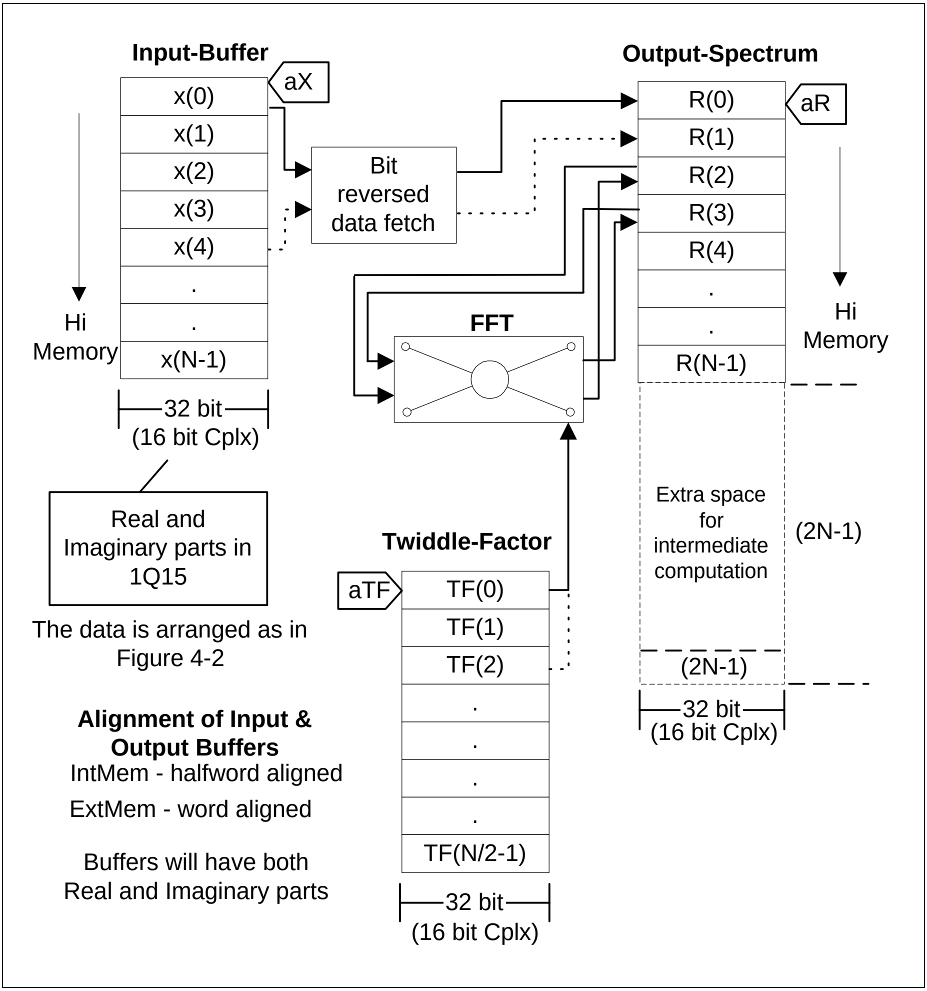


Figure 4-77 FFT_2_16X32

Implementation Refer **Section 4.8.3**

FFT_2_16X32 Complex Forward Radix-2 DIT 16 bit mixed FFT (cont'd)

Example

Trilib\Example\Tasking\Transforms\FFT \expCplxFFT_2_16X32.c, expCplxFFT_2_16X32.cpp
Trilib\Example\GreenHills\Transforms\FFT \expCplxFFT_2_16X32.cpp, expCplxFFT_2_16X32.c
Trilib\Example\GNU\Transforms\FFT \expCplxFFT_2_16X32.c

Cycle Count

Initialization : 8

First Pass Loop : $10 + 9 \times nX/2$

Kernel : $10 \times (\text{Log}_2 N - 1) + 2$
$+7 \times (N/2 - 1) + 2$
$+(16 or 14)(\text{Log}_2 N - 1) \times N/2 + 2$

- Stage Loop : $10 \times (\text{Log}_2 N - 1) + 2$
- Group Loop : $7 \times (N/2 - 1) + 2$
- Butterfly : $(16 or 14)(\text{Log}_2 N - 1) \times N/2 + 2$

Post Processing : $11 + 4 \times nX$

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	269	272	256
256	17508	17508	15712

Code Size

374 bytes

IFFT_2_16X32 Complex Inverse Radix-2 DIT 16 bit mixed IFFT

Signature	short IFFT_2_16X32(CplxS *R, CplxS *X, CplxS *TF, int nX);		
Inputs	X	:	Pointer to Input-Buffer of 16 bit complex value
	TF	:	Pointer to Twiddle- Factor-Buffer of 16 bit complex number value in predefined format
	nX	:	Size of Input-Buffer (power of 2)
Output	R	:	Pointer to Output-Buffer of 16 bit complex value
Return	NF	:	Scaling factor used for normalization

Description

This function computes the Complex Inverse Radix-2 decimation-in-time Fast fourier transform on the given input complex array with better precision where it internally uses 32 bit for computation. The detailed implementation is given in the **Section 4.8**.

IFFT_2_16X32 Complex Inverse Radix-2 DIT 16 bit mixed IFFT (cont'd)

Pseudo code

```
{
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * y->real - k->imag * y->real);
           y'->real = x->real - (k->real * y->real - y->imag * k->imag);
           y'->imag = x->imag - (k->real * y->imag - y->real * k->imag);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q15 format
- Input and Output has real and imaginary part packed as 16 bit data to form 32 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order

IFFT_2_16X32 Complex Inverse Radix-2 DIT 16 bit mixed IFFT (cont'd)

Memory Note

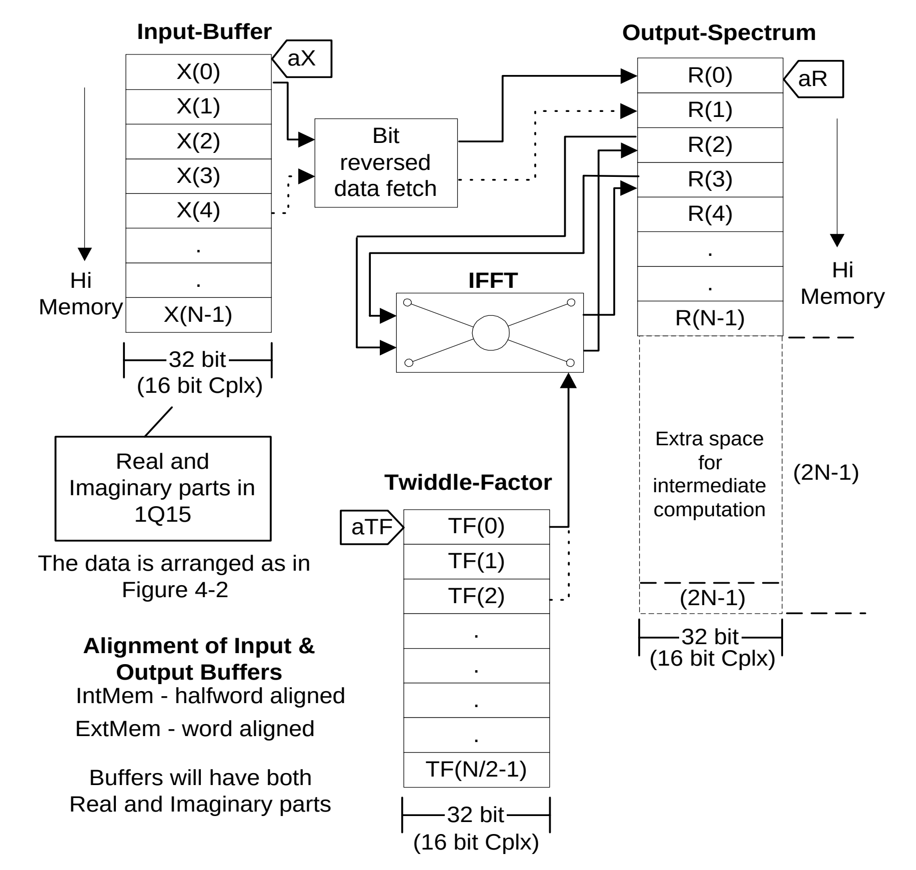


Figure 4-78 IFFT_2_16X32

Implementation Refer **Section 4.8.3**

IFFT_2_16X32 Complex Inverse Radix-2 DIT 16 bit mixed IFFT (cont'd)

Example

Trilib\Example\Tasking\Transforms\FFT \expCplxFFT_2_16X32.c, expCplxFFT_2_16X32.cpp
Trilib\Example\GreenHills\Transforms\FFT \expCplxFFT_2_16X32.cpp, expCplxFFT_2_16X32.c
Trilib\Example\GNU\Transforms\FFT \expCplxFFT_2_16X32.c

Cycle Count

Initialization : 8

First Pass Loop : $10 + 9 \times nX/2$

Kernel : $10 \times (\mathrm{Log}_2 N - 1) + 2$
$+7 \times (N/2 - 1) + 2$
$+(16 or 14)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

- Stage Loop : $10 \times (\mathrm{Log}_2 N - 1) + 2$
- Group Loop : $7 \times (N/2 - 1) + 2$
- Butterfly : $(16 or 14)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

Post Processing : $11 + 4 \times nX$

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	270	272	256
256	17506	17508	15712

Code Size

376 bytes

FFTReal_2_16X32 Real Forward Radix-2 DIT 16 bit mixed FFT

Signature	short FFTReal_2_16X32(CplxS *R, CplxS *X, CplxS *TF, int nX);		
Inputs	X	:	Pointer to Input-Buffer of 16 bit complex value
	TF	:	Pointer to Twiddle- Factor-Buffer of 16 bit complex value in predefined format
	nX	:	Size of Input-Buffer (power of 2)
Output	R	:	Pointer to Output-Buffer of 16 bit complex value
Return	NF	:	Scaling factor used for normalization

Description

This function computes the Real Forward Radix-2 decimation-in-time Fast Fourier Transform on the given input complex array with better precision where it internally uses 32 bit for computation. The detailed implementation is given in the **Section 4.8**. The Real FFT is implemented by using the complex FFT and the output spectrum is split to separate the Real FFT results.

FFTReal_2_16X32 Real Forward Radix-2 DIT 16 bit mixed FFT (cont'd)

Pseudo code

```
{
   Bit reverse input
   for(l=1;l<=L;l++)  //Loop 1 Stage loop
   {
      for(i=1;i<=I;i++);
                      //Loop 2 Group loop
      {
         for(j=1;j<=J;j++)
                      //Loop 3 Butterfly loop
         {
           x'->real = x->real + (k->real * y->real - k->imag * y->imag);
           x'->imag = x->imag + (k->imag * y->real + k->imag * y->real);
           y'->real = x->real - (k->real * y->real - y->imag * k->imag);
           y'->imag = x->imag - (k->real * y->imag + y->real * k->imag);
         }
         initialize k pointer
         initialize x,y pointer
      }
      I = I/2;
      J = J*2;
   }
   Split Spectrum     // separate the real from the complex output
}
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q15 format
- Input and Output has real and imaginary part packed as 16 bit data to form 32 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order with the real part separated from the complex part

FFTReal_2_16X32 Real Forward Radix-2 DIT 16 bit mixed FFT (cont'd)

Memory Note

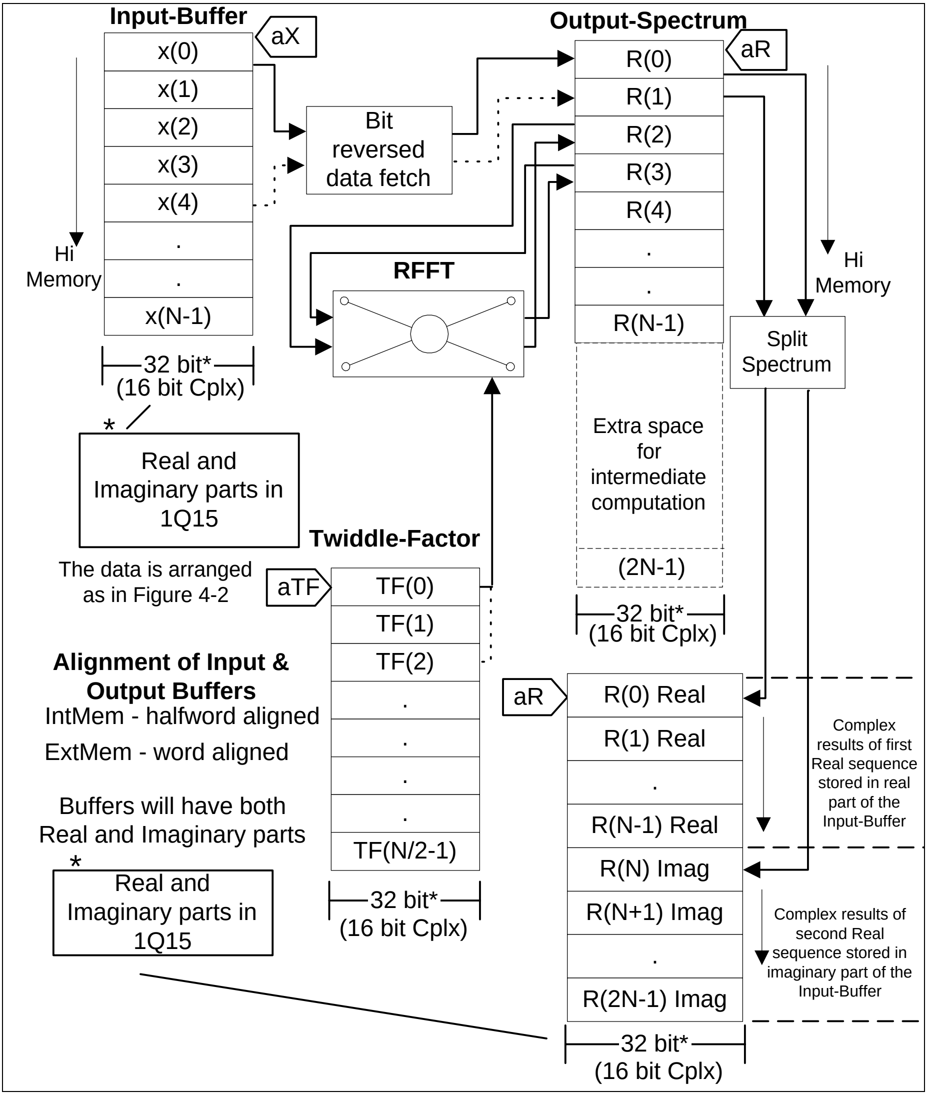


Figure 4-79 FFTReal_2_16X32

FFTReal_2_16X32 Real Forward Radix-2 DIT 16 bit mixed FFT (cont'd)

Implementation Refer **Section 4.8.3**

Example

Trilib\Example\Tasking\Transforms\FFT
\expRealFFT_2_16X32.c, expRealFFT_2_16X32.cpp
Trilib\Example\GreenHills\Transforms\FFT
\expRealFFT_2_16X32.cpp, expRealFFT_2_16X32.c
Trilib\Example\GNU\Transforms\FFT
\expRealFFT_2_16X32.c

Cycle Count

Initialization : 8

First Pass Loop : $10 + 9 \times nX/2$

Kernel : $10 \times (Log_2N - 1) + 2$
$+7 \times (N/2 - 1) + 2$
$+(16or14)(Log_2N - 1) \times N/2 + 2$

- Stage Loop : $10 \times (Log_2N - 1) + 2$
- Group Loop : $7 \times (N/2 - 1) + 2$
- Butterfly : $(16or14)(Log_2N - 1) \times N/2 + 2$

Post Processing : $11 + 4 \times nX$

Split Spectrum : $14 + 11 \times (N/2 - 1) + 5$

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	320	324	308
256	18004	18924	17128

Code Size 662 bytes

IFFTReal_2_16X32 Real Inverse Radix-2 DIT 16 bit mixed IFFT

Signature

```
short IFFTReal_2_16X32(CplxS   *R,
                       CplxS   *X,
                       CplxS   *TF,
                       int     nX,
                       int     SFlg
                      );
```

Inputs

X	:	Pointer to Input-Buffer of 16 bit complex value
TF	:	Pointer to Twiddle-Factor-Buffer of 16 bit complex value in predefined format
nX	:	Size of Input-Buffer (power of 2)
SFlg	:	Indicates scale down the input by 2 if this flag is TRUE

Output

R	:	Pointer to Output-Buffer of 16 bit complex value

Return

NF	:	Scaling factor used for normalization

Description

This function computes the Real Inverse Radix-2 decimation-in-time Fast fourier transform on the given input complex array with better precision where it internally uses 32 bit for computation. The detailed implementation is given in the **Section 4.8**.The Real IFFT is implemented by using the complex IFFT and before processing the input is arranged to form a single valued complex sequence from two complex sequences.

Pseudo code

```
{
  unify spectrum     //Forms a single valued complex sequence from two
                          sequences
   Bit reverse input
```

IFFTReal_2_16X32 Real Inverse Radix-2 DIT 16 bit mixed IFFT (cont'd)

```
    for(l=1;l<=L;l++)  //Loop 1 Stage loop
    {
       for(i=1;i<=I;i++);
                       //Loop 2 Group loop
       {
          for(j=1;j<=J;j++)
                       //Loop 3 Butterfly loop
          {
            x’->real = x->real + (k->real * y->real - k->imag * y->imag);
            x’->imag = x->imag + (k->imag * k->real - k->imag * y->real);
            y’->real = x->real - (k->real * y->real - y->imag * k->imag);
            y’->imag = x->imag - (k->real * y->imag - y->real * k->imag);
          }
          initialize k pointer
          initialize x,y pointer
       }
       I = I/2;
       J = J*2;
    }
 }
```

Techniques

- Packed multiplication
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Inputs are in 1Q15 format
- Input and Output has real and imaginary part packed as 16 bit data to form 32 bit complex data
- Input is halfword aligned in IntMem and word aligned in ExtMem
- Input and Output are in normal order with the real part separated from the complex part
- Input contains two complex blocks each of length N, wherein the first block is for x1 and subsequent block is for x2
- The output spectra contains two real sequences x1 and x2, each of length N. x1 is in real part and x2 is in imaginary part of output complex data

Caution

- The input array gets modified after processing

IFFTReal_2_16X32 Real Inverse Radix-2 DIT 16 bit mixed IFFT (cont'd)

Memory Note

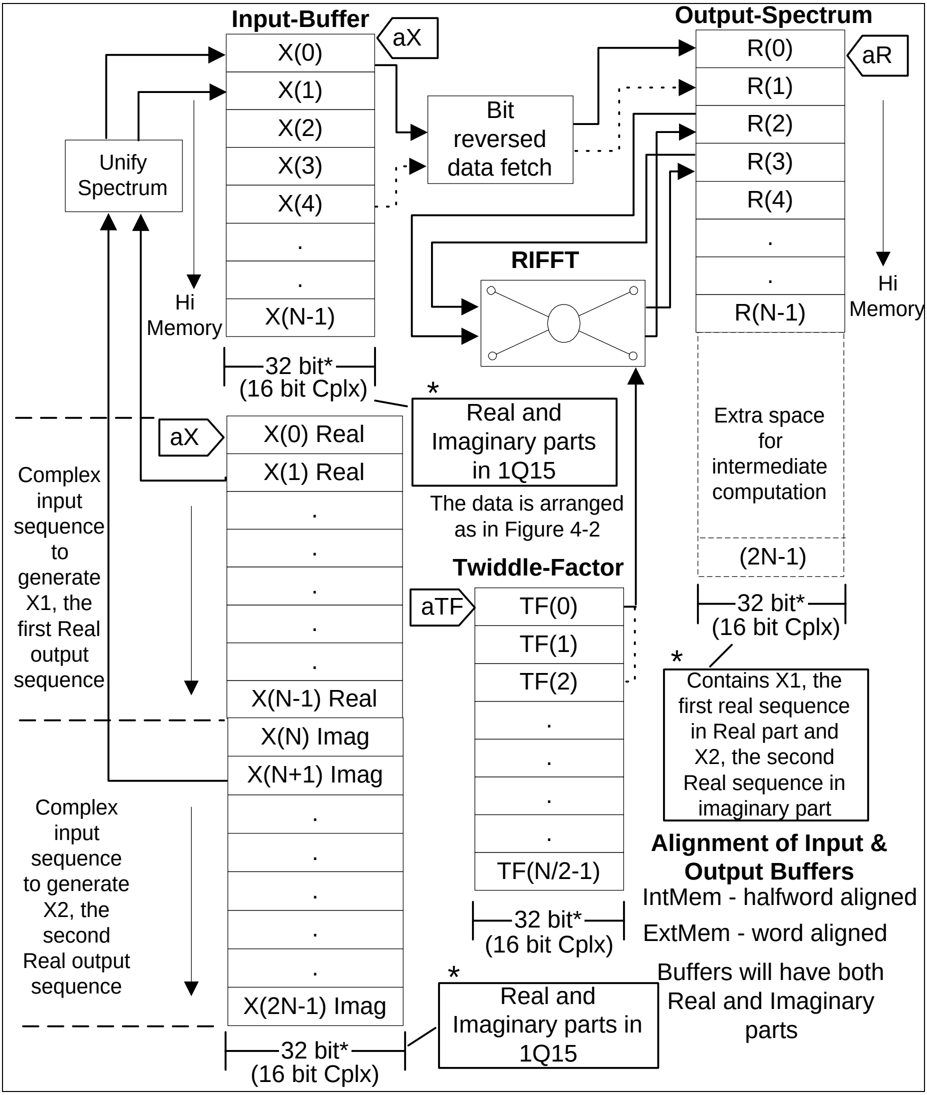


Figure 4-80 IFFTReal_2_16X32

IFFTReal_2_16X32 Real Inverse Radix-2 DIT 16 bit mixed IFFT (cont'd)

Implementation Refer **Section 4.8.3**

Example

Trilib\Example\Tasking\Transforms\FFT \expRealFFT_2_16X32.c, expRealFFT_2_16X32.cpp
Trilib\Example\GreenHills\Transforms\FFT \expRealFFT_2_16X32.cpp, expRealFFT_2_16X32.c
Trilib\Example\GNU\Transforms\FFT \expRealFFT_2_16X32.c

Cycle Count

Initialization : 8

Unify : $5 + (10 \times N/2) + 2$

First Pass Loop : $10 + 9 \times nX/2$

Kernel : $10 \times (\mathrm{Log}_2 N - 1) + 2$
$+7 \times (N/2 - 1) + 2$
$+(16 \mathrm{or} 14)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

- Stage Loop : $10 \times (\mathrm{Log}_2 N - 1) + 2$
- Group Loop : $7 \times (N/2 - 1) + 2$
- Butterfly : $(16 \mathrm{or} 14)(\mathrm{Log}_2 N - 1) \times N/2 + 2$

Post Processing : $11 + 4 \times nX$

Example

N is the number of points of FFT

N	Actual	Higher limit	Lower limit
8	314	319	303
256	17004	18795	16999

Code Size 482 bytes

4.9 Discrete Cosine Transform (DCT)

4.9.1 Algorithm

Similar to the Discrete Fourier Transform (DFT) the Discrete Cosine Transform (DCT) is widely used for transforming a signal or image from the time or spatial domain to the frequency domain. The DCT, especially the two-dimensional (2D) DCT plays an important role in applications such as signal or image compression, e.g. in the JPEG and MPEG standards. In contrast to FFT, DCT is a real valued transform. The one-dimensional (1D) DCT of a discrete time sequence u(n) (n = 0, 1,...,N-1) is defined as

$$v(k) = \sum_{n=0}^{N-1} u(n) \cdot \alpha_N(k) \cos\left[\frac{(2n+1)k\pi}{2N}\right] \quad (k = 0, 1,...,N-1) \qquad [4.126]$$

with

$$\alpha_N(k) = \begin{cases} \sqrt{1/N} & \text{for } k = 0 \\ \sqrt{2/N} & \text{for } k = 1, 2,...N-1 \end{cases}$$

The DCT **Equation [4.126]** can be represented in a matrix vector form

$$v = C_N u \qquad [4.127]$$

where

$$u = \begin{bmatrix} u(0) \\ u(1) \\ u(N-1) \end{bmatrix} \qquad v = \begin{bmatrix} v(0) \\ v(1) \\ v(N-1) \end{bmatrix} \qquad [4.128]$$

$$C_N = \begin{bmatrix} c_N(0,0) & c_N(0,1) & \dots & c_N(0,N-1) \\ c_N(1,0) & c_N(1,1) & \dots & c_N(1,N-1) \\ c_N(N-1,0) & c_N(N-1,1) & \dots & c_N(N-1,N-1) \end{bmatrix} \qquad [4.129]$$

with

$$c_N(k,n) = \alpha_N(k)\cos\left[\frac{(2n+1)k\pi}{2}\right]$$

Notice that C_N is an orthogonal matrix, i.e., its inverse is equal to its transpose.

$$C_N^{-1} = C_N^{T} \qquad [4.130]$$

or

$C_N C_N^T = C_N^T C_N$ = identity matrix

The 2D DCT separates a two dimensional signal (i.e., an image) $u(n_1, n_2)$, (n_1 = 0, 1,...,N_1-1; n_2 = 0, 1,...,N_2-1) into parts or spectral subbands of differing importance (with respect to the visual quality of the image). The transformed image $v(n_1,n_2)$ has the same size $N_1 \times N_2$ and is defined as

$$v(k_1, k_2) = \sum_{n1=0}^{N1-1} \sum_{n2=0}^{N2-1} u(n_1, n_2) \cdot \alpha_{N1}(k_1)\alpha_{N2}(k_2) \tag{4.131}$$

$$\cos\left[\frac{(2n_1+1)k_1\pi}{2N_1}\right]\cos\left[\frac{(2n_2+1)k_2\pi}{2N_2}\right]$$

(k_1 = 0, 1,...,N_1-1; k_2 = 0,1,...,N_2-1)

By using the matrix notation

$$U = \begin{bmatrix} u(0,0) & u(0,1) & u(0, N_2-1) \\ u(1,0) & u(1,1) & u(1, N_2-1) \\ u(N_1-1, 0) & u(N_1-1, 1) & u(N_1-1, N_2-1) \end{bmatrix} \tag{4.132}$$

$$V = \begin{bmatrix} v(0,0) & v(0,1) & v(0, N_2-1) \\ v(1,0) & v(1,1) & v(1, N_2-1) \\ v(N_1-1, 0) & v(N_1-1, 1) & v(N_1-1, N_2-1) \end{bmatrix} \tag{4.133}$$

We can write the 2D DCT as a multiplication of three matrices

$V = C_{N1} U C_{N2}^T$

The $N_1 \times N_2$ matrix C_{N1} and the $N_2 \times N_2$ C_{N2} are defined as in **Equation [4.129]**.

It is easy to see that the 2D DCT is separable into a sequence of 1D DCTs, N_2 times 1D DCTs of the length N_1 applied to the columns of U, followed by another N_1 times 1D DCTs of the length N_2 applied to the rows of $C_{N1}U$. Hence, we can say that the 1D DCT algorithm is the Kernel of the 2D one.

A direct implementation of the DCT given in **Equation [4.126]** requires NxN multiplications and additions/subtractions of the same order. Like the DFT, the DCT can be implemented more efficiently by using a fast algorithm. In the literature many fast DCT algorithms have been developed **"References" on Page 423**. Among them, the sparse matrix factorization algorithms decompose the coefficient matrix C_N into a product of several sparse matrices in order to reduce the number of multiplications and additions. One such algorithm is proposed in **"References" on Page 423**. It is applicable to any DCT whose transform length is a power of 2. For a length N 1D DCT, this algorithm requires $(3N/2)(\log_2 N-1)+2$ real additions and $N\log_2 N-(3N/2)+4$ real multiplications.

The number of additions and multiplications for this particular case is 26 and 16. Note that the input samples u(n) are in natural order while the output samples v'(k) are in bit reversed order. The output samples v'(k) are exactly identical to those defined in **Equation [4.126]** except for scaling

$$v(k) = \sqrt{\frac{2}{N}}\,v'(k) \quad (k = 0, 1,\ldots,N-1) \qquad [4.134]$$

$$= v'(k)/2 \qquad \text{for } N = 8$$

DCT is an orthogonal transform. If we decompose the scaling factor 1/2 in **Equation [4.134]** in two $1/\sqrt{2}$ and scale all butterflies in **Figure 4-81** whose branch coefficients are 1 and -1, by $1/\sqrt{2}$, all butterflies become an orthogonal transform.

In the following, we use this algorithm to compute an 8×8 DCT. A C code is given below. It computes actually 2×8, 8 sample 1D DCTs, based on the signal flow graph in **Figure 4-81**. The first 8 DCTs (j = 8) are applied to the 8 columns of the original image and the last 8 DCTs (j = 1) are applied to the 8 rows of the resulting image. The results we obtain correspond to the transformed image V in **Equation [4.133]** except for a scaling $(\sqrt{2/N})^2 = 2/N$ due to **Equation [4.134]**. The program is for 16 bit fractional data and works in an in-place manner. The 8×8 input image U is stored in the raster scan (row-by-row) order in a buffer of the length 64. The same buffer is also used to store the immediate result C_8U during the processing, as well as the final output V in the same order.

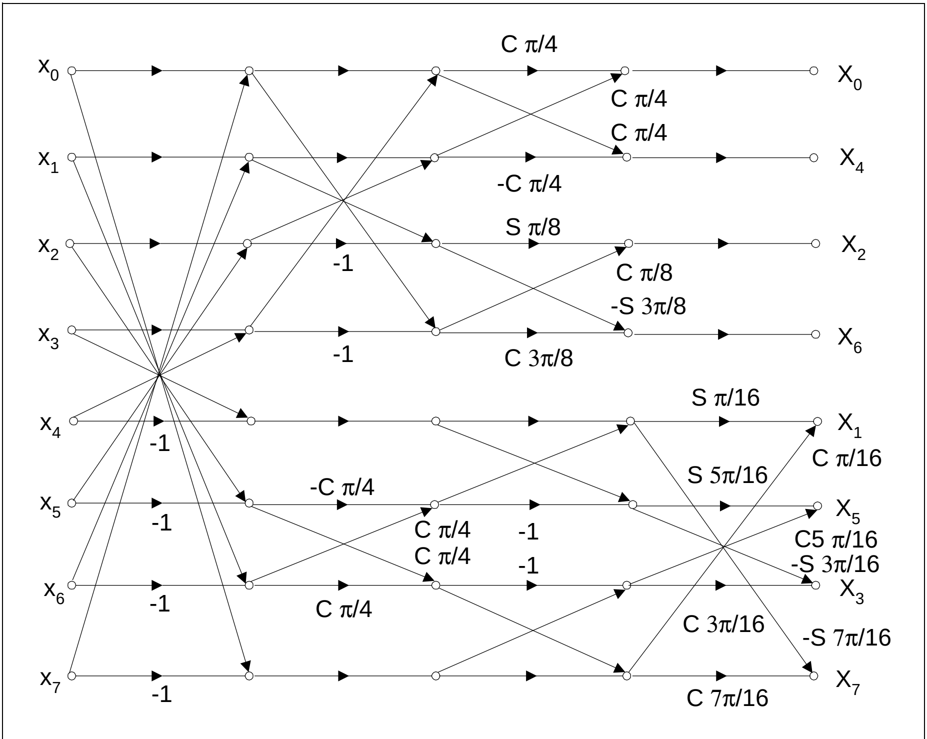


Figure 4-81 Signal Flow Graph for an 8-sample 1D DCT

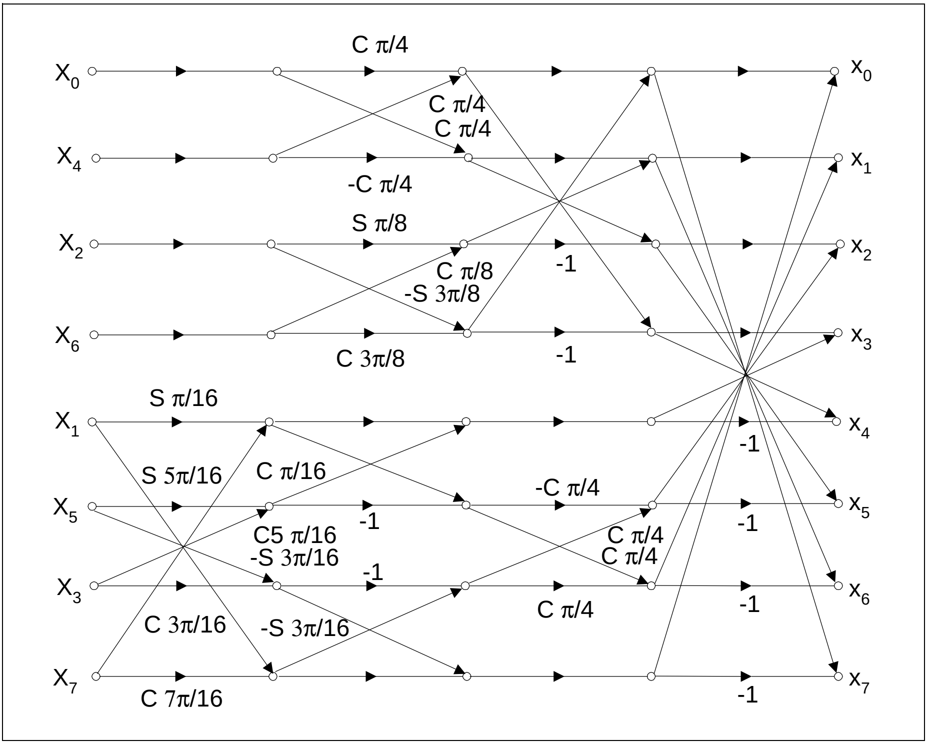


Figure 4-82 Signal Flow Graph for an 8-sample 1D IDCT

4.10 Inverse Discrete Cosine Transform (IDCT)

4.10.1 Algorithm

The Inverse Discrete Cosine Transform (IDCT) is easily derived from the DCT. By multiplying both sides of **Equation [4.127]** with C_N^{-1} from left and considering the orthogonality **Equation [4.130]** we obtain

$u = C_N^T v$

or

$$u(n) = \sum_{k=0}^{N-1} v(k) \cdot \alpha_N(k) \cos\left[\frac{(2n+1)k\pi}{2N}\right] \quad (n = 0, 1, ..., N-1) \qquad [4.135]$$

In other words, to get the IDCT we simply replace the DCT matrix C_N by its transpose C_N^T. The same is true for the 2D IDCT, i.e.

$U = C_{N1}^T V C_{N2}$

or

$$u(n_1, n_2) = \sum_{k1=0}^{N1-1} \sum_{k2=0}^{N2-1} v(k_1, k_2) \cdot \alpha_{N1}(k_1) \alpha_{N2}(k_2) \qquad [4.136]$$

$$\cos\left[\frac{(2n_1+1)k_1\pi}{2N_1}\right] \cos\left[\frac{(2n_2+1)k_2\pi}{2N_2}\right]$$

$(n_1 = 0, 1, ..., N_1-1;\ n_2 = 0, 1, ..., N_2-1)$

For the fast computation of IDCT, we use the same idea **“References” on Page 423** as for DCT. Because each butterfly in **Figure 4-81** represents an orthogonal transform (except for a possible scaling), we only need to reserve the signal flow in **Figure 4-81** in order to get a signal flow graph for IDCT. By introducing the transformed samples v(k) in bit reversed order at the right side, we recover u’(n) in natural order at the left side. The original samples u(n) defined in **Equation [4.135]** are given by

$$u(n) = \sqrt{\frac{2}{N}}\, u'(n) \quad (n = 0, 1, ..., N) \qquad [4.137]$$

$$= u'(n)/2 \qquad \text{for } n = 8$$

like in **Equation [4.134]**. The number of additions and multiplications is exactly the same as for DCT. A C code of 16 bit 8×8 IDCT is given below. It has the same structure as for the DCT and differs only in the reversed signal flow.

4.11 Multidimensional DCT (General Information)

As DCT is a separable transform, 1D DCT, defined in **Equation [4.126]** can be extended to 2D DCT as follows.

2D DCT (separable)

$$X_{u,v}^{c2} = \frac{4}{NM} c_u c_v \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} x_{n,m} \cos\left[\frac{(2n+1)u\pi}{2N}\right] \cos\left[\frac{(2m+1)v\pi}{2M}\right] \qquad [4.138]$$

u = 0, 1,...,N-1, $\quad c_l = 1/\sqrt{2} \quad l = 0$

v = 0, 1,...,M-1, $\quad 1, \quad l \neq 0$

2D IDCT

$$x_{n,m} = \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} c_u c_v X_{u,v}^{c2} \cos\left[\frac{(2n+1)u\pi}{2N}\right] \cos\left[\frac{(2m+1)v\pi}{2M}\right] \qquad [4.139]$$

n = 0, 1,...,N-1

m = 0, 1,...,M-1,

The normalized version of 2D DCT is

2D DCT (normalized)

$$X_{u,v}^{c2} = c_u c_v \frac{2}{\sqrt{NM}} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} x_{n,m} \cos\left[\frac{(2n+1)u\pi}{2N}\right] \cos\left[\frac{(2m+1)v\pi}{2M}\right]$$

$$= \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} c_u \left[\frac{2}{\sqrt{M}} c_v \sum_{m=0}^{M-1} x_{n,m} \cos\frac{(2m+1)v\pi}{2M}\right] \cos\frac{(2n+1)u\pi}{2N} \qquad [4.140]$$

u = 0, 1,...,N-1, $\quad c_l = 1/\sqrt{2} \quad l = 0$

v = 0, 1,...,M-1, $\quad 1, \quad l \neq 0$

2D IDCT (normalized)

$$x_{n,m} = \frac{2}{\sqrt{NM}} \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} c_u c_v X_{u,v}^{c2} \cos\left[\frac{(2n+1)u\pi}{2N}\right] \cos\left[\frac{(2m+1)v\pi}{2M}\right] \qquad [4.141]$$

n = 0, 1,...,N-1

m = 0, 1,...,M-1

DCT is a separable transform, as is IDCT. An implication of this is that 2D DCT can be implemented by a series of 1D DCTs, i.e., 1D DCTs along rows (columns) of a 2D array followed by 1D DCTs along columns (rows) of the semi-transformed array **Figure 4-83**

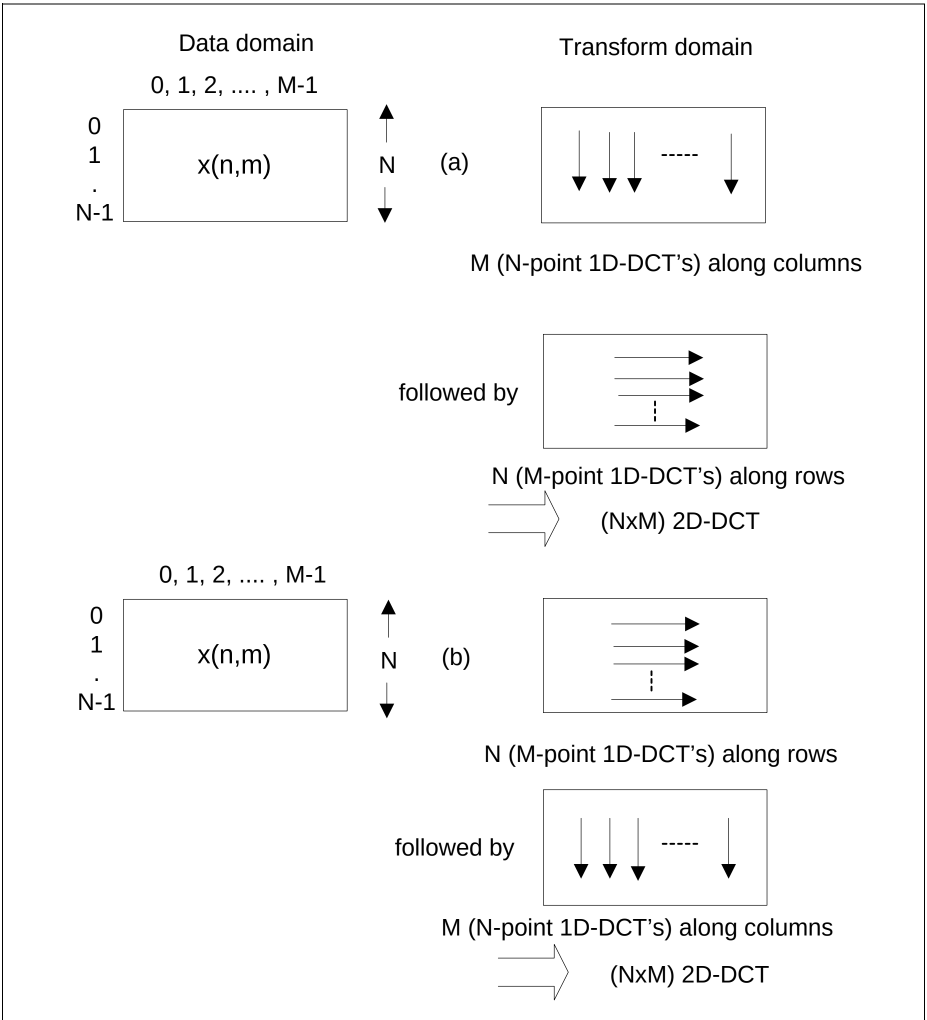


Figure 4-83 Implementation of 2D (NxM) DCT by Series of 1D DCTs

a) 1D DCTs along columns followed by 1D DCTs along rows.
b) 1D DCTs along rows followed by 1D DCTs along columns.

Theoretically, both are equivalent. All the properties of the ID DCT (fast algorithms, recursivity, etc.) extend automatically to the MD-DCT. The separability property can be observed by rewriting **Equation [4.138]** as follows.

$$X_{u,v}^{c2} = \frac{2}{N}\sum_{n=0} c_u\left[\frac{2}{M}\sum_{m=0} c_v x_{n,m}\cos\frac{(2m+1)v\pi}{2M}\right]\cos\frac{(2n+1)u\pi}{2N}$$
$$= \frac{2}{N}\sum^{M-1} c_u\left[\frac{2}{N}\sum^{N-1} c_u x_{n,m}\cos\frac{(2n+1)u\pi}{2N}\right]\cos\frac{(2m+1)v\pi}{2M} \qquad [4.142]$$

u = 0, 1,...,N-1, v = 0, 1,...,M-1,

A similar manipulation on **Equation [4.139]** yields the separability property of the 2D IDCT. This property is illustrated in **Figure 4-83**.

Since DCT is a separable transform, it can be expressed in a matrix form as follows

2D DCT

$$\underset{(N\times N)}{\left[X^{c2}\right]} = \frac{2}{N}\underset{(N\times N)}{\left[C_N^{\pi}\right]}\underset{(N\times N)}{\left[x\right]}\frac{2}{N}\underset{(N\times N)}{\left[C_N^{\pi}\right]^T} \qquad [4.143]$$

2D IDCT

$$\underset{(N\times N)}{\left[x\right]} = \underset{(N\times N)}{\left[C_N^{\pi}\right]^T}\underset{(N\times N)}{\left[X^{c2}\right]}\underset{(N\times N)}{\left[C_N^{\pi}\right]}$$
$$\frac{2}{N}\underset{(N\times N)}{\left[C_N^{\pi}\right]}\underset{(N\times N)}{\left[C_N^{\pi}\right]^T} = \frac{2}{N}\underset{(N\times N)}{\left[C_N^{\pi}\right]^T}\underset{(N\times N)}{\left[C_N^{\pi}\right]} \qquad [4.144]$$
$$= \underset{(N\times N)}{I_N}$$

For the 2D DCT, the sizes (dimensions) along each coordinate need not be the same.

2D DCT

$$\underset{(N\times M)}{\left[X^{c2}\right]} = \frac{2}{N}\underset{(N\times N)}{\left[C_N^{\pi}\right]}\underset{(N\times M)}{\left[x\right]}\frac{2}{M}\underset{(M\times M)}{\left[C_M^{\pi}\right]^T} \qquad [4.145]$$

2D IDCT

$$\underset{(N \times M)}{[x]} = \underset{(N \times N)}{\left[C_N^{\pi}\right]^T} \underset{(N \times M)}{\left[X^{c2}\right]} \underset{(M \times M)}{\left[C_M^{\pi}\right]}$$

$$\frac{2}{N}\left[C_N^{\pi}\right]\left[C_N^{\pi}\right]^T = \frac{2}{N}\left[C_N^{\pi}\right]^T\left[C_N^{\pi}\right] = I_N \qquad [4.146]$$

$$\frac{2}{M}\left[C_M^{\pi}\right]\left[C_M^{\pi}\right]^T = I_M$$

4.11.1 Descriptions

The following DCT functions are described.

- Discrete Cosine Transform
- Inverse Discrete Cosine Transform

4.11.2 2D 8x8 Spatial Block DCT/IDCT Implementation

The DCT, IDCT is implemented using the Chen's **"References" on Page 423** Fast DCT/IDCT one dimensional algorithm which is discussed in the earlier **Section 4.10.1**. The 2D DCT /IDCT exploits the orthogonal property of the algorithm and breaks the 2D 8x8 Spatial block into the 8 rows and 8 columns.

Each row is taken as a whole and is processed by the Chen's ID DCT as in **Equation [4.135]**and the schematic is shown in the signal flow graph **Figure 4-81**. This is achieved by the **RDct1d** macro for the DCT and the **RIdct1d** macro for the IDCT. The column is then processed by the **CDct1d** for the DCT and the **CIDct1d** for the IDCT.

DCT_2_8 Discrete Cosine Transform

Signature	DataS* DCT_2_8(DataS *X);		
Inputs	X	:	Pointer to Real Data block 8×8 array Input coefficients
Output	None		
Return	R	:	Pointer to the Real Data block of 8×8 DCT coefficient

Description

This function implements the 2 dimensional Discrete Cosine Transform. This is implemented using the FDCT algorithm based on the Chen's, that falls in the class of orthogonal DCTs. The data is organized in the 8×8 block, the result is returned in the same block.

DCT_2_8 Discrete Cosine Transform (cont'd)

Pseudo code

```
{
   int t[12],i,j;
   for (j=8; j>0; j-=7,d-=8)
   {
      t[0] = d[0];
      t[1] = d[j];
      t[2] = d[2 * j];
      t[3] = d[3 * j];
      t[4] = d[4 * j];
      t[5] = d[5 * j];
      t[6] = d[6 * j];
      t[7] = d[7 * j];

      t[8] = t[0] + t[7];
      t[7] = t[0] - t[7];
      t[9] = t[1] + t[6];
      t[6] = t[1] - t[6];
      t[10] = t[2] + t[5];
      t[5] = t[2] - t[5];
      t[11] = t[3] + t[4];
      t[4] = t[3] - t[4];

      t[0] = t[8] + t[11];
      t[1] = t[8] - t[11];
      t[2] = t[9] + t[10];
      t[3] = t[9] - t[10];

      t[10] = r[0] * (short) (t[6] - t[5]);
      t[11] = r[0] * (short) (t[6] + t[5]);

      t[8] = t[4] + t[10];
      t[9] = t[4] - t[10];
      t[10] = t[7] + t[11];
      t[11] = t[7] - t[11];

      d[0] = (r[0] * (short)(t[0] + t[2])) >> 15;
      d[j] = (r[3] * t[11] + r[4] * t[8]) >> 15;
      d[2 * j] = (r[1] * t[1] + r[2] * t[3]) >> 15;
      d[3 * j] = (r[5] * t[10] - r[6] * t[9]) >> 15;
      d[4 * j] = (r[0] * (short)(t[0] - t[2])) >> 15;
      d[5 * j] = (r[6] * (t[10] + r[5] * t[9]) >> 15;
      d[6 * j] = (r[2] * t[1] - r[1] * t[3]) >> 15;
      d[7 * j] = (r[4] * t[11] - r[3] * t[8]) >> 15;
```

DCT_2_8 Discrete Cosine Transform (cont'd)

```
    }
}
```

Techniques

- Packed multiplication/addition
- Software pipelining
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Input is real sign extended data packed in 16 bit
- Output is the sign extended data shifted to left by 3 bit positions and packed in 16 bits
- Input is halfword aligned in IntMem and word aligned in ExtMem
- The processing is done inplace so the input block itself gets modified by the program
- Dynamic Input range is -2048 to 2047 before scaling

DCT_2_8 Discrete Cosine Transform (cont'd)

Memory Note

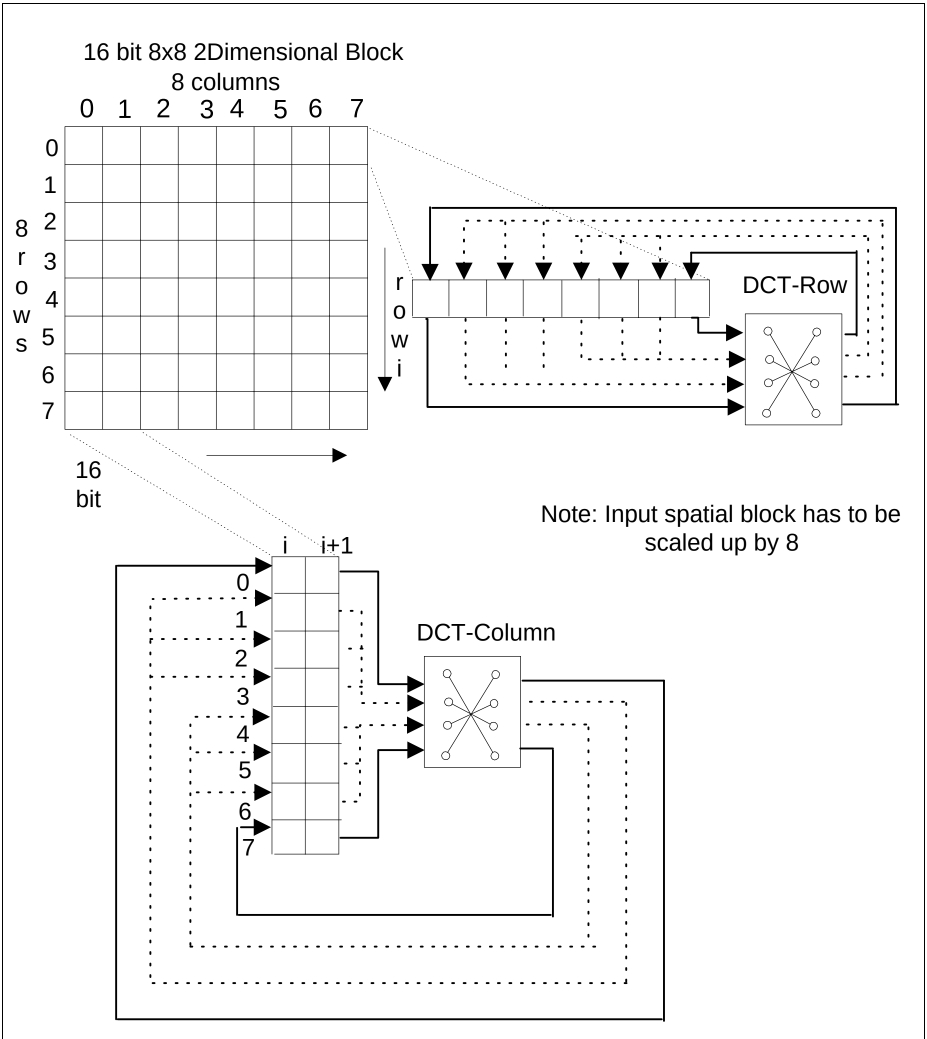


Figure 4-84 DCT_2_8

DCT_2_8 **Discrete Cosine Transform** (cont'd)

Implementation	**Section 4.11.2**
Example	*Trilib\Example\Tasking\Transforms\DCT\expDCT_2_8.c, expDCT_2_8.cpp* *Trilib\Example\GreenHills\Transforms\DCT \expDCT_2_8.cpp, expDCT_2_8.c* *Trilib\Example\GNU\Transforms\DCT\expDCT_2_8.c*
Cycle Count	Initialization : 4 Kernel : 453 Post Processing : 3
Code Size	444 bytes

IDCT_2_8 — Inverse Discrete Cosine Transform

Signature	DataS* IDCT_2_8(DataS *X);		
Inputs	X	:	Pointer to Real Data block 8×8 array Input coefficients
Output	None		
Return	R	:	Pointer to the Real Data block of 8×8 DCT coefficient

Description

This function implements the 2D Inverse Discrete Cosine Transform. This is implemented using the FIDCT algorithm based on the Chen's, that falls in the class of orthogonal DCTs. The data is organized in the 8×8 block, the result is returned in the same block.

IDCT_2_8 Inverse Discrete Cosine Transform (cont'd)

Pseudo code

```
{
   int t[12],i,j;
   for (j=8; j>0; j-=7,d-=8)
   {
      t[0] = d[0];
      t[1] = d[j];
      t[2] = d[2 * j];
      t[3] = d[3 * j];
      t[4] = d[4 * j];
      t[5] = d[5 * j];
      t[6] = d[6 * j];
      t[7] = d[7 * j];

      t[8] = (r[4] * t[1] - r[3] * t[7]) >> 15;
      t[9] = (r[3] * t[1] + r[4] * t[7]) >> 15;
      t[10] = (r[5] * t[5] - r[6] * t[3]) >> 15;
      t[11] = (r[6] * t[5] + r[5] * t[3]) >> 15;

      t[1] = (r[0] * (short) (t[0] + t[4])) >> 15;
      t[3] = (r[0] * (short) (t[0] - t[4])) >> 15;
      t[5] = (r[2] * t[2] - r[1] * t[6]) >> 15;
      t[7] = (r[1] * t[2] + r[2] * t[6]) >> 15;

      t[0] = t[1] + t[7];
      t[2] = t[1] - t[7];
      t[4] = t[3] + t[5];
      t[6] = t[3] - t[5];

      t[1] = t[8] + t[10];
      t[3] = t[8] - t[10];
      t[5] = t[9] - t[11];
      t[7] = t[9] - t[11];

      t[10] = r[0] * (short) (t[5] - t[3]) >> 15;
      t[11] = r[0] * (short) (t[5] + t[3]) >> 15;

      d[0] = t[0] + t[7];
      d[j] = t[4] + t[11];
      d[2 * j] = t[6] + t[10];
      d[3 * j] = t[2] + t[1];
      d[4 * j] = t[2] - t[1];
```

IDCT_2_8 Inverse Discrete Cosine Transform (cont'd)

```
        d[5 * j] = t[6] - t[10];
        d[6 * j] = t[4] - t[11];
        d[7 * j] = t[0] - t[7];
    }
}
```

Techniques

- Packed multiplication/additions
- Load/Store scheduling
- Packed Load/Store

Assumptions

- Input is real sign extended data packed in 16 bit and has to be scaled up by a factor of 8 (left shifted by 3)
- Output is the sign extended data packed in the 16 bit
- Input is halfword aligned in IntMem and word aligned in ExtMem
- The processing is done inplace so the input block itself gets modified by the program
- Dynamic Input range is -2048 to 2047 before scaling

IDCT_2_8 Inverse Discrete Cosine Transform (cont'd)

Memory Note

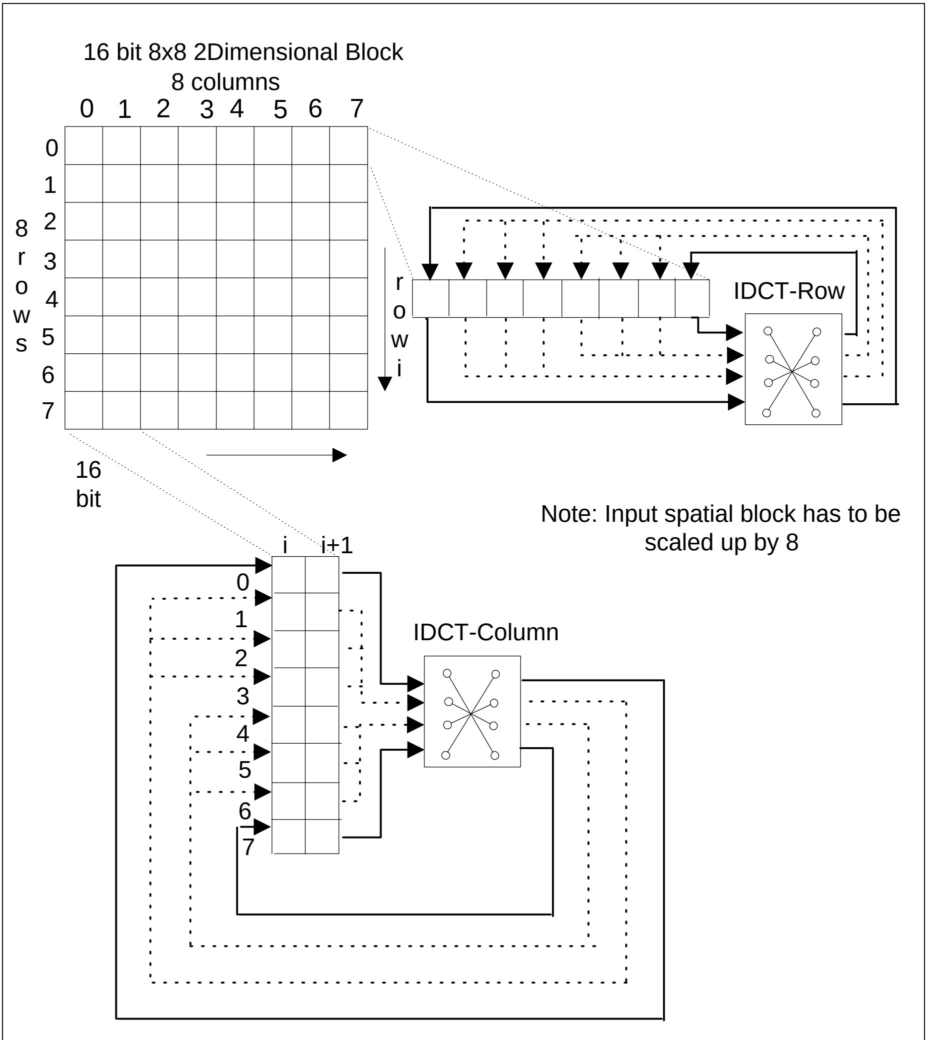


Figure 4-85 IDCT_2_8

IDCT_2_8 **Inverse Discrete Cosine Transform** (cont'd)

Implementation Section 4.11.2

Example *Trilib\Example\Tasking\Transforms\DCT\expDCT_2_8.c, expDCT_2_8.cpp*
Trilib\Example\GreenHills\Transforms\DCT \expDCT_2_8.cpp, expDCT_2_8.c
Trilib\Example\GNU\Transforms\DCT\expDCT_2_8.c

Cycle Count

Initialization	:	4
Kernel	:	439
Post Processing	:	3

Code Size 430 bytes

4.12 Mathematical Functions

4.12.1 Functions using Polynomial Approximation

The Mathematical and Trignometrical functions can be approximated by polynomial expansion. Generally, Taylor & McLaren series are used for expansion of these functions. The function uses the coefficients calculated by statistical analysis technique of regression. Only limited terms of series are used. To improve the accuracy of the output of the function, the optimized coefficients are used.

4.12.1.1 Descriptions

The following series functions are described.

- Sine
- Cosine
- Arctan
- Square Root
- Natural log
- Natural Antilog
- Exponential
- X Power Y

Sine_32 Sine

Signature DataS Sine_32(int X);

Inputs X : The radian input in [-pi,pi] range

Output None

Return R : Output sine value of the function

Description This function calculates the sine of an angle. It takes 32 bit input which represents the angle in radians and returns the 16 bit sine value.

Pseudo code

```
{
    int Xabs;           //Stores Absolute value
    int sign;           //sign of the result
    frac32 XbyPi;       //angle scaled down by pi
    frac32 acc;         //Output of polynomial calculation in 4Q28 format
    frac32 Rf;          //32-bit Sine value in 1Q31
    frac16 R;           //Result in 1Q15 format

    Xabs = |X|;
    if (Xabs != X)
       sign = 1;        //sign = 1 if X is in III or IV quadrant

    if (Xabs > Pi/2)
       Xabs = Pi - Xabs;
                        //if input angle in II or III quadrant subtract
                        //absolute value from pi
    XbyPi = Xabs (*) one_Pi;
                     //angle is scaled down by pi before being used in the
                      //polynomial calculation
    acc = ((((H[4] (*) XbyPi + H[3]) (*) XbyPi + H[2]) (*) XbyPi
          + H[1]) (*) XbyPi + H[0]) (*) XbyPi;
                        //polynomial calculation - acc in 4Q28 format
    acc = acc << 3;     //acc in 1Q31 format
    if (sign == 1)
       Rf = 0 - acc;    //sine is negative in III and IV quadrant
    R = (frac16)Rf;     //16 bit result in 1Q15 format
    return R;           //Returns the calculated sine value
 }
```

Techniques
- Use of MAC instructions
- Instruction ordering provided for zero overhead Load/Store

Assumptions
- Input is the radian value in 3Q29 format, output is the sine value in 1Q15 format and coefficients are in 4Q28 format

Sine_32 — **Sine** (cont'd)

Memory Note None

Implementation Sin(x), where x is in radians is approximated using the polynomial expansion.

$$\sin(x) = 3.140625(x/\pi) + 0.02026367(x/\pi)^2 - 5.325196(x/\pi)^3 + 0.5446778(x/\pi)^4 + 1.800293(x/\pi)^5 \quad [4.147]$$

$0 \le x \le \pi/2$ radians.

Sine value in other quadrants is computed by using the relations,
$\sin(-x) = -\sin(x)$ and $\sin(180 - x) = \sin x$

The function takes 32 bit radian input in 3Q29 format to accommodate the range $(-\pi, \pi)$. The output is 16 bits in 1Q15 format. Coefficients are stored in 4Q28 format. Constants pi, pi/2 and 1/pi are also stored in the data segment in 3Q29, 3Q29 and 1Q31 formats respectively.

The absolute value of the radian input is calculated. If the input angle is negative (III/IV Quadrant), then sign=1. If absolute value of the angle is greater than pi/2 (II/III Quadrant), it is subtracted from pi. The angle is then scaled down by pi, converted to 1Q31 and used in polynomial calculation. The result is negated, if sign=1 to give the final sine result.

To have an optimal implementation with zero overhead load store, the polynomial in **Equation [4.147]** is rearranged as below.

$$\sin(x) = ((((1.800293\,(x/\pi) + 0.5446778)(x/\pi) - 5.325196)(x/\pi) + 0.02026367)(x/\pi) + 3.140625)(x/\pi) \quad [4.148]$$

Hence, 4 multiply-accumulate and 1 multiply instruction will compute the expression **Equation [4.148]** with a load of coefficient done in parallel with each of them.

Sine_32 **Sine** (cont'd)

Example

Trilib\Example\Tasking\Mathematical\expSine_32.c, expSine_32.cpp
Trilib\Example\GreenHills\Mathematical\expSine_32.cpp, expSine_32.c
Trilib\Example\GNU\Mathematical\expSine_32.c

Cycle Count

With DSP Extensions

If input angle is in (I/II Quadrant)	:	15+2
If input angle is in (III/IV Quadrant)	:	18+2

Without DSP EXtensions

If input angle is in (I/II Quadrant)	:	16+2
If input angle is in (III/IV Quadrant)	:	19+2

Code Size

76 bytes

32 bytes (Data)

Cos_32 Cosine

Signature DataS Cos_32(int X);

Inputs X : The radian input in [-pi,pi] range

Output None

Return R : Output cosine value of the function

Description This function calculates the cosine of an angle. It takes 32 bit input which represents the angle in radians and returns the 16 bit cosine value.

Pseudo code

```
{
    int Xabs;             //absolute value of angle
    frac32 XbyPi;         //angle scaled down by pi
    frac32 Pi = pi;
    frac32 one_Pi = 1/pi;
                          //Constant 1/pi in 1Q31 format
    int sign;             //sign of the result
    frac32 acc;          //Output of polynomial calculation in 4Q28 format
    frac32 Rf;            //32-bit Cosine value in 1Q31
    frac16 R;             //Result in 1Q15 format

    Xabs = |X|;
    X = Pi/2 - Xabs;      //Complementary angle is calculated
    Xabs = |X|;
    if (X != Xabs)
       sign = 1;          //sign = 1 if input angle is in the II or III
                          //quadrant
    XbyPi = Xabs (*) one_Pi;
                       //angle is scaled down by pi before being used in the
                          //polynomial calculation
    acc = ((((H[4] (*) XbyPi + H[3]) (*) XbyPi + H[2]) (*) XbyPi
          + H[1]) (*) XbyPi + H[0]) (*) XbyPi;
                          //polynomial calculation - acc in 4Q28 format
    Rf = acc << 3;        //acc in 1Q31 format
   if (sign == 1)       //cosine value is negative in the II or III quadrant
       Rf = 0 - acc;

    R = (frac16)Rf;       //cosine result in 1Q15 format
    return R;             //Returns the calculated cosine value
}
```

Techniques

- Use of MAC instructions
- Instruction ordering provided for zero overhead Load/Store

Cos_32 **Cosine** (cont'd)

Assumptions

- Input is the radian value in 3Q29 format, output is the cosine value in 1Q15 format and coefficients are in 4Q28 format

Memory Note

None

Implementation

Cos(x) is approximated by the same polynomial expression used for sine as $\cos(x) = \sin(90 - x)$.

The function takes 32 bit radian input in 3Q29 format to accommodate the range $(-\pi, \pi)$. The output is 16 bits in 1Q15 format. Coefficients are stored in 4Q28 format. Constants pi, pi/2 and 1/pi are also stored in the data segment in 3Q29, 3Q29 and 1Q31 formats respectively.

Absolute value of the radian input is calculated. Its complementary angle is determined. If the complementary angle is negative, the input angle is in II/III Quadrant where cos is negative. Hence sign=1. The absolute value of complementary angle is scaled down by pi, brought to 1Q31 format and is used in the polynomial calculation. If sign=1, the result of the polynomial calculation is negated, to give the final cosine result.

The implementation of the polynomial is optimal with zero overhead Load/Store.

Example

Trilib\Example\Tasking\Mathematical\expCos_32.c, expCos_32.cpp
Trilib\Example\GreenHills\Mathematical\expCos_32.cpp, expCos_32.c
Trilib\Example\GNU\Mathematical\expCos_32.c

Cycle Count

With DSP Extensions

If input angle is in (I/IV Quadrant)	:	15+2
If input angle is in (III/II Quadrant)	:	18+2

Cos_32 **Cosine** (cont'd)

	Without DSP Extensions		
	If input angle is in (I/IV Quadrant)	:	16+2
	If input angle is in (III/II Quadrant)	:	19+2
Code Size	68 bytes		
	28 bytes (Data)		

Arctan_32 Arctan

Signature	short Arctan_32(int X);	
Inputs	X	: tan value in the range $[-2^{15}, 2^{15})$
Output	None	
Return	R	: Output arctan value of the function

Description This function calculates the arc tangent of the input. The input to the function is 32 bits. The input range is $[-2^{15}, 2^{15})$. The function returns 16 bit value which represents the angle in radians.

Arctan_32 Arctan (cont'd)

Pseudo code

```
{
    frac32 Xabs;        //absolute value of input
   frac32 X1Q31;      //|X| or 1/|X| in 1Q31 format used in the polynomial
                        //calculation
   frac32 acc;       //Output of the polynomial calculation in 1Q31 format
    int sign;           //sign of the result
    frac32 Rf;          //32 bit arctan value in 2Q30 format
    frac16 R;           //16 bit arctan result in 2Q14 format

    Xabs = |X|;

    if (X != Xabs)
       sign = 1;        //if input tan value is negative,sign = 1

    if (Xabs > 1)
       X1Q31 = 1/Xabs;
                        //X1Q31 = 1/|X| in 1Q31 format if |X| > 1
    else
       X1Q31 = Xabs << 15;
                        //X1Q31 = |X| in 1Q31 format

    acc = ((((H[4] (*) X1Q31 + H[3]) (*) X1Q31 + H[2]) (*) X1Q31
          + H[1]) (*) X1Q31 + H[0]) (*) X1Q31;
                        //polynomial calculation - acc in 1Q31 format

    if (Xabs > 1)
       acc = 0.5 - acc;//polynomial result is subtracted from 0.5 if
                        //1/|X| has been used in the calculation

   Rf = acc (*) Pi;   //32 bit arctan value in radians - Rf in 2Q30 format
    R = (frac16)Rf;     //16 bit arctan value in radians in 2Q14 format
    return R;           //Returns the calculated arctan value
}
```

Techniques
- Use of MAC instructions
- Instruction ordering provided for zero overhead Load/Store

Assumptions
- Input tan value is in 16Q16 format, output is the angle in radians in 2Q14 format and coefficients are in 1Q31 format

Memory Note None

Arctan_32 — Arctan (cont'd)

Implementation

Arctan(x) in radians is approximated using the following polynomial expansion.
For x<1,

$$\arctan(x) = \pi(0.318253x + 0.003314x^2 - 0.130908x^3 + 0.068542x^4 - 0.009159x^5) \quad [4.149]$$

For $x \geq 1$ the formula

$$\arctan(x) = \pi/2 - \arctan(1/x) \quad [4.150]$$

can be used.

As 1/x < 1 (for x>1), the polynomial of **Equation [4.149]** can be used to compute arctan(1/x).
Combining **Equation [4.149]** and **Equation [4.150]**,

For $x \geq 1$,
$\arctan(x) = \pi(0.5 - \arctan(1/x))$

The input to the function is 32 bits in 16Q16 format. Hence input is in the range [-2^{15}, 2^{15}). The function returns 16 bit output which is the arctan value in radians. Since arctan values lie in the range [-pi/2, pi/2] output format is 2Q14. 32 bits are used to store coefficients in 1Q31 format in the data segment. π value is also stored in 3Q29 format in data segment. The absolute value of the input is taken in a register and if input is less than 0, sign is set to 1. When input is less than 1, the upper 16 bits of absolute value will be zero and the lower 16 bits represent the tan value in 0Q16. Shifting 15 times to the left will bring the input to 1Q31 format. This value is used in polynomial calculation. The output of the polynomial is multiplied by π and if sign=1, the result is negated to give the final arctan result.

If $|x| > 1$, the reciprocal is calculated by dividing a one in 16Q16 format by the given input. The result gives reciprocal of input in 0Q32, which is converted to 1Q31. This value is now used in the polynomial calculation.

Arctan_32 **Arctan** (cont'd)

The result of the polynomial calculation is subtracted from 0.5 and then multiplied by pi. Once again, it is negated if sign =1 to give the final arctan result in radians.

The implementation of the polynomial is optimal with zero overhead Load/Store.

Example

Trilib\Example\Tasking\Mathematical\expArctan_32.c, expArctan_32.cpp
Trilib\Example\GreenHills\Mathematical\expArctan_32.cpp, expArctan_32.c
Trilib\Example\GNU\Mathematical\expArctan_32.c

Cycle Count

For $\|X\| < 1$ and $X > 0$	:	28+2
For $\|X\| < 1$ and $X < 0$	:	31+2
For $\|X\| > 1$ and $X > 0$	:	50+2
For $\|X\| > 1$ and $X < 0$	:	53+2

Code Size

126 bytes

24 bytes(Data)

Sqrt_32 Square Root

Signature	short Sqrt_32(int X);		
Inputs	X	:	Real input value in the range $[0, 2^{14})$
Output	None		
Return	R	:	Output value of the function

Description

This function calculates the square root of a given number. It takes 32 bit input in the range $[0, 2^{14})$ and returns 16 bit square root value in the range $[0, 2^{7})$.

Sqrt_32 Square Root (cont'd)

Pseudo code

```
{
   int Shcnt;          //Shift count
   int Scale;          //Scaling factor
   frac32 acc;         //Result of Polynomial calculation
   frac32 X1Q31;       //Input scaled to 1Q31 format
   frac16 R;           //Result in 8Q8 format

   Shcnt = count_lead_sign(X);
                       // number of leading sign values
   Scale = Shcnt - 15;//Get the scale factor
   X1Q31 = X << Shcnt;// 1Q31 <- 16Q16

  acc = ((((H5 (*) X1Q31 + H4) (*) X1Q31 + H3) (*) X1Q31 + H2) (*) X1Q31 +
        H1) (*) X1Q31 + H0
                       //polynomial calculation - acc in 1Q31 format
   //Input less than 1
   if (Scale >= 0)
   {
      acc = acc (*) SqrtTab[Scale];
                       //acc = acc * Scale factor
      R = (frac16) acc >> 22;
                       //8Q8 format <- 2Q30 format
   }
   //Input greater than 1
   else
   {
      acc = acc (*) SqrtTab[ShCnt+1];
                       //acc = acc * Scale factor
      R =  (frac16) acc >> 14;
                       //8Q8 format <- 10Q22 format
   }
   return R;           //Returns the calculated square root
}
```

Techniques
- Use of MAC instructions
- Instruction ordering for zero overhead Load/Store

Assumptions
- Inputs are in 16Q16 format and returned output is in 8Q8 format
- Input is always positive

Memory Note None

Sqrt_32 Square Root (cont'd)

Implementation

The square root of the input value x can be calculated by using the following approximation series.

$$\mathrm{sqrt}(x) = 1.454895x - 1.34491x^2 + 1.106812x^3 - 0.536499x^4 + 0.1121216x^5 + 0.2075806 \quad [4.151]$$

where, $0.5 \geq x \geq 1$

The coefficients of polynomial are stored in 2Q30 format. The square root table (table of scale factors) stores $(1/\sqrt{2})^n$ in 1Q31 format where n ranges from 0 to 15. This is same as $(\sqrt{2})^n$ in 9Q23 format, where n ranges from 16 to 1. The 32 bit input given is in 16Q16 format which can take values in the range $[-2^{15}, 2^{15})$. As input should be positive it will be subset of actual input range, i.e., it is in the range $[0, 2^{15})$. The 16 bit output returned is in 8Q8 format. So the output values are in the range of $[0, 2^7)$. So it can accommodate inputs in the range $[0, 2^{14})$.

As the polynomial expansion needs input only in the range 0.5 to 1, the given input has to be scaled up or scaled down. If the given input number is greater than 1, then it is scaled down by powers of two, so that scaled input value lies in the range 0.5 to 1.This scaled input is used in polynomial calculation. The calculated output is scaled up by power of $\sqrt{2}$ to get the actual output.

If the input is less than 1, then it is scaled up by power of two, so that scaled value lies in the range 0.5 to 1. This scaled input is used in polynomial calculation. The calculated output is scaled down by power of $1/\sqrt{2}$ to get actual output.

The CLS instructions of TriCore gives directly the shiftcount, to scale up or scale down the input. When input is shifted by this count, it is brought into 1Q15 format. If shiftcount is15, input already exists in the range of 0.5 to 1. If shiftcount is less than 15, indicates input is greater than 1 and has to be scaled down.

Sqrt_32 **Square Root** (cont'd)

If shiftcount is greater than 15, indicates input is less than 1 and has to be scaled up.

Scale factor is obtained as (15-shiftcount). The output of polynomial calculation is scaled by a value from square root table. The appropriate scale factor is obtained and multiplied to get the square root of given input.

The implementation of the polynomial is optimal with zero overhead Load/Store.

Example

Trilib\Example\Tasking\Mathematical\expSqrt_32.c, expSqrt_32.cpp
Trilib\Example\GreenHills\Mathematical\expSqrt_32.cpp, expSqrt_32.c
Trilib\Example\GNU\Mathematical\expSqrt_32.c

Cycle Count

If X>1	:	14+2
If X<=1	:	16+2

Code Size

88 byes

88 bytes(Data)

Ln_32 Natural logarithm

Signature	short Ln_32(int X);		
Inputs	X	:	Real input value in the range $[2^{-16}, 2^{15})$
Output	None		
Return	R	:	Output value of the function

Description This function calculates logarithm of a function to the base e, i.e., natural logarithm. It takes 32 bit input in the range $[2^{-16}, 2^{15})$ and returns the output logarithm in the range $[-2^{4}, 2^{4})$.

Ln_32 Natural logarithm (cont'd)

Pseudo code

```
{
   int Shcnt           //Shift count
   int Scale;          //Scaling factor
   frac32 acc;         //Result of Polynomial calculation
   frac32 Xu1Q31;      //Input scaled to unsigned 1Q31 format
   frac32 Xsub1;       //X-1
   frac32 Rf;          //Output of polynomial calculation
   frac16 R;           //Result in 5Q11 format

   Shcnt = count_lead_sign(X);
                       // number of leading sign values
   Scale = 14 - Shcnt;//Get the scale factor
   Shcnt = Shcnt + 1; //add 1 to shift count to bring input to
                       //1 to 2(unsigned 1Q15)from 0.5 to 1
   Xu1Q31 = X << Shcnt;
                       //unsigned 1Q15 <- 16Q16
   Xsub1 = Xu1Q31 - 1;//X = X - 1

   acc = ((((H4 * Xsub1 + H3) * Xsub1 + H2) * Xsub1 + H1) * Xsub1 + H0) *
         Xsub1
                       //polynomial calculation - acc in 1Q31 format

   acc = acc << 4;     //5Q27 <- 1Q31
   Add = Scale (*) ln2;
                       //Get the adding factor by scaling Ln2
   Add = Add << 12;    //5Q27 <- 17Q15

   Rf = acc + Add;     //Add the factor to get the result in 5Q27
                       //format
   R = (frac16)Rf;     //result in 5Q11 format

   return R;           //Returns the calculated natural logarithm
}
```

Techniques
- Use of MAC instructions
- Instruction ordering for zero overhead Load/Store

Assumptions
- Inputs are in 16Q16 format and returned output is in 5Q11 format
- Input is always positive

Memory Note None

Ln_32 Natural logarithm (cont'd)

Implementation

The natural logarithm of the input value x can be calculated using the following approximation series.

$$\ln(x) = 0.9991150(x-1) - 0.4899597(x-1)^2 + 0.2856751(x-1)^3 - 0.1330566(x-1)^4 + 0.03137207(x-1)^5 \quad [4.152]$$

where, $1 \geq x \geq 2$ which means $0 \geq (x-1) \geq 1$

The coefficients of polynomial are stored in 1Q31 format. The constant ln2 is also stored in 1Q31 format.

The 32 bit input is in 16Q16 format which can take values in the range [-2^{15}, 2^{15}). As input to logarithm should always be positive it will be subset of actual input range, i.e., it is in the range [2^{-16}, 2^{15}). The 16 bit output returned format is in 5Q11 format.

As the polynomial expansion needs x in the range 1 to 2, the input has to be scaled up or scaled down. If the given input number is greater than 1, then it is scaled down. If less than 1, it is scaled up by powers of two, so that scaled input lies in the range 1 to 2. One is subtracted from this scaled input and this is used in polynomial calculation.

The scale factor is positive, if input is greater than 1 and negative, if input is less than 1. The CLS instruction of TriCore gives the shiftcount. When the input is shifted by this shiftcount it will be scaled in the range 0.5 to 1. The polynomial expects input to be in the range 1 to 2 (unsigned). So 1 is added to the shiftcount.

Scale factor is obtained as (14-shiftcount). The output of polynomial is added with scale times ln2 to get the natural logarithm of given input.

The implementation of the polynomial is optimal with zero overhead Load/Store.

Ln_32 **Natural logarithm** (cont'd)

Example	*Trilib\Example\Tasking\Mathematical\expLn_32.c, expLn_32.cpp* *Trilib\Example\GreenHills\Mathematical\expLn_32.cpp, expLn_32.c* *Trilib\Example\GNU\Mathematical\expLn_32.c*
Cycle Count	For all X : 19+2
Code Size	86 bytes 24 bytes (Data)

AntiLn_16 Natural Antilogarithm

Signature int AntiLn_16(short X);

Inputs X : Real Input value in the range [-8, 8)

Output None

Return R : Output value of the function

Description This function calculates antilog of a function. It takes 16 bit input in the range $[-2^3, 2^3)$ and returns 32 bit antilog value in the range $[2^{-16}, 2^{16})$.

Pseudo code

```
{
   int Shcnt           //Shift count
   int Scale;          //Scaling factor
   frac32 acc;         //Result of Polynomial calculation
   frac32 Rf;          //Result of antilog in Q format
   frac32 X1Q31;       //Input scaled to 1Q31 format
   int Expow;          //Power of calculated polynomial
   frac32 R;           //Result in 16Q16 format

   Shcnt = count_lead_sign(X);
                       //number of leading sign values
   X1Q31 = X << Shcnt;//1Q15 <- 4Q12

   Scale = 19 - Shcnt;//Get the scale factor

   acc = ((((H5 (*) X1Q31 + H4) (*) X1Q31 + H3) (*) X1Q31 + H2) (*) X1Q31
         + H1) (*) X1Q31 + H0
                       //polynomial calculation - acc in 3Q29 format

   if(Scale <= 0)
   {
      R = acc >> 13;   //Final result in 16Q16 format
   }
```

AntiLn_16 Natural Antilogarithm (cont'd)

```
    else{
            Rf = acc;    //Rf <- acc
            Expow = 1 << Scale;
                         // Get power of e^x1Q31
            tmp = Expow - 1;
                         //x^n needs (n-1) multiplications
            for (i=0;i<tmp;i++)
            {
               Rf = Rf (*) acc;
                      //Multiply calculated e^x1Q31 with itself power times
            }

            //Get the shift count to convert final result in 16Q16 format
            Expow = Expow << 1;
            ShCnt = Expow - 15;

            R = Rf << ShCnt;
                         //Final result in 16Q16 format
         }

    return R;            //Returns the calculated natural antilogarithm
 }
```

Techniques

- Use of MAC instructions
- Instruction ordering for zero overhead Load/Store

Assumptions

- Input 4Q12 format, output is the antilog of the input in 16Q16 format and coefficients are in 3Q29 format

Memory Note None

AntiLn_16 Natural Antilogarithm (cont'd)

Implementation

The antilog of the input value x can be calculated by using the following approximation series.

$$\mathrm{AntiLn}(x) = 1.0000 + 1.0001x + 0.4990x^2 + 0.1705x^3 + 0.0348x^4 + 0.0139x^5 \quad [4.153]$$

The coefficients of polynomial are stored in 3Q29 format. The 16 bit input is in 4Q12 format which can take values in the range $[-2^3, 2^3)$. The output returned is in 16Q16 format. The input is scaled in the range -1 to +1. If the given number is greater than 1, it is scaled down and if it is less than -1, it is scaled up by powers of 2. This scaled input is used in polynomial calculation.

The CLS instruction of TriCore gives the shiftcount to scale up or scale down the input. Only when shiftcount is less than 19, input is scaled up or scaled down. Otherwise input is in the range -1 to +1. The scale factor is obtained as (19-shiftcount). This scale factor will always be positive for the inputs greater than 1 and less than -1. The output of polynomial calculation is multiplied with itself scale factor times to get the actual output.

The implementation of the polynomial is optimal with zero overhead Load/Store.

Example

Trilib\Example\Tasking\Mathematical\expAntiLn_16.c, expAntiLn_16.cpp
Trilib\Example\GreenHills\Mathematical\expAntiLn_16.cpp, expAntiLn_16.c
Trilib\Example\GNU\Mathematical\expAntiLn_16.c

Cycle Count

If X in the range -1 to 1	:	14+2
else	:	$16 + (\mathrm{scale} \times 2) + 5 + 2$

Code Size

104 bytes

24 bytes (Data)

Expn_16 Exponential

Signature short Expn_16(DataS X);

Inputs X : Real Input value in the range [-1, 1)

Output None

Return R : Output exponent value of the function

Description This function calculates the exponent of the given input. It takes 16 bit input in the range [-1, 1) and returns the exponential value in 16 bits.

Pseudo code

```
{
   frac32 acc;         //result of polynomial calculation in 3Q29 format
   frac16 R;           //16 bit exponential result in 3Q13 format
   acc = ((((H[5] (*) X + H[4]) (*) X + H[3]) (*) X
         + H[2]) (*) X + H[1]) (*) X + H0;
                    //polynomial calculation - acc is result in 3Q29 format
   R = (frac16)acc;    //16 bit exponential result in 3Q13 format
}
```

Techniques
- Use of packed data Load/Store
- Use of MAC instructions
- Instruction ordering for zero overhead Load/Store

Assumptions
- Input 1Q15 format, output is the exponential of the input in 3Q13 format and coefficients are in 3Q29 format

Memory Note None

Implementation Exp(x) is approximated using the polynomial expansion given below.

$$\exp(x) = 1.0000 + 1.0001x + 0.4990x^2 + 0.1705x^3 + 0.0348x^4 + 0.0139x^5 \quad [4.154]$$

The input to the function is 16 bits in 1Q15 format. Hence input range is [-1, 1). Input outside this range should be scaled to this range before calling the function. Coefficients are stored in 3Q29 format. Output of the function is in 3Q13 format.

The polynomial is implemented in an optimal way so as to have zero overhead Load/Store.

Expn_16 Exponential (cont'd)

Example

Trilib\Example\Tasking\Mathematical\expExpn_16.c, expExpn_16.cpp
Trilib\Example\GreenHills\Mathematical\expExpn_16.cpp, expExpn_16.c
Trilib\Example\GNU\Mathematical\expExpn_16.c

Cycle Count 10+2

Code Size 42 bytes

24 bytes (Data)

XpowY_32 X Power Y

Signature	int XpowY_32(int X, DataS Y);		
Inputs	X	:	Real input value in the range $[2^{-11}, 2^{11})$
	Y	:	power in the range [-1,1)
Output	None		
Return	R	:	Output value of the function in the range $[2^{-11}, 2^{11})$

Description

X power Y is calculated. The input is 32-bit in 12Q20 format but it should lie within the range $[2^{-11}, 2^{11})$. The exponent Y is 16-bit in 1Q15 format and is in the range [-1,1). The output is 32-bit in 12Q20 format and lies in the range $[2^{-11}, 2^{11})$

XpowY_32 X Power Y (cont'd)

Pseudo code

```
{
   int Shcnt           //Shift count
   int Scale;          //Scaling factor
   frac32 acc;         //Result of Polynomial calculation
   frac32 Xu1Q31;      //Input scaled to unsigned 1Q31 format
   frac32 Xsub1;       //X-1
   frac32 Rf;          //Output of polynomial calculation
   frac32 LnX;         //Result of ln in 4Q28 format
   frac32 LnXPowY;     //Y*lnX in 4Q28 format
   int Expow;          //Power of calculated polynomial
   frac32 R;           //Result in 12Q20 format


   Shcnt = count_lead_sign(X);
                       // number of leading sign values
   Scale = 10 - Shcnt;//Get the scale factor
   Shcnt = Shcnt + 1; //add 1 to shift count to bring input to
                       //1 to 2(unsigned 1Q15)from 0.5 to 1
   Xu1Q31 = X << Shcnt;
                       //unsigned 1Q15 <- 16Q16
   Xsub1 = Xu1Q31 - 1;//X = X - 1
   if(Xsub1 == 0)
   go to XpowY_2

   acc = ((((H4 * Xsub1 + H3) * Xsub1 + H2) * Xsub1 + H1) * Xsub1 + H0) *
         Xsub1
                       //polynomial calculation - acc in 1Q31 format

   acc = acc << 3;     //4Q28 <- 1Q31

XpowY_2:
   Scale = Scale << 26;
                       //6Q26 <- 32Q0
   Add = Scale (*) ln2;
                       //Get the adding factor by scaling Ln2
   Add = Add << 2;     //4Q28 <- 6Q26
   LnX = acc + Add;    //Add the factor to get the result in 4Q28
                       //format
   LnXpowY = LnX (*) Y;

   Shcnt = count_lead_sign(LnXpowY);
                       //number of leading sign values
   X1Q31 = LnXpowY << Shcnt;//1Q31 <- 4Q28
```

XpowY_32 X Power Y (cont'd)

```
    Scale = 19 - Shcnt;//Get the scale factor

   acc = ((((H5 (*) X1Q31 + H4) (*) X1Q31 + H3) (*) X1Q31 + H2) (*) X1Q31
             + H1) (*) X1Q31 + H0
                      //polynomial calculation - acc in 3Q29 format
    if(Scale <= 0)
    {
       R = acc >> 9;   //Final result in 12Q20 format

    }
    else
    {
      Rf = acc;         //Rf <- acc
      Expow = 1 << Scale;
                        // Get power of e^x1Q31
      tmp = Expow - 1;
                        //x^n needs (n-1) multiplications
      for (i=0;i<tmp;i++)
      {
         Rf = Rf (*) acc;
                    //Multiply calculated e^x1Q31 with itself power times
      }
      //Get the shift count to convert final result in 12Q20 format
      Expow = Expow << 1;
      ShCnt = Expow - 11;

      R = Rf << ShCnt;
                        //Final result in 12Q20 format
    }

    return R;           //Returns the calculated X power Y
 }
```

Techniques

- Use of MAC instructions
- Instruction ordering for zero overhead Load/Store

Assumptions

- Inputs are in 12Q20 format and should in the range [2^{-11}, 2^{11}) which is a subset of actual range. Exponent is in 1Q15 format and is in the range [-1,1).The returned output is in 12Q20 format and lies in the range [2^{-11}, 2^{11})
- Input is always positive

Memory Note None

XpowY_32 **X Power Y** (cont'd)

Implementation

X power Y can be calculated as $e^{(Y.\ln X)}$. The natural logarithm of the input value x can be calculated using the following approximation series.

$$\begin{aligned}\ln(x) = {} & 0.9991150(x-1) - 0.4899597(x-1)^2 \\ & + 0.2856751(x-1)^3 - 0.1330566(x-1)^4 \\ & + 0.03137207(x-1)^5\end{aligned} \quad [4.155]$$

where, $1 \geq x \geq 2$ which means $0 \geq (x-1) \geq 1$

The coefficients of polynomial are stored in 1Q31 format. The constant ln2 is also stored in 1Q31 format.

The 32 bit input is in 12Q20 format which can take values in the range [-2^{11}, 2^{11}). As input to logarithm should always be positive it will be subset of actual input range, i.e., in the range [2^{-20}, 2^{11}). For proper operation of lnX and antiln(Y.lnX) input should lie in the range [2^{-11}, 2^{11}). The 32 bit output format is 12Q20 which lies in the range [2^{-11}, 2^{11}). Implementation of lnX is same as natural logarithm of X except that scale factor is obtained as (10 - shiftcount) [Refer Natural Logarithm].
The output (lnX) is multiplied with the exponent Y. The resulting product is in 4Q28 format. The antilog of this product gives the desired output.
The antilog of the input value X can be calculated by using the following approximation series.

$$\begin{aligned}\mathrm{AntiLn}(x) = {} & 1.0000 + 1.0001x + 0.4990x^2 + 0.1705x^3 \\ & + 0.0348x^4 + 0.0139x^5\end{aligned} \quad [4.156]$$

The coefficients of polynomial are stored in 3Q29 format. The 32 bit input is in 4Q28 format. The output is in 12Q20 format. Implementation is same as natural antilog of function. [Refer Natural Antilog].

XpowY_32 **X Power Y** (cont'd)

Example

Trilib\Example\Tasking\Mathematical\expXpowY_32.c, expXpowY_32.cpp
Trilib\Example\GreenHills\Mathematical \expXpowY_32.cpp, expXpowY_32.c
Trilib\Example\GNU\Mathematical\expXpowY_32.c

Cycle Count

When X is a power of 2 and X^Y in the range $[e^{-1}, e)$

38+2

When X is a power of 2 and X^Y not in the range $[e^{-1}, e)$

$42 + 2 \times \mathrm{scale} + 1 + 2$ for scale = 1

scale factor for antiln(YlnX)

$42 + 2 \times \mathrm{scale} + 2 + 2$ otherwise

scale factor for antiln(YlnX)

When X is not a power of 2 and X^Y in the range $[e^{-1}, e)$

47+2

When X is not a power of 2 and X^Y not in the range $[e^{-1}, e)$

$51 + 2 \times \mathrm{scale} + 1 + 2$ for scale = 1

scale factor for antiln(YlnX)

$51 + 2 \times \mathrm{scale} + 2 + 2$ otherwise

scale factor for antiln(YlnX)

Code Size

190 bytes

48 bytes (Data)

4.12.2 Random Number Generation

Randomness is typically associated with unpredictability. Mathematics provides a precise definition of randomness that is then applied here to evaluate random number vector. Random numbers within the context of the function Rand_16 refers to "a sequence of independent numbers with a specified distribution and a specified probability of falling in any given range of values".

Here Random Number Generator is implemented using **Linear Congruential Method (L.C.M).** RNG using linear congruential method is also called **pseudo RNG** because they require a seed and produce a deterministic sequence of numbers. Algorithm used here is called **L.C.M** introduced by D. Lehmen in 1951.

Linear Congruential Method

This method produces a sequence of integers X_1, X_2, X_3,... between zero and M-1 according to the following recursive relationship

$$X_{i+1} = (aX_i + c)\mathrm{mod}M \quad i = 0,1,2,... \qquad [4.157]$$

where,

X_i	:	the initial value, called the seed
a	:	constant multiplier (RNDMULT)
c	:	increment (RNDINC)
M	:	modulus

Apart from LCM many Random Number Generators exist, but this method is arguably the fastest for a 16-bit value. If a 32-bit value is needed, the code can be modified by performing a 32-bit multiply and using 32-bit constants (RNDMULT, RNDINC). This method, however, does have one major disadvantage. It is very sensitive to the values of RNDMULT and RNDINC.

Much research has been done to identify the optimal choices of these constants to avoid degeneration. The constants used in the subroutine below were chosen based on this research.

M: The modulus value. This routine returns a random number from 0 to 65536 (64K) and is not internally bounded. If the user needs a min/max limit, this must be coded externally to this routine.

RNDSEED: An arbitrary constant, can be chosen to be any value representable by the (0-64K) word. If zero is chosen, RNDINC should be some larger value than one. Otherwise, the first two values will be zero and one. This is ok if the generator is given three cycles to warm up. To change the set of random numbers generated by this routine, change the RNDSEED value. RNDSEED=21845 is used in this routine because it is 65536/3.

RNDMULT: Should be chosen such that the last three digits are even-2-1 (such as xx821, x421, etc). RNDMULT=31821 is used in this routine.

RNDINC: In general, this constant can be any prime number related to M (or 64K in this case).Two values were actually tested, 1 and 13849. Research shows that RNDINC (the increment value) should be chosen by the following formula

$$\mathrm{RNDINC} = ((1/2 - (1/6 \times \mathrm{SQRT}(3))) \times \mathrm{M}) \qquad [4.158]$$

Using M=65536, RNDINC=13849. (as indicated above.)

RNDINC=13849 is used in this routine.

Because PRNG’s employ a mathematical algorithm for number generation, all PRNG’s possess the following properties:

- A seed value is required to initialize the equation
- The sequence will cycle after a particular period

4.12.2.1 Description

The following Random Number Generation functions are described.

- Random Number Initialization
- Random Number Generator

RandInit_16 Random Number Initialization

Signature void RandInit_16(void);

Inputs None

Output None

Return None

Description RandInit_16 function initializes the value of seed stored in global memory location for 16-bit random number generation routine.

Pseudo code None

Techniques None

Assumptions None

Memory Note

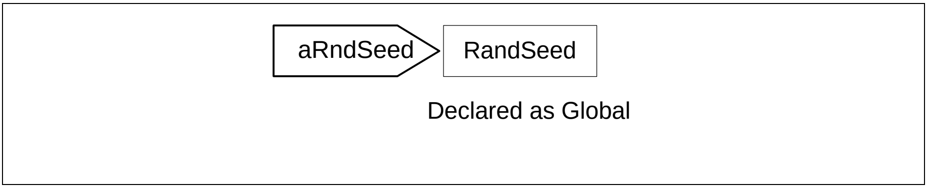


Figure 4-86 RandInit_16

Implementation RndSeed, the seed for Random Vector Generator is initialized from global memory. Assembler directive `.space` is used to reserve a block of memory. The seed value is stored in this memory. This memory is declared as global so that seed value can be accessed while generating random vector.

Example *Trilib\Example\Tasking\Mathematical\expRandInit_16.c, expRandInit_16.cpp*

Trilib\Example\GreenHills\Mathematical \expRandInit_16.cpp, expRandInit_16.c

Trilib\Example\GNU\Mathematical\expRandInit_16.c

Cycle Count 2+2

Code Size 14 bytes

Rand_16 Random Number Generator

Signature

```
void Rand_16(int   nX,
             int   *R
            );
```

Inputs

nX : Size of output vector

R : Pointer to output vector

Output

R[nX] : Output vector

Return

None

Description

Rand_16 function computes vector of 16 bit random numbers. Seed value is initialized by RandInit_16 function. This function uses 16 bit predefined RandMul, RandInc values to calculate output vector of given size. After calculation of random vector the seed in memory is updated. So if this function is called again, will use this new seed value and vector generated will be different from the original one.

Pseudo code

```
{
   int i;
   for (i=0;i<max;i++)
   {
      rndvec[i] = (rndseed*rndmul+rndinc)%modulus;
                     //Rndvec=16-bit random number
                     //RndSeed=Seed value=21845,Userdefined constant
                     //RndMul=Multiplier=31821,Userdefined constant
                     //RndInc=Increment=13849, Userdefined constant
                     //Modulus=65536,Userdefined constant

   }
   rndseed = rndvec[i];
}
```

Techniques

- Instruction ordering for zero overhead Load/Store

Assumptions

- Uses seed value from the memory location which can be initialized by Rand initialization routine

Rand_16 Random Number Generator (cont'd)

Memory Note

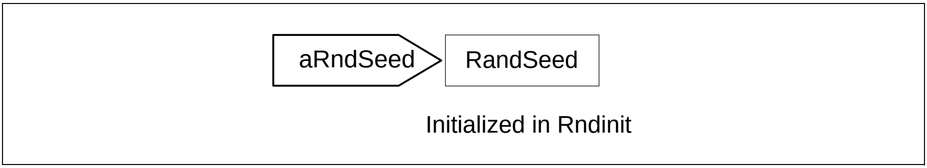


Figure 4-87 Rand_16

Implementation

Random vector generation uses

$$\text{Randvec} = (\text{RndSeed} \times \text{RndMul} + \text{RndInc})\text{Modulus} \quad [4.159]$$

RndSeed is initialized by routine RandInit_16, rest other constant values are stored immediate to data registers. viz.,RndMul, RndInc, Modulus.
Rndseed stored in global memory is accessed as external variable and Random Vector is calculated as per above equation.

Example

Trilib\Example\Tasking\Mathematical\expRand_16.c, expRand_16.cpp

Trilib\Example\GreenHills\Mathematical\expRand_16.cpp, expRand_16.c
Trilib\Example\GNU\Mathematical\expRand_16.c

Cycle Count

With DSP Extensions

$$4 + nX \times (8) + 1 + 2$$

Without DSP Extensions

$$4 + nX \times (8) + 1 + 2$$

Code Size 38 bytes

4.13 Matrix Operations

A matrix is a rectangular array of numbers (or functions) enclosed in brackets. These numbers (or functions) are called entries or elements of the matrix.The number of entries in the matrix is product of number of rows and columns. An $m \times n$ matrix means matrix with m rows and n columns. In the double-subscript notation for the entries, the first subscript always denotes the row and the second the column.

4.13.1 Descriptions

The following Matrix Operations are described.

- Addition
- Subtraction
- Multiplication
- Transpose

MatAdd_16 Addition

Signature

```
void MatAdd_16 (short   X[ ] [MAXCOL],
                short   Y[ ] [MAXCOL],
                short   R[ ] [MAXCOL],
                int     nRow,
                int     nCol
                );
```

Inputs

X	:	Pointer to first matrix
Y	:	Pointer to second matrix
R	:	Pointer to output matrix
nRow	:	Number of rows
nCol	:	Number of columns

Output

R	:	Pointer to output matrix which is the sum of the matrices X and Y

Return None

Description This function performs the addition of two matrices. It takes pointers to the two matrices, pointer to the output matrix, size of row and size of column as input. The entries in the matrices are 16 bit values. The output matrix is stored starting from the address which is sent as input.

Pseudo code

```
{
   short *R;           //Ptr to a two dimensional output array of nRow
                        //rows and nCol columns
   int Tmp;

   Tmp = nRow * nCol; //number of elements
   loopCnt = Tmp/4    //4 additions performed per loop

   for(i=0;i<loopCnt;i+=4)
   {
      *(R+i) = *(X+i) + *(Y+i);
      *(R+i+1) = *(X+i+1) + *(Y+i+1);
      *(R+i+2) = *(X+i+2) + *(Y+i+2);
      *(R+i+3) = *(X+i+3) + *(Y+i+3);
   }
}
```

Techniques

- Loop Unrolling, 4 additions/loop
- Use of packed data Load/Store
- Use of packed addition with saturation
- Instruction ordering provided for zero overhead Load/Store

MatAdd_16 **Addition** (cont'd)

Assumptions

- nRow = 2*m, m = 1,2,3...
- nCol = 2*n, n = 1,2,3...

Memory Note

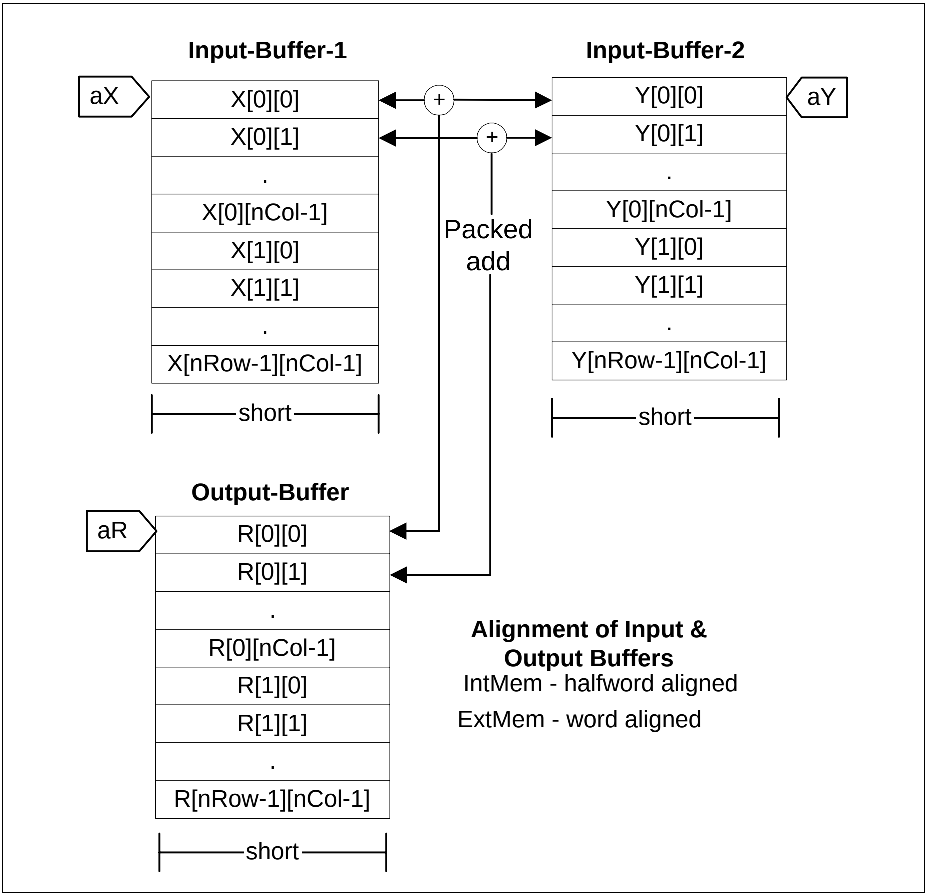


Figure 4-88 MatAdd_16

MatAdd_16 **Addition** (cont'd)

Implementation

The inputs to the function are three pointers (one each to each of the input matrices to be added and one to the output matrix) and the number of rows and number of columns. Both number of rows and number of columns are multiple of two. Hence the number of elements could be 4,8,12,.... This fact is made use of in implementing the matrix addition in an optimal manner. Addition is performed in a loop. Using TriCore's load doubleword instruction, four elements of each matrix are loaded in two data register pairs. Using packed arithmetic on halfwords, two of the 16 bit entries can be added in one cycle. Hence, by using two packed add instructions per loop, the loop count is brought down by a factor of four. The loop is executed (nRow * nCol)/4 times.

Example

Trilib\Example\Tasking\Matrix\expMatAdd_16.c, expMatAdd_16.cpp
Trilib\Example\GreenHills\Matrix\expMatAdd_16.cpp, expMatAdd_16.c
Trilib\Example\GNU\Matrix\expMatAdd_16.c

Cycle Count

Pre-loop : 5

Loop : $\left\lceil \frac{3 \times \mathrm{nRow} \times \mathrm{nCol}}{4} \right\rceil + 2$

Post-loop : 0+2

Code Size

52 bytes

MatSub_16 Subtract

Signature

```
void MatSub_16(short   X[ ] [MAXCOL],
               short   Y[ ] [MAXCOL],
               short   R[ ] [MAXCOL],
               int     nRow,
               int     nCol
               );
```

Inputs

X	:	Pointer to first matrix
Y	:	Pointer to second matrix
R	:	Pointer to output matrix
nRow	:	Number of rows
nCol	:	Number of columns

Output

R	:	Pointer to output matrix which is the subtraction of the matrices X and Y

Return None

Description This function performs the subtraction of two matrices. It takes pointers to the two matrices, pointer to the output matrix, size of row and size of column as input. The entries in the matrices are 16 bit values. The output matrix is stored starting from the address which is sent as input.

Pseudo code

```
{
   short *R;           //Ptr to a two dimensional output array of nRow
                        //rows and nCol columns
   int Tmp;

   Tmp = nRow * nCol; //number of elements
   loopCnt = Tmp/4    //4 subtractions performed per loop

   for(i=0;i<loopCnt;i+=4)
   {
      *(R+i) = *(X+i) - *(Y+i);
      *(R+i+1) = *(X+i+1) - *(Y+i+1);
      *(R+i+2) = *(X+i+2) - *(Y+i+2);
      *(R+i+3) = *(X+i+3) - *(Y+i+3);
   }
}
```

MatSub_16 **Subtract** (cont'd)

Techniques

- Loop Unrolling, 4 subtractions/loop
- Use of packed data Load/Store
- Use of packed subtraction with saturation
- Instruction ordering provided for zero overhead Load/Store

Assumptions

- nRow = 2*m, m = 1,2,3...
- nCol = 2*n, n = 1,2,3...

MatSub_16 **Subtract** (cont'd)

Memory Note

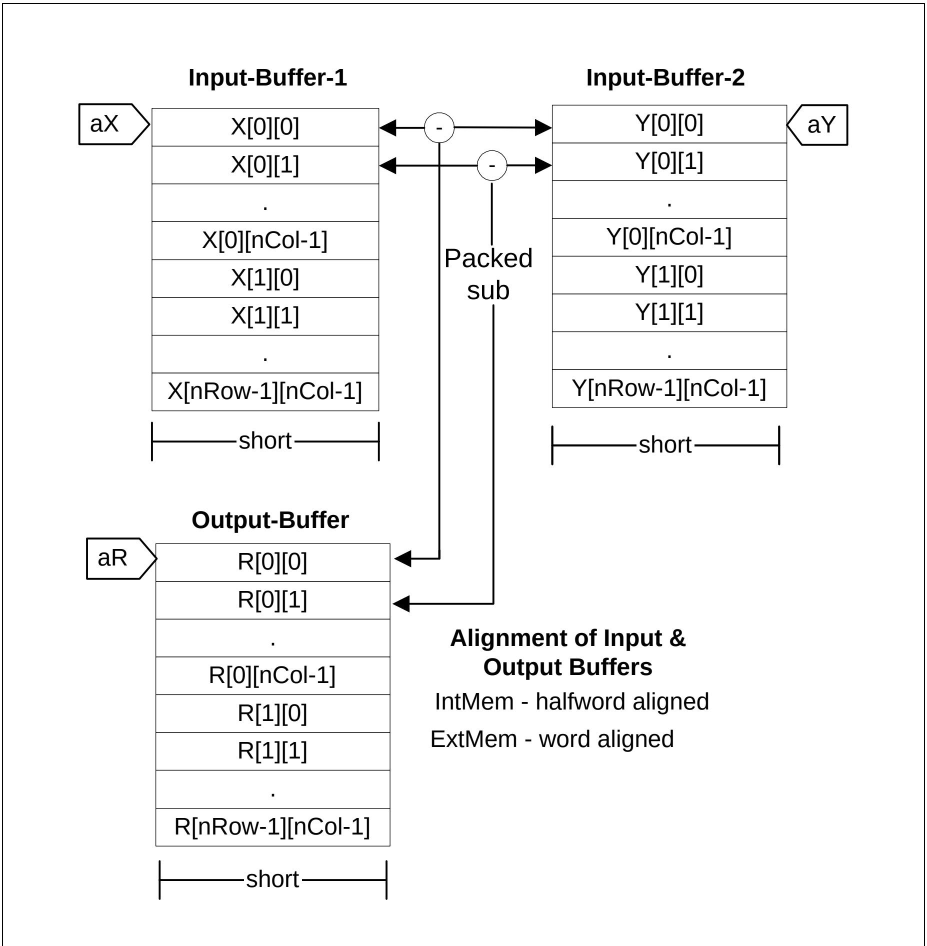


Figure 4-89 MatSub_16

MatSub_16 **Subtract** (cont'd)

Implementation

The inputs to the function are three pointers (one each to each of the input matrices to be subtracted and one to the output matrix) and the number of rows and number of columns. Both number of rows and number of columns are multiple of two. Hence the number of elements could be 4, 8, 12,.... This fact is made use of in implementing the matrix subtraction in an optimal manner. Subtraction is performed in a loop. Using TriCore's load doubleword instruction, four elements of each matrix are loaded in two data register pairs. Using packed arithmetic on halfwords, two of the 16 bit entries can be subtracted in one cycle. Hence by using two packed subtract instructions per loop, the loop count is brought down by a factor of four. The loop is executed (nRow * nCol)/4 times.

Example

Trilib\Example\Tasking\Matrix\expMatSub_16.c, expMatSub_16.cpp
Trilib\Example\GreenHills\Matrix\expMatSub_16.cpp, expMatSub_16.c
Trilib\Example\GNU\Matrix\expMatSub_16.c

Cycle Count

Pre-loop	:	5
Loop	:	$\left\lceil \frac{3 \times \mathrm{nRow} \times \mathrm{nCol}}{4} \right\rceil + 2$
Post-loop	:	0+2

Code Size

52 bytes

MatMult_16 Multiplication

Signature

```
DataS MatMult_16(DataS   X[] [MaxCol],
                 DataS   Y[] [MaxCol],
                 DataS   R[] [MaxCol],
                 int     nRowX,
                 int     nColX,
                 int     nColY
                 );
```

Inputs

X	:	Pointer to first matrix
Y	:	Pointer to second matrix
R	:	Pointer to output matrix
nRowX	:	Number of rows of first matrix
nColX	:	Number of columns of first matrix
nColY	:	Number of columns of second matrix

Output

R	:	Pointer to output matrix which is the multiplication of the matrices X and Y

Return None

Description The multiplication of two matrices X and Y is done. Both the input matrices and output matrix are 16-bit. All the matrices are halfword aligned. All the element of the matrix are stored row-by-row in the buffer.

MatMult_16 **Multiplication** (cont'd)

Pseudo code

```
{
   int nRowX;          //Number of rows of first matrix
   int nColX;          //Number of columns of first matrix
   int nColY;          //Number of columns of second matrix
   frac16 R;           //Result of matrix multiplication
   frac32 acc;

   for(i=0; i<nRowX; i++)
                       //Outer loop is executed nRow times
   {
      for(j=0; j<nColY; j=j+2)
                       //Middle loop is executed nColY/2 times
      {
         acc = 0;
         for(k=0; k<nColX/2; k++)
                       //Inner loop is executed nColX/2 times
         {
           acc += (sat rnd) Y[i][j+1] (*) X[i][j] || Y[i][j] (*) X[i][j]
            acc += (sat rnd) Y[i+1][j+1] (*) X[i][j+1] || Y[i+1][j] (*)
            X[i][j+1]
         }
         R[i][j] = (frac16)accLo;
         R[i][j+1] = (frac16)accHi;
      }
   }
}
```

Techniques

- Use of packed data Load/Store
- Use of packed MAC instruction
- Instruction ordering for zero overhead Load/Store

Assumptions

- nRowX = 2*l, l = 1,2,3...
- nColX = nRowY = 2*m, m = 1,2,3...
- nColY = 2*n, n = 1,2,3...

MatMult_16 Multiplication (cont'd)

Memory Note

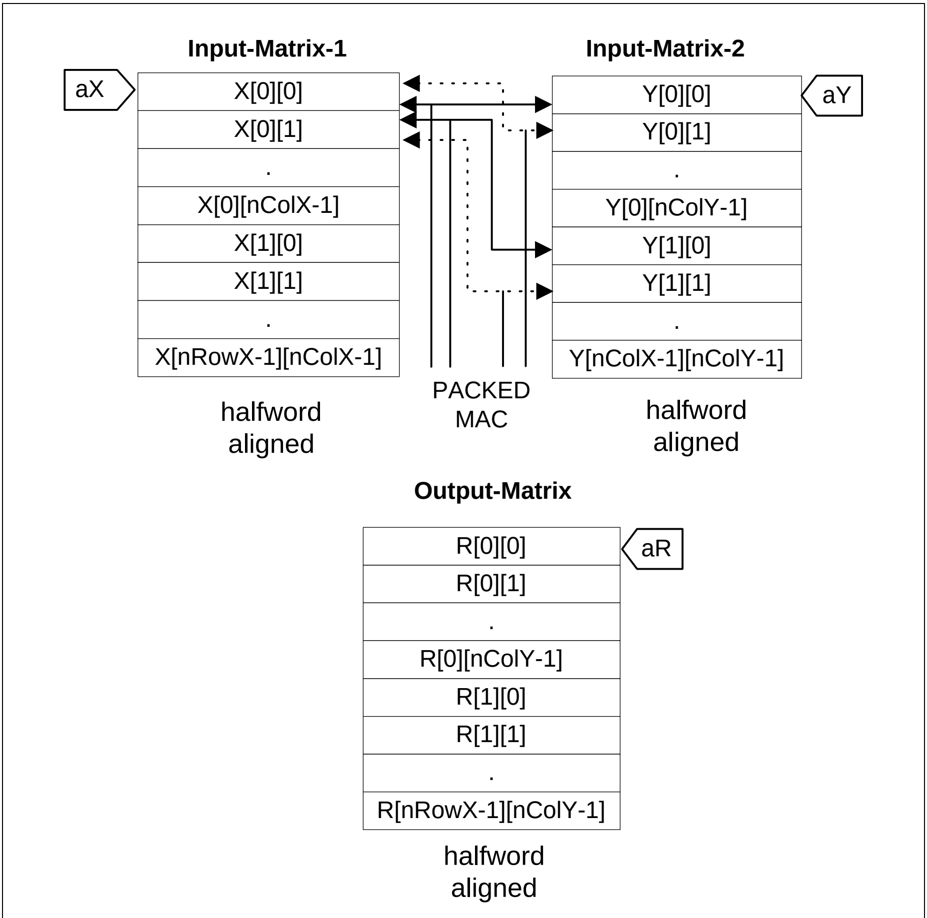


Figure 4-90 MatMult_16

MatMult_16 **Multiplication** (cont'd)

Implementation

The pointer to both the input matrices (X and Y), pointer to output matrix (R), number of rows of X (nRowX), number of columns of X (nColX) and number of columns of Y (nColY) are sent as arguments.

The implementation uses three loops:
The outer loop is executed nRowX times. The middle loop is executed nColY/2 times and the inner loop is executed nColX/2 times.

In the outer loop, the pointer is initialized to first element of X (X[0][0]). For every next iteration of loop it is updated to point to next row (X[i+1][0]). Thus this loop is executed nRowX times.

In the middle loop, the pointer to X is always initialized to point to the row of X selected by outer loop. The pointer to Y is initialized to first element of Y (Y[0][0]). For every next iteration of loop it is updated to point to next to next column of Y (Y[i][j+2]). Since the two columns are considered in one pass of inner loop, this loop is executed nColY/2 times.

In the inner loop two values of X and two values of Y are loaded using load word instruction. Two packed MAC instructions are used in this loop.

First packed MAC uses X[i][j] and following operation is performed.

$$\mathrm{acc} = \mathrm{acc} + Y[i][j+1] \cdot X[i][j] \,\|\, Y[i][j] \cdot X[i][j] \qquad [4.160]$$

Second packed MAC uses X[i][j+1] and following operation is performed.

$$\mathrm{acc} = \mathrm{acc} + Y[i+1][j+1] \cdot X[i][j+1] \,\|\, Y[i+i][j] \\ X[i][j+1] \qquad [4.161]$$

As two values from the selected row of X are used in each pass, this loop is executed nColX/2 times.

MatMult_16 **Multiplication** (cont'd)

Example

Trilib\Example\Tasking\Matrix\expMatMult_16.c, expMatMult_16.cpp
Trilib\Example\GreenHills\Matrix\expMatMult_16.cpp, expMatMult_16.c
Trilib\Example\GNU\Matrix\expMatMult_16.c

Cycle Count

$$8 + \mathrm{nRowX}\left\{\frac{\mathrm{nColY}}{2}\left[6 + \frac{\mathrm{nColX}}{2}(6) + 2(\mathrm{or}1)\right] + 1 + 4\right\} + 1$$

Code Size

100 bytes

MatTrans_16 Transpose

Signature

```
void MatTrans_16(short   X[ ] [MAXCOL],
                 short   R[ ] [MAXROW],
                 int     nRow,
                 int     nCol
                 );
```

Inputs

X	:	Pointer to input matrix
R	:	Pointer to output matrix
nRow	:	Number of rows
nCol	:	Number of columns

Output

R	:	Pointer to output matrix which is the transpose of the matrix X

Return None

Description This function performs transpose of the given matrix. It takes pointers to input and output matrix, size of row and size of column as input. The entries in the matrix are 16 bit values. The output matrix is stored from the address which is sent as input.

Pseudo code

```
{
   int i,j;
   for(i=0;i<nCol;i++)//Column loop
   {  K = 0;
      for(j=0;j<nRow/2;j++)
                      //Row loop
      {
         R[i][k] = X[k][i];
                      //Two elements of input matrix are read
                      //and stored
         R[i][k+1] = X[k+1][i];
         k = K+2;
      }
   }
}
```

Techniques

- Use of packed data Load/Store
- Instruction ordering provided for zero overhead Load/Store

Assumptions

- nRow = 2*m, m = 1,2,3...
- nCol = 2*n, n = 1,2,3...

MatTrans_16 **Transpose** (cont'd)

Memory Note

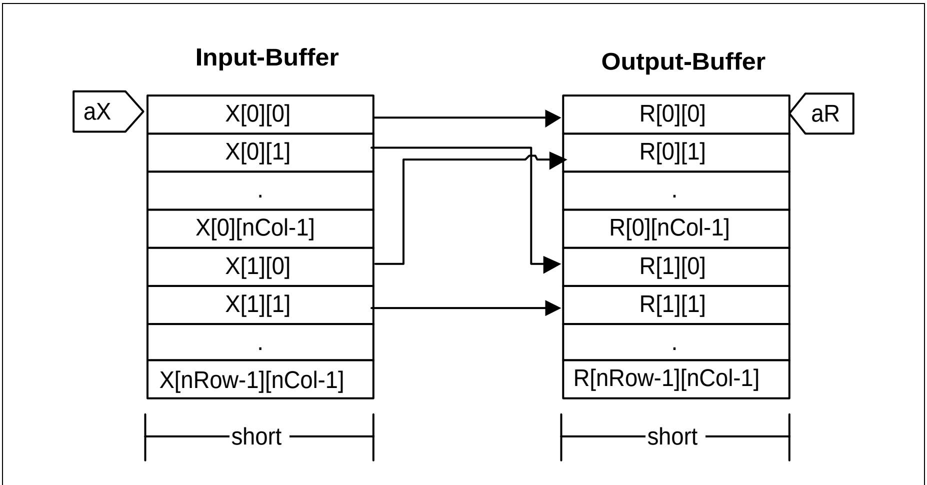


Figure 4-91 MatTrans_16

Implementation The inputs to the function are two pointers to the matrices (input matrix and output matrix respectively), number of rows and number of columns. Both number of rows and number of columns are multiple of 2. The outer loop is executed number of column times. The inner loop is executed nRow/2 times. In the row loop two input elements from first column are read and packed. Using TriCore's store word instruction, it is stored in first row of output matrix. The inner loop is executed for the first column. Then pointer is made to point to second element in the first row. Then inner loop is executed for second column. Thus outer loop is executed number of column times and transpose is obtained.

Example *Trilib\Example\Tasking\Matrix\expMatTrans_16.c, expMatTrans_16.cpp*
Trilib\Example\GreenHills\Matrix\expMatTrans_16.cpp, expMatTrans_16.c
Trilib\Example\GNU\Matrix\expMatTrans_16.c

MatTrans_16 **Transpose** (cont'd)

Cycle Count For all X[nRow][nCol] : $3 + \left[\left(\frac{nRow}{2}\right) \times 5 + 2 + 5\right] \times nCol$

$+2+2$

Code Size 52 bytes

4.14 Statistical Functions

4.14.1 Descriptions

The following Statistical functions are described.

- Autocorrelation
- Convolution
- Mean Value

Autocorrelation

Correlation determines the degree of similarity between two signals. If two signals are identical their correlation coefficient is 1, and if they are completely different it is 0. If the phase shift between them is 180 and otherwise they are identical, then correlation coefficient is -1.

There are two types of correlation Cross Correlation and Autocorrelation.

When two independent signals are compared, the procedure is cross correlation. When the same signal is compared to phase shifted copies of itself, the procedure is autocorrelation. Autocorrelation is used to extract the fundamental frequency of a signal. The distance between correlation peaks is the fundamental period of the signal. Discrete correlation is simply a vector dot product.

$$R(j) = \sum_{i=0}^{N} x(i) \times y(i+j) \qquad [4.162]$$

where,

N = nX - j -1 (j = 0, 1,...,nR-1),

nX = Size of input vector

nR = Desired number of outputs. It can take values from 1 to nX

Autocorrelation is given by

$$R(j) = \sum_{i=0}^{N} x(i) \times x(i+j) \quad (j = 0, 1,...,nR-1) \qquad [4.163]$$

i is the index of the array, j is the lag value, as it indicates the shift/lag considered for the R(j) autocorrelation. N is the correlation length and it determines how much data is used for each correlation result. When R(j) is calculated for a number of j values, it is referred to as autocorrelation function.

Convolution

Discrete convolution is a process, whose input is two sequences, that provide a single output sequence.

Convolution of two time domain sequences results in a time domain sequence. Same thing applies to frequency domain.

Both the input sequences should be in the same domain but the length of the two input sequences need not be the same.

Convolution of two sequences X(k) and H(k) of length nX and nH respectively can be given mathematically as

$$R(n) = \sum_{k=0}^{nX+nH-2} H(k) \cdot X(n-k) \tag{4.164}$$

The resulting output sequence R(n) is of length nX+nH-1.

The convolution in time domain is multiplication in frequency domain and vice versa.

ACorr_16 Autocorrelation

Signature

```
void ACorr_16( DataS  *X,
               DataL  *R,
               int     nX,
               int     nR
             );
```

Inputs

X	:	Pointer to Input-Vector
R	:	Pointer to Output-Vector containing the first nR elements of the positive side of the autocorrelation function of the vector X
nX	:	Size of vector X
nR	:	Size of vector R

Output

R	:	Output-Vector

Return None

Description

The function performs the positive side of the autocorrelation function of real vector X. The arguments to the function are pointer to the input vector, pointer to output buffer to store autocorrelation result, size of input buffer (only even) and number of auto correlated outputs desired. The input values are in 16 bit fractional format and output values are in 32 bit fractional format. The implementation is optimal and works if size of output buffer is even/odd.

ACorr_16 Autocorrelation (cont'd)

Pseudo code

```
{
    frac16 *X1;          //Ptr to input vector
    frac16 *X2;          //Ptr to input vector + LagCount
    frac64 acc;          //Autocorrelation result
    int dCnt;            //Correlation loop count
    //Macro
    macro ACorr;

    {
       int aCorlen;      //Correlation loop count
      aCorlen = dCnt; //Correlation loop count for current autocorrelation
                         //output
       for(i=0; i<aCorlen; i++)
       {
          acc = acc + *(X1++) * *(X2++) + *(X1++) * *(X2++);
                         //acc = acc + X(0) * X(0+aLagCnt) + X(1) *
                         //X(1+aLagCnt)(even correlation length) (or)
                     //acc = acc + X(1) * X(1+aLagCnt) + X(2) * X(2+aLagCnt)
                         //(odd correlation length)
       }
    }

    ACorr_16:
    {
       int lflag = 0;
       int aLagCnt = 0;//First autocorrelation output is with zero lag
       int dCnt = nX/2;
       X1 = X;           //Initialize first Ptr to start of input vector
       if (nR%2 != 0)
       {
          nR++;
          lflag = 1;     //lflag = 1 if nR is odd
       }
       //If desired no. of output is 1 or 2 skip ACorr_OutDataL
       if (nR == 2)
       go to ACorr_R_1or2;

       //ACorr_OutDataL
       for (i=0; i<nR/2-1; i++)
       {
          acc = 0;       //Clear accumulator
         X2 = X + aLagCnt;
                       //Second Ptr initialized to first Ptr plus an offset
```

ACorr_16 Autocorrelation (cont'd)

```
                    //of aLagCnt
        ACorr;      //Autocorrelation computation
        *R++ = (frac32_sat) acc;
                   //Autocorrelation result converted to 32 bits with
                    //saturation and stored to output buffer
        acc = 0;    //Clear accumulator
        aLagCnt = aLagCnt + 2;
                  //Lag count is incremented for the next correlation
        X1 = X;     //Initialize first Ptr to start of input vector
        X2 = X2 + alagCnt;
                 //Second Ptr initialized to first Ptr plus an offset
                    //of aLagCnt

        //Autocorrelation computation
        dCnt--;
        acc = acc + *(X1++) * *(X2++);
                    //acc = acc + X(0) * X(0+aLagCnt)
        ACorr;
        X1 = X;     //Initialize first Ptr to start of input vector
        aLagCnt = aLagCnt + 1;
                //Lag cnt incremented for next autocorrelation
                    //computation
    }

  //Last two results (if nR is even) or last one result (if nR is
  //odd) is calculated outside the loop
  ACorr_R_1or2:
     acc = 0;       //Clear accumulator
     X2 = X + aLagCnt;
     ACorr;
     *R++ = (frac32_sat)acc;
     if (lflag == 1) //Jump to ACorr_16_Ret if lflag = 1
     go to ACorr_Ret;
     else
        acc = 0;    //Clear accumulator
        X1 = X;     //Initialize first Ptr to start of input vector
        X2 = X2 + aLagCnt;
        acc = acc + *(X1++) * *(X2++);
        //If nR = nX, jump to ACorr_Rlast
     if (dCnt = 0)
        go to ACorr_Rlast;
     else
```

ACorr_16 Autocorrelation (cont'd)

```
        {
            dCnt--;
            ACorr;
        }
    ACorr_Rlast:
        (*R++)(frac32_sat)acc;
    ACorr_Ret:
    }
}
```

Techniques

- Loop unrolling is done so that implementation is efficient for both even and odd number of desired outputs. Last two outputs (for nR even) or last one output (for nR odd) is computed outside the loop
- A macro ACorr is used to calculate each autocorrelation output. The macro uses packed load and dual MAC to reduce the number of cycles for a given correlation length
- One pass through the loop calculates two outputs, i.e., there are two calls to the macro
- For odd correlation length one multiplication is performed before calling the macro
- Implementation is optimal for both even and odd values of nR
- Intermediate result stored in 64 bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Input is in 1Q15 format
- Output is in 1Q31 format

ACorr_16 **Autocorrelation** (cont'd)

Memory Note

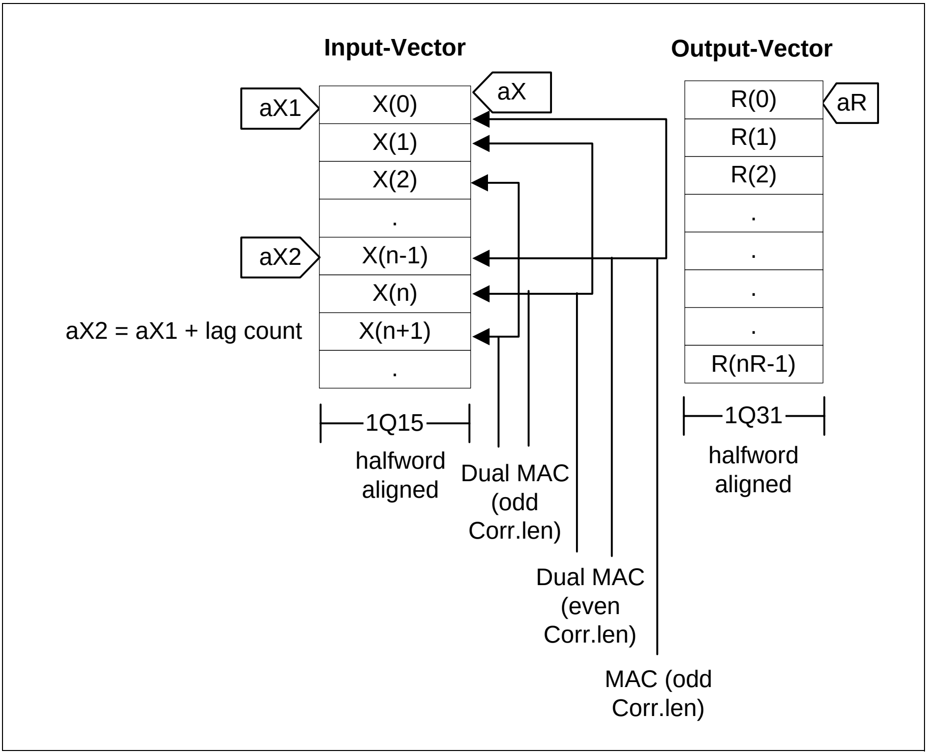


Figure 4-92 ACorr_16

ACorr_16 Autocorrelation (cont'd)

Implementation

Correlation is similar to FIR filtering without the time reversal of the second input variable. In autocorrelation, the signal is multiplied with phase shifted copies of itself. The implementation begins with zero lag, i.e., the value at each instant is squared and added to produce the first autocorrelation output.

The lag value is incremented by one for each next output. Hence, in autocorrelation computation the number of multiplication (correlation length) needed for each R(i) decreases as i increases from 1 to nR-1. Since the given assumption is that the number of input is always even, correlation length is even for all R(j) where j = 0, 2, 4,....,nR-2 and it is odd when j = 1, 3, 5,...,nR-1.

For each autocorrelation output computation, two pointers to input buffer aX1, aX2 are initialized such that aX1 points to beginning of input vector and the difference between them is equal to the lag value for that output, i.e., aX2 = aX1+lag count.

A macro ACorr is used to calculate each autocorrelation output. The macro uses packed load and dual MAC to reduce the number of cycles for a given correlation length. This brings down the loop count for each autocorrelation by a factor of 2. For all R(i), i = 0, 2, 4,...., the call to ACorr will directly give the autocorrelation result in a 64 bit register which is then converted with saturation to 1Q31 format and stored to output buffer. In case of R(i) with i = 1, 3, 5,..., the correlation length is odd. Hence, one MAC is performed before calling the ACorr macro. This makes the implementation optimal for all R(i). The loop in the ACorr_16 function runs (nR/2-1) times. During each pass through the loop two outputs are calculated and written to output buffer (there are two calls to ACorr). The implementation works for both odd and even values of nR, i.e., nR = 1, 2,...,nX.

ACorr_16 **Autocorrelation** (cont'd)

Example

Trilib\Example\Tasking\Statistical\expACorr_16.c, expACorr_16.cpp
Trilib\Example\GreenHills\Statistical\expACorr_16.cpp, expACorr_16.c
Trilib\Example\GNU\Statistical\expACorr_16.c

Cycle Count

For Macro ACorr

$$\mathrm{Mcall}(1) = 1 + \mathrm{nX} + 2$$

$$\mathrm{Mcall}(i) = 1 + 2 \times ((\mathrm{nX})/2 - (i - i\bmod 2)/2) + 2$$

i = 2, 3,...,nX-2

$$\mathrm{Mcall}(i) = 1 + 2 \times ((\mathrm{nX})/2 - (i - i\bmod 2)/1) + 2$$

i = 2, 3,...,nX-2

$$\mathrm{Mcall}(i) = 1 + 2 \times ((\mathrm{nX})/2 - (i - i\bmod 2)/1) + 2$$

i = nX-1

where Mcall(i) refers to the i^{th} call to the macro

For ACorr_16

a) When nR = any Even value less than nX and greater than 2

Pre-loop	:	9
Loop	:	$19 \times (\mathrm{nR}/2 - 1) + \mathrm{Mcall}(1) + \ldots + \mathrm{Mcall}(\mathrm{nR} - 2)$
Post-loop	:	$2 + 2 + \mathrm{Mcall}(\mathrm{nR} - 1) + 14 + \mathrm{Mcall}(\mathrm{nR}) + 6 + 2$
Example	:	When nX = 54, nR = 4
	:	Cycle Count = 274 cycles

b) When nR = any Odd value less than nX and greater than 1

Pre-loop	:	9
Loop	:	$19 \times ((\mathrm{nR} + 1)/2 - 1)) + \mathrm{Mcall}(1) + \ldots + \mathrm{Mcall}(\mathrm{nR} - 1)$
Post-loop	:	$2 + 2 + \mathrm{Mcall}(\mathrm{nR}) + 9 + 2$

ACorr_16 **Autocorrelation** (cont'd)

Example	:	When nX = 54, nR = 5
	:	Cycle Count = 335 cycles
c) When nR = nX		
Pre-loop	:	9
Loop	:	$19 \times (nR/2 - 1) + Mcall(1) + \ldots$ $+ Mcall(nX - 2)$
Post-loop	:	$2 + 2 + Mcall(nX - 1) + 17 + 2$
Example	:	When nR = nX = 54
	:	Cycle Count = 2141 cycles
d) When nR = 1		
The OutData loop is bypassed		
Cycle Count	:	$13 + Mcall(1) + 9 + 2$
Example	:	When nX = 54, nR = 1
	:	Cycle Count = 79 cycles
e) When nR = 2		
The OutData loop is bypassed		
Cycle Count	:	$13 + Mcall(1) + 14 + Mcall(2)$ $+ 6 + 2$
Example	:	When nX = 54, nR = 2
	:	Cycle Count = 145 cycles

Code Size 268 bytes

Conv_16 Convolution

Signature

```
void Conv_16(DataS  *X,
             DataS  *H,
             DataL  *R,
             int    nR,
             int    nH
             );
```

Inputs

X	:	Pointer to First Input-Vector
H	:	Pointer to Second Input-Vector
R	:	Pointer to Output-Vector
nH	:	Size of Second Input-Vector
nR	:	Size of Output-Vector

Output

R(nR) : Output-Vector

Return

None

Description

The convolution of two sequences X and Y is done. The input vectors are 16-bit and returned output is 32-bit. All the vectors are halfword aligned. The length of input vectors is even. Therefore for full convolution length output vector length is always odd.

Conv_16 Convolution (cont'd)

Pseudo code

```
{
   frac16 *X;          //Ptr to First Input-Vector
   frac16 *H;          //Ptr to Second Input-Vector
   frac64 acc;         //Convolution result
   int dCnt;           //Convolution loop count

   //Macro
   macro Conv;
   {
      int aOvlpCnt;    //Convolution loop count
      aOvlpCnt = dCnt;//Convolution loop count for current convolution
                       //output

      for(i=0; i<aOvlpCnt; i++)
      {
         acc = acc + (*(X-K)) (*) H(K) + (*(X-K-1)) (*) H(K+1)
                       //acc += X(n) * H(0) + X(n-1) * H(i)
         K = K + 2;
      }

      Conv_16:
      {
         int anHCnt;
         int anX_nHCnt;
         int anR_nXCnt;
         int dCnt = 1;
         int nX_1;

         dnHCnt = nH/2 - 1;
         anHCnt = dnHCnt;
         X1 = X;       //Store Ptr to First Input-Vector
         H1 = H;       //Store Ptr to Second Input-Vector
         *R++ = X[0].H[0]
         acc = 0.0;

         Conv;         //Convolution computation
         *R++ = (frac32 sat)acc;
                       //Result stored
         X1 = X1 + 2;
         X = X1;
         H = H1;
```

Conv_16 **Convolution** (cont'd)

```
if (nR == 3)
go to Conv_R_3;
for (i=0; i<anHCnt; i++)
{
   acc = 0.0;
   acc = X[n] (*) H[0];
   Conv;     //Convolution computation
   *R++ = (frac16 sat)acc;
             //Result stored
   dCnt++;
   X = X1;
   H = H1;
   acc = 0.0;
   Conv;     //Convolution computation
   X1 = X1 + 2;
   X = X1;
   H = H1;
   *R++ = (frac32 sat)acc;
}
nX_1 = nR - nH;
X1 = X1 - 1;
X = X1;
anR_nXCnt = dnHCnt;
if (nX == nH)
go to Conv_DCntr;

H = H1;
anX_nHCnt = nX - nH;
for (i=0; i<anX_nHCnt; i++)
{
   X = X1;
   acc = 0.0;
   Conv;     //Convolution computation
   X1 = X1 + 1;
   H = H1;
   *R++ = (frac32 sat)acc;
             //Result stored
}
```

Conv_16 **Convolution** (cont'd)

```
        X = X1;
    for (i=0; i<anR_nXCnt; i++)
    {
        dCnt--;
        H1 = H1 + 1;
        H = H1;
        acc = 0.0;
        acc = X(n) (*) H(0);
        Conv;     //Convolution computation
        *R++ = (frac32 sat)acc;
        X1 = X1 - 1;
        H1 = H1 + 1;
        X = X1;
        H = H1;
        acc = 0.0;
        Conv;     //Convolution computation
        *R++ = (frac32 sat)acc;
        X1 = X1 + 1;
        X = X1;
    }

    Conv_R_3;
        acc = 0.0;
        acc = X(nX - 1) (*) H(nH - 1);
        K++ = (frac32)acc;

    return;
  }
}
```

Techniques

- For optimization implementation is divided into three loops. First loop where overlap count increases, second loop overlap count remains same and third loop overlap count decreases
- A macro Conv is used which calculates convolution output. The macro uses packed load and dual MAC to reduce the number of cycles for a given overlap count of two sequences
- Use of dual MAC and MAC instructions
- Intermediate results stored in 64 bit register (16 guard bits)
- Instruction ordering for zero overhead Load/Store

Assumptions

- Inputs are in 1Q15 format, Output is in 1Q31 format
- nX and nH are even and hence nR is always odd

Conv_16 Convolution (cont'd)

Memory Note

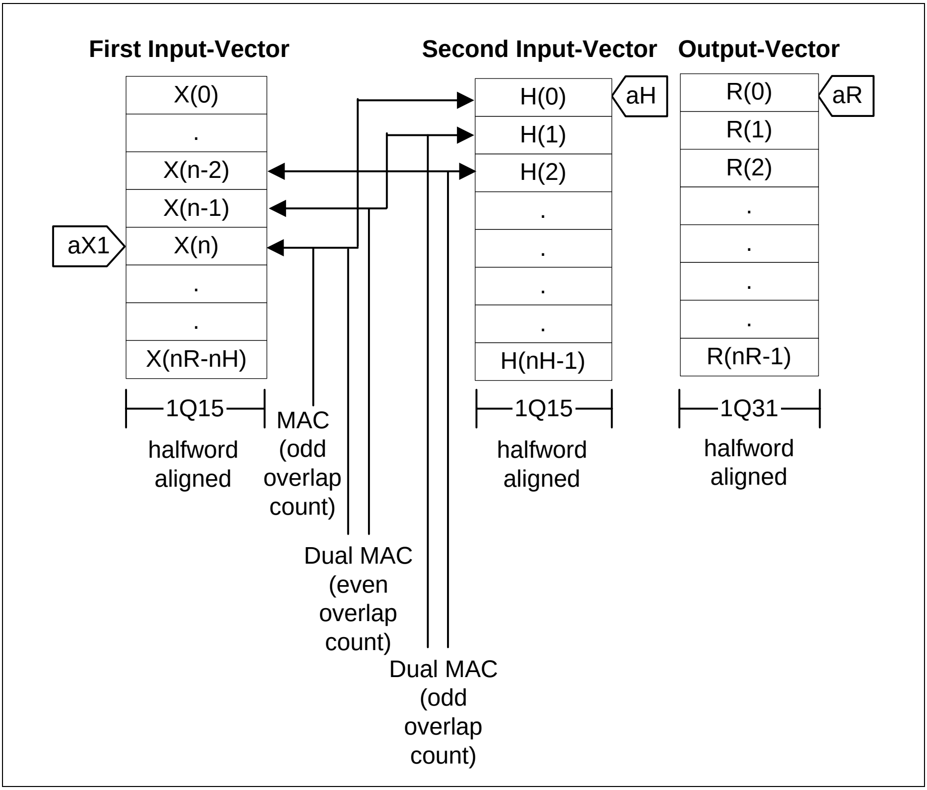


Figure 4-93 Conv_16

Conv_16 **Convolution** (cont'd)

Implementation

Convolution is same as FIR filtering. For convolution one of the two sequences is inverted in time. To implement the convolution, the two sequences are multiplied together and the products are summed to compute the output sample. To calculate next output sample time inverted signal is shifted by one and process is repeated. If two sequences of length nX and nH are convolved the convolution length is given by nR = nX+nH-1.

The pointer to input vectors, output vector, the size of output vector (nR) and size of the input sequence of smaller length (nH) are sent as arguments. The size of the other input sequence is calculated as (nR-nH+1).

Implementation uses macro Conv. The macro uses two load word and one dual MAC instruction. Thus two multiplications and one addition is performed per loop according to the equation

$$acc = acc + X(n) \cdot H(0) + X(n-1) \cdot H(1) \tag{4.165}$$

Thus loop count is always (overlap count/2-2) for even and odd lengths of overlap count. For odd one more MAC is performed before the macro is called.

The convolution is divided into three loops.

First loop: The first two convolution outputs are given as

$$R(0) = X(0) \cdot H(0) \tag{4.166}$$

$$R(1) = X(1) \cdot H(0) + X(0) \cdot H(1) \tag{4.167}$$

The number of multiplication and additions required for computation of R(i) increases as i is increased from 0 to nH-1. The overlap count of the two input sequences is even for i = 1, 3, 5,...,nH-1 and odd for i = 0, 2, 4,...,nH-2. Macro is called for every R(n).

Conv_16 **Convolution** (cont'd)

The first loop is unrolled and first two outputs are calculated outside the loop. One pass through the first loop gives two outputs. Thus loop count for first loop is (nH/2-2). This loop gives first nH outputs.

Second loop: Here the overlap count is always constant and is nH. Macro Conv is called for (nX-nH) times. This loop gives next (nX-nH) outputs.

This loop is skipped if nX = nH.

Third loop: The overlap count decreases from (nH-1) to 1 as i increases from (nX+1) to (nR-1). The loop is unrolled and last output which needs only one multiplication is done outside the loop. Thus loop count for this loop is (nH/2-2).

Example

Trilib\Example\Tasking\Statistical\expConv_16.c, expConv_16.cpp
Trilib\Example\GreenHills\Statistical\expConv_16.cpp, expConv_16.c
Trilib\Example\GNU\Statistical\expConv_16.c

Cycle Count

For i = 1 to nH-1

Mcall(1) and Mcall(2) = 1+2+1

$$\mathrm{Mcall(i)} = 1 + 2 \times (\mathrm{i} + 1)/2 + 2$$
for i = 3, 5,...,(nH-1)

$$\mathrm{Mcall(i)} = 1 + 2 \times \mathrm{i}/2 + 2$$
for i = 4,...,(nH-2)

For i = nH to nX-1

$$\mathrm{Mcall(i)} = 1 + 2 \times \mathrm{nH}/2 + 2$$
for i = nH,nH+1,...,(nX-1)

For i = nX to nR-2

Conv_16 **Convolution** (cont'd)

$$Mcall(i)) = 1 + 2 \times (nH/2 - (i/2 - (nX)/2 + 1)) + 2$$
for i = nX, nX+2,...,(nR-5)

$$Mcall(i)) = 1 + 2 \times (nH/2 - ((i-1)/2 - (nX)/2 + 1)) + 2$$
for i = nX+1, nX+3,...,(nR-4)

Mcall(nR-3) and Mcall(nR-2) = 1+2+1

For nX>nH

14+Mcall(1)

First loop

$$(nH/2 - 1)[18 + Mcall(2) + Mcall(3) + \ldots + Mcall(nH-1)] + 8$$

For nH>4

$$(nH/2 - 1)[18 + Mcall(2) + Mcall(3) + \ldots + Mcall(nH-1)] + 7$$

For nH = 4

Second loop

$$(nX - nH)[8 + Mcall(nH) + Mcall(nH+1) + \ldots + Mcall(nX-1)] + 3$$

Third loop

$$(nH/2 - 1)[19 + Mcall(nX) + Mcall(nX+1) + \ldots + Mcall(nR-2)] + 2$$

2+2

For nX = nH

Second loop is skipped and first loop will take 2 extra cycles for jump

For nH = nX =2

16+Mcall(1)+4

Code Size 420 bytes

Avg_16 Mean Value

Signature DataS Avg_16(DataS *X,
int nX
);

Inputs X : Pointer to Input-Buffer
nX : Size of Input-Buffer

Output None

Return R : Mean value of the input values

Description This function calculates the mean of a given array of values. It takes pointer to the array and size of the array as input. Input range is [-1, 1). The return is the mean value represented using 32 bits.

Pseudo code

```
{
   frac32 acc = 0;     //Sum of inputs
   frac32 one_nX;      //1/no. Of inputs
   frac64 Ra;
   frac32 R;

   for(i=0; i<nx; i++)
   {
      acc = acc + X[i];
                       //acc in 17Q15 format
   }
   one_nX = 1/nX;      //one_nX in 1Q31 format
   Ra = acc (*) one_nX;
                       //Mean value in 17Q47 format
   R = (frac32)Ra;     //32 bit result in 1Q31 format

}
```

Techniques
- 32 bit addition is used to provide 16 guard bits for addition
- Instruction ordering provided for zero overhead Load/Store

Assumptions
- Inputs are in the range [-1,1) and in 1Q15 format. Output is also in 1Q15 format.

Avg_16 **Mean Value** (cont'd)

Memory Note

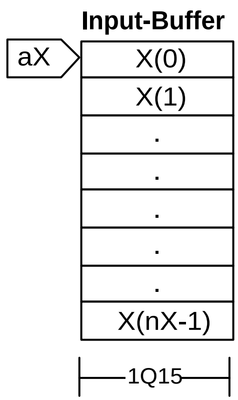


Figure 4-94 Avg_16

Implementation

The function takes a short pointer to an array whose mean is to be calculated and the size of the array as input. The return value is the 32 bit mean value.

$$\text{mean} = \frac{x(0) + x(1) + \ldots + x(nx-1)}{nx} \qquad [4.168]$$

Load of inputs and addition are performed in a loop. The input values are read into the lower 16 bits of a 32 bit register. Hence 32 bit addition is performed on 17Q15 values thereby providing 16 guard bits for addition. The reciprocal of the size is calculated.
The product of the sum and the reciprocal gives the mean value in 17Q47 format. This is converted to 1Q31 and returned.

Avg_16 **Mean Value** (cont'd)

Example	*Trilib\Example\Tasking\Statistical\expAvg_16.c, expAvg_16.cpp* *Trilib\Example\GreenHills\Statistical\expAvg_16.cpp, expAvg_16.c* *Trilib\Example\GNU\Statistical\expAvg_16.c*
Cycle Count	Pre-loop : 3 Loop : nX + 2 Post-loop : 27+2
Code Size	54 bytes

5 Applications

The following applications are described.

- Spectrum Analyzer
- Sweep Oscillator
- Equalizer

5.1 Spectrum Analyzer

To perform a spectral analysis of any signal spectrum analyzer is used. The spectrum analyzer uses radix-2 FFT to get the frequency content of a signal. The FFT algorithm takes N-data-samples x(n), n=0,1,...,N-1 of the input given and produces N-point complex frequency samples X(K), K=0,1,...,N-1. The power spectrum is obtained by squaring the scaled magnitude of complex frequency samples.

$$P(K) = \frac{1}{N}|X(K)|^2 = \frac{1}{N}\{Re[X(K)^2] + Im[X(K)^2]\} \quad K=0,1,...,N/2 \qquad [5.1]$$

The Power Spectrum Density (PSD) gives a measure of the distribution of the average power of a signal over frequency.

The PSD can be actual or averaged. The actual PSD gives N/2 point output from N point complex FFT output. The averaged PSD gives b band output where the number of bands is user input.

A simple example showing functioning of Spectrum Analyzer.

The following are the diagrams where input given is a mixture of 4kHz and 12kHz sine waves sampled at 32kHz. The FIR filter has a cutoff frequency of 8 kHz. So after filtering the input to FFT contains only 4kHz wave. The power spectrum gives the corresponding frequency. Here the number of FFT points taken is 512. The maximum frequency value represented by the spectrum is 16K as sampling frequency is 32K. Since FFT is of 512 complex points it will result in a power spectrum of 256 points. Here 256th doppler bin represents frequency of 16K. So the frequency corresponding to 64th doppler bin is 4K.

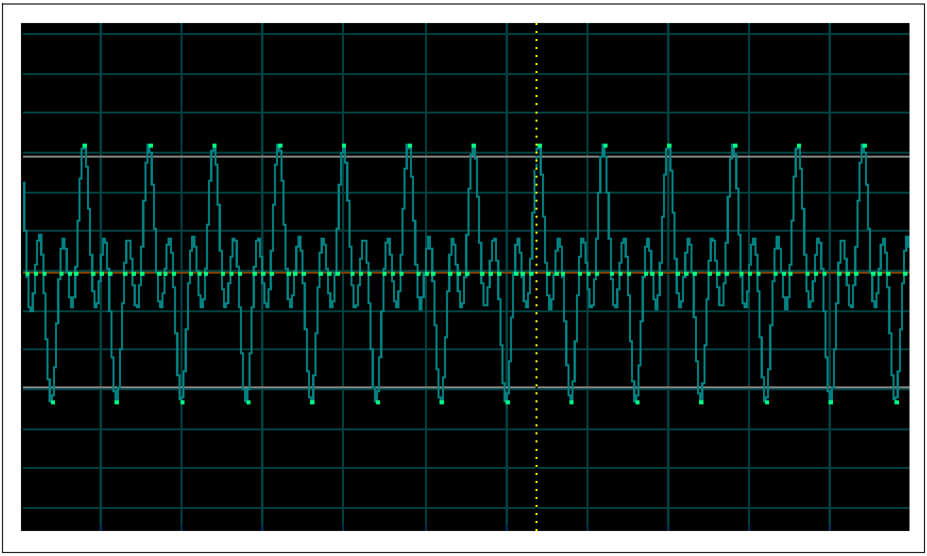


Figure 5-1 Input given to Spectrum Analyzer

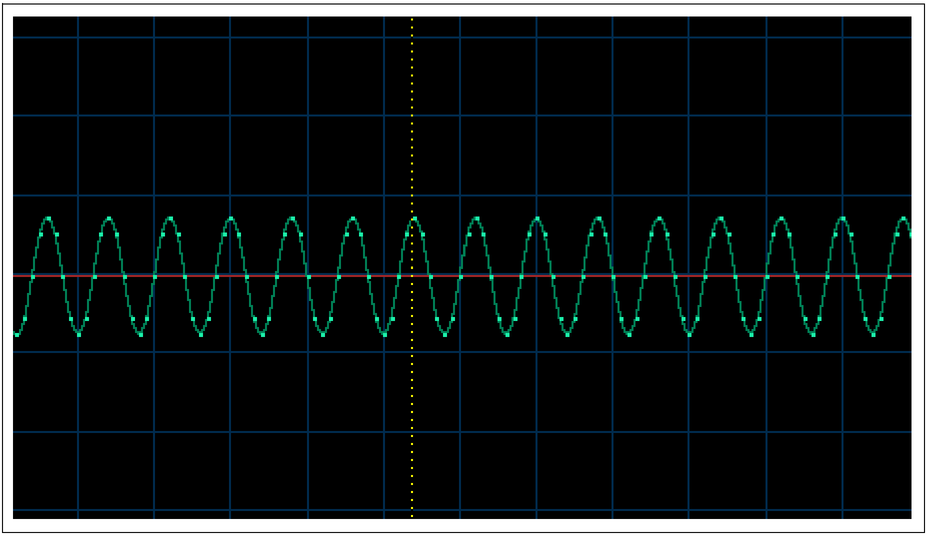


Figure 5-2 Output of FIR filter

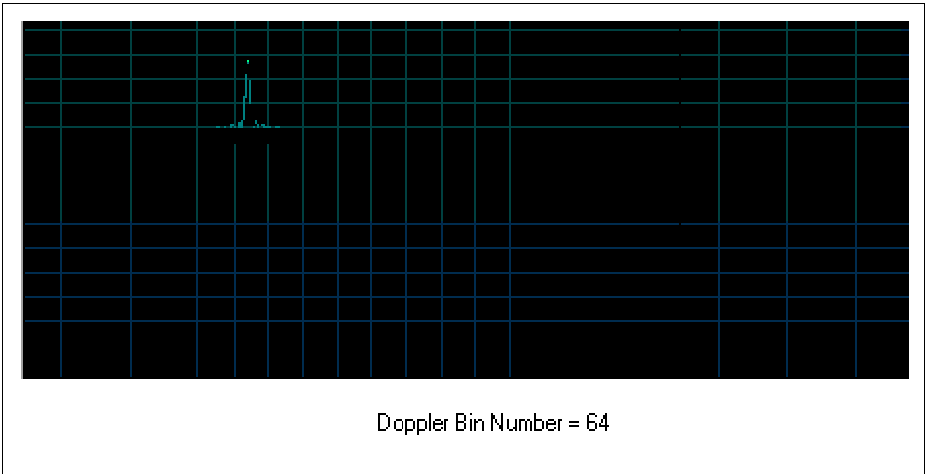


Figure 5-3 Output power spectrum considering actual PSD

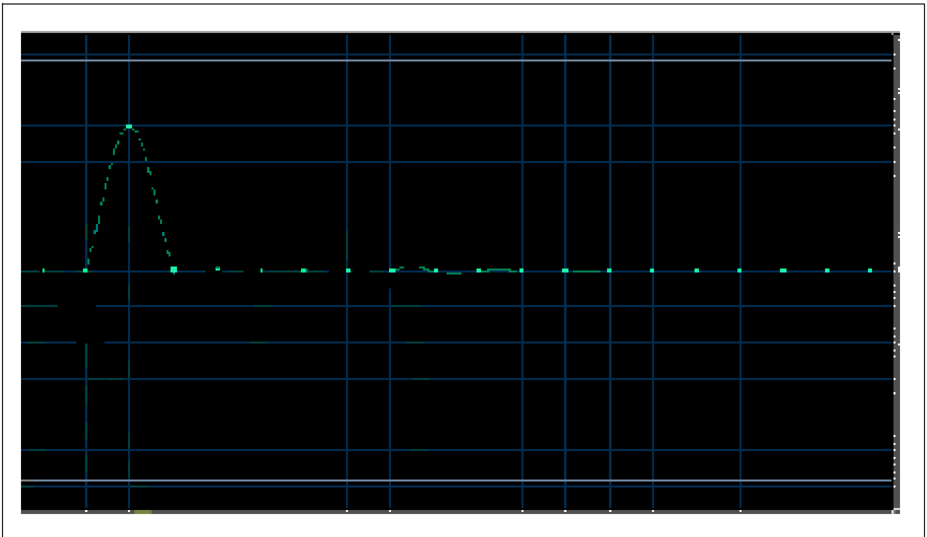


Figure 5-4 20 Band averaged power spectrum

5.2 Sweep Oscillator

The generation of pure tones is often used for testing DSP systems and to synthesize waveforms of required frequencies. The basic oscillator is a special case of an IIR filter where the poles are on the unit circle and the initial conditions are such that the input is an impulse. If the poles are moved outside the unit circle, the oscillator output will grow at an exponential rate. If the poles are placed inside the unit circle, the output will decay toward zero. The state (or history) of the second-order section determines the amplitude and phase of the future output.

The impulse of a continuous second order oscillator is given by

$$R(t) = e^{-dt}\frac{\sin \omega t}{\omega} \qquad [5.2]$$

If d>0 then the output will decay toward zero and the peak will occur at

$$t_{peak} = \frac{\mathrm{Arctan}(\omega/d)}{\omega} \qquad [5.3]$$

The peak value will be

$$R(t_{peak}) = \frac{e^{-dt_{peak}}}{\sqrt{d^2 + \omega^2}} \qquad [5.4]$$

A second order difference can be used to generate an approximation response of this continuous-time output. The equation for a second-order discrete time oscillators is based on an IIR filter and is as follows

$$R_{n+1} = a_1 y_n - a_2 y_{n-1} + b_1 x_n \qquad [5.5]$$

where, the x input is only present for t=0 as an initial condition to start the oscillator and

$$a_1 = 2e^{-d\tau}\cos(\omega\tau) \qquad [5.6]$$

$$a_2 = e^{-d\tau} \qquad [5.7]$$

where, τ is the sampling period (1/fs) and ω is 2π times the oscillator frequency.

The frequency and rate of change of envelope of the oscillator output can be changed by modifying the values of d and ω on a sample by sample basis.

The sweep oscillator implemented here uses the function IirBiq_4_16.

When the oscillator has to be started, the function oscillator is called with one of the arguments indicating to start new oscillator where impulse is given as an input and the delay line gets updated. From the next sample onwards input is made zero, but as the poles lie on the unit circle the output is oscillatory at given frequency. The coefficients, whenever there is frequency change, are calculated for that particular frequency.

Following parameters are programmable

- The sampling frequency
- Start frequency
- The factor, by which frequency has to be incremented or decremented
- The number of cycles for a start frequency
- Number of cycles for changed frequency

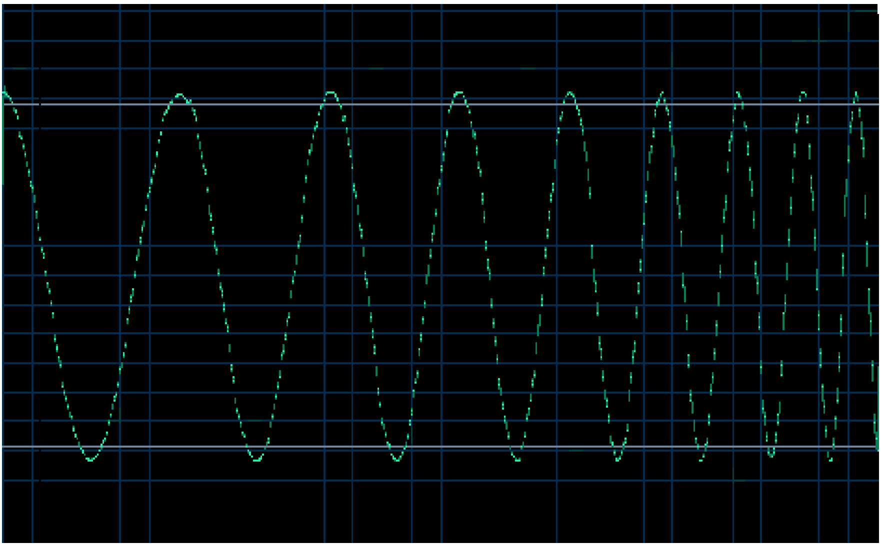


Figure 5-5 Sweep Oscillator

5.3 Equalizer

A Graphic Equalizer is a powerful tool to characterize and enhance audio signals.

Technically it is composed of a bank of band-pass filters, each with a fixed center frequency and a variable gain. This kind of processing unit is called Graphic since the position of the slider resembles the frequency response of the filters bank. Thus its usage is extremely intuitive, moving the slider up boosts a selected band, moving it down will cut it.

Graphic equalizer uses high quality **constant Q** digital filters. This allows to isolate every filter section from the effects of the amplitude with respect to the centre frequency and bandwidth. The result is an accurate control permitting each band not to affect the adjacent ones.

5-band equalizer implemented uses 128-tap FIR filters to get the desired band pass filter response. Here the function FirBlk_16 is used for FIR filtering.

The five bands are

- 0 - 170
- 170 - 600
- 600 - 3K
- 3K - 12K
- 12K - 16K

The gain in dB for each band is programmable. Also the common master gain is programmable. The filters are designed for three sampling frequencies 32kHz, 44.1kHz, 48kHz. The user gives the desired sampling frequency as an input. Depending on this corresponding filter bank is selected. After input is passed through all the five filters the output of each filter is multiplied with the gain for that particular band. All the outputs are added and then finally multiplied with master gain to get the equalizer output.

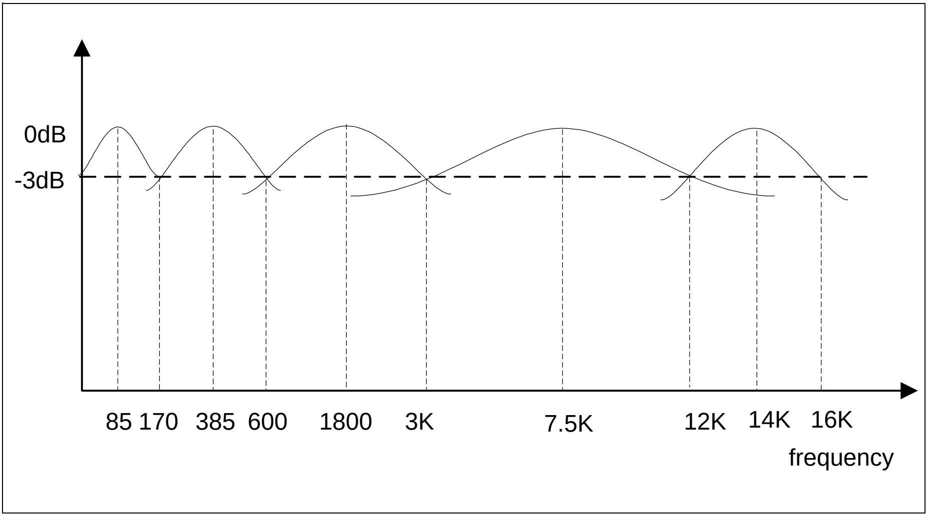


Figure 5-6 5 Band Graphic Equalizer

5.4 Hardware Setup for Applications

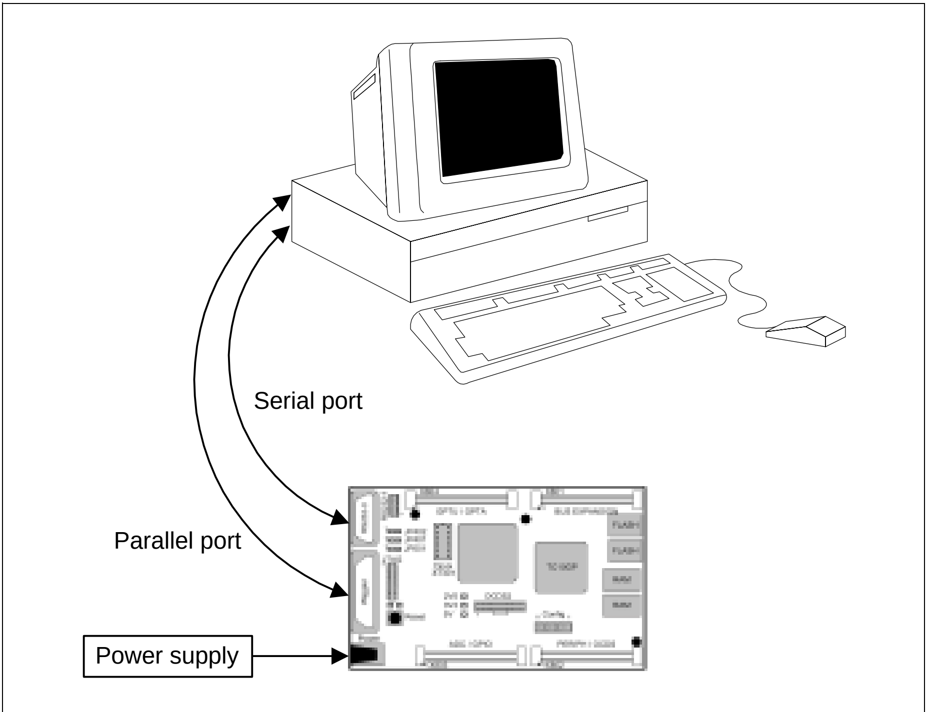


Figure 5-7 Hardware Setup

1. Preparing the TriBoard for Debugging

Connect a parallel cable from the parallel port on the PC to the On Board Wiggler (DB25) on the TriBoard as shown in **Figure 5-7**. Connect a “one to one” serial port cable from the RS232 interface on the PC to the serial interface (RS232-0) on the TriBoard. For details refer TriBoard manual.

2. Starting a Terminal Program

A terminal program can be used to communicate with the TriBoard via RS232. Both transmit and receive of data is possible. The TriBoard has an RS232 transceiver on board to meet the RS232 specification of your PC.

3. Power Up the TriBoard

Connect the power supply (6V to 25V DC, power plug with surrounding ground) to the lower left edge of the card as shown in **Figure 5-7**. Power up the unit. The green LED's next to the OCDS2 Connector indicates the right power status. The red LED near the reset button indicates the reset status.

Once the connections are done the applications can be run over the TriBoard. The spectrum analyzer and the equalizer applications can be run by reading the input from the serial port of TriBoard and calculated output is sent again to serial port of TriBoard.

5.4.1 Spectrum Analyzer

Frontend for Spectrum Analyzer:

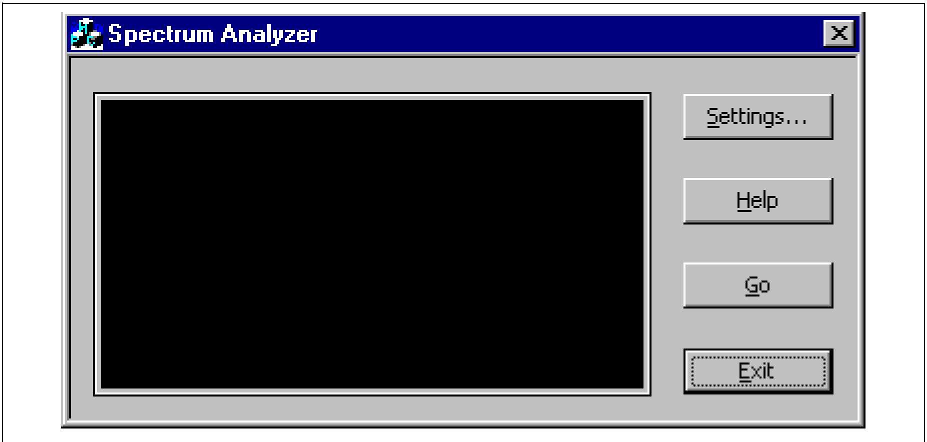


Figure 5-8 Frontend of Spectrum Analyzer

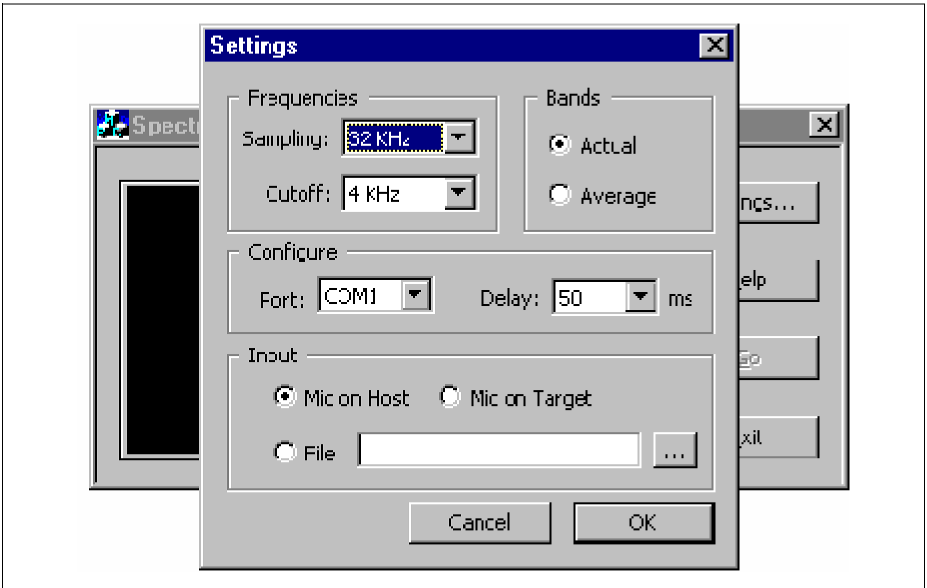


Figure 5-9 Settings for Spectrum Analyzer

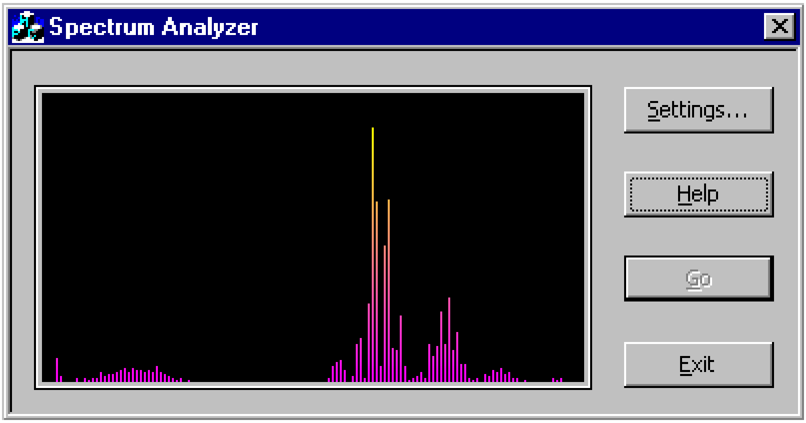


Figure 5-10 Actual PSD of the input (128 point power spectrum)

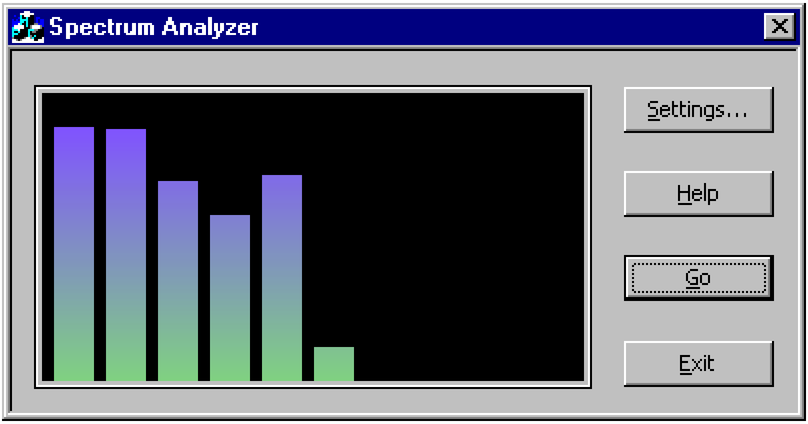


Figure 5-11 Averaged PSD of the input (10 bands)

The inputs taken from the user are

1. Actual band or average band
2. Sampling frequency
3. Cutoff frequency

Actual band gives 128 point power spectrum of the given 1024 input samples.

Sampling frequency can be one of the three choices 32K, 44.1K, and 48K.

Cutoff frequency can be one of the three choices 4K, 8K, and 16K.

From the host machine, first 1 byte is sent to the serial port of TriBoard to get the above user inputs. Then acknowledgement is sent to host machine as 1 byte is received. Then follows the data from the host machine to the TriBoard. 1024, 16 bit data is sent to the TriBoard. This data is read in a buffer. The FFT of 1024 points input data is calculated. From the frequency spectrum, power spectrum density is calculated by squaring the scaled magnitude complex frequency samples. Then 128 point PSD is calculated from 512 point PSD by averaging. If user input is actual PSD, the 128 point PSD is sent to serial port of TriBoard. If the user input is average input then calculated PSD is divided into 10 segments and averaged 10 bands are sent to serial port. The host machine reads the data on the serial port and displays actual or averages spectrum depending on user input.

5.4.2 Equalizer

Frontend for Equalizer:

Settings:

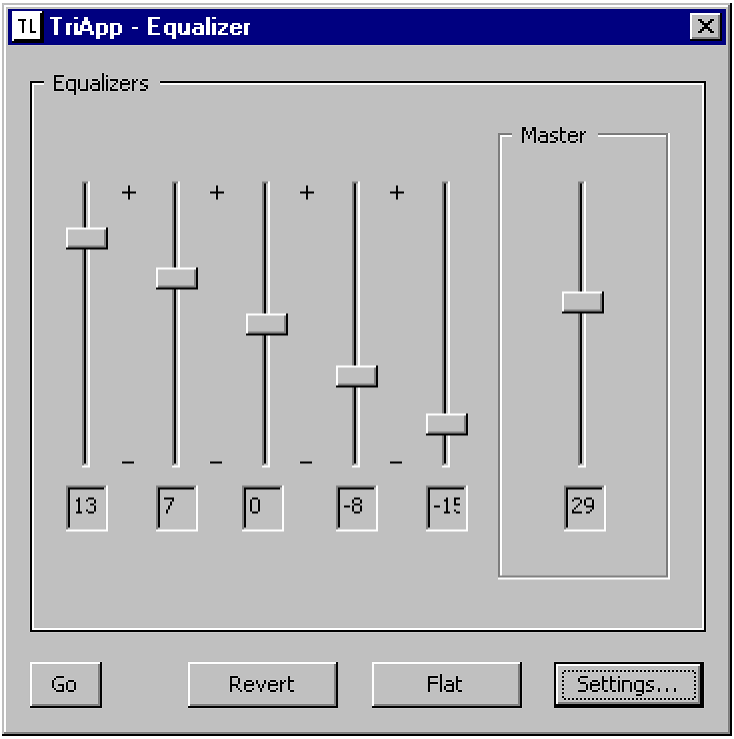


Figure 5-12 Frontend of Equalizer

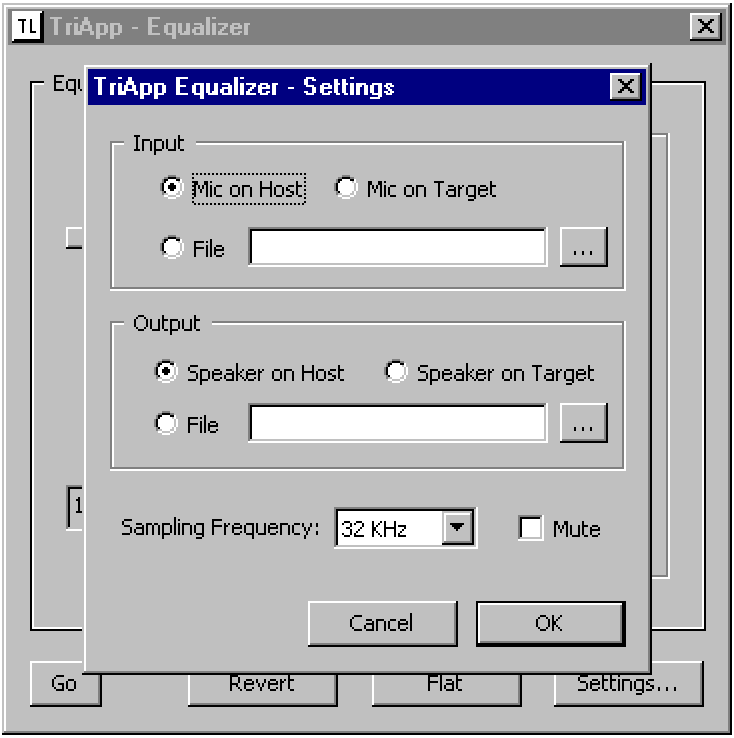


Figure 5-13 Settings for Equalizer

The inputs taken from the user are

1. Sampling frequency
2. 5 band gains in dB
3. Master gain in dB

Sampling frequency can be one of the three choices 32K, 44.1K and 48K.

Band gains can be from -20dB to +20dB.

Master gain can be from 0 to +50dB.

From the host machine, first 13 bytes are sent to the serial port of TriBoard to get the above user inputs. Then a one byte acknowledgement is sent to the host machine. This is followed by the data from the host machine. 128, 16 bit data is sent to the TriBoard. This data is read in a buffer. This is band passed through 5 Band pass filters. Each of the outputs of the filters is multiplied by the respective gain and the final output is generated by their sum. This is then multiplied by the master gain and sent back to the host machine. The host machine then sends this data to an output file.

6 References

1. Digital Signal Processing by Alan V Oppenheim and Ronald W Schafer
2. Digital Signal Processing, A Practical Approach by Emmanuel C Ifeachor and Barrie W Jervis
3. Discrete-Time Signal Processing by Alan V Oppenheim and Ronald W Schafer
4. Advanced Engineering Mathematics by Erwin Kreyszig
5. K. R. Rao and P. Yip, Discrete Cosine Transform: Algorithms, Advantages, Applications
6. W. H. Chen, C. H. Smith, and S. C. Fralick, "A fast computational algorithm for the Discrete Cosine Transform"

7 Frequently Asked Questions

7.1 FIR Basics

1. What are FIR filters?

FIR filters are one of two primary types of digital filters used in Digital Signal Processing (DSP) applications (the other type being IIR). FIR means Finite Impulse Response.

2. Why is the impulse response "finite"?

The impulse response is "finite" because there is no feedback in the filter, if an impulse is given as an input (i.e., a single one sample followed by many zero samples), zeroes will eventually come out after the one sample has made its way in the delay line past all the coefficients.

3. What is the alternative to FIR filters?

DSP filters can also be Infinite Impulse Response (IIR). IIR filters use feedback, so when an impulse is input the output theoretically rings indefinitely.

4. How do FIR filters compare to IIR filters?

Each has advantages and disadvantages. Overall, the advantages of FIR filters outweigh the disadvantages, so they are used much more than IIRs.

a) What are the advantages of FIR Filters as compared to IIR filters?

Compared to IIR filters, FIR filters have the following advantages.

- They can easily be designed to be "linear phase". Simple linear-phase filters delay the input signal, but do not distort its phase.
- They are simple to implement. On most DSP microprocessors, the FIR calculation can be done by looping a single instruction.
- They are suited to multi-rate applications. By multi-rate, we mean either decimation (reducing the sampling rate), interpolation (increasing the sampling rate) or both. Whether decimating or interpolating, the use of FIR filters allows some of the calculations to be omitted, thus providing an important computational efficiency. In contrast, if IIR filters are used, each output must be individually calculated, even if that output is discarded. (so the feedback will be incorporated into the filter.)
- They have desirable numeric properties. In practice, all DSP filters must be implemented using finite-precision arithmetic, i.e., a limited number of bits. The use of finite-precision arithmetic in IIR filters can cause significant problems due to the use of feedback, but FIR filters have no feedback, so they can usually be implemented using fewer bits.

- They can be implemented using fractional arithmetic. Unlike IIR filters, it is always possible to implement an FIR filter using coefficients with magnitude of less than 1.0. (The overall gain of the FIR filter can be adjusted at its output, if desired). This is an important consideration when using fixed-point DSP's, because it makes the implementation much simpler.

b) What are the disadvantages of FIR Filters as compared to IIR filters?

FIR filters sometimes have the disadvantage that they require more memory and/or calculation to achieve a given filter response characteristic. Also, certain responses are not practical to implement with FIR filters.

5. What terms are used in describing FIR filters?

Impulse Response - The impulse response of an FIR filter is actually just the set of FIR coefficients. (If an impulse is put into an FIR filter which consists of a one sample followed by many zero samples, the output of the filter will be the set of coefficients, as the one sample moves past each coefficient in turn to form the output.)

Tap - An FIR tap is simply a coefficient/delay pair. The number of FIR taps, (often designated as N) is an indication of

- The amount of memory required to implement the filter
- The number of calculations required
- The amount of filtering the filter can do

In effect, more taps means more stopband attenuation, less ripple, narrower filters, etc.

7.1.1 FIR Properties

Linear Phase

1. What is the association between FIR filters and linear-phase?

Most FIRs are linear-phase filters. When a linear-phase filter is desired an FIR is usually used.

2. What is a linear phase filter?

Linear Phase refers to the condition where the phase response of the filter is a linear (straight-line) function of frequency (excluding phase wraps at +/- 180 degrees). This results in the delay through the filter being the same at all frequencies. Therefore, the filter does not cause phase distortion or delay distortion. The lack of phase/delay distortion can be a critical advantage of FIR filters over IIR and analog filters in certain systems, for example, in digital data modems.

3. What is the condition for linear phase?

FIR filters are usually designed to be linear-phase (but they don't have to be). An FIR filter is linear-phase if (and only if) its coefficients are symmetrical around the center coefficient, i.e., the first coefficient is the same as the last, the second is the same as the next-to-last, etc. (A linear-phase FIR filter having an odd number of coefficients will have a single coefficient in the center which has no mate.)

4. What is the delay of a linear-phase FIR?

The formula is simple. Given an FIR filter which has N taps, the delay is

(N - 1) / Fs, where Fs is the sampling frequency. So, for example, a 21 tap linear-phase FIR filter operating at a 1 kHz rate has delay (21 - 1) / 1 kHz = 20 milliseconds.

Frequency Response

1. What is the Z transform of an FIR filter?

For an N-tap FIR filter with coefficients h(k), whose output is described by

$$y(n) = h(0)\cdot x(n) + h(1)\cdot x(n-1) + h(2)\cdot x(n-2) + \ldots + h(N-1)\cdot x(n-N-1) \qquad [7.1]$$

The filter's Z transform is

$$H(z) = h(0)z^{-0} + h(1)z^{-1} + h(2)z^{-2} + \ldots + h(N-1)z^{-(N-1)} \qquad [7.2]$$

2. What is the frequency response formula for an FIR filter?

The variable z in H(z) is a continuous complex variable and can be described as

$$z = re^{jw} \qquad [7.3]$$

where,

r is the magnitude and w is the angle of z.

let r = 1, then H(z) around the unit circle becomes the filter's frequency response $H(e^{jw})$. This means that substituting e^{jw} for z in H(z) gives an expression for the filter's frequency response $H(e^{jw})$, which is

$$H(e^{jw}) = h(0)e^{-j0w} + h(1)e^{-j1w} + h(2)e^{-j2w} + \ldots + h(N-1)e^{-j(N-1)w} \text{ or} \qquad [7.4]$$

Using Euler's identity,

$$e^{-ja} = \cos(a) - j\sin(a) \qquad [7.5]$$

H(w) can be written in rectangular form as

$$\begin{aligned} H(jw) &= h(0)[\cos(0w) - j\sin(0w)] + h(1)[\cos(1w) - j\sin(1w)] + \ldots \\ &\quad + h(N-1)[\cos((N-1)w) - j\sin((N-1)w)] \end{aligned} \quad [7.6]$$

3. How to scale the gain of an FIR filter?

Multiply all coefficients by the scale factor.

Numeric Properties

1. Are FIR filters inherently stable?

Yes, since they have no feedback elements, any bounded input results in a bounded output.

2. What makes the numerical properties of FIR filters good?

The key is the lack of feedback. The numeric errors that occur when implementing FIR filters in computer arithmetic occur separately with each calculation, the FIR does not remember its past numeric errors. In contrast, the feedback aspect of IIR filters can cause numeric errors to compound with each calculation, as numeric errors are fed back. The practical impact of this is that FIRs can generally be implemented using fewer bits of precision than IIRs. For example, FIRs can usually be implemented with 16-bits, but IIRs generally require 32-bits, or even more.

6. Why are FIR filters generally preferred over IIR filters in multirate (decimating and interpolating) systems?

Because only a fraction of the calculations that would be required to implement a decimating or interpolating FIR in a literal way actually needs to be done.

Since FIR filters do not use feedback, only those outputs which are actually going to be used have to be calculated. Therefore, in case of decimating FIRs (in which only 1 of N outputs will be used), the other N-1 outputs do not have to be calculated. Similarly, for interpolating filters (in which zeroes are inserted between the input samples to raise the sampling rate) the inserted zeroes need not have to be multiplied with their corresponding FIR coefficients and sum the result, the multiplication-additions that are associated with the zeroes are just omitted. (because they don't change the result anyway.)

In contrast, since IIR filters use feedback, every input must be used, and every input must be calculated because all inputs and outputs contribute to the feedback in the filter.

7.1.2 FIR Design

1. What are the methods of designing FIR filters?

The three most popular design methods are (in order):

a) Parks-McClellan: The Parks-McClellan method is probably the most widely used FIR filter design method. It is an iteration algorithm that accepts filter specifications in terms of passband and stopband frequencies, passband ripple, and stopband attenuation. The fact that all the important filter parameters can be directly specified is what makes this method so popular. The Parks-McClellan method can design not only FIR filters but also FIR differentiators and FIR Hilbert transformers.

b) Windowing: In the windowing method, an initial impulse response is derived by taking the Inverse Discrete Fourier Transform (IDFT) of the desired frequency response. Then, the impulse response is refined by applying a data window to it.

c) Direct Calculation: The impulse responses of certain types of FIR filters (e.g. Raised Cosine and Windowed Sine) can be calculated directly from formulae.

7.2 IIR Basics

1. What are IIR filters?

IIR filters are one of two primary types of digital filters used in Digital Signal Processing (DSP) applications (the other type being FIR). IIR means Infinite Impulse Response.

2. Why is the impulse response "infinite"?

The impulse response is "infinite" because there is feedback in the filter, if an impulse is given as an input (a single 1 sample followed by many 0 samples), an infinite number of non-zero values will come out (theoretically).

3. What is the alternative to IIR filters?

DSP filters can also be Finite Impulse Response (FIR). FIR filters do not use feedback. So, for an FIR filter with N coefficients, the output always becomes zero after putting in N samples of an impulse response.

4. What are the advantages of IIR filters as compared to FIR filters?

IIR filters can achieve a given filtering characteristic using less memory and fewer calculations than a similar FIR filter.

5. What are the disadvantages of IIR filters as compared to FIR filters?

- They are more susceptible to problems of finite-length arithmetic, such as noise generated by calculations and limit cycles. (This is a direct consequence of feedback, when the output is not computed perfectly and is fed back, the imperfection can compound.)
- They are harder (slower) to implement using fixed-point arithmetic.
- They do not offer the computational advantages of FIR filters for multirate (decimation and interpolation) applications.

7.3 FFT

The Fast Fourier Transform is one of the most important topics in Digital Signal Processing but it is a confusing subject which frequently raises questions. Here, we answer Frequently Asked Questions (FAQs) about the FFT.

7.3.1 FFT Basics

1. What is FFT?

The Fast Fourier Transform (FFT) is a fast (computationally efficient) way to calculate the Discrete Fourier Transform (DFT).

2. How does the FFT work?

By making use of periodicities in the sines that are multiplied to do the transforms, the FFT greatly reduces the amount of calculation required.

Functionally, the FFT decomposes the set of data to be transformed into a series of smaller data sets to be transformed. Then, it decomposes those smaller sets into even smaller sets. At each stage of processing, the results of the previous stage are combined in special way. Finally, it calculates the DFT of each small data set. For example, an FFT of size 32 is broken into 2 FFTs of size 16, which are broken into 4 FFTs of size 8, which are broken into 8 FFTs of size 4, which are broken into 16 FFTs of size 2. Calculating a DFT of size 2 is trivial.

This can be explained as follows. It is possible to take the DFT of the first N/2 points and combine them in a special way with the DFT of the second N/2 points to produce a single N-point DFT. Each of these N/2-point DFTs can be calculated using smaller DFTs in the same way. One (radix-2) FFT begins, therefore, by calculating N/2 2-point DFTs. These are combined to form N/4 4-point DFTs. The next stage produces N/8 8-point DFTs and so on, until a single N-point DFT is produced.

3. How efficient is the FFT?

The DFT takes N^2 operations for N points. Since at any stage the computation required to combine smaller DFTs into larger DFTs is proportional to N and there are $\log_2(N)$ stages (for radix-2), the total computation is proportional to $N * \log_2(N)$. Therefore, the ratio between a DFT computation and an FFT computation for the same N is proportional to $N / \log_2(n)$. In cases where N is small this ratio is not very significant, but when N becomes large, this ratio gets very large. (Every time N is doubled, the numerator doubles, but the denominator only increases by 1.)

4. Are FFTs limited to sizes that are powers of 2?

No. The most common and familiar FFTs are radix-2. However, other radices are sometimes used, which are usually small numbers less than 10. For example, radix-4 is especially attractive because the twiddle factors are all 1, -1, j or -j, which can be applied without any multiplications at all.

Also, mixed radix FFTs can be done on composite sizes. In this case, you break a non-prime size down into its prime factors and do an FFT whose stages use those factors. For example, an FFT of size 1000 might be done in six stages using radices of 2 and 5, since 1000 = 2 * 2 * 2 * 5 * 5 * 5. It can also be done in three stages using radix-10, since 1000 = 10 * 10 * 10.

5. Can FFTs be done on prime sizes?

Yes, although these are less efficient than single-radix or mixed-radix FFTs. It is almost always possible to avoid using prime sizes.

7.3.2 FFT Terminology

1. What is an FFT radix?

The radix is the size of an FFT decomposition. For single-radix FFTs, the transform size must be a power of the radix.

2. What are twiddle factors?

Twiddle factors are the coefficients used to combine results from a previous stage to form inputs to the next stage.

3. What is an "in place" FFT?

An "in place" FFT is an FFT that is calculated entirely inside its original sample memory. In other words, calculating an "in place" FFT does not require additional buffer memory. (as some FFTs do.)

4. What is bit reversal?

Bit reversal is just what it sounds like, reversing the bits in a binary word from left to right. Therefore the MSB’s become LSB’s and the LSB’s become MSB’s. The data ordering required by radix-2 FFTs turns out to be in bit reversed order, so bit-reversed indices are used to combine FFT stages. It is possible (but slow) to calculate these bit-reversed indices in software. However, bit reversals are trivial when implemented in hardware. Therefore, almost all DSP processors include a hardware bit-reversal indexing capability. (which is one of the things that distinguishes them from other microprocessors.)

5. What is decimation in time versus decimation in frequency?

FFTs can be decomposed using DFTs of even and odd points, which is called a Decimation-In-Time (DIT) FFT or they can be decomposed using a first-half/second-half approach, which is called a Decimation-In-Frequency (DIF) FFT.

8 Appendix

Convention Document for TriLib

8.1 Introduction

8.1.1 Scope of the Document

This document describes the Programming Conventions for the TriCore DSP Library.

The purpose of the document is to bring out a unified programming style for the TriCore DSP. It is recommended that the guidelines and the conventions be observed to organize each DSP application software. This ensures uniform and well-structured code.

8.2 File Organization

8.2.1 File Extensions

The Software application, TriLib should be organized as a collection of modules or files that belongs to any one of the following categories. The following table brings out the details of the different categories of files.

Table 8-1 Directory Structure

Type	Extension	Description
'C' Source files	*.c	C Language Source files
Include files	*.h, *.inc	The include files for the 'C' and the assembly functions. The C include files generally have *.h as extension. Assembly can have different extensions based on the compiler in use. All the include files should define the global constants and variable types, if any. They should not allocate memory or define functions as this prevents them from being included by multiple source files. All subroutines which form part of the overall interface to a source file should be declared in include file. This provides a convenient overview of the interface and allows the compiler or assembler to check for errors
Testvector files	*.dat	These files should only contain data to be used for test purposes or algorithmic usage. There must not be any code in these data files. These files, if used, will probably be included or copied (`.include` directive) in other source files or assembled as stand-alone modules. These files can also be given as the command line argument for the example programs depending upon the implementation
Build files	*.pjt, *.bld, *.out	It is strongly recommended that a project make file is maintained that checks for any out-of-date target files and builds them automatically. Different compilers use different extension for the build files.
TriCore Source files	*.asm, *.tri, *.S	Different compilers use different extensions for the assembly source files. Generally *.asm file is widely accepted by many compilers.

8.2.2 File Naming Conventions

The Files will be named using the following convention. This helps in easy identification of the file.

- All the Source files of TriCore assembly will have *.asm, *.tri or *.S extension depending upon the compiler being used. The name can be formulated by using the following convention.

<Function class Operation name>_<Suffix info>.asm/(.tri)/(.S)

The suffix has to be numeric that gives the information such as data size (16 or 32 bits) of input in case of arithmetic operations, or constraint on the order of Filters, say multiple of four (this is optional and can be used wherever applicable). When order and bit information are required, the suffix info is exploded as <order>_<no.bits>

Abbreviated function name approximately in multiples of three letters for each concept or words.

a. The initial three letters will be the class of the functions such as Finite Impulse Response filters and can be represented as **’Fir’**

b. The next three letters will be operation name such as for block operation it can be represented as **’Blk’** or for Maximum Index as **’MaxIdx’**

8.2.3 File Header and Guidelines

The following is the format of the file header.

```
//******************************************************************************************
```

// @Module:	Name of the function or module (e.g., main())
// @Filename:	Name of the file with extension (e.g., expFir_4_16.c)
// @Project:	Name of the Project (DSP Library for Tricore V1.2, V1.3)
// @Controller:	Name of the controller (TriCore V1.2, V1.3)
// @Compiler:	Compiler name (Tasking or GHS or GNU)
// @Version:	Version of the S/W
// @Description:	The description of the file
// @See Also:	List the include files used
// @References:	List the reference documents /manuals
// @Caveats:	Caveats if any
// @Date:	Date (only in this format dd mm yy e.g., 14^{th} Jan 2000)
// @History:	Revision history or the modification details

//--

Notes

- The names in the fields - module, file name etc., should match exactly with the existing name of the file and the module. Consistency should be maintained in all the fields wherever there are multiple references.
- The description should provide the information about the implementation in the file and the global issues, if any.

8.3 Coding Rules and Conventions for 'C' and 'C++'

This section describes the coding rules and conventions for C/C++ languages.

8.3.1 File Organization

- It is recommended to have one functional module in one file. This can be relaxed when the functional module is very small and does not justify having a separate file.
- Tab size is always set to four white spaces.

8.3.2 Function Declaration

The general recommendations and rules for the function declaration are as follows.

- Declaration of all global interface functions should be done in a header file, which should be made available to the external programs.
- All local functions should be declared in the respective C files that makes use of them. This should not be visible outside.
- All functions, arguments, and variables must be explicitly declared. If a function does not return a value, then the return type should be *void*.
- Function definition should never be put in a *.h* header file unless it is an inline function this is applicable only for C++.
- Declare all external functions in a *.h* header file.
- Do not #include *.c* files.
- Any module that needs to provide *extern* variables must provide a header file that declares them. Other modules that need to reference the *extern* variable should include that header file.
- All global variables should be declared as *extern* in the common header file. This avoids the multiple declaration if included in multiple files.

Function definition should have the following syntax.

```
<return_type> <func_name>(<data_type><param1>, /* comments */
                          <data_type><param2>, /* comments */
                          ...
                          ...
                          <data_type><paramn>) /* comments */
{
      /*********Declaration of local variables ********/

      /***** Description about the body below**********/
      /**** Body *****/
      ....
      ....
      ....
      /***** Start of loop *****/
```

```
      {
      } /***** Mark end of loop here *****/
      /*****Mark end of body here ******/
}/* Mark end of function here with the <func_name> ***/
```

8.3.3 Variable Declaration

The general recommendations and rules for the variable declaration is as follows.

- All global variables should be defined in a *.c* file and not in a *.h* file. In the *.h* header file, it should be declared as *extern*.
- If different types of variables are declared in a file, there should be a clear demarcation between the global variables for the project and the global variables for a file.
- Declare the class of variables in groups with a general comment. Determination of the class can be done on basis of usage, locality, etc.
- Local variables should be declared only at the beginning of the function for greater visibility.

Example:

```
void func_name()
{
   int x;
   /****** body of the function*****/

   int y; /* improper - never declare a variable inside the body of the
             function */
   /******end of the body***********/
}
```

- Never mix the index variables or pointer variables with that of the other local variables in the declaration.

Example:

```
int i, temp_32, *pTable;   /* Improper */
int i;                     /* Correct */
int *pTable;               /* Correct */
int temp_32;               /* Correct */
```

- Declare and use the variables as per the naming convention that is formalized for each of the projects.
- For pointer variable declaration, use the '*' sign near to the variable name and in case of multiple pointer declaration, use the '*' sign separately for each of the variables.

- Never initialize the pointer in the same line where it is declared, do it explicitly to increase the visibility.

8.3.4 Comments

- Comments should be written at the beginning of the body of the function to describe its activity.
- Comments and code should not cross the 79^{th} column of the line. In case there is a need to further comment, use the next line and start in the same column it was started in previous line.
- Comments should be to the point.
- Comments should be avoided where the code itself is sufficient to understand the flow of the program.
- Comments are mandatory at the beginning of the new block. It should explain the purpose and the operation of that block.
- Arithmetic and logical operations can be represented by means of symbols in the comments to make it short and increase the readability.

8.4 Coding Rules and Conventions for Assembly Language

This section describes the coding rules and conventions for the Assembly language.

8.4.1 File Organization

- It is recommended to have one functional module in one file. This can be relaxed when the functional module is very small and does not justify having a separate file.
- Tab size is always set to four white spaces.

8.4.2 General Coding Guidelines

The following describes the order of declaration and syntax for the same in the assembly language programs.

- Include syntax should start from the 1st column since some assemblers does not accept if it is other than 1st column.

Example:

```
; -------- Section for all include header files --------------
.include file.h
```

- All include files should have a preprocessor directive at the beginning.

Example:

```
#ifndef _TriLib_h
#define _TriLib_h
....
....
#endif // end of _TriLib_h include file
```

- Describe the external references

Example:

```
; -------- Section for external references -------------------
.global    _mpy32      ;here _mpy32 is the global label that
                       ;can be referenced in other files by using extern
.extern    _mpy32      ;used to refer the global labels.
; -------- Section for constants -----------------------------
Pi             .set    3.14
Localvarsize   .set    1
```

Note: `.equ` directive can also be used here but `.set` can be used if one needs to change the value at a later point in the program.

- Constant definitions for the pointer offsets

Example for Tasking Compiler:

```
.define     W16    '2'      ;Two bytes offset
.define     W32    '4'      ;Four bytes offset
.define     W64    '8'      ;Eight bytes offset
```

Example for GHS Compiler:

```
#define     W16    2        ;Two bytes offset
#define     W32    4        ;Four bytes offset
#define     W64    8        ;Eight bytes offset
```

Example for GNU Compiler:

```
.equ        W16    2        ;Two bytes offset
.equ        W32    4        ;Four bytes offset
.equ        W64    8        ;Eight bytes offset
```

- Use the freely available registers for local variables and document the same. Otherwise, use the macros which will set aside a frame for the required size by decrementing the stack.

Example:

```
FEnter  5                    ;will decrement the stack by 5 words
```

(FEnter is the macro that subtracts the stack pointer by the required number which is passed as the argument)

- Labels must be written in the same convention as that of the function naming convention and should start from the 1st column. It is recommended that all labels should have some prefix that relates it to the function it belongs. This helps to avoid duplicate label names in different files.

For instance, all labels in an assembly function named *Function1* could begin with the prefix *F1_*. A label should end with a colon character.

Example:

In case of a *Finite Impulse Response* filter, a label can be written as *FirS4_TapL:* for tap loop of FIR on sample, coefficient multiple of 4. This helps to identify a label from mnemonics and other assembler directives.

- All instruction mnemonics must be written in lower-case letters. Instruction mnemonics must begin from the 5th column of each line. All operands must start from the 17th column. Most text editors can be configured to position tabs to any column number. In case of multiple operands, they should be separated with a comma.
- When writing a complex assembly language function, it is sometimes difficult to keep track of the contents of registers. Use of symbolic names to replace registers can improve readability of code. It is recommended that `.define` or `#define` assembler directives be used depending upon the compiler used to substitute registers with appropriate symbolic names. Since a register may be used for more than one purpose during the execution of a program, more than one symbolic name can be equated to one register. Note that all symbols replacing registers should be in the convention as described in the section 7.4.4, as shown in the following example.

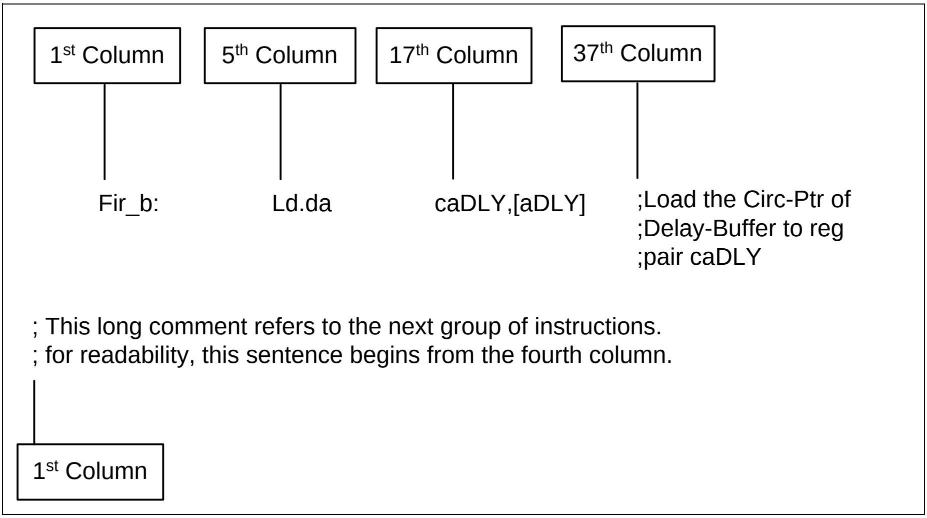
Example for Tasking compiler:

```
.define     caeDLY      "a12"     ;Even-Reg of Circ-Ptr
.define     caoDLY      "a13"     ;Odd-Reg of Circ-Ptr
.define     aTapLoops   "a14"     ;Number of taps
```

Another advantage of using symbolic names to identify registers is maintainability of the code. By using symbolic names for registers, it becomes easier to change register assignments later. For example, if a function uses A1 as an input parameter pointing to an array but the calling function prefers using A2 for that purpose, the `.define` directive in the called function can be modified to equate the input array symbol with A2 instead of A1. If a symbol had not been equated to A1 in the called function, it would have required a search-and-replace operation to find all occurrences of A1 and replace them with A2. Symbolic names should be used whenever it is possible.

- Comments can either begin from the 37th column or from the 1st column if the entire line is required for lengthy comments at the beginning of the block. This rule is for general instruction wise commenting only. In case of block or program commenting, which is trying to explain about the overall function/algorithm, it can start from 1st column. Remember the commenting is inclusive of the semicolon also. Comments should be avoided between parallel instructions. The commenting conventions are described in the later section.

Example:



8.4.3 Function Organization

The general function organization is as follows. Changes can be made to suit the requirements.

```
Function_name_label

----------Prolog of fn starts here--------
SP = SP + Locvarsize     ;Allocate local variables in stack

----------End of prolog-------------------

Body of function......

----------Epilog starts here--------------

SP=SP-Locvarsize         ;Deallocate local variables
                         ;in stack

----------End of epilog-------------------

RETURN
```

- If there is a reference code or pseudocode, use the same variable names for easy debugging and maintenance.
- Loop start and end should be commented for easy identification.

```
;--------------------------loop start----------------------------
         Body of loop
;--------------------------loop end------------------------------
```

8.4.4 Variables and Argument Convention

The variables should have following conventions.

Prefix	**Variables**
s	Short (16 bit value)
ss	Two short values in a 32 bit register
ssss	Four short values in a 64 bit register
l	Long (32 bit) in a 32 bit register
ll	Two long in a 64 bit register
a	Address register or data type prefix
dTmp	Temporary data register
n	Loop count data register
ca	Circular buffer address register pair
aa	Pointer to pointer
o	Odd register
e	Even register

Example:

```
;Registers used for storing input Data Registers (Tasking)
.define   ssXa     "d10"     ;D10-Register holds 2 inputs
.define   ssXb     "d11"     ;D11-Register holds 2 inputs
.define   ssssXab  "d10"     ;E10-Register holds 4 inputs
.define   aVec1    "d11"     ;A1 is the address register
.define   nCnt     "a5"      ;A5 used as loop counter
.define   caH      "a6"      ;A6 is the pointer to circular
```

```
                                 ;buffer address pointer
```

- Define a temporary register of two short values

Example:

```
.define   dTmp     "d4"      ;Generic temp-data-reg
```

- Define the lower half or the upper half of the registers explicitly for GHS and GNU compilers whereas for Tasking it is not needed.

Example for the incorrect implementation:

```
.define   lKa      "d8"      ;d8-Register
.define   lKa_UL   "D8ul"        ;

maddm.h   Acc,Acc,drXb,lKa_UL,1
```

Example for the correct implementation:

```
.define   ssKa     "d8"      ;d8-Register holds


maddm.h   Acc,Acc,ssXb,ssKa ul,#1
```

- Use a consistent notation. Always use the symbolic name that is defined. Do not mix the symbolic names with the register names.

Example for the incorrect implementation:

```
.define   caCoef   "a6/a7"       ;A6/A7-Circ-buf

ld.da       caDelay,[A7]         ;Use absolute
                                 ;register name
ld.w        lKb,[caCoef+c]2*w16  ;Use define
```

If the defines are changed then the absolute names will not match. Also the probability of making errors is high, and the code is not readable. In case of defines that use a register pair `(e.g. caH)`, additional defines can be used for individual odd and even registers.

8.4.5 Function Header and Guidelines

The format of the function header is as follows.

```
;*************************************************************************
;  Return_Value                     Function_Name ( Arg1,
                                                    Arg2,
                                                    ……..
                                                    ……..
                                                    Arg N);
; INPUTS:                          Input parameters
; OUTPUTS:                         Output parameters
; RETURN:                          Return value and type and its significance
; DESCRIPTION:                     Describe the function if relevant give the formula,
                                   C code, Error conditions, etc.
; ALGORITHM:                       Algorithm of the implementation in simple english or
                                   in the pseudo C syntax equations etc.
; TECHNIQUES:                      List the different techniques of optimization used in
                                   the implementation
; ASSUMPTIONS:                     List the assumptions made
; MEMORY NOTE:                     Table to depict the variables and the its type, name,
                                   alignment, etc.
; REGISTER USAGE:                  List of registers used in this function
; CYCLE COUNTS:                    Profiled result in terms of number of cycles
; CODE SIZE:                       Size in terms of words of memory
; DATE:                            Date
; VERSION:                         Version of the function
;****************************************************************************************************
```

Notes

- The signature of the function should be same as what is declared as the function prototype.
- The input/output parameters are passed to the function as arguments. Sometimes the input parameters can also act as the output parameters, such as a pointer variable getting used and updated inside the function. This information should be explained in this field. This field should have information about the type of parameter, its normal value or range of values and it's significance.
- Return values should not be mixed with the output parameters. Sometimes return values are themselves the output values of the function. In DSPLIB implementation, the return values are generally void in many cases as the output will be in form of an array, etc. The return value should give information about the type, range of values and its significance.
- The description field should contain the required description of the function, without any redundant information. It should contain equations wherever applicable. The purpose of the description is to give a good overview of the function and the methodology of implementation. It should also contain information on the implementation with right justification for a specific method, which is followed in the implementation. Alternative methodologies can also be discussed which are optional. Error conditions should be discussed wherever applicable.
- Any assumptions that are made in the implementation such as bits of precision, range of values etc., should be mentioned under assumptions. The assumption should deal only with the implicit requirements of the function. Any direct given data or the requirements should not be listed in the assumptions list.

8.5 Testing

8.5.1 Test Methodology

- Testing of the DSP library is done using the test vectors that are developed internally.
- The reference 'C' code is developed and reviewed critically.
- For few codes the input test vectors (test cases) are used to generate the reference output test vectors using the reference 'C' code.
- The module under test will be tested using the test vector. The output of the module will be cross-examined for correctness with the reference output test vectors. This is test for the PASS/FAIL criterion.
- For all the codes the input test vectors are given in the example main of the function. Same test case can be given to test code and outputs of both can be verified.

8.5.2 Convention

Refer Test Design Specification: INF_DSP.1.0.TD.1.0 dated March 01, 2000.

8.6 Compiler Support

8.6.1 General Common System

The TriLib implementation is designed for multiple compilers. TriCore processor is supported by three compilers at present namely,

- Tasking
- GHS
- GNU

TriLib should be implemented with and without language extensions. It is intended not to have any changes in the organization of the code to support the different compilers. Since the implementation of each of the compilers varies from one another, it is expected that the implementation of the TriLib cannot be uniform across the compilers.

The following sections will bring in the details of how to support the TriLib in Tasking, GHS and the GNU compilers. The main idea of this is to bring in the aspects of portability and extensibility across different platforms.

8.6.2 Distinguishing Tasking, GHS and GNU Specific Directives

Tasking compiler, GHS and GNU have a specific set of assembler directives, refer the individual documentation for more details.

Principally, all the compilers have some directive which are same by syntax and usage perspective. There are also some equivalent directives whose syntax differs. Finally there are some distinctive sets of directives, which are specific to each of the compilers.

Refer individual documentation for more details on the language extensions part of each of the compilers.

8.6.3 Note on Implementation on Different Compilers

Table 8-2 Equal Directives

Tasking Compiler	GHS Compiler	GNU Compiler
.align	.align	.align
.byte	.byte	.byte
.word	.word	.word
.double	.double	.double
.float	.float	.float

Table 8-2 Equal Directives

.space	.space	.space
.set	.set	.set
.extern	.extern	.extern
.include	.include	.include
.macro	.macro	.macro
.endm	.endm	.endm
.exitm	.exitm	.exitm
.if	.if	.if
.else	.else	.else
.endif	.endif	.endif

Table 8-3 Directives with the same functionality but different syntax

Tasking Compiler	GHS Compiler	GNU Compiler
.define	#define	#define
.global	.globl	.global/.globl
.sect ".text"	.text	.text
.sect ".data"	.data	.data
.half	.hword	.hword

Table 8-4 Datatypes with DSPEXT

Tasking Compiler	GHS Compiler	GNU Compiler
_sfract	fract16	Not applicable
_fract	fract32	Not applicable
_sfract_circ	circptr<frac16>	Not applicable
_fract_circ	circptr<frac32>	Not applicable

Table 8-4 Datatypes with DSPEXT

`struct {     _sfract imag;     _sfract real; } CplxS;`	`struct {     frac16 imag;     frac16 real; } CplxS;`	Not applicable
`struct {     _fract imag;     _fract real; } CplxL;`	`struct {     frac32 imag;     frac32 real; } CplxL;`	Not applicable

Datatypes without DSPEXT are same for all compilers. They are as shown

Table 8-5 Datatypes without DSPEXT

Data Size	Data Type
16-bit	short
32-bit	int
Circular buffer structure 16-bit	`struct {     short *base;     short index;     short base; } CptrDataS`
Circular buffer structure 32-bit	`struct {     int *base;     short index;     short base; } CptrDataL`
Complex 16-bit	`{     short imag;     short real; } CplxS`
Complex 32-bit	`{     int imag;     int real; } CplxL`

The instructions which need to be changed for porting.

1. **Instructions using address register pair**: In case of instruction using address register pair for GNU one need to specify even address register of the register pair.

Example for Tasking Compiler:

```
ld.da caDLY,[aDLY]0
```

Example for GHS Compiler:

```
ld.da caDLY,[aDLY]0
```

Example for GNU Compiler:

```
ld.da caeDLY,[aDLY]0
```

2. **Definition of data register pair:** It should be as shown below.

Example for Tasking Compiler:

```
.define   llAcc   "d12/d13" or
.define   llAcc   "e12"
```

Example for GHS Compiler:

```
#define   llAcc   "d12/d13 or
#define   llAcc   e12
```

Example for GNU Compiler:

```
#define   llAcc   %e12
```

3. **Instructions using packed multiply-add:** For instructions using packed multiply-add where lower or upper 16-bits of registers have to be specified, in case of GHS and GNU those registers need to be explicitly defined.

Example for Tasking Compiler:

```
maddm llAcc, llAcc, ssex, ssOH ul, #1
```

In case of GHS the `ssOH_ul` need to be defined as

```
#define ssOH d9
#define ssoH_ul d9ul
```

Example for GHS Compiler:

```
maddm llAcc, llAcc, ssex, ssOH_ul, 1
```

In case of GNU the `ssOH_ul` need to be defined as

```
#define ssOH %d9
#define ssoH_ul %d9ul
```

Example for GNU Compiler:

```
maddm llAcc, llAcc, ssex, ssOH_ul, 1
```

4. **Arithmetic Instruction using same source and destination register:** Any arithmetic instruction where source and destination registers are same GHS needs to explicitly specify registers but it works on Tasking.

Example for Tasking Compiler:

```
add dTmp, #1 or
add dTmp, dTmp, #1
```

Example for GHS Compiler:

```
add dTmp, dTmp, 1
```

Example for GNU Compiler:

```
add dTmp, dTmp, 1
```

5. **Reading data from the data section:** While reading data from the data section of the code the label of data section should be preceded by `%sdaoff` in case of GHS

Example for Tasking Compiler:

```
lea aH, CoeffTab
```

Example for GHS Compiler:

```
lea aH, %sdaoff(CoeffTab)
```

Example for GNU Compiler:

```
lea aH, CoeffTab
```

6. Macro definition:

Example for Tasking Compiler:

```
macro_name .macro
```

Example for GHS Compiler:

```
.macro macro_name
```

Example for GNU Compiler:

```
.macro macro_name
```

7. The arguments sent to macro:

For Tasking and GHS they will be used as it is where as in case of GNU it is preceded by \ in the code of macro.

Example for Tasking Compiler:

```
FirDec .macro Ev_Coef,Ev_Coef_Od_Df
.if Ev_Coef == TRUE
sh     dTmp1, dTmp1, #-1    ;>>1 2Taps/loop
```

Example for GHS Compiler:

```
.macro FirDec Ev_Coef,Ev_Coef_Od_Df
.if Ev_Coef == TRUE
sh     dTmp1, dTmp1, -1     ;>>1 2Taps/loop
```

Example for GNU Compiler:

```
.macro FirDec Ev_Coef,Ev_Coef_Od_Df
.if \Ev_Coef == TRUE
sh     dTmp1, dTmp1, -1     //>>1 2Taps/loop
```

8. Loop within macro:

For Tasking the label for loop within macro should always have first character as ^ , e.g. ^`conv_conL` where as for GHS label need to be a number and where the loop instruction encounters the label should be that number with a letter b as it is a backward jump. For forward jump it should be f.

Example:

For Tasking:
```
^conv_conL :
        .
        .
        loop aloopcount, ^conv_conL
```

For GHS:
```
1:
    .
    .
    loop aloopcount, 1b
```

For GNU:
```
1:
    .
    .
    loop aloopcount, 1b
```

9. **cmov instruction:** Instruction `cmovn` does not work for GHS ver 2.0 it has to be replaced by `seln`.

Example for Tasking Compiler:
```
cmovn loAcc, dTmp2, dTmp1
```

Example for GHS Compiler:
```
seln loAcc, dTmp2, dTmp1, loAcc
```

Example for GNU Compiler:
```
seln loAcc, dTmp2, dTmp1, loAcc
```

10. **Jump Instruction:** Jump instruction syntax is different across these compilers.

Example for Tasking Compiler:
```
jnz.t  dTmp:0, label
```

Example for GHS Compiler:
```
jnz.t  dTmp,0, label
```

Example for GNU Compiler:

```
jnz.t  dTmp,0, label
```

Note:

The instruction jz works only for the GreenHills V2.0.2. For old versions of GreenHills this instruction is not supported.

9 Glossary

A

Acquisition Time	The time required for a sample-and-hold (S/H) circuit to capture an input analog value. Specifically, the time for the S/H output to approximately equal its input.
Adaptive Delta Modulation (ADM)	A variation of delta modulation in which the step size may vary from sample to sample.
ADC (or A/D, Analog-to-Digital Converter)	The electronic component which converts the instantaneous value of an analog input signal to a digital word (represented as a binary number) for Digital Signal Processing. The ADC is the first link in the digital chain of signal processing.
ADPCM (Adaptive Differential Pulse Code Modulation)	A very fast data compression algorithm based on the differences occurring between two samples.
Algorithm	A structured set of instructions and operations tailored to accomplish a signal processing task. For example, a Fast Fourier Transform (FFT), or a Finite Impulse Response (FIR) filter are common DSP algorithms.
Aliasing	The problem of unwanted frequencies created when sampling a signal of a frequency higher than half the sampling rate.
All-Pass Filter	A filter that provides only phase shift or phase delay without appreciable changing the magnitude characteristic.
Amplitude	1. Greatness of size, magnitude. 2. Physics. The maximum absolute value of a periodically varying quantity. 3. Mathematics. a) The maximum absolute value of a periodic curve measured along its vertical axis. b) The angle made with the positive horizontal axis by the vector representation of a complex number. 4. Electronics. The maximum absolute value reached by a voltage or current waveform.

Analog	A real world physical quantity or data, characterized by being continuously variable (rather than making discrete jumps), and can be as precise as the available measuring technique.
ANSI (American National Standards Institute)	A private organization that develops and publishes standards for voluntary use in the U.S.A.
Anti-Aliasing Filter	A low-pass filter used at the input of digital audio converters to attenuate frequencies above the half-sampling frequency to prevent aliasing.
Anti-Imaging Filter	A low-pass filter used at the output of digital audio converters to attenuate frequencies above the half-sampling frequency to eliminate image spectra present at multiples of the sampling frequency.
ASCII (pronounced "ask-ee") (American Standard Code for Information Interchange)	An ANSI standard data transmission code consisting of seven information bits, used to code 128 letters, numbers, and special characters. Many systems now use an 8-bit binary code, called ASCII-8, in which 256 symbols are represented (for example, IBM's "extended ASCII").
Asymmetrical (non-reciprocal) Response	Term used to describe the comparative shapes of the boost/cut curves for variable equalizers. The cut curves do not mirror the boost curves, but instead are quite narrow, intended to act as notch filters.
Asynchronous	A transmission process where the signal is transmitted without any fixed timing relationship between one word and the next (and the timing relationship is recovered from the data stream).

B

Bandpass Filter	A filter that has a finite passband, neither of the cutoff frequencies being zero or infinite. The bandpass frequencies are normally associated with frequencies that define the half power points, i.e., the -3 dB points.
Band-Limiting Filters	A low-pass and a high-pass filter in series, acting together to restrict (limit) the overall bandwidth of a system.

Bandwidth Abbreviation. BW	The numerical difference between the upper and lower -3 dB points of a band of audio frequencies. Used to figure the Q, or quality factor for a filter.
Bilinear Transform	A mathematical method used in the transformation of a continuous time (analog) function into an equivalent discrete time (digital) function. Fundamentally important for the design of digital filters. A bilinear transform ensures that a stable analog filter results in a stable digital filter, and it exactly preserves the frequency-domain characteristics, albeit with frequency compression.
Bit Error Rate	The number of bits processed before an erroneous bit is found (e.g. 10E13), or the frequency of erroneous bits (e.g. 10E-13).
Bit Rate	The rate or frequency at which bits appear in a bit stream. The bit rate of raw data from a CD, for example, is 4.3218 MHz.
Bit Stream	A binary signal without regard to grouping.
Bit-Mapped Display	A display in which each pixel's color and intensity data are stored in a separate memory location.
Boost/Cut Equalizer	The most common graphic equalizer. Available with 10 to 31 bands on octave to 1/3-octave spacing. The flat (0 dB) position locates all sliders at the center of the front panel. Comprised of bandpass filters, all controls start at their center 0 dB position and boost (amplify or make larger) signals by raising the sliders, or cut (attenuate or make smaller) the signal by lowering the sliders on a band-by-band basis. Commonly provide a center-detent feature identifying the 0 dB position. Proponents of boosting in permanent sound systems argue that cut-only use requires adding make-up gain which runs the same risk of reducing system headroom as boosting.
Buffer	In data transmission, a temporary storage location for information being sent or received.
Burst Error	A large number of data bits lost on the medium because of excessive damage to or obstruction on the medium.

Bus	One or more electrical conductors used for transmitting signals or power from one or more sources to one or more destinations. Often used to distinguish between a single computer system (connected together by a bus) and multi-computer systems connected together by a network.

C

Cartesian Coordinate System	1. A two-dimensional coordinate system in which the coordinates of a point in a plane are its distances from two perpendicular lines that intersect at an origin, the distance from each line being measured along a straight line parallel to the other. 2. A three-dimensional coordinate system in which the coordinates of a point in space are its distances from each of three perpendicular lines that intersect at an origin. After the Latin form of Descartes, the mathematician who invented it.
Codec (Code-Decode)	A device for converting voice signals from analog to digital for use in digital transmission schemes, normally telephone based, and then converting them back again. Most codecs employ proprietary coding algorithms for data compression, common examples being Dolby's AC-2, ADPCM, and MPEG schemes.
Compander	A contraction of compressor-expander. A term referring to dynamic range reduction and expansion performed by first a compressor acting as an encoder, and second by an expander acting as the decoder. Normally used for noise reduction or headroom reasons.
Complex Frequency Variable	An AC frequency in complex number form.
Complex Number Mathematics	Any number of the form a + bj, where a and b are real numbers and j is an imaginary number whose square equals -1 and a represents the real part (e.g., the resistive effect of a filter, at zero phase angle) and b represents the imaginary part (e.g., the reactive effect, at 90 phase angle).

Compression	1. An increase in density and pressure in a medium, such as air, caused by the passage of a sound wave. 2. The region in which this occurs.
Compression Wave	A wave propagated by means of the compression of a fluid, such as a sound wave in air.
Constant-Q Equalizer (also Constant-Bandwidth)	Term applied to graphic and rotary equalizers describing bandwidth behavior as a function of boost/cut levels. Since Q and bandwidth are inverse sides of the same coin, the terms are fully interchangeable. The bandwidth remains constant for all boost/cut levels. For constant-Q designs, the skirts vary directly proportional to boost/cut amounts. Small boost/cut levels produce narrow skirts and large boost/cut levels produce wide skirts.
Convolution	A mathematical operation producing a function from a certain kind of summation or integral of two other functions. In the time domain, one function may be the input signal, and the other the impulse response. The convolution than yields the result of applying that input to a system with the given impulse response. In DSP, the convolution of a signal with FIR filter coefficients results in the filtering of that signal.
Correlation	A mathematical operation that indicates the degree to which two signals are alike.
Crest Factor	The term used to represent the ratio of the peak (crest) value to the RMS value of a waveform.
Critical Band Physiology of Hearing	A range of frequencies that is integrated (summed together) by the neural system, equivalent to a bandpass filter (auditory filter) with approximately 10-20% bandwidth (approximately one-third octave wide). [Although the latest research says critical bands are more like 1/6-octave above 500 Hz, and about 100 Hz wide below 500 Hz]. The ear can be said to be a series of overlapping critical bands, each responding to a narrow range of frequencies. Introduced by Fletcher (1940) to deal with the masking of a pure-tone by wideband noise.

Cut-Only Equalizer	Term used to describe graphic equalizers designed only for attenuation. (Also referred to as notch equalizers, or band-reject equalizers). The flat (0 dB) position locates all sliders at the top of the front panel. Comprised only of notch filters (normally spaced at 1/3-octave intervals), all controls start at 0 dB and reduce the signal on a band-by-band basis. Proponents of cut-only philosophy argue that boosting runs the risk of reducing system headroom.
Cutoff Frequency Filters	The frequency at which the signal falls off by 3 dB (the half power point) from its maximum value. Also referred to as the -3 dB points, or the corner frequencies.

D

DAC (or D/A, Digital-to-Analog Converter)	The electronic component which converts digital words into analog signals that can then be amplified and used to drive loudspeakers, etc. The DAC is the last link in the digital chain of signal processing.
Decibel Abbreviation. dB	A unit used to express relative difference in power, intensity, voltage or other, between two acoustic or electric signals, equal to ten times (for power ratios - twenty times for all other ratios) the common logarithm of the ratio of the two levels. Equal to one-tenth of a bel.
Delta Modulation	A single-bit coding technique in which a constant step size digitizes the input waveform. Past knowledge of the information permits encoding only the differences between consecutive values.

Delta-Sigma Modulation (also Sigma-Delta)	An analog-to-digital conversion scheme rooted in a design originally proposed in 1946, but not made practical until 1974 by James C. Candy. The name delta-sigma modulation was coined by Inose and Yasuda at the University of Tokyo in 1962, but due to a misunderstanding the words were interchanged and taken to be sigma-delta. Both names are still used for describing this modulator. Characterized by oversampling and digital filtering to achieve high performance at low cost, a delta-sigma A/D thus consists of an analog modulator and a digital filter. The fundamental principle behind the modulator is that of a single-bit A/D converter embedded in an analog negative feedback loop with high open loop gain. The modulator loop oversamples and processes the analog input at a rate much higher than the bandwidth of interest. The modulator's output provides 1-bit information at a very high rate and in a format that a digital filter can process to extract higher resolution (such as 20-bits) at a lower rate.
Digital Audio Data Compression, commonly shortened to "Audio Compression."	Any of several algorithms designed to reduce the number of bits (hence, bandwidth and storage requirements) required for accurate digital audio storage and transmission. Characterized by being "lossless" or "lossy". The audio compression is "lossy" if actual data is lost due to the compression scheme, and "lossless" if it is not. Well designed algorithms ensure "lost" information is inaudible.
Digital Audio	The use of sampling and quantization techniques to store or transmit audio information in binary form. The use of numbers (typically binary) to represent audio signals.
Digital Filter	Any filter accomplished in the digital domain.
Digital Signal	Any signal which is quantized (i.e., limited to a distinct set of values) into digital words at discrete points in time. The accuracy of a digital value is dependent on the number of bits used to represent it.
Digitization	Any conversion of analog information into a digital form.
Discrete Fourier Transform (DFT)	A DSP algorithm used to determine the fourier coefficient corresponding to a set of frequencies, normally linearly spaced.

DSP (Digital Signal Processing)	A technology for signal processing that combines algorithms and fast number-crunching digital hardware and is capable of high-performance and flexibility.

F

FFT (Fast Fourier Transform)	A DSP algorithm that is the computational equivalent to performing a specific number of discrete fourier transforms, but by taking advantage of computational symmetries and redundancies, significantly reduces the computational burden.
FIR (Finite Impulse-Response) Filter	A commonly used type of digital filter. Digitized samples of the audio signal serve as inputs and each filtered output is computed from a weighted sum of a finite number of previous inputs. An FIR filter can be designed to have completely linear phase (i.e., constant time delay, regardless of frequency). FIR filters designed for frequencies much lower than the sample rate and/or with sharp transitions are computationally intensive with large time delays. Popularly used for adaptive filters.
Floating Point	An encoding technique consisting of two parts: 1. A mantissa representing a fractional value with magnitude less than one 2. An exponent providing the position of the decimal point. Floating point arithmetic allows the representation of very large or very small numbers with fewer bits.
Fourier Analysis Mathematics	The approximation of a function through the application of a Fourier Series to periodic data.
Fourier Series	Application of the Fourier theorem to a periodic function, resulting in sine and cosine terms which are harmonics of the periodic frequency. (After Baron Jean Baptiste Joseph Fourier.)
Fourier Theorem	A mathematical theorem stating that any function may be resolved into sine and cosine terms with known amplitudes and phases.

Frequency

1. The property or condition of occurring at frequent intervals.
2. Mathematics. Physics. The number of times a specified phenomenon occurs within a specified interval as
   a) The number of repetitions of a complete sequence of values of a periodic function per unit variation of an independent variable.
   b) The number of complete cycles of a periodic process occurring per unit time.
   c) The number of repetitions per unit time of a complete waveform, as of an electric current.

G

Graphic Equalizer

A multi-band variable equalizer using slide controls as the amplitude adjustable elements. Named for the positions of the sliders “graphing” the resulting frequency response of the equalizer. Only found on active designs. Center frequency and bandwidth are fixed for each band.

H

Harmonic Series

1. Mathematics. A series whose terms are in harmonic progression as 1 + 1/3 + 1/5 + 1/7 +...
2. Music. A series of tones consisting of a fundamental tone and the overtones produced by it and whose frequencies are consecutive integral multiples of the frequency of the fundamental.

High-Pass Filter

A filter having a passband extending from some finite cutoff frequency (not zero) up to infinite frequency. An infrasonic filter is a high-pass filter.

I

IIR (Infinite Impulse-Response) Filter

A commonly used type of digital filter. This recursive structure accepts as inputs digitized samples of the audio signal and then each output point is computed on the basis of a weighted sum of past output (feedback) terms, as well as past input values. An IIR filter is more efficient than its FIR counterpart, but poses more challenging design issues. Its strength is in not requiring as much DSP power as FIR, while its weakness is not having linear group delay and possible instabilities.

Interpolating Response	Term adopted by Rane Corporation to describe the summing response of adjacent bands of variable equalizers using buffered summing stages. If two adjacent bands, when summed together, produce a smooth response without a dip in the center, they are said to interpolate between the fixed center frequencies, or combine well.
Inverse Square Law Sound Pressure Level	Sound propagates in all directions to form a spherical field, thus sound energy is inversely proportional to the square of the distance, i.e., doubling the distance quarters the sound energy (the inverse square law), so SPL is attenuated 6dB for each doubling.
Interleaving	The process of rearranging data in time. Upon de-interleaving, errors in consecutive bits or words are distributed to a wider area to guard against consecutive errors in the storage media.

L

Linear PCM	A pulse code modulation system in which the signal is converted directly to a PCM word without companding, or other processing.
Low-Pass Filter	A filter having a passband extending from DC (zero Hz) to some finite cutoff frequency (not infinite). A filter with a characteristic that allows all frequencies below a specified rolloff frequency to pass and attenuate all frequencies above. Anti-aliasing and anti-imaging filters are low-pass filters.

M

Minimum-Phase Filters	Electrical circuits from an electrical engineering viewpoint, the precise definition of a minimum-phase function is a detailed mathematical concept involving positive real transfer functions, i.e., transfer functions with all zeros restricted to the left half s-plane (complex frequency plane using the Laplace transform operator s). This guarantees unconditional stability in the circuit. For example, all equalizer designs based on 2nd-order bandpass or band-reject networks have minimum-phase characteristics.

MIPS (Million Instructions Processed Per Second)	A measure of computing power.
MLS (Maximum-Length Sequences)	A time-domain-based analyzer using a mathematically designed test signal optimized for sound analysis. The test signal (a maximum-length sequence) is electronically generated and characterized by having a flat energy-vs-frequency curve over a wide frequency range. Sounding similar to white noise, it is actually periodic, with a long repetition rate. Similar in principle to impulse response testing - think of the maximum-length sequence test signal as a series of randomly distributed positive- and negative-going impulses.

N

Narrow-Band Filter	Term popularized by equalizer pioneer C.P. Boner to describe his patented (tapped toroidal inductor) passive notch filters. Boner's filters were very high Q (around 200) and extremely narrow (5 Hz at the -3 dB points). Boner used 100-150 of these sections in series to reduce feedback modes. Today's usage extends this terminology to include all filters narrower than 1/3-octave. This includes parametrics, notch filter sets, and certain cut-only variable equalizer designs.
Noise Shaping	A technique used in oversampling low-bit converters and other quantizers to shift (shape) the frequency range of quantizing error (noise and distortion). The output of a quantizer is fed back through a filter and summed with its input signal. Dither is sometimes used in the process. Oversampling A/D converters shift much of it out of the audio range completely. In this case, the in-band noise is decreased, which allows low-bit converters (such as delta-sigma) to equal or out-perform high-bit converters (those greater than 16 bits). When oversampling is not involved, the noise still appears to decrease by 12dB or more because it is redistributed into less audible frequency areas. The benefits of this kind of noise shaping are usually reversed by further digital processing.

Nyquist Frequency — The highest frequency that may be accurately sampled. The Nyquist frequency is one-half the sampling frequency. For example, the theoretical Nyquist Frequency of a CD system is 22.05 kHz.

O

Octave —
1. Audio. The interval between any two frequencies having a ratio of 2 to 1.
2. Music
   a) The interval of eight diatonic degrees between two tones, one of which has twice as many vibrations per second as the other.
   b) A tone that is eight full tones above or below another given tone.
   c) An organ stop that produces tones an octave above those usually produced by the keys played.

One-Third Octave —
1. Term referring to frequencies spaced every one-third of an octave apart. One-third of an octave represents a frequency 1.26-times above a reference, or 0.794-times below the same reference. The math goes like this: 1/3-octave = 2E1/3 = 1.260 and the reciprocal, 1/1.260 = 0.794. Therefore, for example, a frequency 1/3-octave above a 1kHz reference equals 1.26kHz (which is rounded-off to the ANSI-ISO preferred frequency of "1.25 kHz" for equalizers and analyzers), while a frequency 1/3-octave below 1 kHz equals 794 Hz (labeled "800 Hz"). Mathematically it is significant to note that, to a very close degree, 2E1/3 equals 10E1/10 (1.2599 vs. 1.2589). This bit of natural niceness allows the same frequency divisions to be used to divide and mark an octave into one-thirds and a decade into one-tenths.
2. Term used to express the bandwidth of equalizers and other filters that are 1/3-octave wide at their -3dB (half-power) points.
3. Approximates the smallest region (bandwidth) humans reliably detect change. Compare with third-octave.

Oversampling — A technique where each sample from the converter is sampled more than once, i.e., oversampled. This multiplication of samples permits digital filtering of the signal, thus reducing the need for sharp analog filters to control aliasing.

P

Parametric Equalizer	A multi-band variable equalizer offering control of all the "parameters" of the internal bandpass filter sections. These parameters being amplitude, center frequency and bandwidth. This allows the user not only to control the amplitude of each band, but also to shift the center frequency and to widen or narrow the affected area. Available with rotary and slide controls. Subcategories of parametric equalizers exist which allow control of center frequency but not bandwidth. For rotary control units the most used term is quasi-parametric. For units with slide controls the popular term is paragraphic. The frequency control may be continuously variable or switch selectable in steps. Cut-only parametric equalizers (with adjustable bandwidth or not) are called notch equalizers or band-reject equalizers.
Passive Equalizer	A variable equalizer requiring no power to operate. Consisting only of passive components (inductors, capacitors and resistors) passive equalizers have no AC line cord. Favored for their low noise performance (no active components to generate noise), high dynamic range (no active power supplies to limit voltage swing), extremely good reliability (passive components rarely break), and lack of RFI interference (no semiconductors to detect radio frequencies). Disliked for their cost (inductors are expensive), size (and bulky), weight (and heavy), hum susceptibility (and need careful shielding) and signal loss characteristic (passive equalizers always reduce the signal). Also inductors saturate easily with large low frequency signals, causing distortion. Rarely seen today, but historically they were used primarily for notching in permanent sound systems.
PCM (Pulse Code Modulation)	A conversion method in which digital words in a bit stream represent samples of analog information. The basis of most digital audio systems.
Peaking Response	Term used to describe a bandpass shape when applied to program equalization.

Period
Abbreviation T, t

1. The period of a periodic function is the smallest time interval over which the function repeats itself. (For example, the period of a sine wave is the amount of time T, it takes for the waveform to pass through 360 degrees. Also, it is the reciprocal of the frequency itself, i.e., T = 1/f.)
2. Mathematics.
   a) The least interval in the range of the independent variable of a periodic function of a real variable in which all possible values of the dependent variable are assumed.
   b) A group of digits separated by commas in a written number.
   c) The number of digits that repeat in a repeating decimal. For example, 1/7 = 0.142857142857... has a six-digit period.

Phaser also called a "Phase Shifter,"

This is an electronic device creating an effect similar to flanging, but not as pronounced. Based on phase shift (frequency dependent), rather than true signal delay (frequency independent), the phaser is much easier and cheaper to construct. Using a relatively simple narrow notch filter (all-pass filters also were used) and sweeping it up and down through some frequency range, then summing this output with the original input, creates the desired effect. Narrow notch filters are characterized by having sudden and rather extreme phase shifts just before and just after the deep notch. This generates the needed phase shifts for the ever-changing magnitude cancellations.

Phase Shift

The fraction of a complete cycle elapsed as measured from a specified reference point and expressed as an angle out of phase. In an un-synchronized or un-correlated way.

Phase Delay

A phase-shifted sine wave appears displaced in time from the input waveform. This displacement is called phase delay.

Phasor

1. A complex number expressing the magnitude and phase of a time-varying quantity. It is math shorthand for complex numbers. Unless otherwise specified, it is used only within the context of steady-state alternating linear systems. (Example: 1.5 /27° is a phasor representing a vector with a magnitude of 1.5 and a phase angle of 27 degrees.)
2. For some unknown reason, used a lot by Star Fleet personnel.

Pink Noise	Pink noise is a random noise source characterized by a flat amplitude response per octave band of frequency (or any constant percentage bandwidth), i.e., it has equal energy, or constant power, per octave. Pink noise is created by passing white noise through a filter having a 3 dB/octave roll-off rate. See white noise discussion for details. Due to this roll-off, pink noise sounds less bright and richer in low frequencies than white noise. Since pink noise has the same energy in each 1/3-octave band, it is the preferred sound source for many acoustical measurements due to the critical band concept of human hearing.
Polarity	A signal's electromechanical potential with respect to a reference potential. For example, if a loudspeaker cone moves forward when a positive voltage is applied between its red and black terminals, then it is said to have a positive polarity. A microphone has positive polarity if a positive pressure on its diaphragm results in a positive output voltage.
Pre-Emphasis	A high-frequency boost used during recording, followed by de-emphasis during playback, designed to improve signal-to-noise performance.
Proportional-Q Equalizer (also Variable-Q)	Term applied to graphic and rotary equalizers describing bandwidth behavior as a function of boost/cut levels. The term "proportional-Q" is preferred as being more accurate and less ambiguous than "variable-Q." If nothing else, "variable-Q" suggests the unit allows the user to vary (set) the Q, when no such controls exist. The bandwidth varies inversely proportional to boost (or cut) amounts, being very wide for small boost/cut levels and becoming very narrow for large boost/cut levels. The skirts, however, remain constant for all boost/cut levels.
Psychoacoustics	The scientific study of the perception of sound.
PWM (Pulse Width Modulation)	A conversion method in which the widths of pulses in a pulse train represent the analog information.

Q

Quantization Error	Error resulting from quantizing an analog waveform to a discrete level. In general the longer the word length, the less the error.

Quantization	The process of converting, or digitizing, the almost infinitely variable amplitude of an analog waveform to one of a finite series of discrete levels. Performed by the A/D converter.

R

Real-Time Operation	What is perceived to be instantaneous to a user (or more technically, processing which completes in a specific time allotment).
Reconstruction Filter	A low-pass filter used at the output of digital audio processors (following the DAC) to remove (or at least greatly attenuate) any aliasing products (image spectra present at multiples of the sampling frequency) produced by the use of real-world (non-brickwall) input filters.
Recursive	A data structure that is defined in terms of itself. For example, in mathematics, an expression, such as a polynomial, each term of which is determined by application of a formula to preceding terms. Pertaining to a process that is defined or generated in terms of itself, i.e., its immediate past history.
Rotary Equalizer	A multi-band variable equalizer using rotary controls as the amplitude adjustable elements. Both active and passive designs exist with rotary controls. Center frequency and bandwidth are fixed for each band.

S

Sample Rate Conversion	The process of converting one sample rate to another, e.g. 44.1kHz to 48kHz. Necessary for the communication and synchronization of dissimilar digital audio devices, e.g., digital tape machines to CD mastering machines.
Sample-and-Hold (S/H)	A circuit which captures and holds an analog signal for a finite period of time. The input S/H proceeds the A/D converter, allowing time for conversion. The output S/H follows the D/A converter, smoothing glitches.
Sampling (Nyquist)Theorem	A theorem stating that a bandlimited continuous waveform may be represented by a series of discrete samples if the sampling frequency is at least twice the highest frequency contained in the waveform.

Sampling Frequency or Sampling Rate	The frequency or rate at which an analog signal is sampled or converted into digital data. Expressed in Hertz (cycles per second). For example, compact disc sampling rate is 44,100 samples per second or 44.1kHz, however in pro audio other rates exist, common examples being 32kHz, 48kHz and 50kHz.
Sampling	The process of representing the amplitude of a signal at a particular point in time.
S/N ratio (Signal-to-Noise ratio)	The ratio of signal level (or power) to noise level (or power), normally expressed in decibels.

T

Third-Octave	Term referring to frequencies spaced every three octaves apart. For example, the third-octave above 1kHz is 8kHz. Commonly misused to mean one-third octave. While it can be argued that "third" can also mean one of three equal parts and as such might be used to correctly describe one part of an octave spit into three equal parts, it is potentially too confusing. The preferred term is one-third octave.
Transversal Equalizer	A multi-band variable equalizer using a tapped audio delay line as the frequency selective element, as opposed to bandpass filters built from inductors (real or synthetic) and capacitors. The term "transversal filter" does not mean "digital filter". It is the entire family of filter functions done by means of a tapped delay line. There exists a class of digital filters realized as transversal filters, using a shift register rather than an analog delay line, with the inputs being numbers rather than analog functions.

W

Wavelength Symbol (Greek lower-case Lambda)	The distance between one peak or crest of a sine wave and the next corresponding peak or crest. The wavelength of any frequency may be found by dividing the speed of sound by the frequency.

White Noise

Analogous to white light containing equal amounts of all visible frequencies, white noise contains equal amounts of all audible frequencies (technically the bandwidth of noise is infinite, but for audio purposes it is limited to just the audio frequencies). From an energy standpoint white noise has constant power per hertz (also referred to as unit bandwidth), i.e., at every frequency there is the same amount of power (while pink noise, for instance, has constant power per octave band of frequency). A plot of white noise power vs. frequency is flat if the measuring device uses the same width filter for all measurements. This is known as a fixed bandwidth filter. For instance, a fixed bandwidth of 5 Hz is common, i.e., the test equipment measures the amplitude at each frequency using a filter that is 5 Hz wide. It is 5 Hz wide when measuring 50 Hz or 2 kHz or 9.4 kHz, etc. A plot of white noise power vs. frequency change is not flat if the measuring device uses a variable width filter. This is known as a fixed percentage bandwidth filter. A common example of which is 1/3-octave wide, which equals a bandwidth of 23%. This means that for every frequency measured the bandwidth of the measuring filter changes to 23% of that new center frequency. For example the measuring bandwidth at 100 Hz is 23 Hz wide, then changes to 230 Hz wide when measuring 1 kHz, and so on. Therefore the plot of noise power vs. frequency is not flat, but shows a 3 dB rise in amplitude per octave of frequency change. Due to this rising frequency characteristic, white noise sounds very bright and lacking in low frequencies.

Z

Z-Transform

A mathematical method used to relate coefficients of a digital filter to its frequency response, and to evaluate stability of the filter. It is equivalent to the Laplace transform of sampled data and is the building block of digital filters.

A

B

C

M

O

R

S

T

V

Infineon goes for Business Excellence

“Business excellence means intelligent approaches and clearly defined processes, which are both constantly under review and ultimately lead to good operating results.
Better operating results and business excellence mean less idleness and wastefulness for all of us, more professional success, more accurate information, a better overview and, thereby, less frustration and more satisfaction.”

Dr. Ulrich Schumacher

h t t p : / / w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG