User's Manual for Keil Compiler, V 1.2, November 2003

XC166Lib

A DSP Library for XC16x Family

Microcontrollers

Never stop thinking.

Edition 2003-11

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.

User's Manual for Keil Compiler, V 1.2, November 2003

XC166Lib

A DSP Library for XC16x Family

Guangyu Wang

AI MC MA TM (Munich, Germany)

Microcontrollers

N e v e r   s t o p   t h i n k i n g .

Revision History:2003-11V 1.2

Previous Version: 2002-02 V 1.0, 2002-09 V1.1	
Page	Subjects (major changes since last revision)

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:

C166Lib-support@infineon.com

Table of Contents

Page

1 Introduction . . . 14
1.1 Introduction to XC166Lib, a DSP Library for XC16x . . . 14
1.2 Features . . . 14
1.3 Future of the XC166Lib . . . 14
1.4 Support Information . . . 14

2 Installation and Build . . . 16
2.1 XC166Lib Content . . . 16
2.2 Installing XC166Lib . . . 16
2.3 Building XC166Lib . . . 16
2.4 Source Files List . . . 17

3 DSP Library Notations . . . 19
3.1 XC166Lib Data Types . . . 19
3.2 Calling a DSP Library Function from C Code . . . 19
3.3 XC166Lib Example Implementation . . . 19
3.4 XC166Lib Implementation - A Technical Note for Keil Compiler . . . 19
3.4.1 Memory Issues . . . 19
3.4.2 Memory Models for Keil C Compiler . . . 20
3.4.3 Optimization Techniques . . . 21
3.4.4 Cycle Count . . . 24
3.4.5 Testing Methodology . . . 24

4 Function Descriptions . . . 25
4.1 Conventions . . . 25
4.1.1 Argument Conventions . . . 25
4.2 Arithmetic Functions . . . 27
4.2.1 Complex Numbers . . . 27
4.2.2 Complex Number Representation . . . 27
4.2.3 Complex Plane . . . 27
4.2.4 Complex Arithmetic . . . 28
4.2.5 Complex Number Schematic . . . 29
4.2.6 Descriptions . . . 29
4.3 FIR Filters . . . 40
4.3.1 Transversal Form . . . 40
4.3.2 Lattice Form . . . 41
4.3.3 Multirate FIR Filters . . . 42
4.3.4 Descriptions . . . 44
4.4 IIR Filters . . . 73
4.4.1 Direct Form 1 . . . 73
4.4.2 Direct Form 2 . . . 73
4.4.3 Cascaded Biquad IIR Filter with Direct Form 2 . . . 75
4.4.4 Cascaded Biquad IIR Filter with Direct Form 1 . . . 75
4.4.5 Lattice IIR Filter . . . 76

4.4.6 Descriptions 77
4.5 Adaptive Digital Filters 111
4.5.1 Delayed LMS algorithm for an adaptive filter 112
4.5.2 Descriptions 113
4.6 Fast Fourier Transforms 123
4.6.1 Radix-2 Decimation-In-Time FFT Algorithm 126
4.6.2 Radix-2 Decimation-In-Frequency FFT Algorithm 128
4.6.3 Complex FFT Algorithm 130
4.6.4 Calculation of Real Forward FFT from Complex FFT 133
4.6.5 Calculation of Real Inverse FFT from Complex FFT 134
4.7 C166S V2 Core Implementation Note 136
4.7.1 Organization of FFT functions 136
4.7.2 Implementation of Real Forward FFT 136
4.7.3 Implementation of Real Inverse FFT 138
4.7.4 Description 138
4.8 Matrix Operations 159
4.8.1 Descriptions 159
4.9 Mathematical Operations 168
4.10 Statistical Functions 187
4.10.1 Correlation 187
4.10.2 Implementation Description 190
4.10.3 Convolution 212

5 References 216

List of Figures

Page

Figure 4-1 The Complex Plane 28
Figure 4-2 16-bit Complex number representation 29
Figure 4-3 Complex Number addition for 16 bits. 31
Figure 4-4 Complex number subtraction for 16 bits 34
Figure 4-5 Complex number multiplication for 16 bits 36
Figure 4-6 32 bit real number multiplication 39
Figure 4-7 Block Diagram of the FIR Filter 41
Figure 4-8 Lattice FIR Filter. 42
Figure 4-9 Block Diagram of FIR Decimation Filter 43
Figure 4-10 Equivelent Implementation of FIR Decimation Filter 43
Figure 4-11 Block Diagram of FIR Interpolation Filter 44
Figure 4-12 Equivelent Implementation of FIR Interpolation Filter 44
Figure 4-13 Fir_16. 47
Figure 4-14 Fir_32. 51
Figure 4-15 Fir_cplx 55
Figure 4-16 Fir_sym 59
Figure 4-17 Fir_lat. 63
Figure 4-18 Fir_dec. 67
Figure 4-19 Fir_inter 71
Figure 4-20 Canonical Form (Direct Form 2) 74
Figure 4-21 Cascaded Biquad IIR Filter with Direct Form 2 Implementation 75
Figure 4-22 Cascaded Biquad IIR Filter with Direct Form 1 Implementation 76
Figure 4-23 Lattice IIR Filter Structure 77
Figure 4-24 IIR_1 81
Figure 4-25 IIR_2 85
Figure 4-26 IIR_bi_1 90
Figure 4-27 IIR_bi_2 95
Figure 4-28 IIR_bi2_32 100
Figure 4-29 IIR_bi_24 105
Figure 4-30 IIR_lat. 109
Figure 4-31 Adaptive filter with LMS algorithm 111
Figure 4-32 Adap_filter_16 116
Figure 4-33 Adap_filter_32 120
Figure 4-34 Complexity Graph 124
Figure 4-35 8-point DIT FFT 126
Figure 4-36 Alternate Form of 8-point DIT FFT. 127
Figure 4-37 8-point DIF FFT 128
Figure 4-38 Alternate Form of 8-point DIF FFT. 129
Figure 4-39 8-point DIT complex FFT 130
Figure 4-40 Butterfly of Radix-2 DIT complex FFT 130
Figure 4-41 8-point DIF complex FFT 131
Figure 4-42 Butterfly of Radix-2 DIF complex FFT 132
Figure 4-43 Bit_reverse. 139

Figure 4-44 FloatTo1Q15 142
Figure 4-45 Real_DIT_FFT 146
Figure 4-46 Real_DIF_IFFT 152
Figure 4-47 Q15toFloat 155
Figure 4-48 Matrix_mul 160
Figure 4-49 Matrix_trans 164
Figure 4-50 Sine 169
Figure 4-51 P_series 172
Figure 4-52 Windowing 175
Figure 4-53 Division of two 1Q15 fractional inputs 178
Figure 4-54 Division of 1Q31 fractional by 1Q15 fractional 180
Figure 4-55 Raw autocorrelation 190
Figure 4-56 Biased autocorrelation 193
Figure 4-57 Unbiased autocorrelation 196
Figure 4-58 Raw cross-correlation 200
Figure 4-59 Biased cross-correlation 204
Figure 4-60 Unbiased cross-correlation 208
Figure 4-61 Convolution 212

List of Tables

Page

Table 2-1	Directory Structure	16
Table 2-2	Source files	17
Table 3-1	XC166Lib Data Types	19
Table 4-1	Argument Conventions	25

Preface

This is the User Manual of version 1.2 for XC166Lib - a DSP library for Infineon XC16x microcontroller family. To make the user easy by using the XC166Lib we provide a separate User Manual for each compiler to describe the implementations. **This User Manual describes the implementations for Keil compiler**.

XC16x microcontroller family is the most recent generation of the popular C166 microcontroller families. The core of XC16x family is C166S V2 Core that combines high performance with enhanced modular architecture. Impressive DSP performance and advanced interrupt handling with fast context switching make XC16x the instrument of choice for powerful applications.

This manual describes the implementation of essential algorithms for general digital signal processing applications on the XC16x using Keil compiler.

For Keil compiler the DSP library is developed mainly in **inline** assembly and most of the source code files is stored in c files. The DSP library can be used as a library of basic functions for developing bigger DSP applications on XC16x microcontroller. The library serves as a user guide and a reference for XC16x microcontroller DSP programmers. It demonstrates how the processor's architecture can be exploited for achieving high performance.

The various functions and algorithms implemented and described are:

- Arithmetic Functions
- Filters
– FIR
– IIR
– Adaptive filters
- Transforms
– FFT
– IFFT
- Matrix Operations
- Mathematical Operations
- Statistical Functions
– Auto-correlation
– Cross-correlation
– Convolution

Each function is described in detail under the following headings:

Signature:

This gives the function interface.

Inputs:

Lists the inputs to the function.

Outputs:

Lists the output of the function if any.

Return:

Gives the return value of the function if any.

Implementation 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.

Register Usage:

Lists the registers that are used for parameter transfer from C to Assembly or inverse.

Assumptions:

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

Memory Note:

A detailed sketch showing how the arrays are stored in memory, the alignment requirements of the different memories, the nature of the arithmetic performed on them. The diagrams give a great insight into the actual implementation.

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 XC166Lib and its features.

Chapter 2, Installation and Build, describes the XC166Lib content, how to install and build the XC166Lib.

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 arithmetic functions, FIR filters, IIR filters, Adaptive filters, Fast Fourier Transforms, Matrix operations and Mathematical operations. Each function is described with its signature, inputs, outputs, return, implementation description, pseudocode, techniques used, registers used for parameter transfer, assumptions made, memory note, example, cycle count and code size.

Chapter 5, gives the list of related references.

Acknowledgements

All source codes in XC166Lib have been designed, developed and tested using Tasking and Keil Tool chains. We in advance would like to acknowledge users for their feedback and suggestions to improve this product.

Guangyu Wang

XC166Lib Developer - Infineon

Acronyms and Definitions

Acronyms	Definitions
DIT	Decimation-In-Time
DIF	Decimation-In-Frequency
LMS	Least Mean Square
DSP	Digital Signal Processing
XC166Lib	DSP Library functions for XC16x
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
Times-italic	File name

1 Introduction

1.1 Introduction to XC166Lib, a DSP Library for XC16x

The XC166Lib, a DSP Library for XC16x microcontroller is C-callable, hand-coded assembly, general purpose signal processing routines. The XC166Lib includes commonly used DSP routines. The throughput of the system using the XC166Lib routines is considerably better than those achieved using the equivalent code written in ANSI C language. The XC166Lib significantly helps in understanding the general purpose signal processing routines, its implementation on XC16x microcontroller family. It also reduces the DSP application development time. Furthermore, The XC166Lib is also a good reference for XC16x microcontroller DSP programmer.

The routines are broadly classified into the following functional categories:

- Arithmetic functions
- FIR filters
- IIR filters
- Adaptive filters
- Fast Fourier Transforms
- Matrix operations
- Mathematical operations
- Statistical functions

1.2 Features

- Common DSP algorithms with source codes
- Hand-coded and optimized assembly modules with CoMAC instructions
- C-callable functions on Keil compiler
- Multi platform support - Win 95, Win 98, Win NT
- Examples to demonstrate the usage of functions
- Example input test vectors and the output test data for verification
- Complete User's manual covering many aspects of implementation

1.3 Future of the XC166Lib

The planned future releases will have the following improvements.

- Expansion of the library by adding more functions in the domain of 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

C166Lib-support@infineon.com.

Visit ***www.infineon.com /C166DSPLIB*** for update on XC166Lib releases.

2 Installation and Build

2.1 XC166Lib Content

The following table depicts the XC166Lib content with its directory structure.

Table 2-1 Directory Structure

Directory name	Contents	Files
XC166Lib	Directories which has all the files related to the XC166Lib	None
Tasking	Directories which has all the files related to the XC166Lib for Tasking compiler	None
Keil	Directories which has all the files related to the XC166Lib for Keil compiler	None
Source	Directories of source files. Each directory has respective assembly language implementation files of the library functions	*.c *.asm *.a66
Include	Directory and common include files for 'C' of the Keil compiler	DspLib_Keil.h
Docs	User's Manual for Keil compiler	*.pdf
Examples	Example directories . Each directory contains example "c" functions to depict the usage of XC166Lib.	*.c

2.2 Installing XC166Lib

XC166Lib is distributed as a ZIP file. To install the XC166Lib on the system, unzip the ZIP file and extract them to the defined directory.

The directory structure is as given in **"XC166Lib Content" on Page 2-16**

2.3 Building XC166Lib

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

```
#include "DspLib_Keil.h"
```

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

Complier:	**Keil**
Arithmetic Functions	
CplxAdd_16.c *CplxSub_16.c* *CplxMul_16.c* *Mul_32.c*	
FIR Filters	
Fir_16.c *Fir_32.c* *Fir_cplx.c* *Fir_dec.c* *Fir_inter.c* *Fir_lat.c* *Fir_sym.c*	
IIR Filters	
IIR_1.c *IIR_2.c* *IIR_bi_1.c* *IIR_bi_2.c* *IIR_bi_24.c* *IIR_bi2_32.c* *IIR_lat.c*	
Adaptive Filters	
Adap_filter_16.c *Adap_filter_32.c*	
FFT	
Bit_reverse.c *real_DIT_FFT.a66* *real_DIF_IFFT.a66*	
Matrix Operations	

Table 2-2 Source files

Matrix_mul.c *Matrix_trans.c*
Mathematical Operations
Power_series.c *Windowing.c* *Sine.c* *div_q15.c* *div_q31.c* *Sqrt.c*
Statistical Functions
Auto_raw.c *Auto_bias.c* *Auto_unbias.c* *Cross_raw.c* *Cross_bias.c* *Cross_unbias.c* *Convol.c*
Miscellaneous
FloatTo1Q15.c *Q15toFloat.c*

3 DSP Library Notations

3.1 XC166Lib Data Types

The XC166Lib handles the following fractional data types.

Table 3-1 XC166Lib Data Types

1Q15 (DataS)	1Q15 operand is represented by a short data type that is predefined as DataS in header files *DspLib_Keil.h*.
1Q31 (DataL)	1Q31 operand is represented by a long data type that is predefined as DataL in header files *DspLib_Keil.h*.
CplxS	Complex data type that contains the two 1Q15 data arranged in Re-Im format.
CplxL	Complex data type that contains the two 1Q31 data arranged in Re-Im format.

3.2 Calling a DSP Library Function from C Code

After installing the XC166Lib, include a XC166Lib function in the source code as follows:

1. Choose the memory model for C compiler
2. Include the header file *DspLib_Keil.h*
3. Include the source file that contains required DSP functions into the project along with the other source files
4. Include the Keil compiler system files TRAPS.C and START_V2.A66
5. Set the Options for Target - Device to select an MCU with XC16x
6. Build the system

3.3 XC166Lib Example Implementation

The examples of how to use the XC166Lib functions are implemented and are placed in *examples* subdirectory. This subdirectory contains a subdirectory for set of functions.

3.4 XC166Lib Implementation - A Technical Note for Keil Compiler

3.4.1 Memory Issues

The XC16x microcontroller family uses the C166S V2 Core that is a 16 bit microcontroller core, with impressive DSP performance. There are two sets of instructions for C166S V2 Core, namely Normal Instruction Set and DSP Instruction Set (MAC-instructions). Normal instruction set is compatible with the microcontroller family C166, while the DSP instruction set is especially designed for implementing DSP algorithms. XC166Lib was developed mainly using DSP instruction set. But the normal instruction set has been also often used in the routines, in particular, for initializing memories and registers.

For each instruction set there is a different addressing mode. DSP instructions use some standard C166 addressing modes such as GPR direct and $\#data_5$ for immediate shift value. To supply the MAC instructions with up to 2 new operands in one CPU cycle, new MAC instruction addressing models have been added in C166S V2 Core. These allow indirect addressing with address pointer post-modification. Double indirect addressing requires 2 pointers, one of which can be supplied by any GPR, the other is provided by one of two Specific Function Registers (SFRs) IDX0 and IDX1. Two pairs of offset registers QR0/QR1 and QX0/QX1 are associated with each pointer (GPR or IDX_i). The GPR pointer gives access to the entire memory space, whereas IDX_i are limited to the internal **Dual-Port RAM** (DPRAM), except for the CoMOV instruction.

The XC166Lib is implemented with the C166S V2 Core memory addressing architecture. The following information gives memory conditions in order to work properly.

Because the specific function register IDX_i is limited to the internal DPRAM, in order to use MAC instructions properly we must first initialize IDX_i with the address pointed to DPRAM space from $00'F200_H$ to $00'FE00_H$ (3KBytes) before using MAC instructions. This means that we must locate one of operands in MAC-instructions with double indirect addressing modes in range from $00'F200_H$ to $00'FE00_H$. Using Keil complier we can easily realize it through defining the variables which should be located in the DPRAM area as on-chip memory type ***idata***, e.g.

short idata x[n] ;

After compiling the vector x will be automatically located in the DPRAM area because Keil compiler locates all variables with the memory type *idata* in the DPRAM area.

Note that C166S V2 Core has defined 3 KBytes of DPRAM. However, the XC16x microcontroller family is equipped with only 2 KBytes of DPRAM in the range from $00'F600_H$ to $00'FE00_H$. The limited DPRAM area can make difficulty by executing DSP algorithms on XC16x, if the size of the vector is larger than 2 KBytes, e.g. a 1024-point Fir filter.

When using pointer post-modification addressing models, the address pointed to must be a legal address, even if its content is not modified. An odd value will trigger the class-B hardware Trap (Illegal Word Operand Access Trap (ILLOPA)).

3.4.2 Memory Models for Keil C Compiler

Just as we said, the DSP library is developed mainly for calling from a C program. Therefore, the memory modes selected by C and assembly modules must be same in order to avoid memory model conflicts. Keil tool chain supports seven memory models: tiny, small, compact, hcompact, medium, large and hlarge. The basic differences between them are:

Beside the tiny model operating in the non-segmented CPU mode the other six memory models operate in the segmented CPU mode. The variables by the tiny and small memory models are located in the **near** area and the functions calls generate near calls (up to 64Kb code size). The both memory modes provide the same efficient code. The difference of tiny and small memory models is that the code and data in the tiny model are limited to the first segment of 64K, while the small model allows code and data to locate anywhere in the space.

The compact, hcompact and medium models operate as small model does, except that the compact model uses the memory type **far** for variables, the hcompact model uses the memory type **huge** for variables and the medium model generates function calls as **far** call.

For both large and hlarge models the function calls are generated as far calls by default. The difference between them is that the large memory locates the variables in the **far** memory, while the hlarge locates the variables in the **huge** memory.

Because the most functions in the library are written in the inline assembly, for those library functions the parameter passing will be done by the compiler automatically. So the library functions written in the inline assembly can be used in all memory models without regarding the memory types of pointers passed from C to assembly code. But for the library functions which are pure assembly module and stored in assembly file *.a66, if we want to use them in the memory models with far or huge variable memory types, we need to redefine the pointers in the function arguments as near pointers, because the library functions use only 16 bit pointers by parameter passing from C to assembly routine.

There are only 16 registers R0-R15 on XC16x that can be used for programming. By calling the XC166Lib routines from C, according to Keil Compiler conventions the registers from R8 to R12 will be used to translate parameters from C to the assembly code, and the rest parameters will be translated through using stack. R4 and R5 are used to store the output values after executing the XC166Lib routines. R1-R7 can be used in the assembly routine without regard for preserving their contents.

3.4.3 Optimization Techniques

DSP optimization techniques depend strongly on the core architecture. So, different cores have different optimization techniques. Furthermore, the number of tricks that can be played by a DSP programmer are endless. In the development of XC166Lib the following optimization techniques have been used.

- *data dependencies removing*

Due to the pipeline requirement of the C166S V2 CPU there are a lot of possible data dependencies between instructions using GPRs. In the C166S V2 Core the dedicated hardware is added to detect and resolve the data dependencies. However, in the C166S V2 Core none of the instructions using indirect addessing modes are capable of using a GPR, which is to be updated by one of the two immediately preceding instructions. This means that the instruction using indirect addressing modes will lead two cycles stall. To use these two cycles for the optimization we can insert before this instruction a multicycle or two single cycle instructions that must not update the GPR used for indirect addressing.

Example:

Assembly without optimization (6 cycles)

```
............
ADD     R1, R2
MOV     R8, [R1]    ; instruction using indirect addressing mode
ADD     R5, R1
ADD     R6, R1
............
```

Assembly with optimization (4 cycles)

```
............
ADD     R1, R2
ADD     R5, R1      ; inserted one cycle instruction
ADD     R6, R1      ; inserted one cycle instruction
MOV     R8, [R1]    ; instruction using indirect addressing mode
............
```

- *memory bandwidth conflicts removing*

Memory bandwidth conflicts can occur if instructions in the pipeline access the same memory ares at the same time. The CoXXX instructions are specially designed for DSP implementation. To avoid the memory bandwidth conflicts in the DPRAM ares, one of the operands should be located in the internal SRAM to guarantee a single cycle execution time of the CoXXX instructions.

- *instruction re-ordering*

By writing DSP routines with CoXXX instructions it is often needed to change and update the Special Function Registers (SFRs), such as IDX0, IDX1, QX0, QX1, QR0, QR1, and so on. CPU-SFRs control the CPU functionality and behavior. Therefore, special care is required to ensure that instructions in the pipeline always work with the correct SFRs values. With instruction re-ordering the flow of instructions through the pipeline can be improved to optimize the routines.

Example:

Assembly code without optimization (7 cycles)

```
............
```

```
EXTR    #1
MOV     IDX1, #12      ; initialize IDX1 with 12
CoMUL   [IDX1],  [R1]
MOV     R6, R1
ADD     R2, R1
............
```

Assembly code with optimization (5 cycles)

```
............
EXTR    #1
MOV     IDX1, #12      ; initialize IDX1 with 12
MOV     R6, R1         ; instruction re-ordering
ADD     R2, R1         ; instruction re-ordering
CoMUL   [IDX1],  [R1]
............
```

- *loop unrolling*

  The equation is written twice or more inside a loop.

Example (unrolling factor 2):

Assembly code without loop unrolling (17 cycles)

```
        ...........
        MOV       R3, #3
loop:
        CoMAC     [IDX0+], [R1+]
        CoMAC     [IDX0+], [R1+]
        CMPD1     R3,#0h
        JMPR      cc_NZ,loop
        ...........
```

Assembly code with loop unrolling (13 cycles)

```
        ...........
        MOV       R3, #1
loop:
        CoMAC     [IDX0+], [R1+]
        CoMAC     [IDX0+], [R1+]
        CoMAC     [IDX0+], [R1+]
```

```
CoMAC       [IDX0+], [R1+]

CMPD1       R3,#0h
JMPR        cc_NZ,loop
............
```

3.4.4 Cycle Count

The cycle count given for each function in this User's Manual represents the cycles to be needed for executing the assembly instructions in the function. They have been verified on XC16x boards. To some degree one can understand as the theoretical cycle count. The given values of cycle count can only be achieved only on conditions that all data and assembly codes are located in the internal memory area and no pipeline conflicts occur in the program. Note that the real cycle count may be much larger than the given values, if the data or source code are located in the external memory.

3.4.5 Testing Methodology

The XC166Lib is tested on XC16x board. The test is believed to be accurate and reliable. However, the developer assumes no responsibility for the consequences of use of the XC166Lib.

4 Function Descriptions

Each function is described with its signature, inputs, outputs, return, implementation description, pseudocode, techniques used, assumptions made, register usage, memory note, example, cycle count and code size.

Functions are classified into the following categories.

- Arithmetic functions
- FIR filters
- IIR filters
- Adaptive filters
- Fast Fourier Transforms
- Matrix operations
- Mathematical 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, x	Input data or input data vector beside in FFT functions, where X representing the FFT spectra of the input vector x
Y, y	Output data or output data vector
D_buffer, d_buffer	Delay buffer
N_x	The size of input vectors
H, h	Filter coefficient vector
N_h	The size of coefficient vector H
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

Table 4-1 Argument Conventions

Argument	Convention
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

4.2 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

To geometrically represent complex numbers as points in the plane two perpendicular coordinate axis in the Cartesian coordinate system are chosen. 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 + 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.

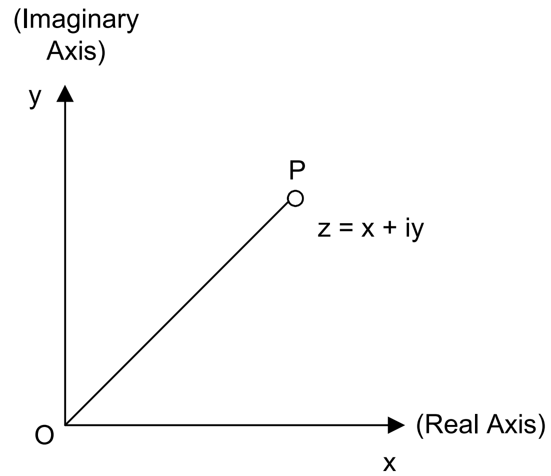

Figure 4-1 The Complex Plane

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 $z_2 = x_2 + iy_2$,

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

Subtraction

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

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

Multiplication

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

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

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

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.

4.2.5 Complex Number Schematic

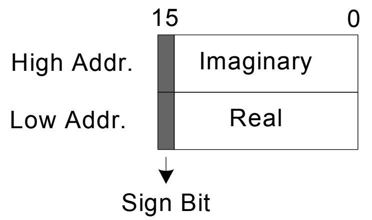

Figure 4-2 16-bit Complex number representation

4.2.6 Descriptions

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

- 16 bit complex addition
- 16 bit complex subtraction
- 16 bit complex multiplication
- 32 bit real multiplication

CplxAdd_16 16 bit Complex Number Addition

Signature void CplxAdd_16 (CplxS* X, CplxS* Y, ClpxS* R)

Inputs

X	:	Pointer to 16 bit Complex input value in 1Q15 format
Y	:	Pointer to 16 bit Complex input value in 1Q15 format

Output None

Return Pointer to the sum of two complex numbers as a 16 bit complex number in 1Q15 format

Implementation 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

$$\begin{aligned} R_r &= x_r + y_r \\ R_i &= x_i + y_i \end{aligned} \qquad [4.9]$$

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

Assumption

Register Usage

- From .c file to inline assembly file:
  Decided by compiler

CplxAdd_16 16 bit Complex Number Addition (cont'd)

Memory Note

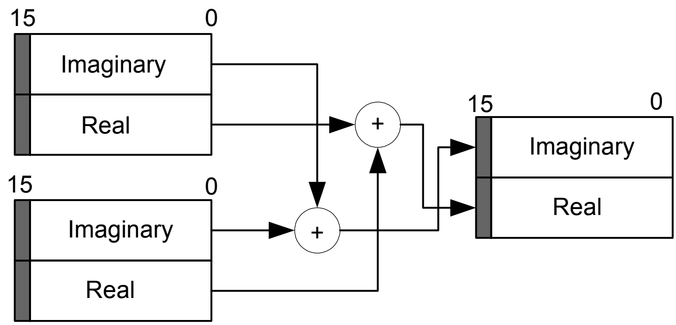

Figure 4-3 Complex Number addition for 16 bits

Example *C166Lib\Keil\Examples\Arith_16\Arith_16.c*

Cycle Count

Initialization and read input values	6
Real Addition	3
Imaginary Addition	3
Total	**12**

Code Size

Initialization and read input values	12 bytes
Real Addition	10 bytes

CplxAdd_16 **16 bit Complex Number Addition** (cont'd)

Imaginary Addition	10 bytes
Total	**32** bytes

CplxSub_16 16 bit Complex Number Subtraction

Signature void CplxSub_16 (CplxS* X, CplxS* Y, CplxS* R)

Inputs

X : Pointer to 16 bit complex input value in 1Q15 format

Y : Pointer to 16 bit complex input value in 1Q15 format

Output None

Return Pointer to the difference of two complex numbers as a 16 bit complex number

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

$$R_r = x_r - y_r$$
$$R_i = x_i - y_i \qquad [4.10]$$

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

Assumption

Register Usage

- From .c file to inline assembly file:
  Decided by compiler

CplxSub_16 16 bit Complex Number Subtraction (cont'd)

Memory Note

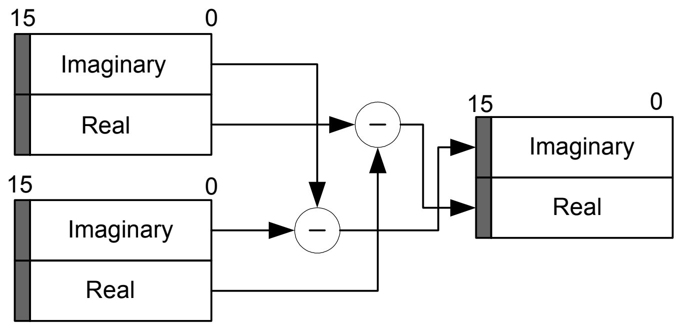

Figure 4-4 Complex number subtraction for 16 bits

Example *C166Lib\Keil\Examples\Arith_16\Arith_16.c*

Cycle Count

Initialization and read input values	6
Real Subtraction	3
Imaginary Subtraction	3
Total	**12**

Code Size

Initialization and read input values	12 bytes
Real Subtraction	10 bytes
Imaginary Subtraction	10 bytes
Total	**32** bytes

CplxMul_16 — 16 bit Complex Number Multiplication

Signature void CplxMul_16 (CplxS* X, CplxS* Y, CplxS* R)

Inputs

X	:	Pointer to 16 bit Complex input value in 1Q15 format
Y	:	Pointer to 16 bit Complex input value in 1Q15 format

Return Pointer to the multiplication result in 1Q15 format

Implementation 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

Assumption

Register Usage

- From .c file to inline assembly file:
  Decided by compiler

CplxMul_16 16 bit Complex Number Multiplication (cont'd)

Memory Note

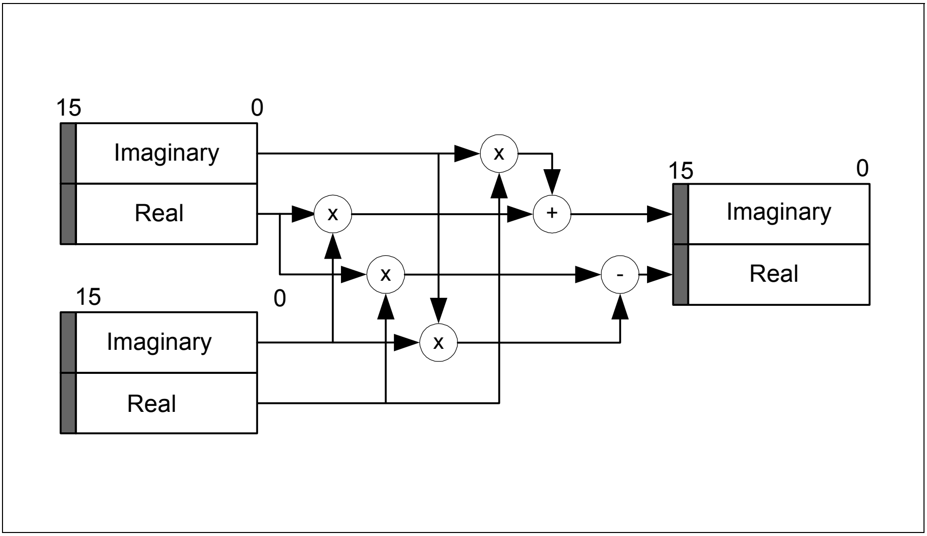

Figure 4-5 Complex number multiplication for 16 bits

Example *C166Lib\Keil\Examples\Arith_16\Arith_16.c*

Cycle Count

Initialization and read input values	5
Real multiplication	3
Imaginary multiplication	3
Total	**11**

Code Size

Initialization and read input values	10 bytes
Real multiplication	12 bytes

CplxMul_16 **16 bit Complex Number Multiplication** (cont'd)

Imaginary multiplication	12 bytes
Total	**34** bytes

Mul_32 32 bit Real Multiplication

Signature DataL Mul_32 (DataL X, DataL Y)

Inputs

X	:	32 bit real input value in 1Q31
Y	:	32 bit real input value in 1Q31

Return Multiplication result in 1Q31 format

Implementation Description This function computes the product of the two 32 bit real numbers. Wraps around the result in case of overflow. The multiplication is computed as follows.

$$R = x_L \times y_L + x_L \times y_H + (x_H \times y_L + x_H \times y_H) \gg 16$$

Pseudo code

```
{
   R = xL*yL + xL*yH + (xH*yL + xH*yH)>>16 ;
}
```

Techniques None

Assumption

Register Usage

- From .c file to inline assembly file:
  Decided by compiler.
- From .asm file to .c file:
  R_L(LSW) is stored in R4.
  R_H(MSW) is stored in R5.

Mul_32 **32 bit Real Multiplication** (cont'd)

Memory Note

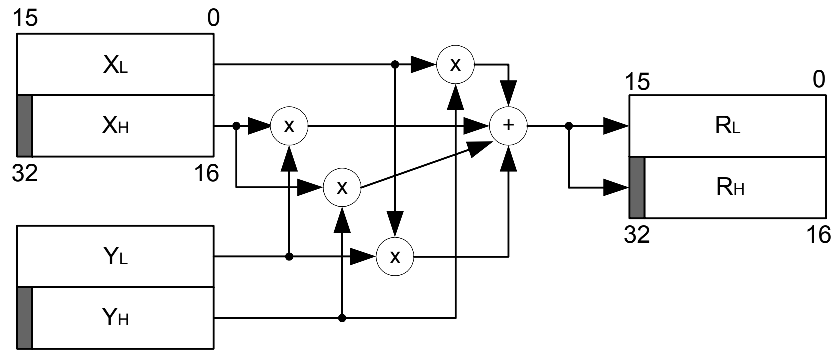

Figure 4-6 32 bit real number multiplication

Example *C166Lib\Keil\Examples\Arith_16\Arith_16.c*

Cycle Count

Initialization	5	
Multiplication	8	
Output	2	
Total	**15**	

Code Size

Initialization	10	bytes
Multiplication	32	bytes
Output	8	
Total	**50**	bytes

4.3 FIR Filters

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 realization of an FIR filter has many forms. But the transversal and lattice forms are most useful structures in practice.

4.3.1 Transversal Form

The transversal form of an Fir filter showed in **Figure 4-7** is realized by a tapped delay line. The 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 N tap FIR filter, the difference equation is

$$y(n) = \sum_{i=0}^{N-1} h_i \cdot x(n-i) \qquad [4.11]$$

where,

x(n)	:	the filter input for n^{th} sample
y(n)	:	output of the filter for n^{th} sample
h_i	:	filter coefficients
N	:	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

$$H[z] = \frac{Y[z]}{X[z]} = \sum_{i=0}^{N-1} h_i \cdot z^{-i} \qquad [4.12]$$

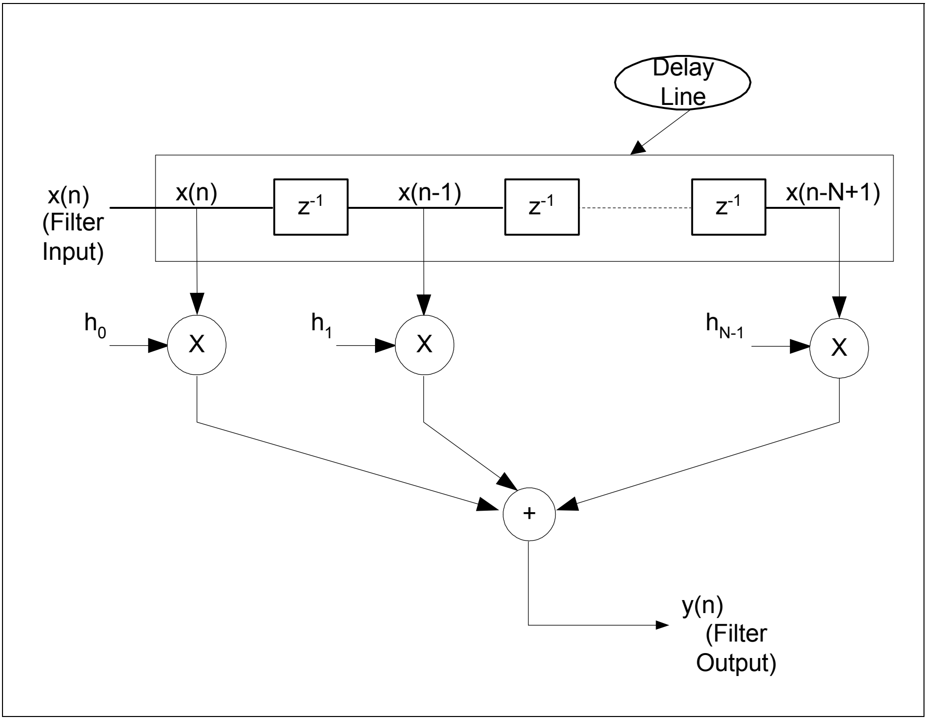

Figure 4-7 Block Diagram of the FIR Filter

4.3.2 Lattice Form

The structure of a lattice FIR filter is showed in **Figure 4-8** . Each stage of the filter has an input and output that are related by the equations:

$$\begin{aligned} y_i(n) &= y_{i-1}(n) + k_i u_i(n-1) \\ u_i(n) &= k_i y_{i-1}(n) + u_{i-1}(n-1) \end{aligned} \qquad 1 \le i \le M \qquad [4.13]$$

The initial values are equal to the filter input x(n):

$$y_0(n) = x(n)$$
$$u_0(n) = x(n)$$

At the last stage we have the output of the lattice FIR filter $y(n)=y_M(n)$.

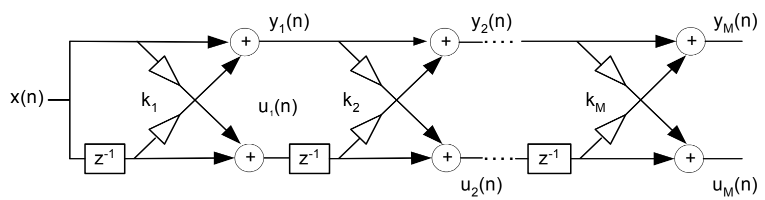

Figure 4-8 Lattice FIR Filter

4.3.3 Multirate FIR Filters

Multirate filters are a kind of digital filters that change the sampling rate of a digital signal. A multirate filter converts a digital signal with sampling rate M to another digital signal with sampling rate N. The both digital signals represent the same analog signal at differtent sampling rates. A multirate filter can be realized in an FIR filter or an IIR filter. Due to the advantages of FIR filters, such as linear phase, unconditional stability, simple structure and easy coefficient design, most of the multirate filters are implemented with FIR filters. Hier we describe only FIR multirate filters.

The basic operations of the multirate filters are decimation and interpolation. Decimation reduces the sample rate of a signal and can be used to eliminate redundant or unnecessary information contained in the signal. Interpolation increases the sample rate of a signal through filling in missing information between the samples of a signal based on the calculation on the existing data.

4.3.3.1 FIR Decimation Filter

The FIR decimation filter can be described using the equation

$$y(m) = \sum_{k=0}^{N-1} h(k)x(mM-k) \qquad [4.14]$$

where h(k) is filter coefficient vector, x(n) is the input signal and M represents the decimation factor. **Figure 4-9** shows the block diagram of an FIR decimation filter. Its equivelent form is showed in **Figure 4-10** .

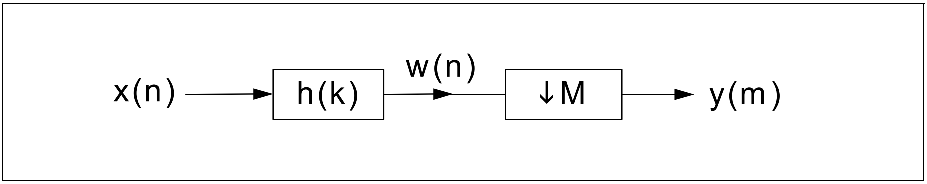

Figure 4-9 Block Diagram of FIR Decimation Filter

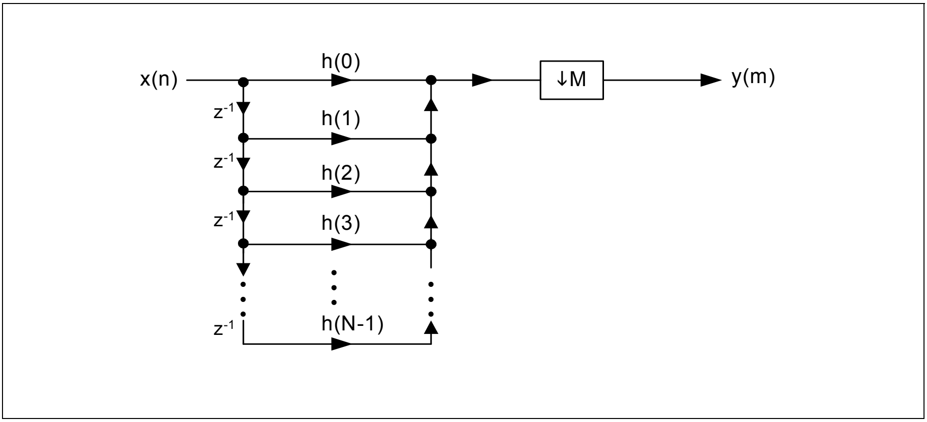

Figure 4-10 Equivelent Implementation of FIR Decimation Filter

4.3.3.2 FIR Interpolation Filter

In comparison with decimation filter the interpolation filter can be used in the reconstruction of a digital signal from another digital signal. **Figure 4-11** shows a block diagram of an interpolation filter, where the low-pass filter of the interpolator uses an Fir filter structure. An Fir interpolation filter can be described as

$$y(m) = \sum_{k=0}^{N-1} h(k)x((m-k)/L) \qquad [4.15]$$

for m-k=0, L, 2L, ··· . An equivelent implementation of the Fir interpolation filter is showed in **Figure 4-12**.

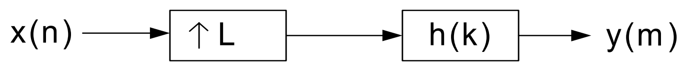

Figure 4-11 Block Diagram of FIR Interpolation Filter

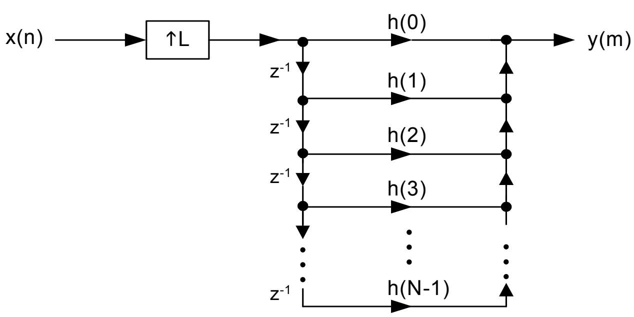

Figure 4-12 Equivelent Implementation of FIR Interpolation Filter

4.3.4 Descriptions

The following FIR filter routines are implemented in the library.

- Fir filter with transversal form, 16 bit filter coefficients, Sample processing
- Fir filter with transversal form, 32 bit filter coefficients, Sample processing
- Complex Fir filter with transversal form, 16 bit filter coefficients, Vector processing
- Symmetric Fir filter, 16 bit filter coefficients, Vector processing
- Lattice Fir filter, 16 bit filter coefficients, Vector processong
- Fir decimation filter, 16 bit filter coefficients, Vector processing
- Fir interpolation filter, 16 bit filter coefficients, Vector processing

Fir_16 FIR Filter with transversal form, 16 bit coefficients, Sample processing

Signature	DataS Fir_16 (DataS* H, DataS* IN, DataS N_h, DataS* D_buffer)		
Inputs	H	:	Pointer to filter coefficients in 1Q15 format
	IN	:	Pointer to the new input sample in 1Q15 format
	L	:	Filter order
	D_buffer		Pointer to delay buffer
Output		:	
Return	Y	:	Filter output in 1Q15 format
Implementation 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. The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM. Delay buffer will be located in the DPRAM area.		

Fir_16 FIR Filter with transversal form, 16 bit coefficients, Sample processing (cont'd)

Pseudo code

```
{

    short x(N_h)={0,...};      //Input vector
    short Y;                   //Filter result
    short i;

    //Update the input vector with the new input value
    for(i=0; i<N_h-1; i++)
      x(i) = x(i+1);
    x(N_h-1) = IN;             //move the new input unto X[N_h-1]

    //Calculate the current FIR output
    Y = 0;
    for(i=0; i<N_h; i++)
      Y = Y + h(i)*x(N_h-1-i)      //FIR filter output

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

Techniques
- Memory bandwidth conflicts removing

Assumptions
- Delay buffer must be located in DPRAM area

Register Usage
- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:
  Y is stored in R4

Fir_16 **FIR Filter with transversal form, 16 bit coefficients, Sample processing** (cont'd)

Memory Note

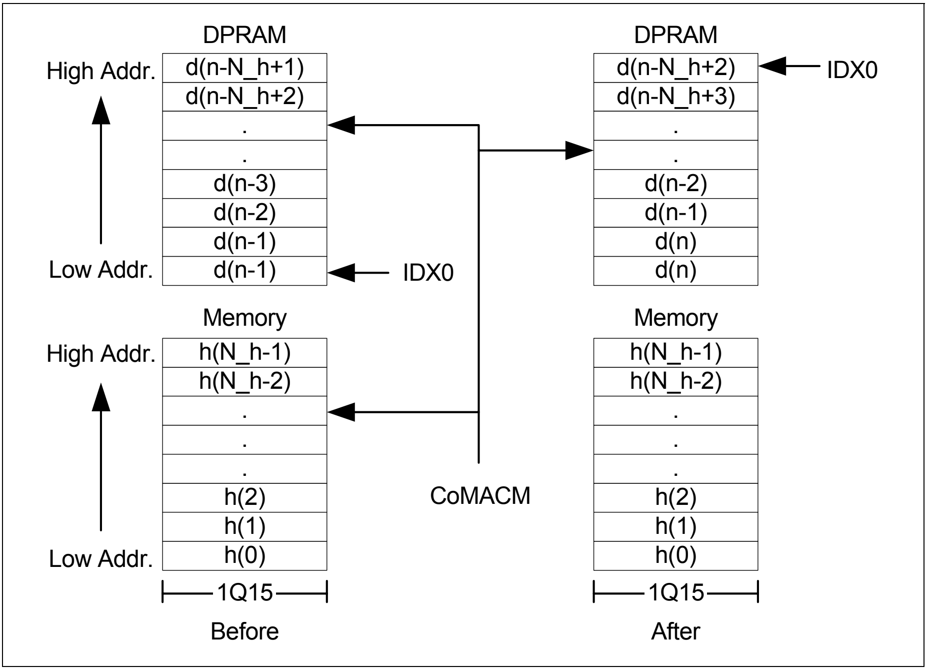

Figure 4-13 Fir_16

Example *C166Lib\Keil\Examples\Filters\Fir\Fir16.c*

Cycle Count

Memory Initialization	9
Read new input into DPRAM	2
FIR loop	N_h

Fir_16 **FIR Filter with transversal form, 16 bit coefficients, Sample processing** (cont'd)

Write output	3
Total	**N_h + 14**
	Example: N_h = 14 cycle = 28

Code Size

Memory initialization	18	bytes
Read new input into DPRAM	8	bytes
FIR loop	10	bytes
Write output	10	bytes
Total	**46**	bytes

Fir_32 — FIR Filter with transversal form, 32 bit coefficients, Sample processing

Signature	DataS Fir_32 (DataL* H, DataS* IN, DataS N_h, DataS* D_buffer)		
Inputs	H	:	Pointer to filter coefficients in 1Q31 format
	IN	:	Pointer to the new input sample in 1Q15 format
	N_h	:	Filter order
	D_buffer		Pointer to delay buffer located in DPRAM area from 0xf200 to 0xfe00 (3 KBytes)
Output		:	
Return	Y	:	Filter output in 1Q15 format

Implementation 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, 32-bit coefficients and gives 16-bit real output. The number of coefficients given by the user is arbitrary. The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM. Delay buffer will be located in the DPRAM area.

Fir_32 FIR Filter with transversal form, 32 bit coefficients, Sample processing (cont'd)

Pseudo code

```
{

    short x(N_h)={0,...};       //Input vector
    short Y;                    //Filter result
    short i;
    long  temp;

    //Update the input vector with the new input value
    for(i=0; i<N_h-1; i++)
      x(i) = x(i+1);
    x(N_h-1) = IN;              //move the new input unto X[N_h-1]

    //Calculate the current FIR output
    temp = 0;
    for(i=0; i<N_h; i++)
      temp = temp + h(i)*x(N_h-1-i)

    Y = (short)temp;          //FIR filter output


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

Techniques

- Memory bandwidth conflicts removing
- Instruction re-ordering
- Loop unrolling

Assumptions

- Delay buffer must be located in DPRAM area

Register Usage

- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:
  Y is stored in R4

Fir_32 FIR Filter with transversal form, 32 bit coefficients, Sample processing (cont'd)

Memory Note

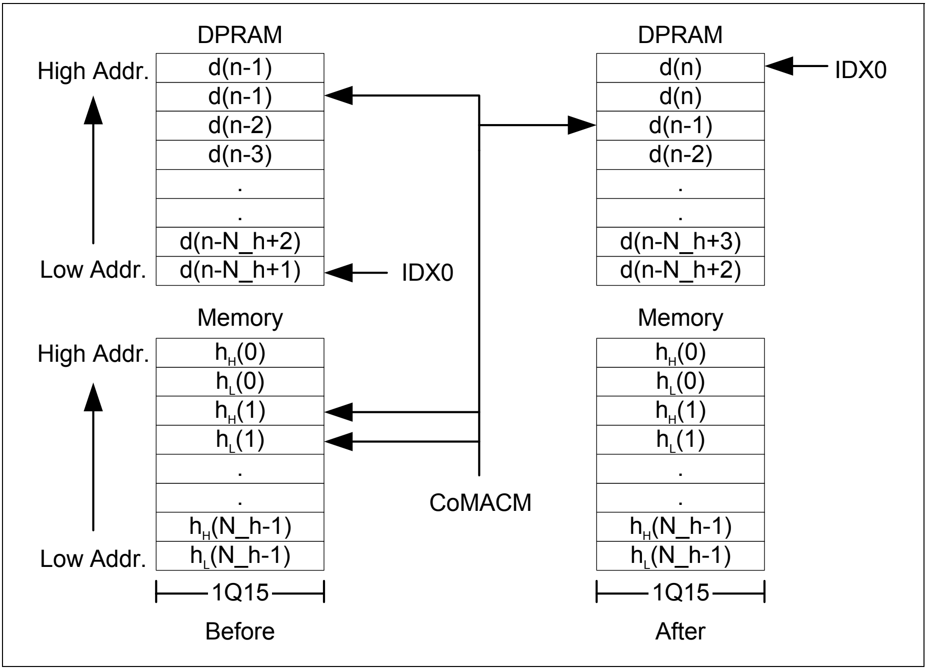

Figure 4-14 Fir_32

Example *C166Lib\Keil\Examples\Filters\Fir\Fir32.c*

Cycle Count

Memory Initialization	16
Read new input into DPRAM	2
FIR loop (LSW)	N_h + 4
FIR loop (MSW)	N_h + 1

Fir_32 **FIR Filter with transversal form, 32 bit coefficients, Sample processing** (cont'd)

Write output	1
Total	**2*N_h + 24**
	Example: N_h = 12 cycle = 48

Code Size

Memory initialization	32 bytes
Read new input into DPRAM	8 bytes
FIR loop (LSW)	28 bytes
FIR loop (MSW)	14 bytes
Write output	4 bytes
Total	**86** bytes

Fir_cplx — Complex Fir filter with transversal form, 16 bit filter coefficients, Vector processing

Signature

DataS Fir_cplx (CplxS* x, CplxS* h, CplxS* y,
DataS* d_buffer, DataS N_x, DataS N_h)

Inputs	x	:	Pointer to the input vector in 1Q15 format
	h	:	Pointer to filter coefficients in 1Q15 format
	d_buffer	:	Pointer to delay buffer Pointer to the new input sample in 1Q15 format
	N_x	:	Size of input vector
	N_h		Filter order
Output	y	:	Filter output vector in 1Q15 format
Return			

Implementation Description

The implementation of the complex FIR filter uses transversal structure (direct form). A vector of the inputs is processed at a time and the output vector is returned. The filter operates on 16-bit complex input vector, 16-bit complex coefficients and gives 16-bit complex output vector. The number of coefficients given by the user is arbitrary. The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM. Delay buffer must be located in the DPRAM area.

Fir_cplx

Complex Fir filter with transversal form, 16 bit filter coefficients, Vector processing (cont'd)

Pseudo code

```
{

   CplxS x[N_x];              //Input vector
   CplxS Y[N_x];              //Filter result
   CplxS h[N_h];              //filter coefficient vector
   CplxS d_buffer[N_h]={0,...}
   short i, j;

   for(j=0; j<N_x; j++)
   {
     //Update the delay buffer with the new input value
     for(i=0; i<N_h-1; i++)
       d_buffer[i] = d_buffer[i+1];
     d_buffer[N_h-1] = x[j];  //move the new input unto d_buffer[N_h-1]

    //Calculate the current FIR output
     y_real[j] = 0;
     y_imag[j] = 0;
     for(i=0; i<N_h; i++)
     {
       y_real[j] = y_real[j] + h_real[i]*d_buffer_real[j-i]-
                   h_imag[i]*d_buffer_imag[j-i];
       y_imag[j] = y_imag[j] + h_real[i]*d_buffer_imag[j-i]-
                   h_imag[i]*d_buffer_real[j-i];
     }
   }
}
```

Techniques

Assumptions

- Delay buffer must be located in DPRAM area

Register Usage

- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:

Fir_cplx **Complex Fir filter with transversal form, 16 bit filter coefficients, Vector processing** (cont'd)

Memory Note

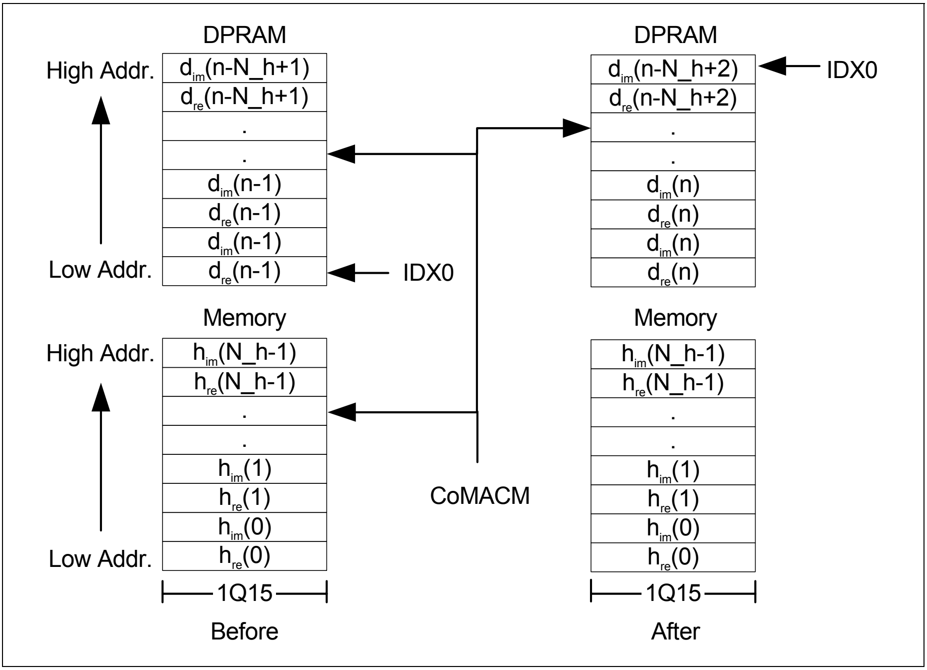

Figure 4-15 Fir_cplx

Example *C166Lib\Keil\Examples\Filters\Fir\FirCplx.c*

Cycle Count

Read parameters	1
Memory initialization	10
Set counters	3
Fir loop	N_x(4N_h + 24)

Fir_cplx **Complex Fir filter with transversal form, 16 bit filter coefficients, Vector processing** (cont'd)

Return	1
Total	**N_x(4N_h + 24) + 15**
	Example: N_x = 1 N_h = 14 cycle = 95

Code Size

Read parameters	2	bytes
Memory initialization	20	bytes
FIR loop	124	bytes
Return	2	bytes
Total	**148**	bytes

Fir_sym — Symmetric Fir filter, 16 bit filter coefficients, Vector processing

Signature DataS Fir_sym (DataS* x, DataS* h, DataS* y, DataS* d_buffer, DataS N_x, DataS N_h)

Inputs

x	:	Pointer to the input vector in 1Q15 format
h	:	Pointer to symmetric filter coefficients in 1Q15 format
d_buffer	:	Pointer to delay buffer
N_x	:	Size of input vector
N_h		Half of the filter order

Output

y	:	Filter output vector in 1Q15 format

Return

Implementation Description

The implemented Fir filter has the symmetric filter coefficient, i.e. h(i)=h(2N_h-1-i), for i=0,1,..., N_h-1. The implementation uses transversal structure (direct form) and vector processing. The filter operates on 16-bit real input vector, 16-bit real coefficients and gives 16-bit real output vector. The number of coefficients should be even.The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM. Delay buffer must be located in the DPRAM area.

Fir_sym **Symmetric Fir filter, 16 bit filter coefficients, Vector processing** (cont'd)

Pseudo code

```
{
   short x[N_x];                      //Input vector
   short y[N_x];                      //Filter result vector
   short h[2N_h];                     //filter coefficient vector
   short d_buffer[2N_h]={0,...};      //delay buffer
   short i, j;

   for(j=0; j<N_x; j++)
   {
     //Update the input vector with the new input value
     for(i=0; i<2N_h-1; i++)
       d_buffer[i] = d_buffer[i+1];
    d_buffer(2N_h-1) = x[j];  //move the new input unto d_buffer[2N_h-1]

     //Calculate the current FIR output
     y[j] = 0;
     for(i=0; i<2N_h; i++)
       y[j] = y[j] + h(i)*d_buffer(2N_h-1-i)      //FIR filter output
   }
   return;

}
```

Techniques

Assumptions
- Delay buffer must be located in DPRAM area

Register Usage
- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:

Fir_sym **Symmetric Fir filter, 16 bit filter coefficients, Vector processing** (cont'd)

Memory Note

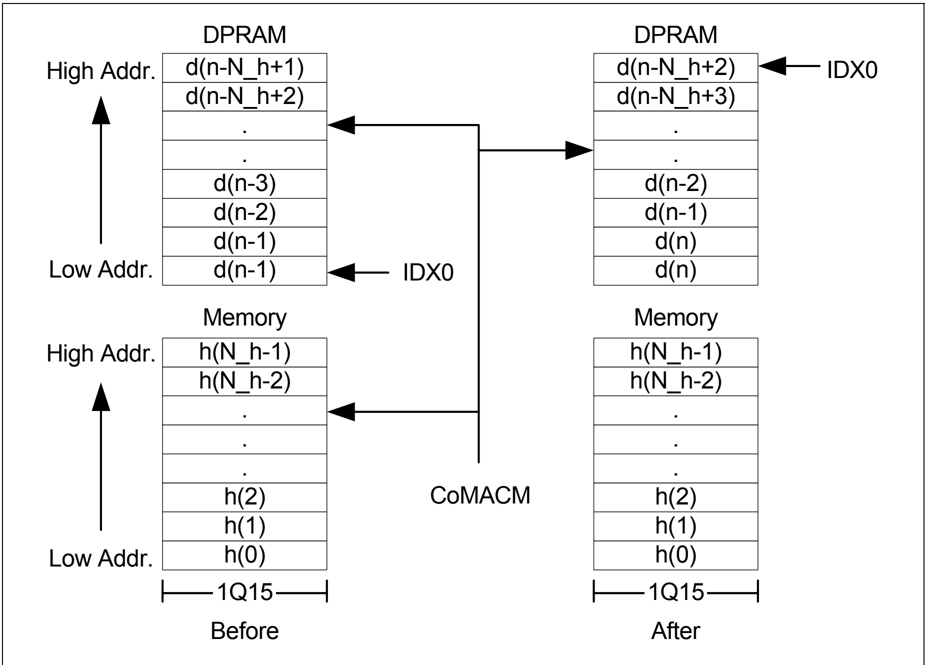

Figure 4-16 Fir_sym

Example *C166Lib\Keil\Examples\Filters\Fir\FirSym.c*

Cycle Count

Read parameters	1
Memory initialization	7
Set counters	3
Fir loop	N_x(2N_h + 12)

Fir_sym **Symmetric Fir filter, 16 bit filter coefficients, Vector processing** (cont'd)

Return	1
Total	**N_x(2N_h + 12) + 11**
	Example: N_x = 1 N_h = 14 cycle = 51

Code Size

Read parameters	2	bytes
Memory initialization	14	bytes
Set counters	6	
FIR loop	68	bytes
Return	2	bytes
Total	**92**	bytes

Fir_lat — Lattice Fir filter, 16 bit filter coefficients, Vector processing

Signature

DataS Fir_lat (DataS* x, DataS* K, DataS* y,
DataS* u, DataS N_x, DataS M)

Inputs	x	:	Pointer to the input vector in 1Q15 format
	K	:	Pointer to lattice coefficients in 1Q15 format
	u	:	Pointer to state variable vector
	N_x	:	Size of input vector
	M		Nummber of stages of the lattice filter
Output	y	:	Filter output vector in 1Q15 format
Return			

Implementation Description

The implementation uses lattice structure showed in **Figure 4-8** and vector processing. The filter operates on 16-bit real input vector, 16-bit real coefficients and gives 16-bit real output vector. The number of stages used in the lattice Fir filter implementation is arbitrary. Lattice coefficient K must be located in the DPRAM area.

Fir_lat — Lattice Fir filter, 16 bit filter coefficients, Vector processing (cont'd)

Pseudo code

```
{
    short x[N_x];              //Input vector
   short yn[M]={0,..};       //Output vector of different stages at time n
    short K[M];                //Lattice filter coefficient vector
    short un[M];                   //State variable vector at time n
    short un-1[M]={0,...};    //State variable vector at time n-1

    short i, j;

    for(j=0; j<N_x; j++)
    {
      //Initialization
      yn[j] = x[j];

      //Calculate the output and state vector
      for(i=1; i<M; i++)
      {
         un[i] = K[i]*yn[j] + un-1[i-1];     //State variable
         yn[j] = yn[j] + K[i]*un-1[i-1];     //Output
         un-1[i] = un[i];                    //Update the state vector
      }

      //Update the first state variable
      un-1[0] = x[j];

    }
    return;

}
```

Techniques

Assumptions

- Lattice coefficient vector must be located in DPRAM area.

Register Usage

- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:

Fir_lat **Lattice Fir filter, 16 bit filter coefficients, Vector processing** (cont'd)

Memory Note

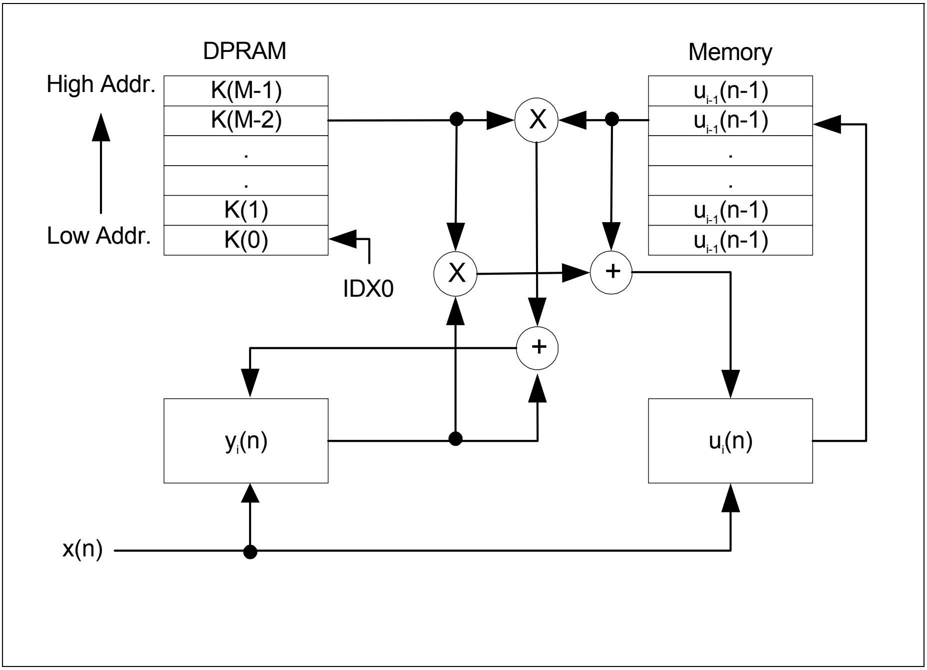

Figure 4-17 Fir_lat

Example *C166Lib\Keil\Examples\Filters\Fir\FirLat.c*

Cycle Count

Read parameters	1
Memory initialization	9
Set counters	1
Fir loop	N_x(15M - 6)

Fir_lat

Lattice Fir filter, 16 bit filter coefficients, Vector processing (cont'd)

Return	1
Total	**N_x(15M - 6) + 12**
	Example: N_x = 1 M = 3 cycle = 51

Code Size

Read parameters	2	bytes
Memory initialization	18	bytes
Set counters	2	bytes
FIR loop	62	bytes
Return	2	bytes
Total	**86**	bytes

Fir_dec — Fir decimation filter, 16 bit filter coefficients, Vector processing

Signature	DataS Fir_dec (DataS* x, DataS* h, DataS* y, DataS* d_buffer, DataS N_x, DataS N_h, DataS D)		
Inputs	x	:	Pointer to the input vector in 1Q15 format
	h	:	Pointer to the filter coefficient vector in 1Q15 format
	d_buffer	:	Pointer to the delay buffer
	N_x	:	Size of the input vector
	N_h	:	Filter order
	D		Decimation factor
Output	y	:	Pointer to output vector
Return		:	

Implementation Description

The implementation of the decimation filter is after **Figure 4-9**, where the FIR filter uses transversal structure (direct form). The vector processing is used in the implementation. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients and the decimator factor given by the user is arbitrary. The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM. Delay buffer will be located in the DPRAM area.

Fir_dec Fir decimation filter, 16 bit filter coefficients, Vector processing (cont'd)

Pseudo code

```
{

    short x[N_x];               //Input vector
    short d_buffer[N_h];        //Delay buffer
    short y[N_x/D];             //Filter result
    short i, j, k;

    for(j=0; j<N_x/D; j++)
    {
      for(k=0; k<D; k++)
      {
        //Update the delay buffer with the new input value
        for(i=0; i<N_h-1; i++)
          d_buffer[i] = d_buffer[i+1];
        d_buffer[N_h-1] = x[j*D+k];

        if(k==0)
        {//Calculate the current decimation FIR filter
          y[j] = 0;
          for(i=0; i<N_h; i++)
            y[j] = y[j] + h(i)*d_buffer[N_h-i];  //filter output
        }
      }
    }
    return;
}
```

Techniques
- Memory bandwidth conflicts removing

Assumptions
- Delay buffer must be located in DPRAM area

Register Usage
- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:
  Ouput in vector y

Fir_dec **Fir decimation filter, 16 bit filter coefficients, Vector processing** (cont'd)

Memory Note

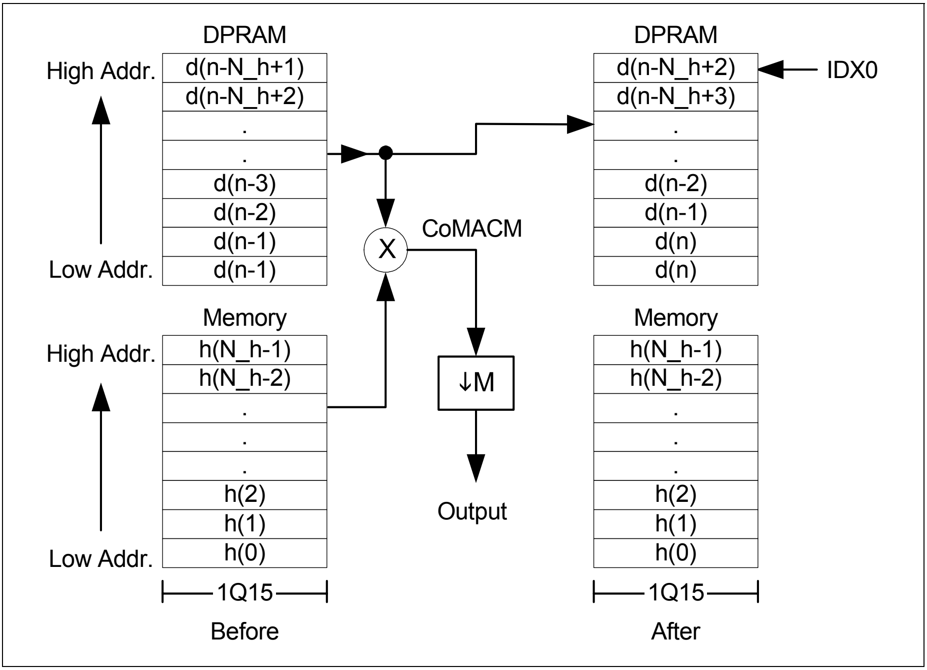

Figure 4-18 Fir_dec

Example *C166Lib\Keil\Examples\Filters\Fir\FirDec.c*

Cycle Count

Read parameter	1
Memory Initialization	7
Set counters	6
FIR loop	N_x(N_h + 13)
Return	1

Fir_dec **Fir decimation filter, 16 bit filter coefficients, Vector processing** (cont'd)

Total	**N_x(N_h + 13) + 15**
	Example: N_x = 1 N_h = 10 cycle = 33

Code Size

Read parameter	2	bytes
Memory initialization	14	bytes
Set counters	12	bytes
FIR loop	44	bytes
Return	2	bytes
Total	**74**	bytes

Fir_inter — Fir interpolation filter, 16 bit filter coefficients, Vector processing

Signature	DataS Fir_inter (DataS* x, DataS* h, DataS* y, DataS* d_buffer, DataS N_x, DataS N_h, DataS I)		
Inputs	x	:	Pointer to the input vector in 1Q15 format
	h	:	Pointer to the filter coefficient vector in 1Q15 format
	d_buffer	:	Pointer to the delay buffer
	N_x	:	Size of the input vector
	N_h	:	Filter order
	I		Interpolation factor
Output	y	:	Pointer to output vector
Return		:	

Implementation Description

The implementation of the interpolation filter is after **Figure 4-11**, where the FIR filter uses transversal structure (direct form). The vector processing is used in the implementation. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients and the interpolation factor given by the user is arbitrary. The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM. Delay buffer will be located in the DPRAM area.

Fir_inter — Fir interpolation filter, 16 bit filter coefficients, Vector processing (cont'd)

Pseudo code

```
{

   short x[N_x];              //Input vector
   short d_buffer[N_h];       //Delay buffer
   short y[N_x*I];            //Filter result
   short i, j, k, m;

   for(j=0; j<N_x; j++)
   {
     for(k=0; k<I; k++)
     {
       //Update the delay buffer with the new input value
       for(i=0; i<N_h-1; i++)
         d_buffer[i] = d_buffer[i+1];
       if(k==0)
         d_buffer[N_h-1] = x[j];
       else
         d_buffer[N_h-1] = 0;

       //Calculate the current decimation FIR filter
       y[j*I+k] = 0;
         for(i=0; i<N_h; i++)
           y[j*I+k] = y[j*I+k] + h(i)*d_buffer[N_h-i];  //filter output
     }

   }
   return;
}
```

Techniques
- Memory bandwidth conflicts removing

Assumptions
- Delay buffer must be located in DPRAM area

Register Usage
- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:
  Ouput in vector y

Fir_inter — Fir interpolation filter, 16 bit filter coefficients, Vector processing (cont'd)

Memory Note

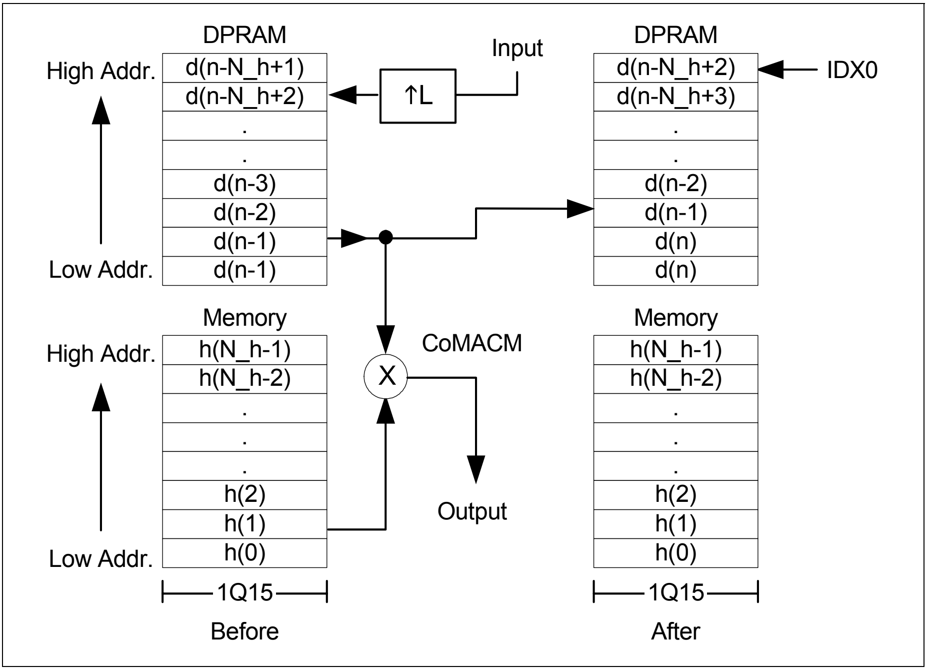

Figure 4-19 Fir_inter

Example *C166Lib\Keil\Examples\Filters\Fir\FirInter.c*

Cycle Count

Read parameter	1
Memory Initialization	7
Set counters	6
FIR loop	N_x(N_h + 13)

Fir_inter

Fir interpolation filter, 16 bit filter coefficients, Vector processing (cont'd)

Return	1
Total	**N_x(N_h + 13) + 15**

Example:
N_x = 1
N_h = 10
cycle = 33

Code Size

Read parameter	2	bytes
Memory initialization	14	bytes
Set counters	12	bytes
FIR loop	44	bytes
Return	2	bytes
Total	**74**	bytes

4.4 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 computational efficient but because of feedback not all IIR structures are realizable (stable). An IIR filter has different implementation structures according to its mathematical description.

4.4.1 Direct Form 1

The N^{th} order difference equation for the direct form 1 of the IIR filter is given by

$$y(n) = \sum_{i=1}^{N} a(i-1) \cdot y(n-i) + \sum_{i=0}^{M} b(i) \cdot x(n-i) \qquad [4.16]$$

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

If M=N=2, we have the biquad (second order) IIR filter as

$$\begin{aligned} y(n) &= b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) \\ &\quad + a(0)y(n-1) + a(1)y(n-2) \end{aligned} \qquad [4.17]$$

where a(0), a(1) correspond to the poles and b(0), b(1), b(2) correspond to the zeroes of the filter.

The equivalent transform function is

$$H[z] = \frac{Y[z]}{X[z]} = \frac{b(0) + b(1)z^{-1} + b(2)z^{-2}}{1 - a(0)z^{-1} - a(1)z^{-2}} \quad . \qquad [4.18]$$

4.4.2 Direct Form 2

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. Breaking **Equation [4.16]** into two parts in terms of zeroes and poles of transfer function (M=N), we have

$$u(n) = x(n) + \sum_{i=1}^{N} a(i-1) \cdot u(n-i) \qquad [4.19]$$

$$y(n) = \sum_{i=0}^{N} b(i) \cdot u(n-i) \qquad [4.20]$$

where intermediate state variable u(n) is used to calculate the filter output y(n). This representation is called "Direct Form 2" implementation of an IIR filter and is illustrated in **Figure 4-20**. Direct Form 2 has an advantage over Direct Form 1 as it requires less data memory.

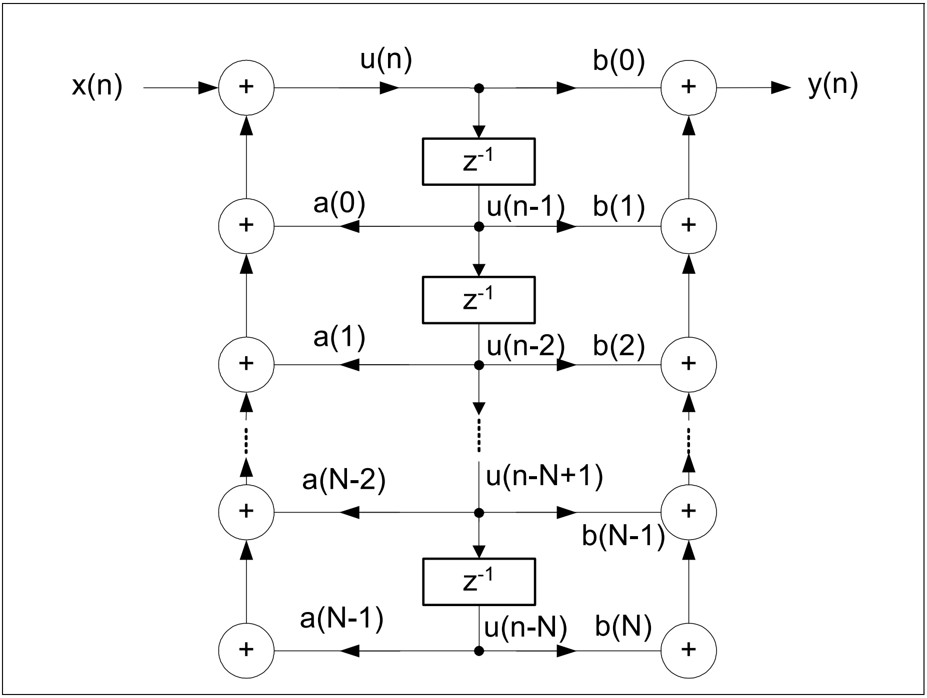

Figure 4-20 Canonical Form (Direct Form 2)

4.4.3 Cascaded Biquad IIR Filter with Direct Form 2

If N=2, **Equation [4.19]** and **Equation [4.20]** are reduced to the biquad (second order) IIR filter with direct form 2 implementation:

$$u(n) = x(n) + a(0) \cdot u(n-1) + a(1) \cdot u(n-2) \qquad [4.21]$$

$$y(n) = b(0) \cdot u(n) + b(1) \cdot u(n-1) + b(2) \cdot u(n-2) \qquad [4.22]$$

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 2 realization of each biquad subsystem would be as shown in the following diagram.

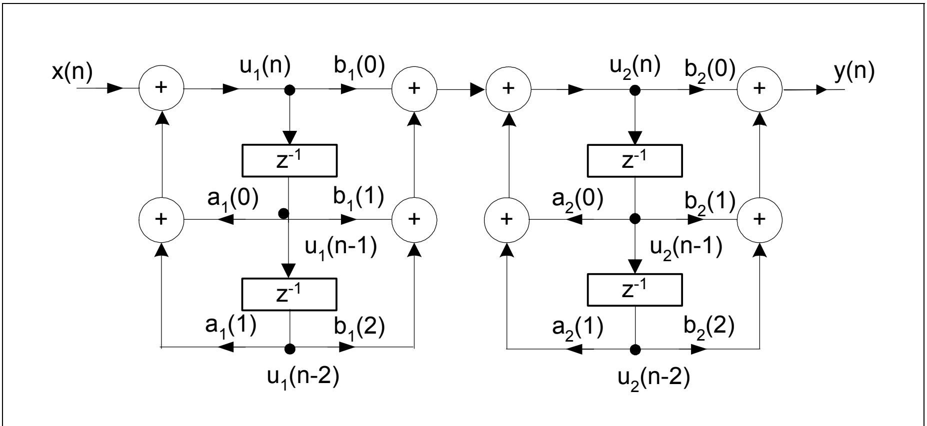

Figure 4-21 Cascaded Biquad IIR Filter with Direct Form 2 Implementation

4.4.4 Cascaded Biquad IIR Filter with Direct Form 1

In **Figure 4-21** each biquad subsystem is realized with direct form 2 structure. Similarly, the biquad subsystem can also be implemented with direct form 1 structure. Rewriting **Equation [4.17]** we have

$$y(n) = b(0) \cdot x(n) + u(n-1) \qquad [4.23]$$

$$u(n) = a(1) \cdot y(n) + b(1) \cdot x(n) + w(n-1) \qquad [4.24]$$

$$w(n) = a(2) \cdot y(n) + b(2) \cdot x(n) \qquad [4.25]$$

where u(n) and w(n) are state variables at time n. According to **Equation [4.23]**, **Equation [4.24]** and **Equation [4.25]** we have another cascaded biquads IIR filter implementation showed in **Figure 4-22**.

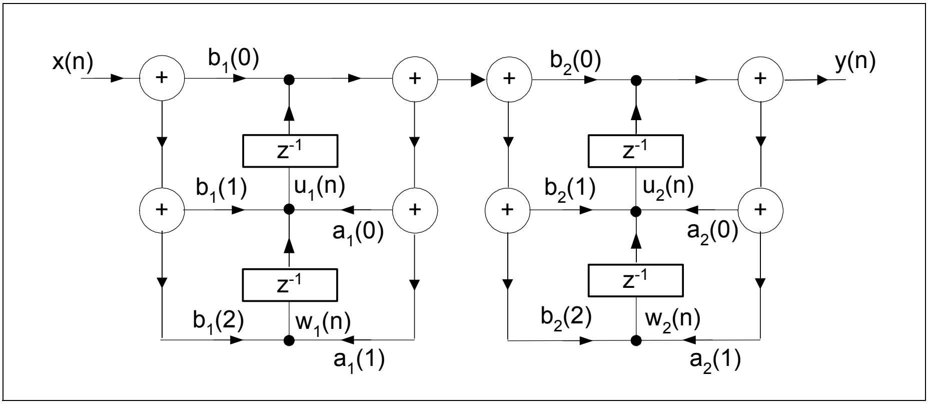

Figure 4-22 Cascaded Biquad IIR Filter with Direct Form 1 Implementation

A Comparison between FIR and IIR filters:

- IIR filters are computational 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.

4.4.5 Lattice IIR Filter

Similar with lattice Fir filter the IIR filter can be also realized using the lattice filter structure. The typical application of the lattice IIR filter is found in voice analysis and sythesis system, where lattice Fir filer is used to analyse the speech and the lattice IIR filter is used in the synthesis of speech.

The lattice IIR filter showed in **Figure 4-23** can be described using the following equations.

$$\begin{aligned} y_{i-1}(n) &= y_i(n) - k_i u_{i-1}(n-1) \\ u_i(n) &= k_i y_{i-1}(n) + u_{i-1}(n-1) \end{aligned} \qquad 0 \le i \le M-1 \qquad [4.26]$$

The initial value is equal to the filter input x(n):

$$y_M(n) = x(n)$$

At the last stage (i=0) we have the output of the lattice IIR filter $y(n)=y_0(n)$.

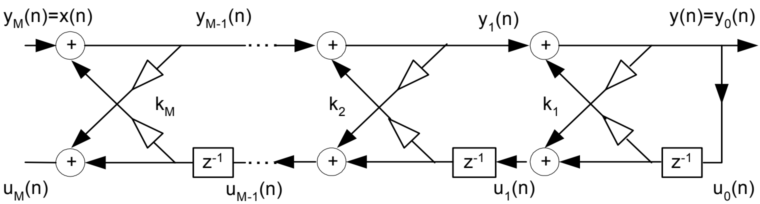

Figure 4-23 Lattice IIR Filter Structure

4.4.6 Descriptions

The following IIR filter routines are described.

- 16 bit filter coefficients, direct form 1, sample processing
- 16 bit filter coefficients, direct form 2, sample processing
- 16 bit filter coefficients, N-cascaded real biquads, direct form 1, 5-coefs per biquad, vector processing
- 16 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing
- 32 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing

- 16 bit filter coefficients, N-cascaded real biquads, direct form 2, 4-coefs per biquad, vector processing
- Lattice IIR filter, 16 bit filter coefficients, vector processong

IIR_1 16 bit filter coefficients, Direct form 1, Sample processing

Signature	DataS IIR_1(DataS* h, DataS* IN, DataS N, DataS* x_y)		
Inputs	h	:	Pointer to filter coefficients in 2Q14
	IN	:	Pointer to new input sample in 1Q15
	N	:	Filter order
	x_y	:	Pointer to delay buffer containing the past input and output samples
Output		:	
Return	Y	:	Filter output in 1Q15 format
Implementation Description	The routine is implemented according to the **Equation [4.16]** , which is called direct form 1 implementation. The vector h contains IIR filter coefficient a(i) and b(i), and is stored in memory. Before starting the routine the delay buffer x_y containing the past input and output signals should be located in DPRAM area. This IIR filter routine processes one sample at a time and returns the output for that sample in 1Q15 format.		

IIR_1 16 bit filter coefficients, Direct form 1, Sample processing (cont'd)

Pseudo code

```
{
    ; x(n) = input signal at time n
    ; y(n) = output signal at time n
    ; a(k), b(k) = IIR filter coefficients
    ; N refer to the filter order in Equation [4.16]

    DataS  a[N], b[N+1];     //Filter vectors
    DataS  y[N], x[N+1];     //input and output signal vectors
    DataS  i, temp;

    ;Move the new input sample into input vector in DPRAM
    for(i=N; i>0; i--)
       x(n-i-1) = x(n-i);
    x(n) = IN;               //New input sample

    ;IIR filtering
    y(n) = 0;
    for(i=0 to N)
       y(n) = y(n)+b(i)*x(n-i);
    for(i=1 to N)
       y(n) = y(n)+a(i)*y(n-i);

    Y = y(n)

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

Techniques
- Memory bandwidth conflicts removing
- Instruction re-ordering

Assumptions
- Delay buffer must be located in DPRAM area

Register Usage
- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:
  Y is stored in R4

Memory Note

IIR_1 **16 bit filter coefficients, Direct form 1, Sample processing** (cont'd)

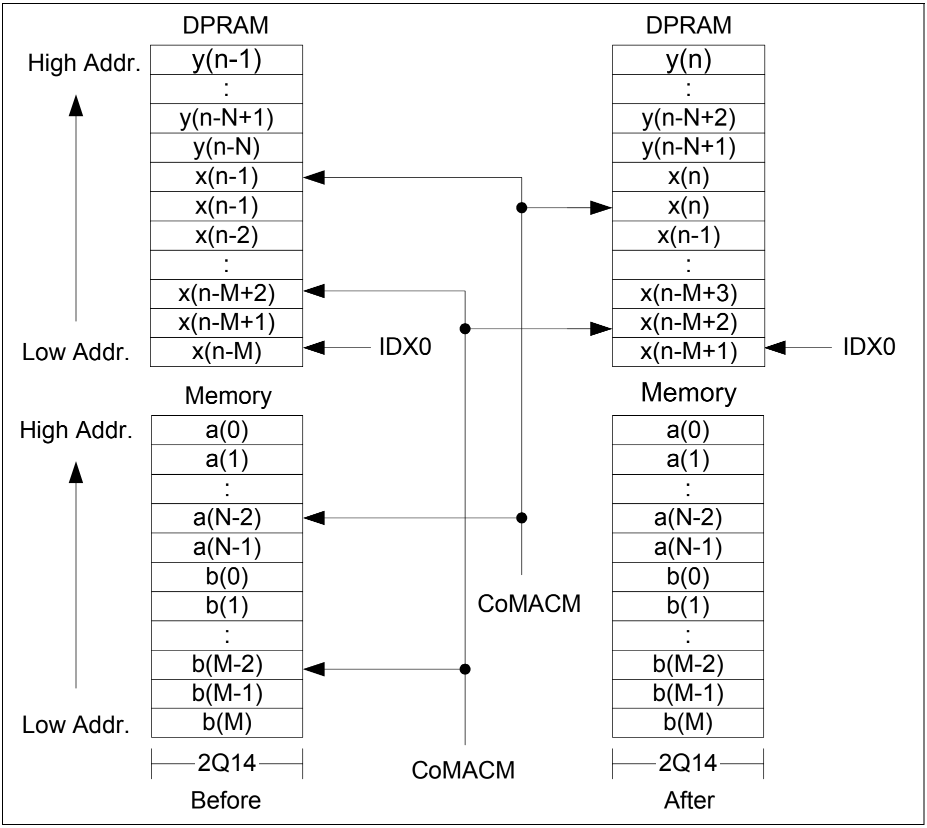

Figure 4-24 IIR_1

Example *C166Lib\Keil\Examples\Filters\IIR\ IIRform1.c*

Cycle Count

Memory Initialization	10
Read the new sample into DPRAM	2

IIR_1 **16 bit filter coefficients, Direct form 1, Sample processing** (cont'd)

First IIR loop	N+1
Repeat count re-initialization	2
Second IIR loop	N
Write the output	6
Total	**2N + 21**
	Example: N=4 cycle = 29

Code Size

Memory initialization	20 bytes
Read the new sample into DPRAM	8 bytes
First IIR loop	10 bytes
Repeat count re-initialization	4 bytes
Second IIR loop	10 bytes
Write the output	22 bytes
Total	**74** bytes

IIR_2 16 bit filter coefficients, Direct form 2, Sample processing

Signature DataS IIR_2 (DataS* h, DataS* IN, DataS N, DataS* u)

Inputs	h	:	Pointer to filter coefficients in 2Q14
	IN	:	Pointer to new input sample in 1Q15
	N	:	Filter order
	u	:	Pointer to state variable vector
Output		:	
Return	Y	:	Filter output in 1Q15 format

Implementation Description

The IIR filter routine is implemented based on direct form II implementation (N^{th} Order) showed in **Figure 4-20**. The filter operates on 16-bit real input, 16-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary.

Pseudo code

```
{
    ; x(n) = input signal at time n
    ; u(n) = state variable at time n
    ; y(n) = output signal at time n
    ; a(k), b(k) = IIR filter coefficients
    ; N = M refer to the filter order in Equation [4.16]

    DataS  a[N], b[N+1];     //Filter vectors
    DataS  u[N+1];           //state variable vector
    DataS  i;

    ;Calculate the sate variable at time n, u(n)
    u(n) = x(n);
    for(i=1; i<N; i++)
       u(n) = u(n) + a(i-1)*u(n-i);

    ;Calculate the output at time n
    y(n) = 0;
    for(i=0; i<=N; i++)
       y(n) = y(n)+b(i)*u(n-i);

    Y = y(n)

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

IIR_2 **16 bit filter coefficients, Direct form 2, Sample processing** (cont'd)

Techniques
- Memory bandwidth conflicts removing
- Use of CoMAC and CoMACM instructions
- Filter output converted to 16-bit with saturation

Assumptions
- The state variable vector must be located in DPRAM area.

Register Usage
- From .c file to .asm file:

  Decided by compiler
- From .asm file to .c file:

  Y is stored in R4

IIR_2 16 bit filter coefficients, Direct form 2, Sample processing (cont'd)

Memory Note

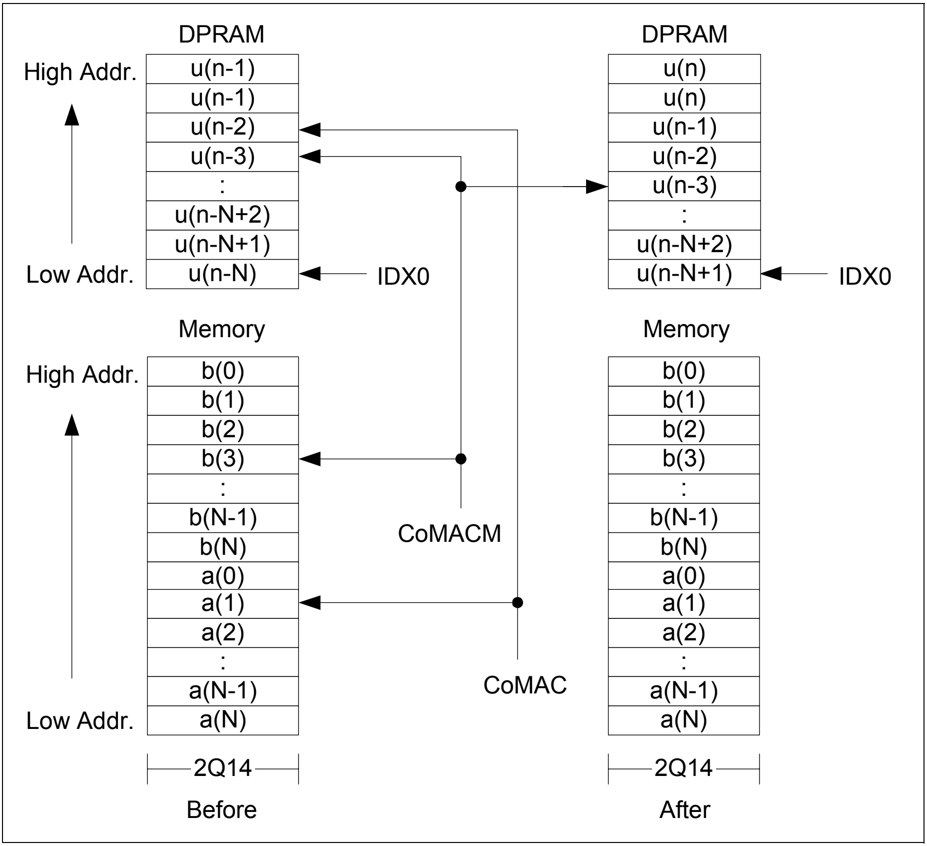

Figure 4-25 IIR_2

Example *C166Lib\Keil\Examples\Filters\IIR\ IIRform2.c*

Cycle Count

Memory Initialization	9

IIR_2 16 bit filter coefficients, Direct form 2, Sample processing (cont'd)

Read the new input sample	2
First IIR loop	N + 4
Repeat count re-initialization	1
Second IIR loop	N+1
Write the output	3
Return	1
Total	**2N + 21**
	Example: N = 4 cycle = 29

Code Size

Memory initialization	18	bytes
Read the new input sample	6	bytes
First IIR loop	20	bytes
Repeat count re-initialization	2	bytes
Second IIR loop	8	bytes
Write the output	12	bytes
Return	2	bytes
Total	**68**	bytes

IIR_bi_1 — 16 bit filter coefficients, N-cascaded real biquads, direct form 1, 5-coefs per biquad, vector processing

Signature	DataS IIR_bi_1(DataS* x, DataS* h, DataS* y, DataS* u_w, DataS N_x, DataS N_biq)		
Inputs	x	:	Pointer to input vector in 1Q15
	h	:	Pointer to filter coefficent vecotr in 1Q15
	u_w	:	Pointer to the state variable vector
	N_x	:	Size of input vector
	N_biq	:	Number of the biquads
Output	y	:	Pointer to output vector in 1Q15 format
Return		:	

Implementation

The IIR filter is implemented as a cascade of direct form 1 biquads according to **Figure 4-21**. If the number of biquads is 'N', the filter order is 2*N. A vector of samples is processed at a time and the output vector for those samples is returned. The filter operates on 16-bit real vector input, 16-bit real coefficients and returns 16-bit real vector output. The number of inputs is arbitrary. Length of delay line is 2*N.

IIR_bi_1 16 bit filter coefficients, N-cascaded real biquads, direct form 1, 5-coefs per biquad, vector processing (cont'd)

Pseudo code

```
{

    ; X[N_x], input vector
    ; xi(n) = input buffer at time n of biquad number i
    ; ui(n), wi(n) = state variables at time n of biquad number i
    ; yi(n) = output signal at time n of biquad number i
    ; ai(k), bi(k) = Filter coefficients of biquad number i
    ; N = number of biquads

    DataS  ai(n), bi(n);        //Filter vectors
    DataS  ui(n), wi(n);        //state variable vector
    DataS  yi(n), Y[N_x];       //output
    DataS  i, j;

    for(j=0;j<N_x;j++)
    {
      xi(n) = X[j];
      for(i=1; i<=N_biq; i++)
      {
        ;Calculate the output at time n
         yi(n) = bi(0)*xi(n) + ui(n-1);

        ;Update the sate variable ui(n)at time n
         ui(n) = bi(1)*xi(n) + ai(0)*yi(n) + wi(n-1);

        ;Update the sate variable wi(n)at time n
         wi(n) = bi(2)*xi(n) + ai(1)*yi(n);

        ;Set i-th biquad output to (i+1)-th biquad input
         xi+1(n) = yi(n);
      }

      ;Output of the last biquad
      Y[j] = yN-1(n);
    }

    return;

}
```

IIR_bi_1 **16 bit filter coefficients, N-cascaded real biquads, direct form 1, 5-coefs per biquad, vector processing** (cont'd)

Techniques

- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering
- Filter output converted to 16-bit with saturation

Assumptions

- The state variable vector must be located in DPRAM area.

Register Usage

- From .c file to .asm file:

  Decided by compiler
- From .asm file to .c file:

  Y is stored in R4

Memory Note

IIR_bi_1 **16 bit filter coefficients, N-cascaded real biquads, direct form 1, 5-coefs per biquad, vector processing** (cont'd)

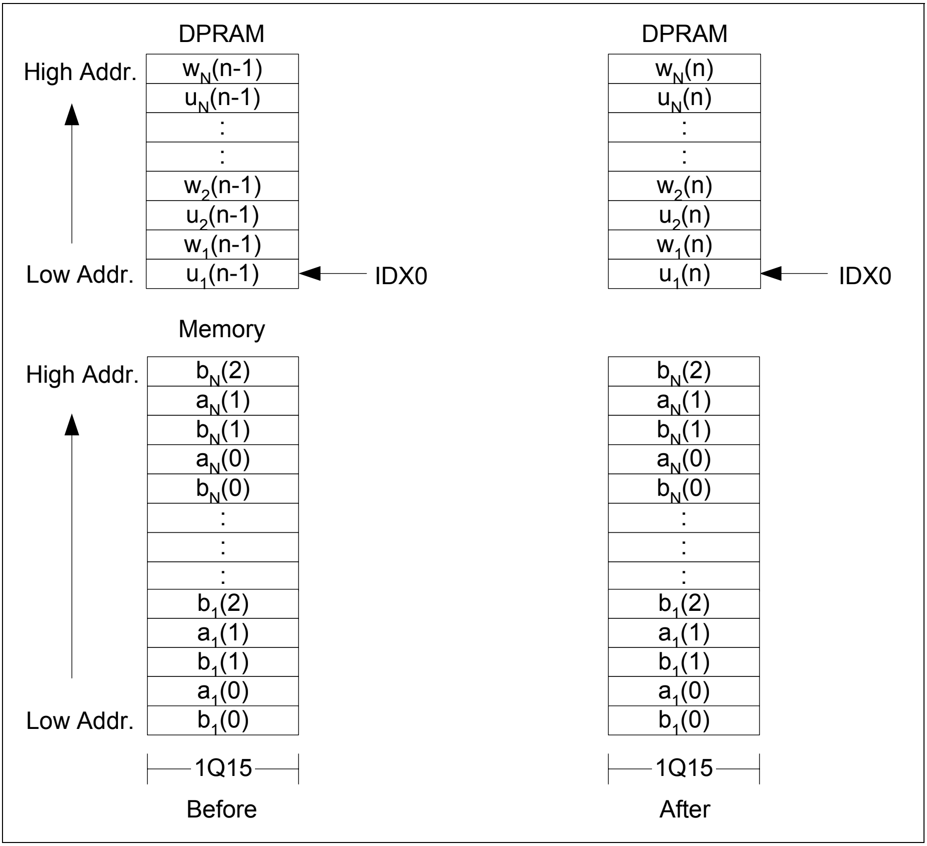

Figure 4-26 IIR_bi_1

Example *C166Lib\Keil\Examples\Filters\IIR\ IIRbi_1.c*

Cycle Count

Read parameter 1

IIR_bi_1 **16 bit filter coefficients, N-cascaded real biquads, direct form 1, 5-coefs per biquad, vector processing** (cont'd)

Memory Initialization	11
Biquad loop	N_x(17*N_biq + 7)
Return	1
Total	**N_x(17*N_biq +7) + 13**
	Example: N_x = 1 N_biq = 2 cycle = 54

Code Size

Read parameter	2	bytes
Memory initialization	22	bytes
Biquad loop	118	bytes
Return	2	bytes
Total	**144**	bytes

IIR_bi_2 — 16 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing

Signature

DataS IIR_bi_2 (DataS* x, DataS* h, DataS* y, DataS* u,
DataS N_x, DataS N_biq)

Inputs	x	:	Pointer to input vector in 1Q15
	h	:	Pointer to filter coefficients in 1Q15
	u	:	Pointer to state variable vector
	N_x	:	Size of input vector
	N_biq	:	Number of the biquads
Output	y	:	Pointer to filter output in 1Q15
Return		:	

Implementation Description

The IIR filter is implemented as a cascade of direct form 2 biquads according to **Figure 4-21**. If number of biquads is 'N', the filter order is 2*N. The filter operates on 16-bit real input, 16-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary. Length of delay line is 2*N.

IIR_bi_2

16 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing (cont'd)

Pseudo code

```
{
    ; X[N_x], input vector
    ; xi(n) = input buffer at time n of biquad number i
    ; ui(n) = state variables at time n of biquad number i
    ; yi(n) = output signal at time n of biquad number i
    ; ai(k), bi(k) = Filter coefficients of biquad number i
    ; N = number of biquads

    DataS  ai(n), bi(n);      //Filter vectors
    DataS  ui(n);             //state variable vector
    DataS  yi(n), Y;             //output
    DataS  i, j;

    for(j=0; j<N_x; j++)
    {
      xi(n) = X[j];
      for(i=1; i<=N_biq; i++)
      {
        ;Update the sate variable ui(n)at time n
         ui(n) = xi(n) + ai(0)*ui(n-1) + ai(1)*ui(n-2);

        ;Calculate the output at time n
         yi(n) = bi(0)*ui(n) + bi(1)*ui(n-1) + bi(2)*ui(n-2);

        ;Set i-th biquad output to (i+1)-th biquad input
         xi+1(n) = yi(n);
      }

      ;Output of the last biquad
      Y[j] = yN-1(n);
    }

    return;          //Filter Output returned

}
```

Techniques

- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering
- Filter output converted to 16-bit with saturation

IIR_bi_2 — 16 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing (cont'd)

Assumptions

- The state variable vector must be located in DPRAM area.

Register Usage

- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:
  Y is stored in R4

IIR_bi_2 **16 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing** (cont'd)

Memory Note

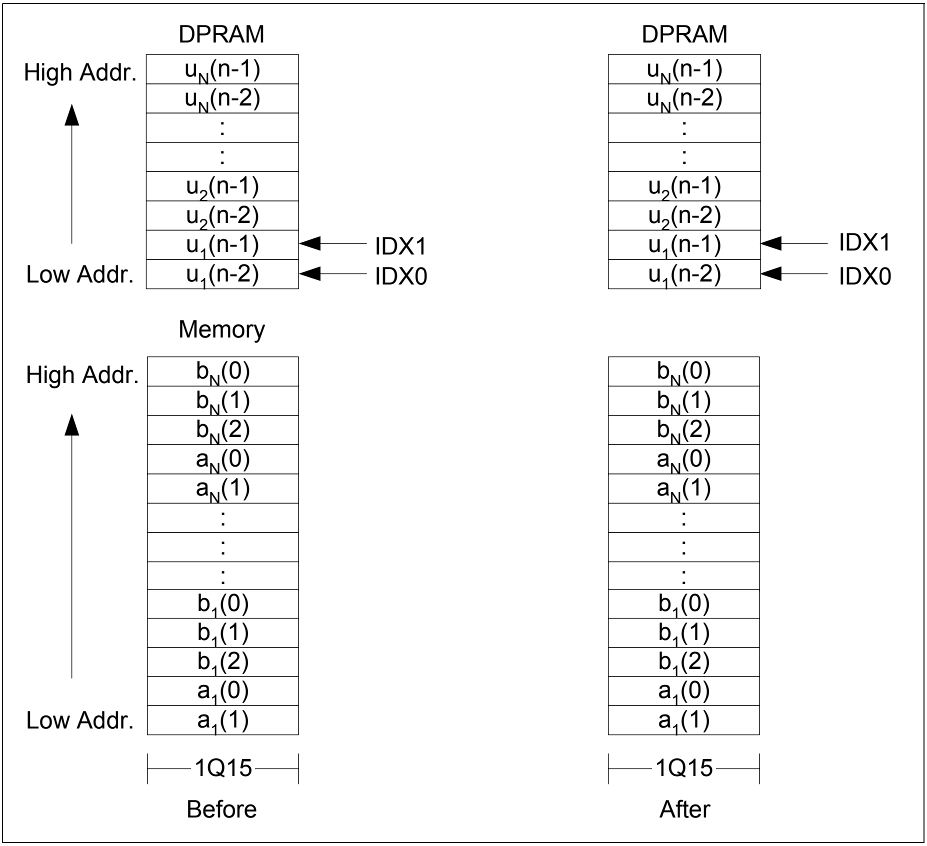

Figure 4-27 IIR_bi_2

Example *C166Lib\Keil\Examples\Filters\IIR\ IIRbi_2.c*

Cycle Count

IIR_bi_2 **16 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing** (cont'd)

Parameter read	1
Memory Initialization	16
Biquad loop	N_x(13*N_biq + 7)
Return	1
Total	**N_x(13*N_biq + 7) + 18**

Example:
N_x = 1
N_biq = 2
cycle = 51

Code Size

Parameter read	2	bytes
Memory Initialization	32	bytes
Biquad loop	54	bytes
Return	2	bytes
Total	**90**	bytes

IIR_bi2_32 — 32 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing

Signature

DataS IIR_bi2_32 (DataS* x, DataSL* h, DataS* y, DataS* u,
DataS N_x, DataS N_biq)

Inputs

x	:	Pointer to input vector in 1Q15
h	:	Pointer to filter coefficients in 1Q31
u	:	Pointer to state variable vector
N_x	:	Size of input vector
N_biq	:	Number of the biquads

Output

y	:	Pointer to filter output in 1Q15

Return :

Implementation Description

The IIR filter is implemented as a cascade of direct form 2 biquads according to **Figure 4-21** . If number of biquads is 'N', the filter order is 2*N. A vector of samples is processed at a time and the output vector for those samples is returned. The filter operates on 16-bit real input, 32-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary. Length of delay line is 2*N.

IIR_bi2_32 — 32 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing (cont'd)

Pseudo code

```
{
    ; X[N_x], input vector
    ; xi(n) = input buffer at time n of biquad number i
    ; ui(n) = state variables at time n of biquad number i
    ; yi(n) = output signal at time n of biquad number i
    ; ai(k), bi(k) = Filter coefficients of biquad number i
    ; N = number of biquads

    DataS  ai(n), bi(n);       //Filter vectors
    DataS  ui(n);              //state variable vector
    DataS  yi(n), Y;              //output
    DataS  i, j;

    for(j=0; j<N_x; j++)
    {
      xi(n) = X[j];
      for(i=1; i<=N_biq; i++)
      {
        ;Update the sate variable ui(n)at time n
         ui(n) = xi(n) + ai(0)*ui(n-1) + ai(1)*ui(n-2);

        ;Calculate the output at time n
         yi(n) = bi(0)*ui(n) + bi(1)*ui(n-1) + bi(2)*ui(n-2);

        ;Set i-th biquad output to (i+1)-th biquad input
         xi+1(n) = yi(n);
      }

      ;Output of the last biquad
      Y[j] = yN-1(n);
    }

    return;          //Filter Output returned

}
```

Techniques

- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering
- Filter output converted to 16-bit with saturation

IIR_bi2_32 **32 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing** (cont'd)

Assumptions

- The state variable vector must be located in DPRAM area.

Register Usage

- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:
  Y is stored in R4

IIR_bi2_32 **32 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing** (cont'd)

Memory Note

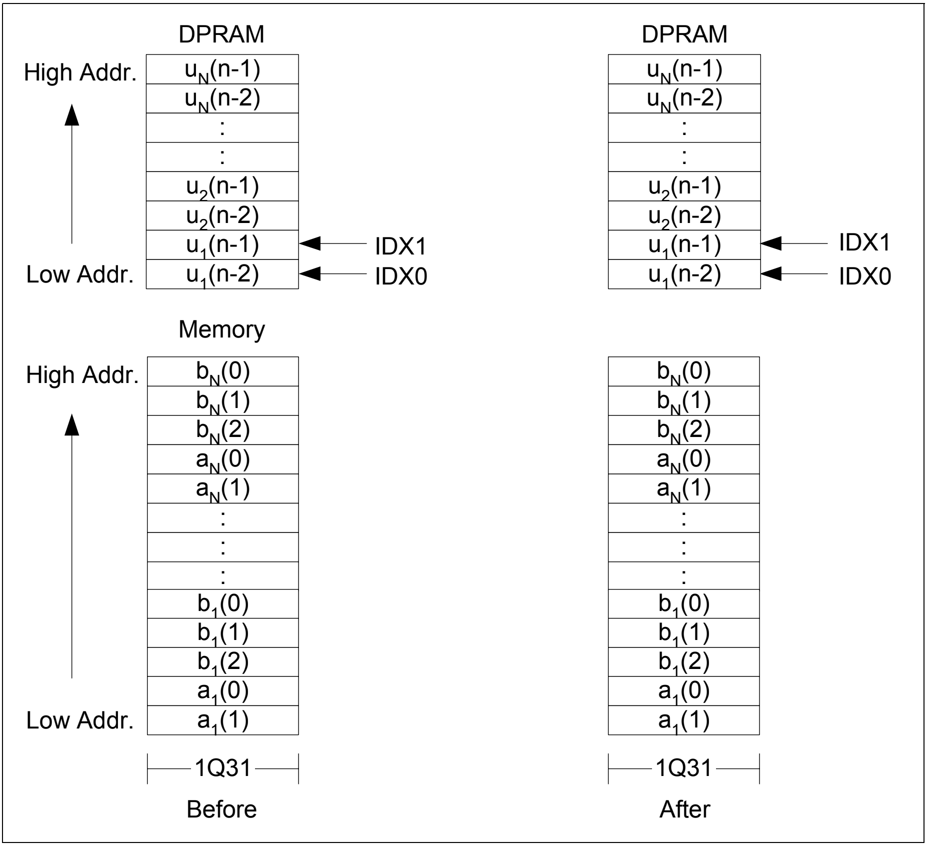

Figure 4-28 IIR_bi2_32

Example *C166Lib\Keil\Examples\Filters\IIR\ IIRbi2_32.c*

Cycle Count

IIR_bi2_32 **32 bit filter coefficients, N-cascaded real biquads, direct form 2, 5-coefs per biquad, vector processing** (cont'd)

Parameter read	1
Memory Initialization	18
Biquad loop	N_x(24*N_biq + 7)
Return	1
Total	**N_x(24*N_biq + 7) + 20**

Example:
N_x = 1
N_biq = 2
cycle = 75

Code Size

Parameter read	2	bytes
Memory Initialization	36	bytes
Biquad loop	156	bytes
Return	2	bytes
Total	**196**	bytes

IIR_bi_24 — 16 bit filter coefficients, N-cascaded real biquads, direct form 2, 4-coefs per biquad, vector processing

Signature

DataS IIR_bi_24 (DataS* x, DataS* h, DataS* y, DataS* u, DataS N_x, DataS N_biq)

Inputs	x	:	Pointer to input vector in 1Q15
	h	:	Pointer to filter coefficients in 1Q15
	u	:	Pointer to state variable vector
	N_x	:	Size of input vector
	N_biq	:	Number of the biquads
Output	y	:	Pointer to filter output in 1Q15
Return		:	

Implementation Description

The IIR filter is implemented as a cascade of direct form 2 biquads according to **Figure 4-21** . If number of biquads is 'N', the filter order is 2*N. A vector of samples is processed at a time and the output vector for those samples 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. Length of delay line is 2*N.

IIR_bi_24 — 16 bit filter coefficients, N-cascaded real biquads, direct form 2, 4-coefs per biquad, vector processing (cont'd)

Pseudo code

```
{
     ; X[N_x], input vector
     ; xi(n) = input buffer at time n of biquad number i
     ; ui(n) = state variables at time n of biquad number i
     ; yi(n) = output signal at time n of biquad number i
     ; ai(k), bi(k) = Filter coefficients of biquad number i
     ; N = number of biquads

     DataS  ai(n), bi(n);      //Filter vectors
     DataS  ui(n);             //state variable vector
     DataS  yi(n), Y;             //output
     DataS  i, j;

     for(j=0; j<N_x; j++)
     {
       xi(n) = X[j];
       for(i=1; i<=N_biq; i++)
       {
         ;Update the sate variable ui(n)at time n
          ui(n) = xi(n) + ai(0)*ui(n-1) + ai(1)*ui(n-2);

         ;Calculate the output at time n
          yi(n) = bi(0)*ui(n) + bi(1)*ui(n-1) + bi(2)*ui(n-2);

         ;Set i-th biquad output to (i+1)-th biquad input
          xi+1(n) = yi(n);
       }

       ;Output of the last biquad
       Y[j] = yN-1(n);
     }

     return;          //Filter Output returned

}
```

Techniques

- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering
- Filter output converted to 16-bit with saturation

IIR_bi_24 — **16 bit filter coefficients, N-cascaded real biquads, direct form 2, 4-coefs per biquad, vector processing** (cont'd)

Assumptions

- The state variable vector must be located in DPRAM area.

Register Usage

- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:
  Y is stored in R4

IIR_bi_24 — **16 bit filter coefficients, N-cascaded real biquads, direct form 2, 4-coefs per biquad, vector processing** (cont'd)

Memory Note

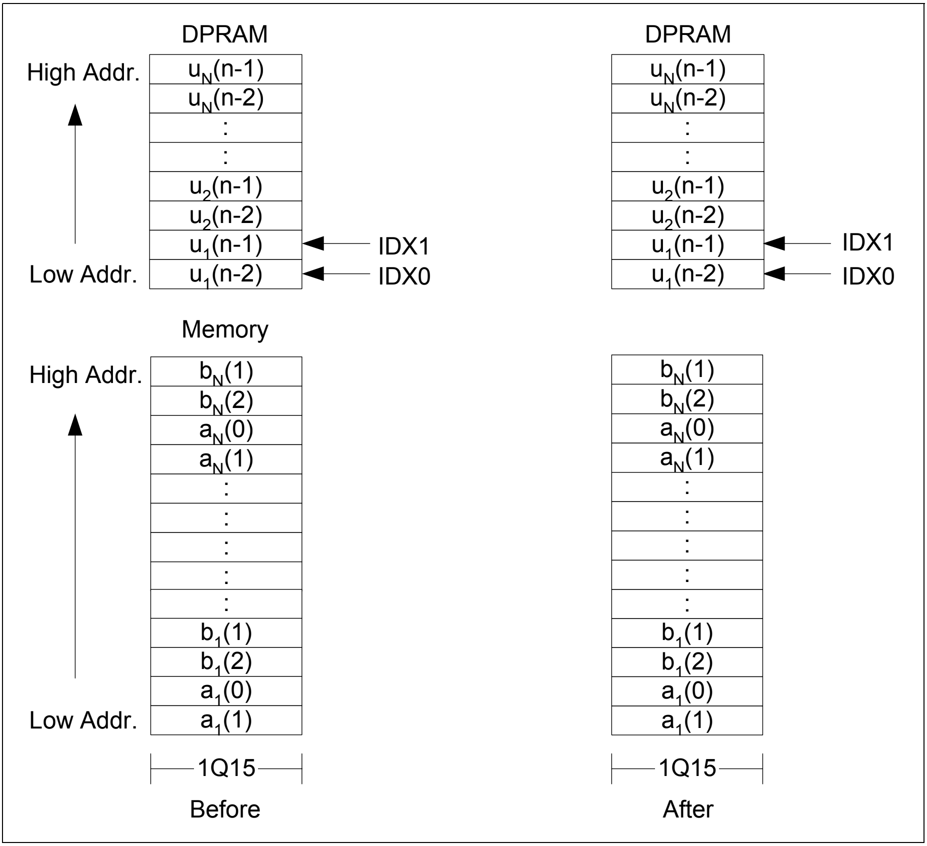

Figure 4-29 IIR_bi_24

Example *C166Lib\Keil\Examples\Filters\IIR\ IIRbi_24.c*

Cycle Count

IIR_bi_24 **16 bit filter coefficients, N-cascaded real biquads, direct form 2, 4-coefs per biquad, vector processing** (cont'd)

Parameter read	1
Memory Initialization	16
Biquad loop	N_x(13*N_biq + 5)
Return	1
Total	**N_x(13*N_biq + 5) + 17**

Example:
N_x = 1
N_biq = 2
cycle = 48

Code Size

Parameter read	2	bytes
Memory Initialization	32	bytes
Biquad loop	100	bytes
Return	2	bytes
Total	**136**	bytes

IIR_lat — Lattice IIR filter, 16 bit filter coefficients, Vector processing

Signature

DataS Fir_lat (DataS* x, DataS* K, DataS* y,
DataS* u, DataS N_x, DataS M)

Inputs	x	:	Pointer to the input vector in 1Q15 format
	K	:	Pointer to lattice coefficients in 1Q15 format
	u	:	Pointer to state variable vector
	N_x	:	Size of input vector
	M		Nummber of stages of the lattice filter
Output	y	:	Filter output vector in 1Q15 format
Return			

Implementation Description

The implementation uses lattice structure showed in **Figure 4-23** and vector processing. The filter operates on 16-bit real input vector, 16-bit real coefficients and gives 16-bit real output vector. The number of stages used in the lattice IIR filter implementation is arbitrary. Lattice coefficient K must be located in the DPRAM area.

IIR_lat Lattice IIR filter, 16 bit filter coefficients, Vector processing (cont'd)

Pseudo code

```
{
    short x[N_x];                //Input vector
   short yn[M]={0,..};        //Output vector of different stages at time n
    short K[M];                  //Lattice filter coefficient vector
    short un[M];                 //State variable vector at time n
    short un-1[M]={0,...};     //State variable vector at time n-1

    short i, j;

    for(j=0; j<N_x; j++)
    {
      //Initialization
      yn[j] = x[j];

      //Calculate the output and state vector
      for(i=1; i<M; i++)
      {
         yn[j] = yn[j] - K[i]*un-1[i-1];     //Output
         un[i] = K[i]*yn[j] + un-1[i-1];     //Update the state variable
      }

    }
    return;

}
```

Techniques

Assumptions
- Lattice coefficient vector must be located in DPRAM area.

Register Usage
- From .c file to .asm file:
  Decided by compiler
- From .asm file to .c file:

IIR_lat — Lattice IIR filter, 16 bit filter coefficients, Vector processing (cont'd)

Memory Note

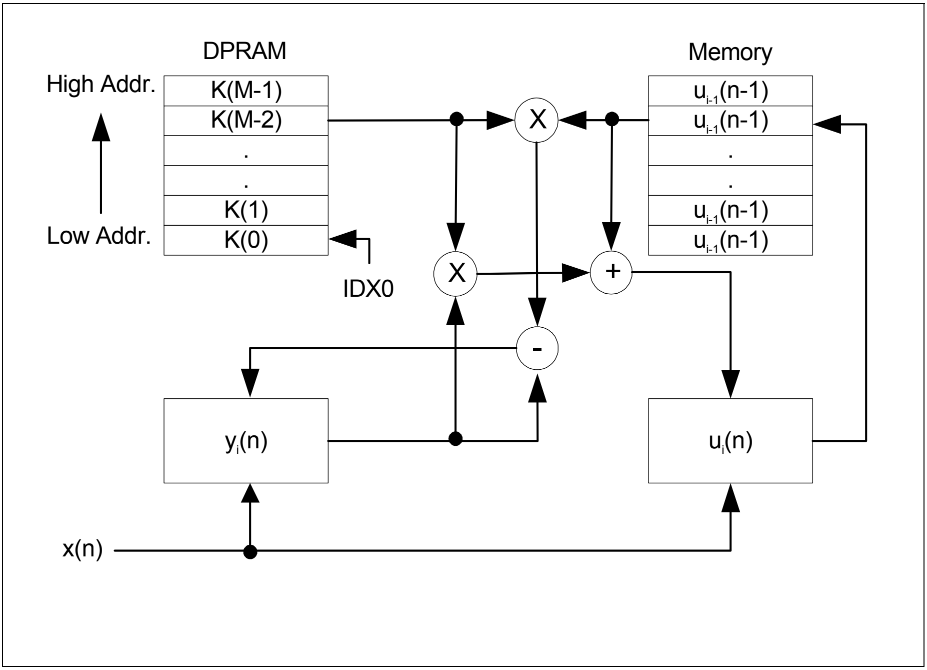

Figure 4-30 IIR_lat

Example *C166Lib\Keil\Examples\Filters\IIR\IIRLat.c*

Cycle Count

Read parameters	1
Memory initialization	9
IIR loop	N_x(11*M + 1)
Return	1

IIR_lat **Lattice IIR filter, 16 bit filter coefficients, Vector processing** (cont'd)

Total	**N_x(11*M + 1) + 11**
	Example: N_x = 1 M = 3 cycle = 45

Code Size

Read parameters	2	bytes
Mmeory initialization	18	bytes
FIR loop	66	bytes
Return	2	bytes
Total	**88**	bytes

4.5 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.

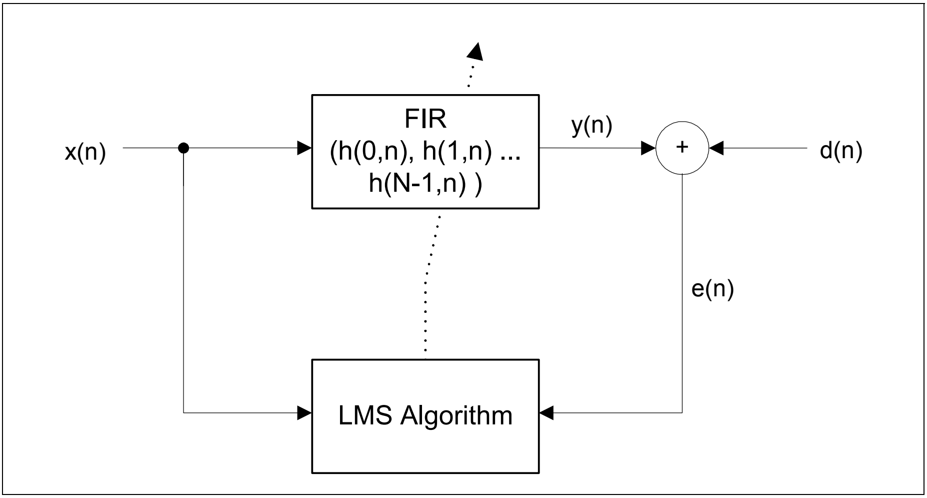

Figure 4-31 Adaptive filter with LMS algorithm

The filter part is an N-tap FIR filter with coefficients h(0,n), h(1,n),..., h(N-1,n), whose input signal is x(n) and output is y(n). The difference between the actual output y(n) and a desired output d(n), gives an error signal

$$e(n) = d(n) - y(n) \qquad [4.27]$$

The algorithm uses the input signal x(n) and the error signal e(n) to adjust the filter coefficients h(0,n), h(1,n),..., h(N-1,n), such that the difference, e(n) is minimized on a criterion. The LMS algorithm uses the minimum mean square error criterion

$$\min_{h(0,n),\, h(1,n),\ldots,\, h(N-1,n)} E(e^2(n)) \qquad [4.28]$$

Where E denotes statistical expectation. The algorithm of a regular (non-delayed) LMS adaptive filter is mathematically expressed as follows.

$$\begin{aligned} y(n) &= h(0,n)\cdot x(n) + h(1,n)\cdot x(n-1) + h(2,n)\cdot x(n-2) + \ldots \\ &\quad + h(N-1,n)\cdot x(n-N+1) \end{aligned} \qquad [4.29]$$

$$e(n) = d(n) - y(n) \qquad [4.30]$$

$$h(k, n+1) = h(k,n) + x(n-k) \times \mu \times e(n) \qquad [4.31]$$

where $\mu > 0$ is a constant called step-size. Note that the filter coefficients are time varying. h(i,n) denotes the value of the i-th coefficient at time n. The algorithm has three stages:

1. calculate the current filter output y(n).
2. calculate the current error value e(n) using the current expected value d(n) and currently calculated output y(n).
3. update the filter coefficients for next iteration using current error e(n) and samples x(n-k).

Step-size μ controls the convergence of the filter coefficients to the optimal (or stationary) state. The larger the μ value, the faster the convergence of the adaptation. On the other hand, a large value of μ also leads to a large variation of h(i,n) (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.

4.5.1 Delayed LMS algorithm for an adaptive filter

In the regular LMS adaptive filter the update of filter coefficients makes use of current error value and input value. This makes the choice of step-size μ more difficult due to the effect of μ on convergence of adaptive filter. To minimize the effect of μ on the filter convergence a delayed LMS algorithm for an adaptive filter is introduced. The algorithm of a delayed LMS adaptive filter can be represented by the following mathematical equations.

$$y(n) = \sum_{k=0}^{N-1} h(k, n) \cdot x(n-k) \qquad [4.32]$$

$$h(k, n+1) = h(k, n) + x(n-k-1) \times \mu \times e(n-1) \qquad [4.33]$$

$$e(n) = d(n) - y(n) \qquad [4.34]$$

where,

y(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(0,n),h(1,n),..	:	filter coefficients at index n
N	:	filter order (number of coefficients)
μ	:	step-size
e(n)	:	error value at index n

The algorithm has three stages:

1. calculate the current filter output y(n).
2. update filter coefficients for the next iteration using previous error value e(n-1) and the delayed input sample x(n-k-1).
3. calculate the current error value e(n) and store it in memory.

4.5.2 Descriptions

In the DSP library, the delayed LMS adaptive filters are implemented. The following are the implemented delayed LMS adaptive FIR filter functions with 16 bit input and 16 bit or 32 bit filter coefficients.

- 16 bit real coefficients, delayed LMS, Sample processing
- 32 bit real coefficients, delayed LMS, Sample processing

Adap_filter_16 — 16 bit real coefficients, Delayed LMS, Sample Processing

Signature

```
DataS Adap_filter_16 (  DataS*   h,
                        DataS*   IN,
                        DataS*   D,
                        DataS*   error,
                        DataS    N_h,
                        DataS    Step,
                        DataS*   d_buffer
                     )
```

Inputs

h	:	Pointer to filter coefficients in 1Q15
IN	:	Pointer to new input value
D	:	Pointer to the expected signal at time n
error	:	Pointer to error signal
N_h		Filter size
Step	:	Adaptive gain
d_buffer		Pointer to delay buffer

Return

Y	:	Output value of the filter in 1Q15

Implementation Description

Delayed LMS algorithm has been used to realize an adaptive FIR filter. That is, the update 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. In this routine, both of signals and filter coefficients have 16 bit precision.

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.

Adap_filter_16 — 16 bit real coefficients, Delayed LMS, Sample Processing (cont'd)

Pseudo code

```
{
    ; x(n) = input signal at time n
    ; d(n) = desired signal at time n
    ; y(n) = output signal at time n
    ; h(k,n) = k-th coefficient at time n
    ; gain = adaptive gain
    ; N = number of coefficient taps in the filter

    DataS  x(n);                //input signal
    DataS  d(n);                //desired signal
    DataS  h(k,n);              //filter coefficient
    DataS  y(n), Y;             //output
    DataS  k;

    ;Calculate the output at time n
    y(n) = 0;
    for(k=0; k<N_h; k++)
      y(n) = y(n) + h(k,n)*x(n-k);

    ;Update the filter coefficients
    for(k=0; k<N_h; k++)
      h(k,n+1) = h(k,n) + gain*e(n-1)*x(n-k-1);

    ;Error signal at time n
      e(n) = d(n) - y(n);

    Y = y(n);
    return Y;         //Filter Output returned

}
```

Techniques

- Memory bandwidth conflicts removing
- Use of CoMACM instructions
- Instruction re-ordering
- Filter output converted to 16-bit with saturation

Assumptions

- Delay buffer must be located in DPRAM area.

Register Usage

- From .c file to .asm file:

  Defined by the Compiler
- From .asm file to .c file:

  (R4) = Y

Adap_filter_16 16 bit real coefficients, Delayed LMS, Sample Processing (cont'd)

Memory Note

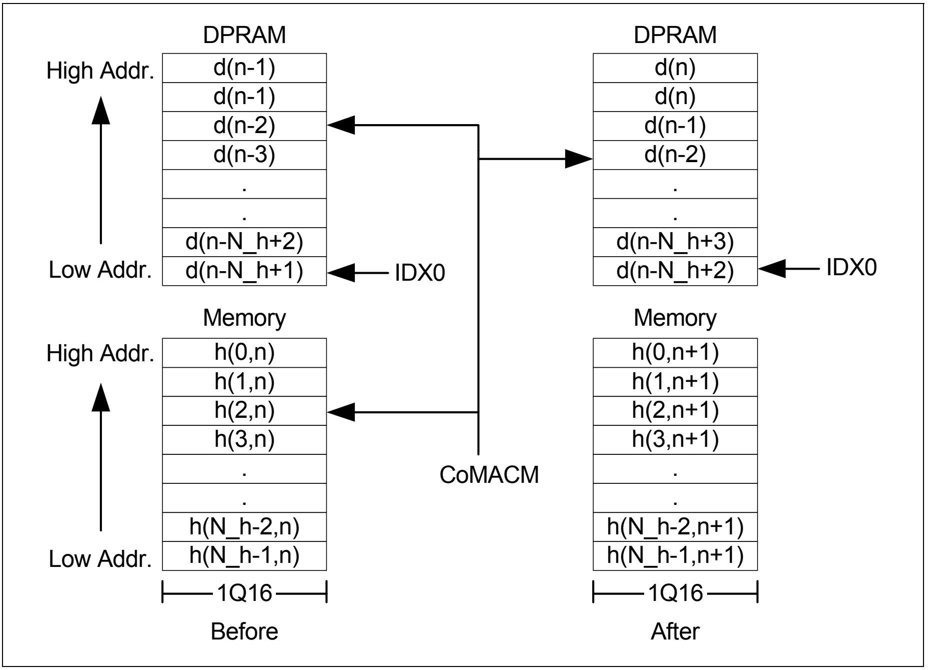

Figure 4-32 Adap_filter_16

Example *C166Lib\Keil\Example\Adaptive_filter\Ad_filter_16.c*

Cycle Count

Memory Initialization	13
Read the new input sample	3
Calculate the current output	N_h + 6

Adap_filter_16 16 bit real coefficients, Delayed LMS, Sample Processing (cont'd)

Filter coefficient adaptation	7N_h + 1
Calculation of the error signal	3
Total	**8*N_h + 26**
	Example: N_h = 12 cycle = 122

Code Size

Memory Initialization	26 bytes
Read the new input sample	12 bytes
Calculate the current output	32 bytes
Filter coefficient adaptation	52 bytes
Calculation of the error signal	6 bytes
Total	**128** bytes

Adap_filter_32 — 32 bit real coefficients, Delayed LMS, Sample Processing

Signature

```
DataS Adap_filter_32(  DataL*    h,
                       DataS*    IN,
                       DataS*    D,
                       DataS*    error,
                       DataS     N_h,
                       DataS     Step,
                       DataS*    d_buffer
                     )
```

Inputs

h	:	Pointer to filter coefficients in 1Q31
IN	:	Pointer to new input value
D	:	Pointer to the expected signal at time n
error	:	Pointer to error signal
N_h		Filter size
Step	:	Adaptive gain
d_buffer		Pointer to delay buffer

Return

Y	:	Output value of the filter

Implementation Description

LMS algorithm has been used to realize an adaptive FIR filter. That is, the update 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. Here, the filter coefficients have 32 bit precision, while input and output signals have 16 bit precision.

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.

Adap_filter_32 — 32 bit real coefficients, Delayed LMS, Sample Processing (cont'd)

Pseudo code

```
{
    ; x(n) = input signal at time n
    ; d(n) = desired signal at time n
    ; y(n) = output signal at time n
    ; h(k,n) = k-th coefficient at time n
    ; gain = adaptive gain
    ; N = number of coefficient taps in the filter

    DataS  x(n);              //input signal
    DataS  d(n);              //desired signal
    DataL  h(k,n);            //filter coefficient
    DataS  y(n), Y;           //output
    DataS  k;

    ;Calculate the output at time n
    y(n) = 0;
    for(k=0; k<N_h; k++)
      y(n) = y(n) + (short)h(k,n)*x(n-k);

    ;Update the filter coefficients
    for(k=0; k<N_h; k++)
      h(k,n+1) = h(k,n) + (long)gain*e(n)*x(n-k);

    ;Error signal at time n
      e(n) = d(n) - y(n);

    Y = y(n);
    return Y;          //Filter Output returned

}
```

Techniques

- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering
- Filter output converted to 16-bit with saturation

Assumptions

- Delay buffer must be located in DPRAM area.

Register Usage

- From .c file to .asm file:
  Defined by the compiler
- From .asm file to .c file:

  (R4) = Y

Adap_filter_32 — 32 bit real coefficients, Delayed LMS, Sample Processing (cont'd)

Memory Note

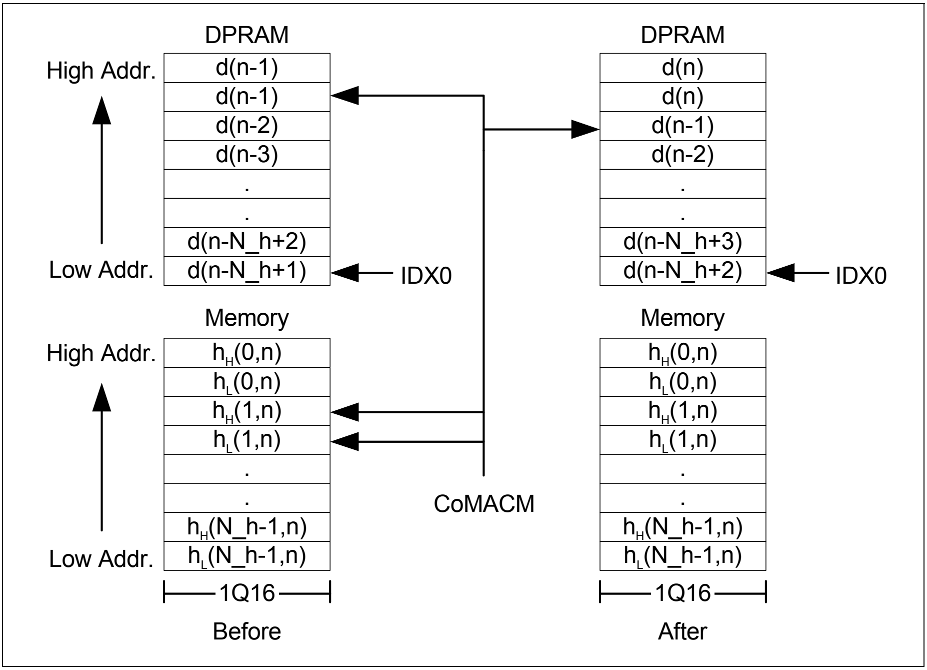

Figure 4-33 Adap_filter_32

Example *C166Lib\Keil\Example\Adaptive_filter\Ad_filter_32.c*

Cycle Count

Memory Initialization	20
Read the new input	3
Calculation of the current output y(n) (LSW)	N_h + 3

Adap_filter_32 — 32 bit real coefficients, Delayed LMS, Sample Processing (cont'd)

Calculation of the current output y(n) (MSW)	N_h + 3
Filter coefficient adaptation	6N_h +1
Calculation of the error signal	4
Total	**8*N_h + 34**
	Example: N_h = 12 cycle = 130

Code Size

Memory Initialization	40 bytes
Read the new input	12 bytes
Calculation of the current output y(n) (LSW)	24 bytes
Calculation of the current output y(n) (MSW)	26 bytes
Filter coefficient adaptation	48 bytes
Calculation of the error signal	10 bytes
Total	**160** bytes

4.6 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. The DFT is defined by

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \qquad [4.35]$$

where

$$W_N = e^{-j2\pi/N} = \cos(2\pi nk/N) - j\sin(2\pi nk/N) \quad . \qquad [4.36]$$

Using **Equation [4.36]** in **Equation [4.35]** we can rewrite X(k) as

$$X(k) = \sum_{n=0}^{N-1} x(n)[\cos(2\pi nk/N) - j\sin(2\pi nk/N)] \qquad [4.37]$$

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

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k) e^{j2\pi nk/N} \quad \text{[exponential form]} \qquad [4.38]$$

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)[\cos(2\pi nk/N) + j\sin(2\pi nk/N)] \qquad [4.39]$$

Notice the difference between DFT in **Equation [4.37]** and **Equation [4.39]** , 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.

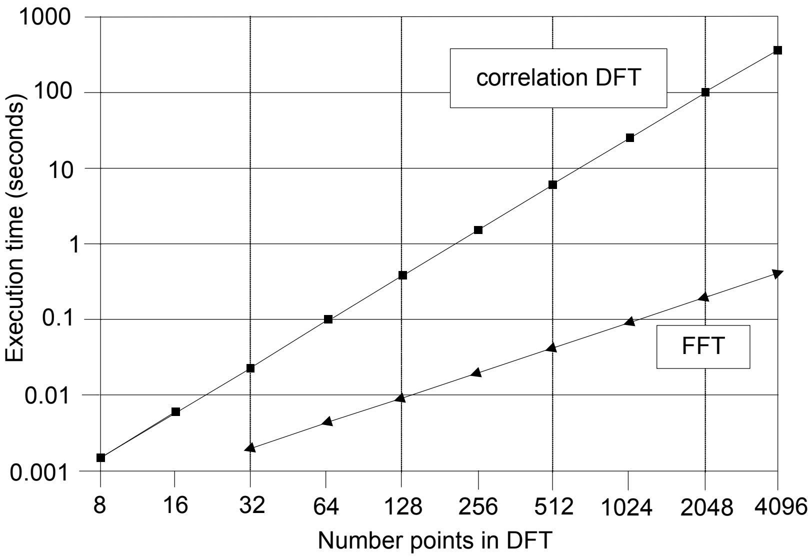

Figure 4-34 Complexity Graph

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.

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 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. In this library the basic radix-2 decimation-in-frequency FFT algorithm is implemented.

4.6.1 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 [4.35]** is divided into even and odd sub-sequences.

$$\begin{aligned} X(k) &= \sum_{n=0}^{\frac{N}{2}-1} x(2n)W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)W_N^{(2n+1)k} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(2n)W_N^{2nk} + W_N{}^k \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)W_N^{2nk} \end{aligned} \qquad \text{for k=0 to N-1} \qquad [4.40]$$

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 [4.40]**

h(n)=x(2n)

g(n)=x(2n+1) ,

Equation [4.40] becomes

$$\begin{aligned} X(k) &= \sum_{n=0}^{N/2-1} h(n)W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} g(n)W_{N/2}^{nk} \\ &= H(k) + W_N{}^k G(k) \end{aligned} \qquad \text{for k=0 to N-1} \qquad [4.41]$$

Equation [4.41] is the radix-2 DIT FFT equation. It consists of two N/2-point DFTs H(k) and G(k) performed on the subsequences of even and odd samples of the input sequence x(n), respectively. Multiples of W_N, the Twiddle-Factors are the coefficients in the FFT calculation.

Further,

$$W_N{}^{k+N/2} = (e^{-j2\pi/N})^k \times (e^{-j2\pi/N})^{N/2} = -W_N{}^k . \qquad [4.42]$$

Equation [4.41] can be expressed in two equations

$$X(k) = H(k) + W_N^{\ k} G(k) \qquad [4.43]$$

$$X(k + N/2) = H(k) - W_N^{\ k} G(k) \quad \text{for k=0 to N/2-1} \qquad [4.44]$$

The decomposition procedure can be continued until two-point DFTs are reached. **Figure 4-36** illustrates the flow graph of a real 8-point DIT FFT.

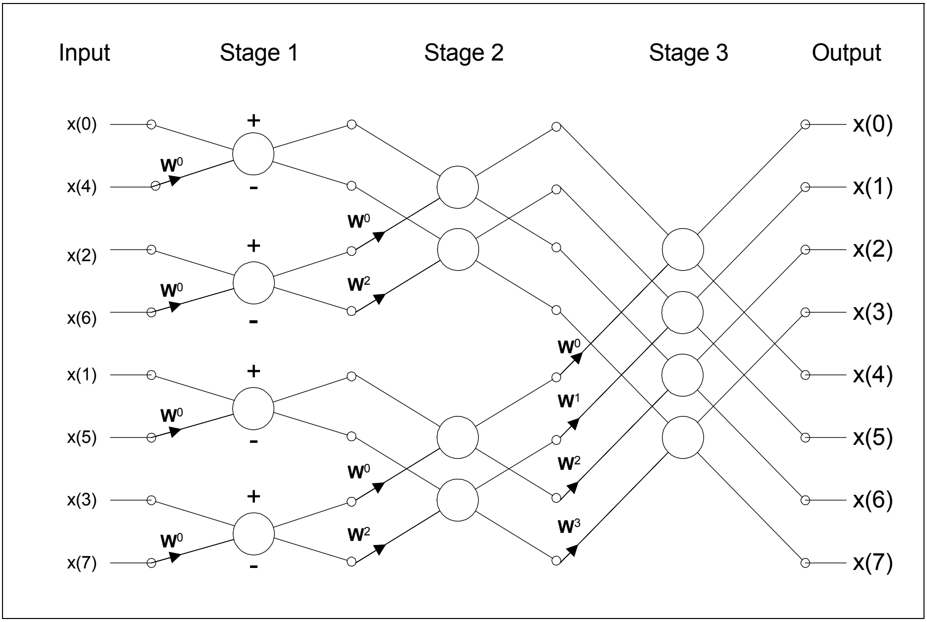

Figure 4-35 8-point DIT FFT

Note that the input sequence x(n) in **Figure 4-35** is in the scrabbled order. The same procedure can be repeated with the linear input order. Then we have the alternate form of the DIT FFT shown in **Figure 4-36** , where the output sequence X(n) is rearranged to appear in bit-reversed order. The only difference between **Figure 4-35** and **Figure 4-36** is a rearrangement of the butterflies and the computational load and final results are identical. In the library the implementation of real-valued forward FFT is based on **Figure 4-36** .

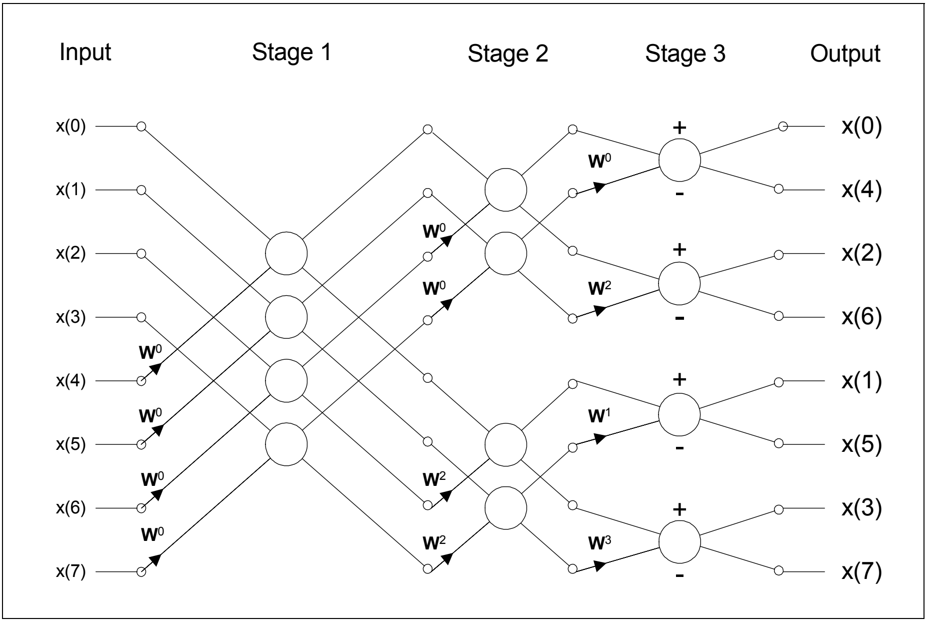

Figure 4-36 Alternate Form of 8-point DIT FFT

4.6.2 Radix-2 Decimation-In-Frequency FFT Algorithm

A second variant of the radix-2 FFT is the decimation-in-frequency algorithm. In order to get this algorithm, we split the input sequence into the first and second halves and write the DFT as

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) W_N^{nk} \\ &= \sum_{n=0}^{N/2-1} x(n) W_N^{nk} + \sum_{n=0}^{N/2-1} x(n+N/2) W_N^{(n+N/2)k} \\ &= \sum_{n=0}^{N/2-1} [x(n) + (-1)^k x(n+N/2)] W_N^{nk} \end{aligned} \quad , \quad \text{for k=0 to N.} \qquad [4.45]$$

For the even and odd numbered DFT points we get

$$X(2k) = \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)] W_{N/2}^{nk} \qquad [4.46]$$

$$X(2k+1) = \sum_{n=0}^{N/2-1} [x(n) - x(n+N/2)] W_N^n W_{N/2}^{nk} \quad \text{, for k=0 to N/2.} \qquad [4.47]$$

As with the decimation-in-time algorithm, the N-point DFT is decomposed into two N/2-point DFTs. Using the principle repeatedly results in an FFT algorithm where the input values appear in their natural order, but where the output values appear in bit reversed order. The complexity is the same as for the decimation-in-time FFT. **Figure 4-37** shows the flow graph of an 8-point DIF FFT. The comparison of **Figure 4-35** and **Figure 4-37** shows that the two graphs can be viewed as transposed versions of one another.

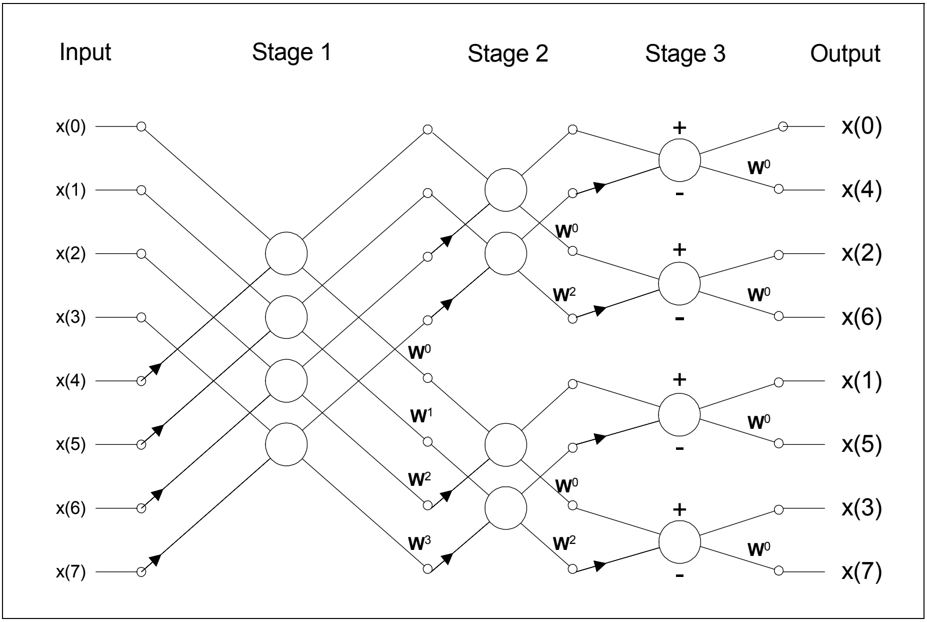

Figure 4-37 8-point DIF FFT

Similar to FFT with decimation-in-time, if the inputs in **Figure 4-37** are rearranged in bit-inverse order and the outputs in linear order, we have an alternate implementation for DIF FFT showed in **Figure 4-38** .

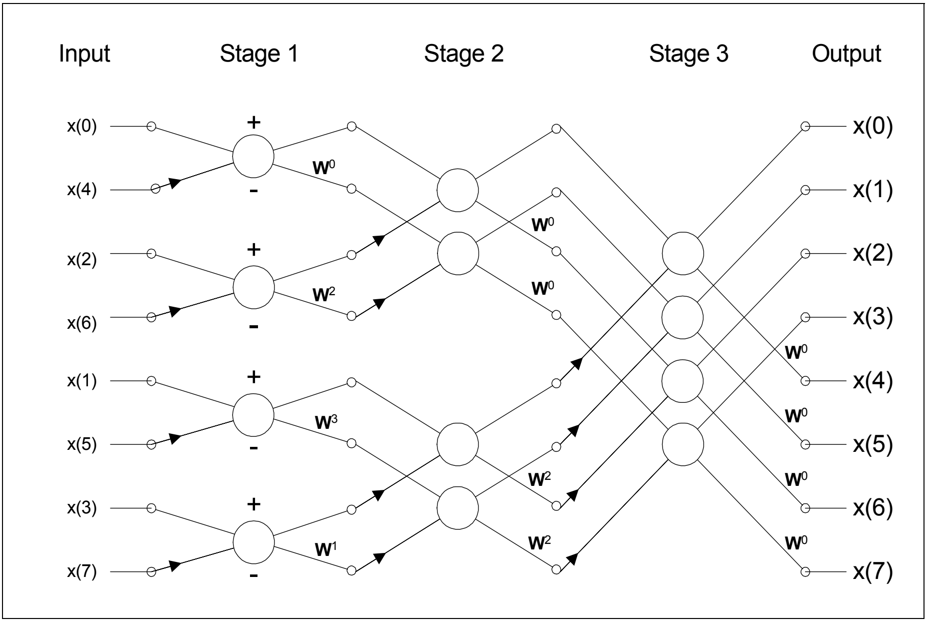

Figure 4-38 Alternate Form of 8-point DIF FFT

4.6.3 Complex FFT Algorithm

If the input sequence is complex, according to **Equation [4.43]** and **Equation [4.44]**, we can write the complex DFT as

$$X(k) = P(k) + W_N^k Q(k)$$

$$X(k + N/2) = P(k) - W_N^k Q(k) \qquad \text{for k=0 to N/2-1,} \qquad [4.48]$$

where P(k) and Q(k) are the complex even and odd partial DFTs. As same as real FFT, the complex FFT has also two implementations, i.e. decimation-in-time and decimation-in-frequency complex FFT. **Figure 4-39** shows an 8-point DIT complex FFT implementation.

In **Figure 4-39**, each pair of arrows represents a Butterfly. The whole of the complex FFT is computed by different patterns of Butterflies. These are called groups and stages.

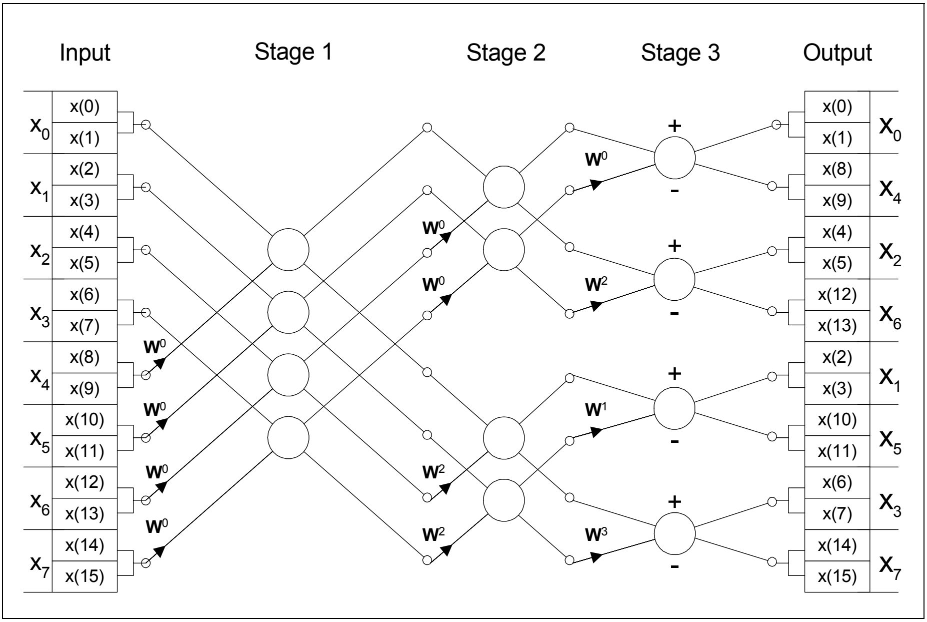

Figure 4-39 8-point DIT complex FFT

For 8-point DIT FFT the first stage consists of one groups of four Butterfly, second consists of two groups of two butterflies and third stage has four group of one Butterflies. Each Butterfly is represented as in **Figure 4-40** .

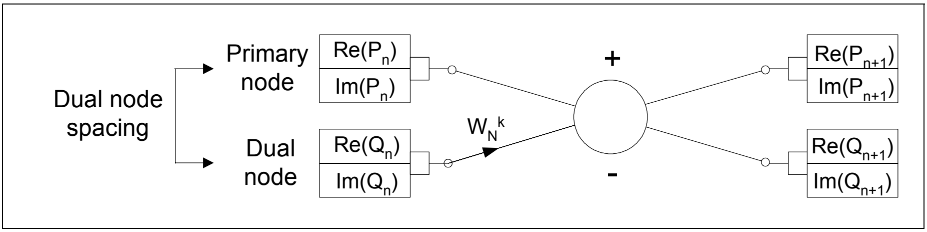

Figure 4-40 Butterfly of Radix-2 DIT complex FFT

The output is derived as follows

$$P_{n+1} = P_n + Q_n \cdot W_N^k \qquad [4.49]$$

$$Q_{n+1} = P_n - Q_n \cdot W_N^k \quad , \qquad [4.50]$$

where n represents the number of stages.

Of course, the complex FFT can also be implemented with decimation-in-frequency FFT showed in **Figure 4-38** . In this case, an 8-point complex FFT has the implementation structure depicted in **Figure 4-41** .

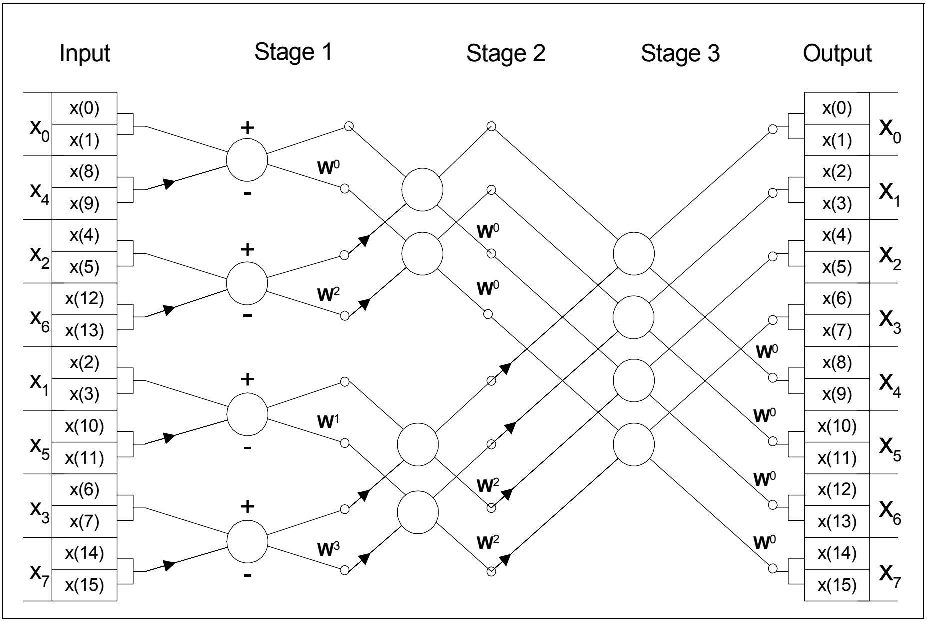

Figure 4-41 8-point DIF complex FFT

The corresponding butterfly is illustrated in **Figure 4-42**, in that the input-output relationship writes

$$P_{n+1} = P_n + Q_n \qquad [4.51]$$

$$Q_{n+1} = (P_n - Q_n) \cdot W_N^k \qquad [4.52]$$

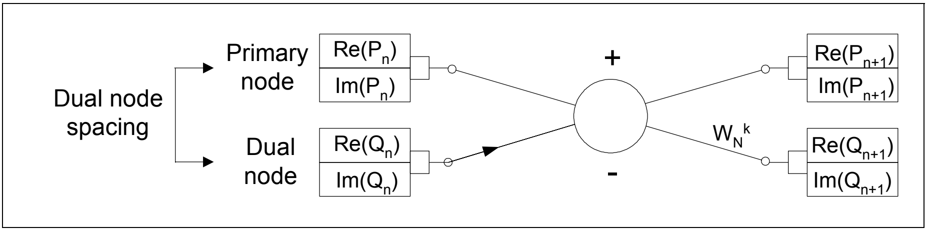

Figure 4-42 Butterfly of Radix-2 DIF complex FFT

4.6.4 Calculation of Real Forward FFT from Complex FFT

Having only real valued input data x(n), the computational effort of a N point real FFT can be reduced to a N/2 point complex FFT. Firstly, even indexed data h(n)=x(2n) and odd indexed data g(n)=x(2n+1) are separated. According to **Equation [4.43]** and **Equation [4.44]**, The spectrum X(k) can be decomposed into the spectra H(k) and G(k) as follows:

$$X(k) = H(k) + \left(\cos\frac{2\pi k}{N} - j\sin\frac{2\pi k}{N}\right)G(k) \qquad \text{for k=0 to N/2-1.} \qquad [4.53]$$

In order to cut the N-point real FFT into N/2-point complex transformation a complex input vector y(n) = h(n)+jg(n) is formed with the index n running from 0 to N/2-1. The real part values are formed by the even indexed input data h(n). The imaginary part is formed by the odd indexed input data g(n). Then y(n) is transformed into the frequency domain resulting in a spectrum consisting of the spectra H(k) and G(k).

$$Y(k) = H(k) + jG(k) = \mathrm{Re}\{Y(k)\} + j\mathrm{Im}\{Y(k)\} \qquad [4.54]$$

Now the complex spectra H(k) and G(k) have to be extracted out of the complex spectrum Y(k). By employing symmetry relations, the spectra H(k) and G(k) can be derived from the spectrum Y(k) as follows:

$$\begin{aligned}
\mathrm{Re}\{H(k)\} &= \frac{\mathrm{Re}\{Y(N/2-k)\} + \mathrm{Re}\{Y(k)\}}{2} \\
\mathrm{Im}\{H(k)\} &= \frac{\mathrm{Im}\{Y(k)\} - \mathrm{Im}\{Y(N/2-k)\}}{2} \\
\mathrm{Re}\{G(k)\} &= \frac{\mathrm{Im}\{Y(k)\} + \mathrm{Im}\{Y(N/2-k)\}}{2} \\
\mathrm{Im}\{G(k)\} &= \frac{\mathrm{Re}\{Y(N/2-k)\} - \mathrm{Re}\{Y(k)\}}{2}
\end{aligned} \qquad [4.55]$$

Therefore, computing an N-point real forward FFT has the following steps:

1. Generating the N/2 point complex sequence $y(n)=x(2n)+jx(2n+1)$,
2. Computing the complex spectra Y(k) using complex FFT,
3. Extracting H(k) and G(k) from computed Y(k) according to ,**Equation [4.55]**
4. Calculating the first N/2+1 points of X(k) based on **Equation [4.53]** using H(k) and G(k),
5. Using the symmetry relation of the spectra of the real sequence to get the complete N-point X(k).

4.6.5 Calculation of Real Inverse FFT from Complex FFT

Using the algorithm in **Section 4.6.4** we can calculate a N-point real-valued FFT through N/2-point complex FFT. Now we present the corresponding inverse FFT algorithm to repeat the original real-valued input sequences based on the FFT spectrum. Usually a N-point real-valued forward FFT produces N-point complex spectrum. To get the original real-valued sequences one can simply input this N-point complex spectrum to an inverse FFT. But it needs N-point real inverse FFT. With the symmetry properties of real-valued FFT a N-point real inverse FFT can be reduced to N/2-point complex inverse FFT.

This can be realized through two steps. In the first step the real FFT spectrum will be unpacked to the corresponding complex spectrum similar with the unpack stage in the forward FFT. Then, the results are inputted into a N/2-point inverse complex FFT to get real-valued sequences.

In the unpack stage only the first (N/2+1)-point spectrum are needed because the first (N/2+1)-point spectrum of a N-point real-valued FFT contain all information. According to **Equation [4.53]** and the symmetry properties of H(k) and G(k), we have

$$\begin{aligned} \mathrm{Re}\{H(k)\} &= \frac{\mathrm{Re}\{X(k)\} + \mathrm{Re}\{X(N/2-k)\}}{2} \\ \mathrm{Im}\{H(k)\} &= \frac{\mathrm{Im}\{X(k)\} - \mathrm{Im}\{X(N/2-k)\}}{2} \end{aligned} \qquad \text{for } k=0 \text{ to } N/2-1. \qquad [4.56]$$

Defining

$$G'(k) = X(k) - H(k) \ , \qquad [4.57]$$

and replacing **Equation [4.56]** into **Equation [4.57]** we get

$$\mathrm{Re}\{G'(k)\} = \frac{\mathrm{Re}\{X(k)\} - \mathrm{Re}\{X(N/2-k)\}}{2}$$
$$\mathrm{Im}\{G'(k)\} = \frac{\mathrm{Im}\{X(k)\} + \mathrm{Im}\{X(N/2-k)\}}{2} \qquad [4.58]$$

Finally we have the expression of the complex spectrum Y(k) from **Equation [4.54]**

$$\mathrm{Re}\{Y(k)\} = \mathrm{Re}\{H(k)\} - \sin\frac{2\pi k}{N}\cdot\mathrm{Re}\{G'(k)\} - \cos\frac{2\pi k}{N}\cdot\mathrm{Im}\{G'(k)\}$$
$$\mathrm{Im}\{Y(k)\} = \mathrm{Im}\{H(k)\} + \cos\frac{2\pi k}{N}\cdot\mathrm{Re}\{G'(k)\} - \sin\frac{2\pi k}{N}\cdot\mathrm{Im}\{G'(k)\} \qquad [4.59]$$

where K=0 to N/2-1.

Summarily, computing an N-point real inverse FFT has the following steps:

1. Extracting H(k) according to **Equation [4.56]** ,
2. Extracting G'(k) according to **Equation [4.58]**,
3. Computing the complex spectrum Y(k) based on H(k) and G'(k) using **Equation [4.59]**,
4. Calculating N/2-point inverse complex FFT with input Y(k).

4.7 C166S V2 Core Implementation Note

4.7.1 Organization of FFT functions

In the library the radix-2 FFT is implemented. Basically there are following three kinds of functions:

- Bit reverse function (Bit_reverse.c)
- Floating point to 1Q15 format function (FloatTo1Q15.c)
- FFT kernel function

The first two functions will be as subroutines called by FFT kernel function. The subroutine Bit_reverse.c is used for bit reversing the binary presentation of the input indices to get the output data indices. FloatTo1Q15.c is used to change the floating point format to 1Q15 fractional format. The kernel FFT realizes the kernel FFT algorithm that may be complex and real FFT. Butterflies are implemented in the form of macros.

The kernel FFT implementation in the C166S V2 Core Library is based on an application note performed early by Siemens HL MCB AT, in which an implementation of a real-valued 1024-point radix-2 decimation-in-time forward FFT for C166 microcontroller family is described. However, there are many basic differences between the two implementations. Firstly, the FFT implementation for C166S V2 Core is not only C-callable but also Assembler-callable, while the early implementation for C166 is only Assembler-callable. Secondly, the FFT implementation for C166S V2 Core is performed with the new DSP MAC instruction set, while the early implementation for C166 has only used the normal instruction set. Therefore, the FFT implementation for C166S V2 Core is a more optimal implementation in comparison with early implementation. Thirdly, the size of input data of the new FFT implementation for C166S V2 Core can be changed from 2 to 2048 points, while the early implementation is only suitable for 1024-point FFT.

This library provides not only forward FFT but also inverse FFT implementation as well as their implementation theory basis.

4.7.2 Implementation of Real Forward FFT

A real valued N point FFT can be reduced to a N/2 point complex FFT followed by an unweave phase in order to compute the N/2 point spectrum. The N/2 complex FFT consists of $\log_2(N/2)$ stages and each stage calculates N/4 butterflies.

The input data is stored in the vector FFT_in that consists of N 16 bit words and is defined in C main function. Since we perform an in-place FFT, the input data field will be overwritten by the output data. For providing the trigonometric functions, the precomputed sinus and cosines values are stored in memory. Because the input data and the trigonometric function values are represented by a 15 bit fixed-point fraction in two’s complement (1Q15 format), the floating-point sinus and cosines values have to be changed into 1Q15 format with the routine FloatToQ15. To rearrange the bit reversed output of the complex FFT and to calculate the twiddle factor Wk, a bit reversal index table has been precomputed.

Regarding the N/2 point complex FFT, the number of twiddle factors amounts N/2. However, due to the symmetry only the first N/4 are used. The implementation consists of $\log_2(N/2)$ stages. Each stage contains basically three loops, i.e. Outloop, Mitloop and Inloop. One stage has one Outloop and N/2 Inloops, but different Mitloops due to different twiddle factors. Each Inloop implements one butterfly with a twiddle factor. For example, an 8-point DIT complex FFT in **Figure 4-39** has the following structure:

Stage 1 (Outloop_1): 4 butterflies with W_0

Stage 2 (Outloop_2): 2 butterflies with W_0

2 butterflies with W_2

Stage 3 (Outloop_3): 1 butterfly with W0

1 butterfly with W1

1 butterfly with W2

1 butterfly with W3

The output of the decimation-in-time FFT shows a bit reversed order that has to be ordered to calculate the final frequency spectrum. Supposing the input data has been in a sequential order, the indices of the output data can be easily computed by bit reversing the binary presentation of the input indices. The table below gives an example of the bit reversal for an 8-point FFT.

Order of input data		Order of output data	
index	binary	binary	index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

The second part of the program unweaves the bit reversed output of the N/2 point FFT to extract the N point real valued FFT.

4.7.3 Implementation of Real Inverse FFT

From **Section 4.6.5** we know that the N-point real inverse FFT can be realized by N/2 point complex inverse FFT. Different from forward FFT the unpack stage will be performed at beginning to get the N/2 point complex input spectra according to N point real FFT spectra.

The input data is the first (N/2+1)-point real FFT spectra and stored in the vector with size N/2+1 that consists of N+2 16 bit words and is defined in C main function. After unpacking the N/2 point complex spectra are strode in vector FFT_out. Note that the input data should be by 1/N scaled real FFT spectra. All data has 1Q15 format. Here we use the name real inverse FFT, which doesn't mean that the input data is real value. Actually here the input data is complex value coming from a real-valued FFT. The word real indicates that the input value come from a real-valued forward FFT instead of a complex forward FFT.

The butterfly of inverse FFT has the same structure as forward FFT beside the twiddle factor. The twiddle factor of the inverse FFT butterfly has the form W_N^{-nk} instead of W_N^{nk} . Similar with forward FFT the inverse FFT can be also implemented with DIT and DIF. For example, for an 8-point real inverse FFT, if the DIF implementation in **Figure 4-41** is used, there are following structure:

Stage 1 (Outloop_1): 1 butterfly with W0
1 butterfly with W1
1 butterfly with W2
1 butterfly with W3
Stage 2 (Outloop_2): 2 butterflies with W_0
2 butterflies with W_2
Stage 3 (Outloop_3): 4 butterflies with W_0

In this case the output shows the linear order while the input has bit-reversed order. The corresponding butterfly has the structure showed in **Figure 4-42**.

4.7.4 Description

The following functions are described.

- Bit_reverse.c
- FloatTo1Q15.c
- real_DIT_FFT.c
- real_DIF_IFFT.a66
- Q15toFloat.c

Bit_reverse — Reverse the binary bit of the input data

Signature DataS Bit_reverse(DataS* X, DataS N)

Inputs	X	:	16 bit Input data vector
	N	:	Size of the vector, $N=2^n$ (n<12)
Output	X	:	Bit reversed data vector
Return	n	:	Exponent of N

Implementation Description

This routine is a subroutine in the DIT FFT implementation packet and used to obtain a bit reversed data in respect to the input data. The bit reversed data is also stored in the vector X.

For an 8 bit input data there is following algorithm:

Before bit reverse: b7.b6.b5.b4.b3.b2.b1.b0
After bit reverse: b0.b1.b2.b3.b4.b5.b6.b7

Pseudo code

```
{
    ; X = input/output data vector
    ; N = the size of the vector, N=2^n (n<12)

    DataS*  X;            //input/output vector
    DataS   N;            //size of vector N
    DataS   i, k, n;

    for(k=0; k<N; k++)
    {
      temp = 0;
      for(i=15; i>0; i--)
         temp = temp + (X(k)<<i)>>(15-i);
    //write the output into X
      X(k) = temp;
    }

    n = (DataS)log10(N);

    Y = n;
    return Y;         //Filter Output returned


}
```

Techniques

Bit_reverse **Reverse the binary bit of the input data** (cont'd)

Assumptions
- 16 bit Input data vector X
- $N = 2^n$ (n < 12)

Register Usage
- From .c file to .asm file:
  Defined by the compiler
- From .asm file to .c file:
  (R4) = n (exponent of N)

Memory Note

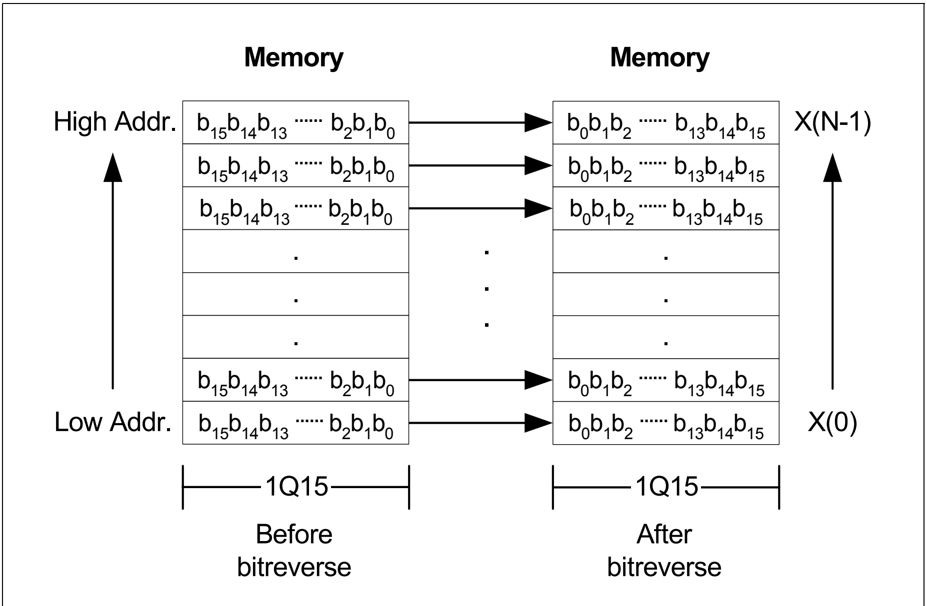

Figure 4-43 Bit_reverse

Example *C166Lib\Keil\Examples\FFT\real_FFT\bit_rev.c*

Cycle Count

Exponent determination	6
Bit reverse:	

Bit_reverse **Reverse the binary bit of the input data** (cont'd)

if $N=2^1$	3
if $N=2^2$, $N=2^3$	10
if $N=2^4$, $N=2^5$	12
if $N=2^6$, $N=2^7$	14
if $N=2^8$, $N=2^9$	16
if $N=2^{10}$, $N=2^{11}$	18
Restore state	3
Return	1
Total	

Code Size

Exponent determination	12 bytes
Bit reverse:	
if $N=2^1$	6 bytes
if $N=2^2$, $N=2^3$	20 bytes
if $N=2^4$, $N=2^5$	24 bytes
if $N=2^6$, $N=2^7$	28 bytes
if $N=2^8$, $N=2^9$	32 bytes
if $N=2^{10}$, $N=2^{11}$	36 bytes
Restore state	6 bytes
Return	2 bytes
Total	**304** bytes

FloatTo1Q15 — Change the floating point format to fixed point format 1Q15

Signature DataS FloatTo1Q15 (float x)

Inputs x : Input in float format

Output :

Return : Output in 1Q15 format

Implementation Description

This routine is a subroutine in the DIT FFT implementation packet and used to change the floating point data into a 1Q15 fixed point format.

The floating point format has the structure:

1. word: s e e e e e e e e mmmmmmm
2. word: mmmmmmmmmmmmmmmm

where s =sign, e=exponent, m=mantissa. After format change we have the 1Q15 fixed point data with the structure:

s.	b1	b2	b3	b4	......	b15
sig.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	......	2^{-15}

For example:

binary	*hex*	*value*
0111 1111 1111 1111	7FFF	+1
0110 0000 0000 0000	6000	+0.75
1010 0000 0000 0000	A000	- 0.75
1000 0000 0000 0000	8000	-1

Pseudo code

Assumption DPRAM begins with the address # f200h (it can be changed)

Register Usage

- From .c file to .asm file:
  Defined by the Compiler
- From .asm file to .c file:
  The output is stored in R4.

FloatTo1Q15 — Change the floating point format to fixed point format 1Q15 (cont'd)

Memory Note

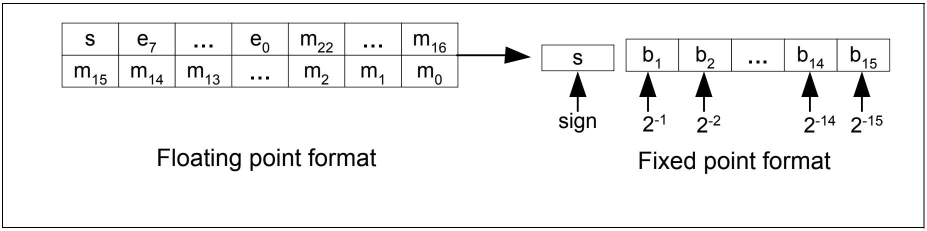

Figure 4-44 FloatTo1Q15

Example *C166Lib\Keil\Examples\Misc\Float_1Q15.c*

Cycle Count

Read exponent	10
Read mantissa	7
Output	12
Return	1
Total	**30**

Code SIze

Read exponent	20 bytes
Read mantissa	14 bytes
Output	20 bytes
Return	2 bytes
Total	**56** bytes

real_DIT_FFT — Real Forward Radix-2 Decimation-in-Time Fast FourierTransformation

Signature

```
void real_DIT_FFT ( DataS*   x,
                    DataS*   index,
                    DataS    exp,
                    DataS*   table,
                    DataS*   X
                  )
```

Inputs	x	:	16 bit real input vector
	index		Bit reversed input index vector
	exp	:	Exponent of the input block size
	table	:	The precalculated trigonometric function (sinus and cosine) table
Output	X	:	The FFT output vector in 1Q15 format
Return		:	

Implementation Description

This function computes the N-point real forward radix-2 decimation-in-time Fast Fourier Transform on the given N-point real input array. The detailed implementation is given in the **Section 4.7.2** .

The function is implemented as a complex FFT of size N/2 followed by a unpack stage to unpack the real FFT results. Normally an FFT of a real sequence of size N produces a complex sequence of size N or 2N real numbers that will not fit in the input sequence. To accommodate all the results without requiring extra memory locations, the output reflects only half of the complex spectrum plus the spectrum at the nyquist point (N/2). This still provides the full information because an FFT of a real sequence has even symmetry around the nyquist point.

real_DIT_FFT Real Forward Radix-2 Decimation-in-Time Fast FourierTransformation (cont'd)

Pseudo code

```
{
    DataS*  table;          //sinus and cosine table
    DataS*  x;              //16 bit real input vector
    CplxS*  X;             //FFT output vector
    CplxS   P(n), P(n+1), Q(n), Q(n+1), Y(N/2), H(N/2), G(N/2);
    DataS   k,i,n;

    //building the complex sequences P(n) and Q(n)
    Q(n) = x(2n) + jx(2n+1);
    P(n) = x(2n+N/4) + jx(2n+N/4+1);

    //Outloop = 1 to exp=log2(N/2)
    for(k=0; k++; k<exp)
    {
      //Midloop = 1 to N/4 according to which stage
      for(i=0; i<Midloop; i++)
      {
         //Inloop = N/2 to 1 (number of butterflies)
         for(n=0; n<Inloop; n++)
         {
            P(n+1) = Re(P(n))+Re(Q(n))*cos(X)+Im(Q(n))*sin(X)
                     +j[Im(P(n))-Im(Q(n))*cos(X)-Re(Q(n)*sin/X)];
            Q(n+1) = Re(P(n))-Re(Q(n))*cos(X)-Im(Q(n))*sin(X)
                     +j[Im(P(n))-Im(Q(n))*cos(X)-Re(Q(n)*sin/X)];
         }
         //if all elements processed, jump out of the Midloop
      }
    }
    //output the first half of N/2 point complex FFT values Y,
    //Extracting H and G from computed Y according to Equation [4.55],
    //Calculating the first N/2 points of X based on
      Equation [4.53] using H and G

}
```

Assumption
- Scale factor = 1/N ($N = 2^{exp}$), preventing overflow
- FFT spectra = N * X(k)

Techniques
- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering
- output converted to 16-bit with saturation

real_DIT_FFT **Real Forward Radix-2 Decimation-in-Time Fast FourierTransformation** (cont'd)

Register Usage

- From .c file to .asm file:

  R8 points to input vector x

  R9 points to index vector

  (R10) = exp

  R11 points to the trigonometric function table

  R12 contains the pointer of output vector X

real_DIT_FFT Real Forward Radix-2 Decimation-in-Time Fast FourierTransformation (cont'd)

Memory Note

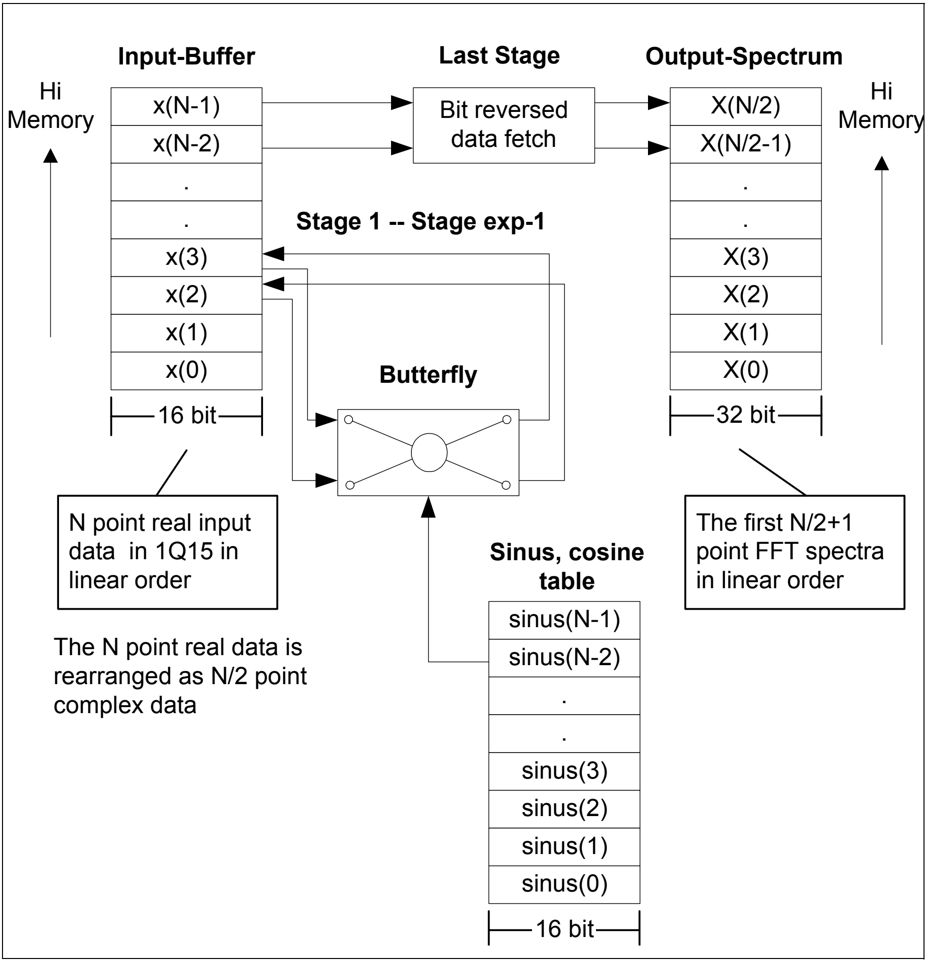

Figure 4-45 Real_DIT_FFT

real_DIT_FFT — Real Forward Radix-2 Decimation-in-Time Fast FourierTransformation (cont'd)

Example *C166Lib\Keil\Examples\FFT\real_FFT\ real_T_FFT.c*

Cycle Count

Store state	3
Read parameters and memory Initialization	12
Butterfly	31
FFT kernel (FFT_cycle)	$\sum_{n=0}^{exp-2}\left\{6 + 2^{n} \bullet 29 + \frac{N}{4} \bullet 36\right\}$ where exp=Log_2N.
Unpack stage (U_cycle)	71 + (N/4-1)*44
Restore state	3
Return	1
Total	**19 + FFT_cycle + U_cycle**

Examples:
N = 8 : cycle = 360
N = 16 : cycle = 876
N = 1024: cycle = 109111

Code Size

Store state	6 bytes
Read parameters and memory initialization	24 bytes
Butterfly	96 bytes
FFT kernel	88 bytes

real_DIT_FFT **Real Forward Radix-2 Decimation-in-Time Fast FourierTransformation** (cont'd)

Unpack Stage	178	bytes
Restore state	6	bytes
Return	2	bytes
Total	**400**	bytes

real_DIF_IFFT **Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation**

Signature	void real_DIF_IFFT (DataS* x, DataS* index, DataS exp, DataS* table, DataS* X)		
Inputs	index	:	Bit reversed input index vector
	exp		Exponent of the input block size
	table	:	The precalculated trigonometric function (sinus and cosine) table
	X	:	Input vector, FFT spectra in 1Q15 format
Output	x	:	16 bit output of inverse FFT, real sequences
Return		:	

Implementation Description

This function computes the N-point real inverse radix-2 Fast Fourier Transform with decimation-in-frequency on the given N-point FFT spectra. The detailed implementation is given in the **Section 4.7.3**.

The function is implemented as a unpack stage followed by a complex inverse FFT of size N/2. The unpack stage aims to unpack the N-point real FFT as N/2-point complex FFT. The N/2-point complex inverse FFT is implemented with decimation-in-frequency structure showed in **Figure 4-41**, where the butterfly W_N^{nk} should be replaced by W_N^{-nk} .

real_DIF_IFFT Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation (cont'd)

Pseudo code

```
{
    CplxS*  X;             //input FFT spectra
    DataS*  x;             //16 bit real output vector
    CplxS   P(n), P(n+1), Q(n), Q(n+1), Y(N/2), H(N/2), G'(N/2);
    DataS   k,i,n;

    //extracting H(k) according to Equation [4.56]
    Re{H(n)} = {Re{X(n)} + Re{X(N/2-n)}}/2;
    Im{H(n)} = {Im{X(n)} - Im{X(N/2-n)}}/2;

    //extracting G'(k) according to Equation [4.58]
    Re{G'(n)} = {Re{X(n)} - Re{X(N/2-n)}}/2;
    Im{G'(n)} = {Im{X(n)} + Im{X(N/2-n)}}/2;

    //computing the complex spectrum Y according to Equation [4.59]
    Re{Y(n)} = Re{H(n)}-sin(2*pi*n/N)*Re{G'(n)}-
               cos(2*pi*n/N)*Im{G'(k)};
    Im{Y(n)} = Im{H(n)}+cos(2*pi*n/N)*Re{G'(n)}-
               sin(2*pi*n/N)*Im{G'(k)};
    //define
    P(n) = Y(2n); Q(n) = Y(2n+1);

    //N/2-point inverse complex FFT
    //Outloop = 1 to exp=log2(N/2)
    for(k=0; k++; k<exp)
    {
      //Midloop = 1 to N/4 according to which stage
      for(i=0; i<Midloop; i++)
      {
        //Inloop = 1 to N/4 (number of butterflies)
        for(n=0; n<Inloop; n++)
        {
         P(n+1) = Re{P(n)}+Re{Q(n)}+j[Im{P(n)}+Im{Q(n)}];
         Re{Q(n+1)} = (Re{P(n)}-Re{Q(n)})*cos(x)-
                      (Im{Q(n)}-Im{Q(n)}*sin(X);
         Im[Q(n+1)} = (Im{P(n)}-Im{Q(n)})*cos(X)+
                      (Re(Q(n)-Re{Q(n)})*sin/X);
        }
         //if all elements processed, jump out of the Midloop
      }
    }
}
```

real_DIF_IFFT **Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation** (cont'd)

Assumption

- The inputs are the scaled FFT spectra with factor = 1/N (N = 2^{exp})
- Output without scale

Techniques

- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering
- output converted to 16-bit with saturation

Register Usage

- From .c file to .asm file:

R8 pointer of output vector x

R9 pointer of index vector

(R10) = exp

R11 pointer of trigonomic function table

R12 contains the pointer of input vector X

real_DIF_IFFT Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation (cont'd)

Memory Note

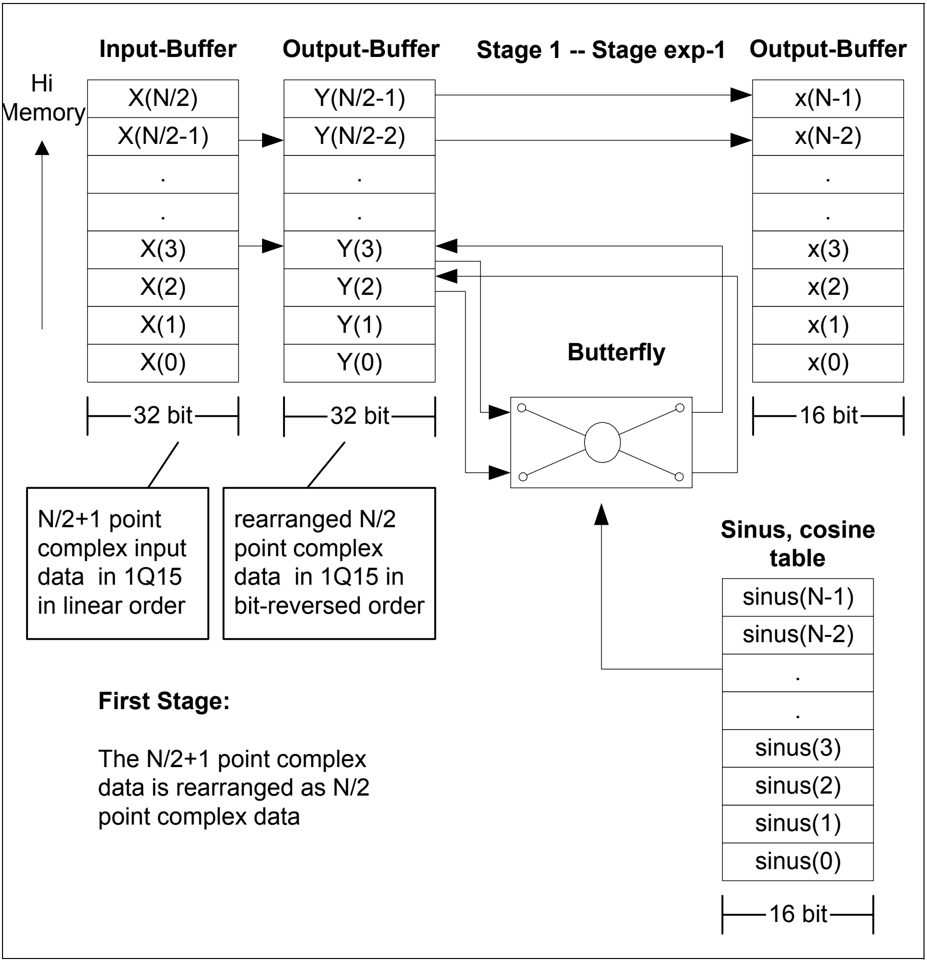

Figure 4-46 Real_DIF_IFFT

real_DIF_IFFT — Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation (cont'd)

Example *C166Lib\Keil\Examples\real_IFFT\ real_F_IFFT.c*

Cycle Count

Store state	4
Read parameters and memory Initialization	7
Butterfly	28
Unpack stage (U_cycle)	6 + 48*N/4
IFFT kernel (IFFT_cycle)	$\sum_{n=0}^{exp-2}\left\{7 + 2^{n} \bullet 28 + \frac{N}{4} \bullet 33\right\}$ where $exp=Log_2N$.
Restore state	4
Return	1
Total	**16 + IFFT_cycle + U_cycle**

Examples:
N = 8 : cycle = 338
N = 16 : cycle = 817
N = 1024: cycle = 77654

Code Size

Store state	8 bytes
Read parameters and memory initialization	14 bytes
Butterfly	78 bytes
Unpack Stage	122 bytes
IFFT kernel	102 bytes

real_DIF_IFFT **Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation** (cont'd)

Restore state	8	bytes
Return	2	bytes
Total	**334**	bytes

Q15toFloat Change the Fixed point 1Q15 format to Floating point format

Signature float Q15toFloat (DataS X)

Inputs X : The input value in 1Q15 format

Output None

Return Y : The output value in floating format

Implementation Description

This routine is used to change the 1Q15 fixed point format into the floating point data.

we have the 1Q15 fixed point data with the structure:

s. b1 b2 b3 b4 ...... b15
sig. 2^{-1} 2^{-2} 2^{-3} 2^{-4} ...... 2^{-15}

The floating point format has the structure:

1. word: mmmmmmmmmmmmmmmm
2. word: s e e e e e e e e mmmmmmm

where s =sign, e=exponent, m=mantissa. After format change

Pseudo code

Techniques None

Assumption

Register Usage

- From .c file to .asm file:
  Defined by the Compiler
- Output:
  R4 =Mantissa, R5 =Exponent

Q15toFloat Change the Fixed point 1Q15 format to Floating point format (cont'd)

Memory Note

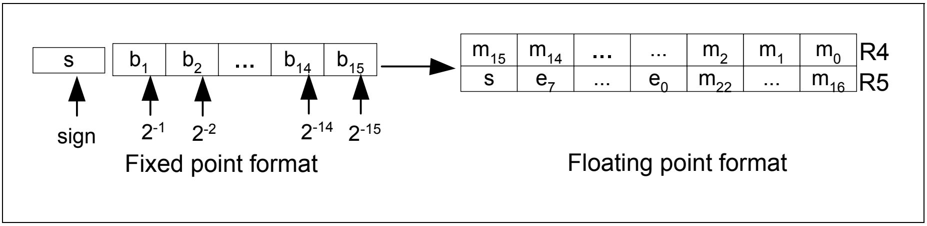

Figure 4-47 Q15toFloat

Example *C166Lib\Keil\Examples\Misc\Q15_Float.c*

Cycle Count

Check if X=0 or X=1	4
Get sign bit	6
Initialise Exponent and Mantissa	3
Check Most significant bit	1+3N
Compute floating output	8
Write Output	6
Return	1
Total	**28+3N**
	Example: N=3 Cycle =37

Code Size

Check if X=0 or X=1	12 bytes
Get sign bit	12 bytes

Q15toFloat Change the Fixedpoint 1Q15 format to Floating point format (cont'd)

Initialise Exponent and Mantissa	6	bytes
Check Most significant bit	12	bytes
Compute floating output	16	bytes
Write Output	12	bytes
Return	2	bytes
Total	72	bytes

4.8 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.

4.8.1 Descriptions

The following Matrix Operations are described.

- Multiplication
- Transpose

Matrix_mul [MxN][NxP] Matrix Multiplication

Signature

```
void  Matrix(    DataS*    x_1,
                 DataS*    x_2,
                 DataS*    y,
                 DataS     row1,
                 DataS     col1,
                 DataS     row2,
                 DataS     col2
               )
```

Inputs

x_1	:	Pointer to the first matrix in 1Q15 format
x_2	:	Pointer to the second matrix in 1Q15 format
row1	:	M, the number of rows in the first matrix
col1, row2	:	N, the number of columns in the first matrix, the number of rows in the second matrix
col2	:	P, M, the number of rows in the second matrix

Output

y	:	Pointer to the output in 1Q15 format

Return

Implementation Description

The multiplication of two matrices A and B is done. The multiplication results will be stored in the [MxP] matrix y. Both the input matrices and output matrix are 16-bit. All the element of the matrix are stored row-by-row in the buffer.

$$\begin{bmatrix} y_{11} & y_{12} & \dots y_{1P} \\ y_{21} & y_{22} & \dots y_{2P} \\ , & , & , \\ , & , & , \\ y_{M1} & y_{M2} & \dots y_{MP} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots a_{1N} \\ a_{21} & a_{22} & \dots a_{2N} \\ , & , & , \\ , & , & , \\ a_{M1} & a_{M2} & \dots a_{MN} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \dots b_{1P} \\ b_{21} & b_{22} & \dots b_{2P} \\ , & , & , \\ , & , & , \\ b_{N1} & b_{N2} & \dots b_{NP} \end{bmatrix}$$

Matrix_mul [MxN][NxP] Matrix Multiplication (cont'd)

Pseudo code

```
{
   DataS   x_1[M*N];          //first input matrix
   DataS   x_2[N*P];          //second input matrix
   DataS   y[M*P];            //output matrix
   DataS   i,j;

   for (i=0 to M-1)
     for (k=0 to P-1)
      {
        y(i,k) = 0;
        for (j=0 to N-1)
           y(i,k) = y(i,k) + x_1(i,j)*x_2(j,k);
      }

}
```

Assumption

Techniques

- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering

Register Usage

- From .c file to .asm file:

  Decided by the compiler

Matrix_mul **[MxN][NxP] Matrix Multiplication** (cont'd)

Memory Note

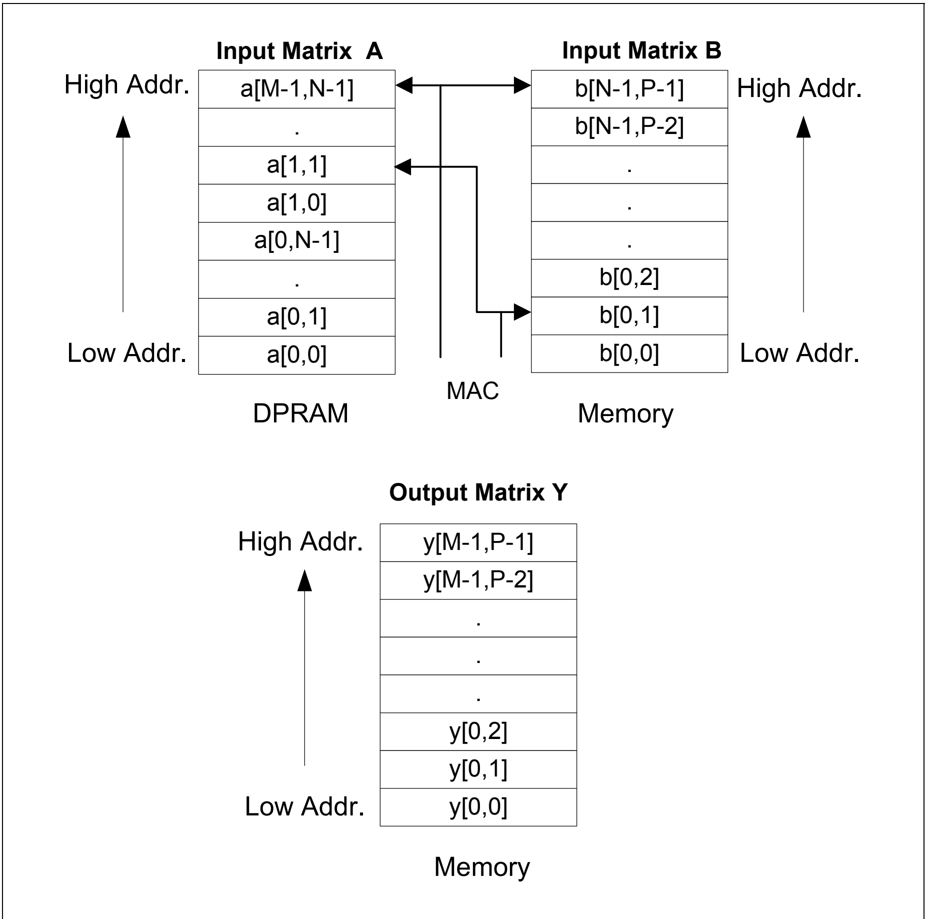

Figure 4-48 Matrix_mul

Example *C166Lib\Keil\Examples\Matrix\Matrixmul.c*

Cycle Count

Matrix_mul **[MxN][NxP] Matrix Multiplication** (cont'd)

Read parameters and memory Initialization	13
Matrix loop	M(4+P(6+3N))
Return	1
Total	**14 + M(4 + P(6 + 3N))**

Example: M = 4, N = 3, P = 5
cycle = 330

Code Size

Read parameters and memory initialization	26 bytes
Matrix loop	40 bytes
Return	2 bytes
Total	**68** bytes

Matrix_trans [MxN] Matrix Transpose

Signature

```
void  Matrix_trans( DataS*    x,
                    DataS*    y,
                    DataS     row,
                    DataS     col
                  )
```

Inputs

x	:	Pointer to the input matrix in 1Q15 format
row	:	M, the number of rows in the matrix
col	:	N, the number of columns in the matrix

Output

y	:	Pointer to the output in 1Q15 format

Return

Implementation Description

This function performs Transpose of the given matrix.It takes pointer to input matrix,size of row and size of column as input. multiplication of two matrices A and B is done. The Transpose results will be stored in the [MxN] matrix y. Both the input matrices and output matrix are 16-bit. All the element of the input matrix are stored row-by-row in the buffer.

$$\begin{bmatrix} x_{11} & x_{21} & \ldots x_{M1} \\ x_{12} & x_{22} & \ldots x_{M2} \\ , & , & , \\ , & , & , \\ x_{1N} & x_{2N} & \ldots x_{MN} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \ldots x_{1N} \\ x_{21} & x_{22} & \ldots x_{2N} \\ , & , & , \\ , & , & , \\ x_{M1} & x_{M2} & \ldots x_{MN} \end{bmatrix}$$

Matrix_trans [MxN] Matrix Transpose (cont'd)

Pseudo code

```
{
   DataS*   x;          //input matrix
   DataS*   y;          //output matrix
   DataS    M,N;        //Number of rows,columns
   DataS    i,k;

   for (i=0 to M-1)
     for (k=0 to N-1)
      {
        y(k,i) =x(i,k);
      }

}
```

Assumption

Techniques

- Memory bandwidth conflicts removing
- Instruction re-ordering

Register Usage

- From .c file to .asm file:
  Decided by the compiler

Matrix_trans [MxN] Matrix Transpose (cont'd)

Memory Note

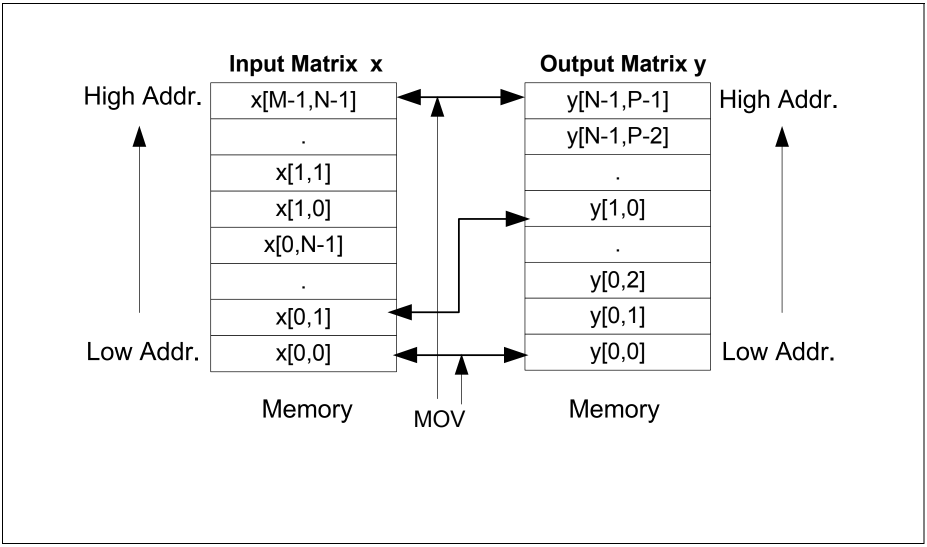

Figure 4-49 Matrix_trans

Example *C166Lib\Keil\Examples\Matrix\Matrixtrans.c*

Cycle Count

Memory Initialization	10
Matrix loop	M(2N+4)
Return	1
Total	**11+M(2N+4))**

Example: M = 4, N = 3
cycle = 51

Code Size

Matrix_trans **[MxN] Matrix Transpose** (cont'd)

Memory initialization	20 bytes
Matrix loop	22 bytes
Return	2 bytes
Total	**44** bytes

4.9 Mathematical Operations

Here we provide some assembly routines of basic mathematical operations which can be used in many DSP algorithms, such as speech codecs and spectrum analysis.

The following routines are described.

- Sine
- N-th order power series
- Windowing with N coefficients
- div_q15
- div_q31
- Sqrt

Sine **Sine function**

Signature DataS Sine(DataS* x)

Inputs x : By pi normalized input value between [-1,1] in 1Q15 format, x=*xrad*/pi, where *xrad* contains the angle in radians [-p1,p1].

Output :

Return y : output in 1Q15 format

Implementation Description For the input value in 1th quadrant $(0, \pi/2)$ the Sin(x) can be approximated using the polynomial expansion.

$$\begin{aligned}\sin(x) &= 3.140625(x) + 0.02026367(x)^2 \\ &\quad - 5.325196(x)^3 + 0.5446778(x)^4 \\ &\quad + 1.800293(x)^5\end{aligned} \qquad [4.60]$$

$0 \le x \le 0\dot{5}$.

The values of Sine function in other quadrants are computed by using the relations,

$\sin(-x) = -\sin(x)$ and $\sin(180 - x) = \sin x$.

The function takes 16 bit input in 1Q15 format to accommodate the range $(-1, 1)$. The output is 16 bits in 1Q15 format. Coefficients are stored in 4Q12 format.

The absolute value of the input is calculated. If the input is negative (III/IV Quadrant), then sign=1. If absolute value of the input is greater than 1/2 (II/III Quadrant), it is subtracted from 1. If sign=1, the result is negated to give the final sine result.

To have an optimal implementation with zero overhead load store, the polynomial in **Equation [4.60]** is rearranged as below.

Sine — Sine function (cont'd)

$$\sin(x) = ((((1.800293\ x + 0.5446778)x - 5.325196)x + 0.02026367)x + 3.140625)x \quad [4.61]$$

Hence, 4 multiply-accumulate and 1 multiply instruction will compute the Equation [4.61].

Pseudo code

```
{
   DataS   a[5];          //coefficient vector
   DataS   x;             //input value
   DataS   y;             //output value
   DataS   i;
   DataS   sign;          //sign of input

   //determine the sign of input
   sign = 0;
   if(x<0)
      sign = 1

   //if x in III/IV quadrant
   if(abs(x)>0.5)
      x = 1-abs(x);

   //polynomial
   y = a(0)*x + a(1);
   y = y*x + a(2);
   y = y*x + a(3);
   y = y*x + a(4);
   y = y*x;

   //x in I/II quandrant
   if(sign == 0)
      y = y;
   //x in III/IV quandrant
   else if(sign == 1)
      y = -y;

}
```

Assumption

Sine | **Sine function** (cont'd)

Techniques

- Use of CoMAC instructions
- Instruction re-ordering

Register Usage

- Defined by the compiler
- From .asm file to.c file

  y is stored in R4

Memory Note

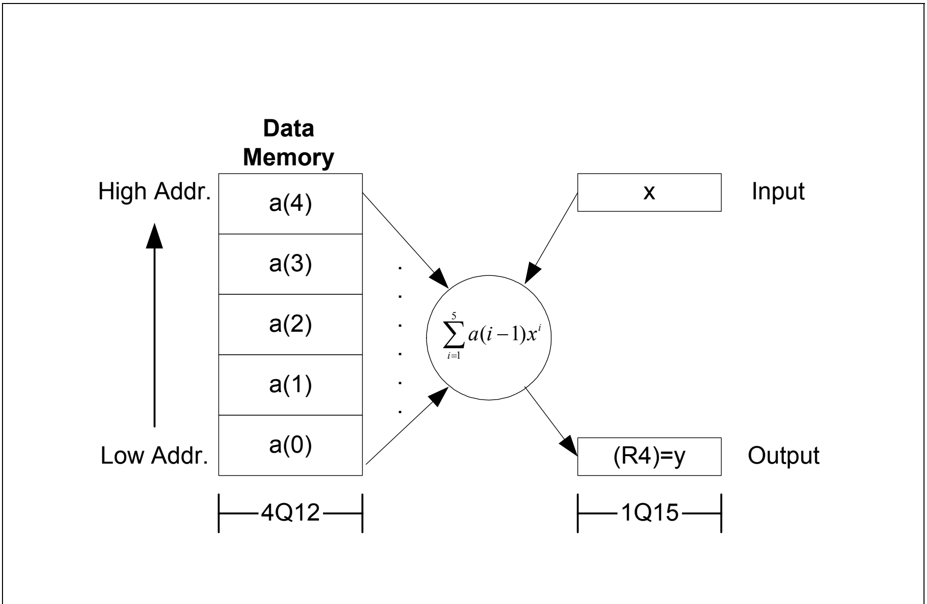

Figure 4-50 Sine

Example *C166Lib\Keil\Examples\Math\ Sinus.c*

Cycle Count

Memory Initialization	1
Polynomial	26

Sine **Sine function** (cont'd)

Return	1	
Total	**28**	

Code Size

Memory initialization	2	bytes
Polynomial	88	bytes
Return	2	bytes
Total	**92**	bytes

P_series N-th order power series

Signature DataS P_series(DataS* a, DataS* IN, DataS N)

Inputs

a	:	Pointer to the coefficient vector in 1Q15 format
IN	:	Input value
N	:	Size of series that must be even number

Output

Return

Y	:	Series output in 1Q15 format

Implementation Description

The routine is implemented with CoMUL and CoADD instructions. To optimize the routine, the implementation uses "loop unrolling" technique and assumes that N is even. The implementation formula is:

$$
\begin{aligned}
y &= \sum_{i=0}^{N} a(i) \cdot x^{i} \\
&= [[[[a(N) \cdot x + a(N-1)]x + a(N-2)]x + a(N-3)]x + \ldots]
\end{aligned}
$$

Pseudo code

```
{
   DataS   a[N];          //coefficient vector
   DataS   X;             //input value
   DataS   Y;             //output value
   DataS   i;

   Y = a(N);
   for(i=0; i<N; i++)
     Y = Y*X + a(N-1);

}
```

Assumption

- N = even number

Techniques

- Use of CoMAC instructions
- Loop unrolling
- Instruction re-ordering

P_series N-th order power series (cont'd)

Register Usage

- From .c file to .asm file:
  Defined by the compiler
- From .asm file to.c file
  Y is stored in R4

Memory Note

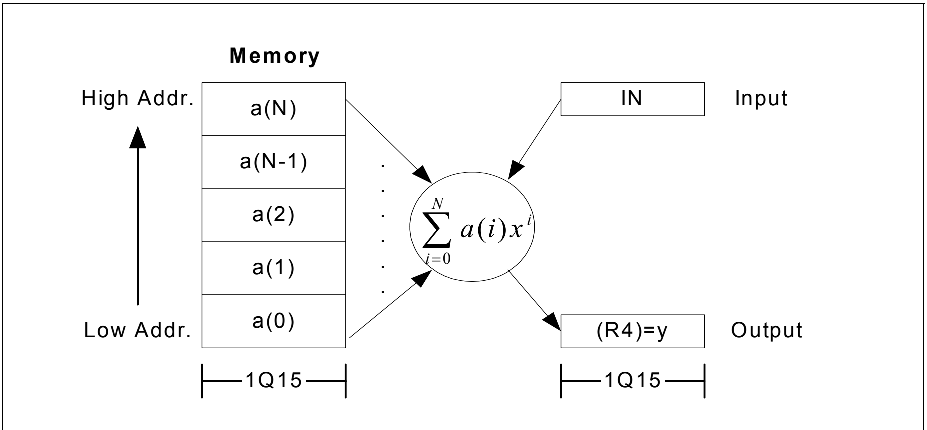

Figure 4-51 P_series

Example *C166Lib\Keil\Examples\Math\ Power_Serie.c*

Cycle Count

Memory Initialization	13
Series loop	7N/2
Return	1

P_series **N-th order power series** (cont'd)

Total	**14 + 7N/2**
	Example: N = 10 cycle = 49

Code Size

Memory initialization	26 bytes
Series loop	28 bytes
Return	2 bytes
Total	**56** bytes

Windowing — Windowing with N coefficients

Signature

```
void  Windowing(   DataS*     h,
                   DataS*     x,
                   DataS*     y,
                   DataS      N
                  )
```

Inputs

- h : Pointer to the coefficient vector in 1Q15 format
- x : Pointer to the input vector in 1Q15 format
- N : The length of the window

Output

- y : Output vector in 1Q15 format

Return :

Implementation Description

To optimize the routine, the implementation uses "loop unrolling" technique and assumes that N is even number. The implementation formula is:

$$y(i) = x(i) \cdot w(i), \quad \text{for} \quad i = 0, 1, \ldots N-1$$

Pseudo code

```
{
   DataS   w[N];          //coefficient vector
   DataS   x[N];          //input value
   DataS   y[N];          //output value
   DataS   i;

   for(i=0; i<N; i++)
     y(i) = x(i)*w(i);

}
```

Assumption

- N should be even number.

Techniques

- Use of CoMUL instructions
- Loop unrolling with the factor 2

Register Usage

- From .c file to .asm file:

  Defined by the compiler

Windowing — Windowing with N coefficients (cont'd)

Memory Note

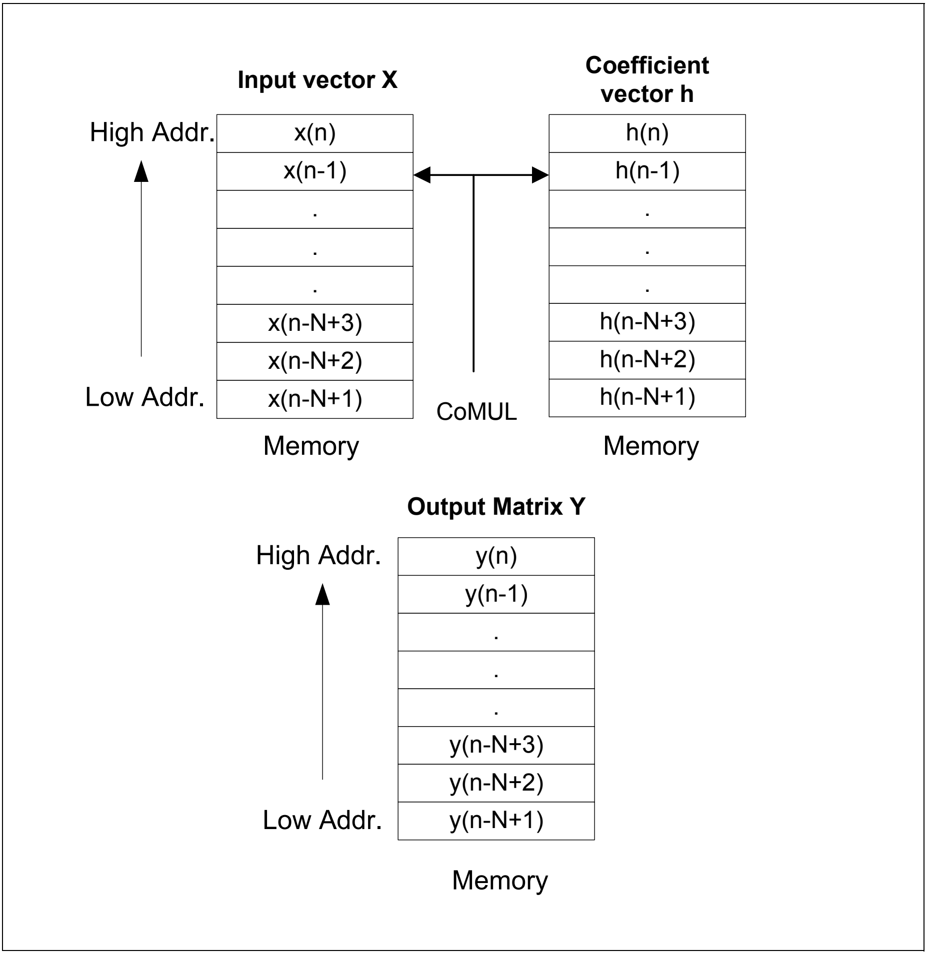

Figure 4-52 Windowing

Example *C166Lib\Keil\Examples\Math\ Window.c*

Windowing — Windowing with N coefficients (cont'd)

Cycle Count

Memory Initialization	4
Loop	7N/2
Return	1
Total	**5 + 7N/2**
	Example: N = 128 cycle = 453

Code Size

Memory initialization	8 bytes
Loop	24 bytes
Return	2 bytes
Total	**34** bytes

div_q15 Division of two 1Q15 fractional inputs

Signature	DataS div_q15(DataS x, DataS y)		
Inputs	x	:	Dividend in 1Q15 format
	y	:	Divisor in 1Q15 format
Output		:	
Return	r	:	Result in 1Q15 format

Implementation Description

This function performs the division of two fractional inputs. The dividend and the divisor are in 1Q15 format.The result is saturated to +1 or -1 if |x|>|y|.

$$\frac{x}{y} = \frac{x \times 2^{31}}{y \times 2^{15}} \times 2^{-15}$$

Pseudo code

Assumption

|x| should be less than |y|

Techniques

Register Usage

- From .c file to .asm file:
  Defined by the compiler
- Output in R4

div_q15 Division of two 1Q15 fractional inputs (cont'd)

Memory Note

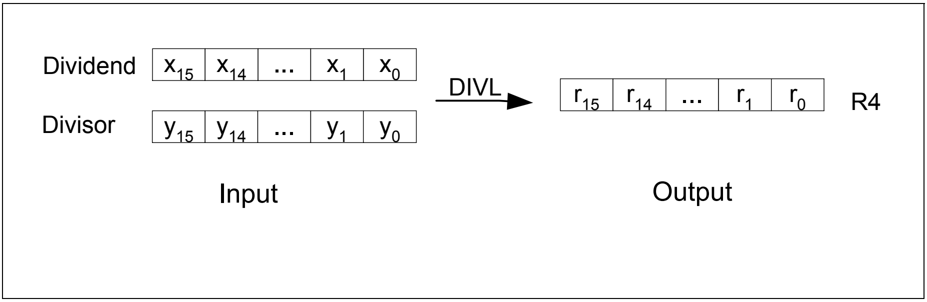

Figure 4-53 Division of two 1Q15 fractional inputs

Example *C166Lib\Keil\Examples\Math\ div_16.c*

Cycle Count

Compute absolute values	6
Compute signs	8
Division	11
Return	1
Total	**26**

Code Size

Compute absolute values	20	bytes
Compute signs	24	bytes
Division	18	bytes
Return	2	bytes
Total	**64**	**bytes**

div_q31 Division of 1Q31 fractional by 1Q15 fractional

Signature	DataS div_q31(	DataL x, DataS y)	
Inputs	x	:	Dividend in 1Q31 format
	y	:	Divisor in 1Q15 format
Output			
Return	r	:	Result in 1Q15 format

Implementation Description

This function performs the division of two fractional inputs. The dividend is in1Q31 format and the divisor is in 1Q15 format.The result is saturated to +1 or -1 if |x|>|y|,which is in 1Q15 format.

$$\frac{x}{y} = \frac{x \times 2^{31}}{y \times 2^{15}}$$

Pseudo code

Assumption
- |x|should be less than |y|

Techniques
- Use of CoMUL instructions

Register Usage
- From .c file to .asm file:
  Defined by the compiler
- Output in R4

div_q31 Division of 1Q31 fractional by 1Q15 fractional (cont'd)

Memory Note

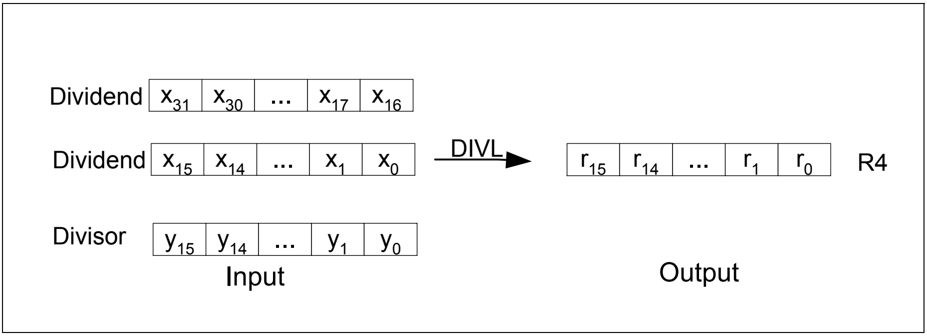

Figure 4-54 Division of 1Q31 fractional by 1Q15 fractional

Example *C166Lib\Keil\Examples\Math\div_32.c*

Cycle Count

Compute absolute values	8
Compute signs	8
Division	11
Return	1
Total	**28**

Code Size

Compute absolute values	22	bytes
Compute signs	24	bytes
Division	18	bytes

div_q31 Division of 1Q31 fractional by 1Q15 fractional (cont'd)

Return	2	bytes
Total	**60**	**bytes**

Sqrt	**Square root**		
Signature	DataS Sqrt(DataS x)		
Inputs	x	:	Input value in 1Q15 format in the range [0,1)
Output	y	:	Output value in 1Q15 format
Return		:	

Implementation Description

The square root of the input value x can be calculated by using the following polynomial expansion

$$Sqrt(x) = 1.454895(x) - 1.34491(x)^2 + 1.106812(x)^3$$
$$-0.536499(x)^4 + 0.1121216(x)^5 + 0.2075806$$

where,0.5<=x<=1

The coefficents of polynomial are stored in 2Q14 format.The square root table(table of scale factors)stores 1/sqrt(2^n) in 1Q15 format where n ranges from 0 to15. As the polynomial expansion needs input only in the range 0.5 to 1,the given input has to be scaled up.If the input value is less than 0.5,then it is scaled up by the powers of two,so that the scaled input value lies in the range 0.5 to 1.This scaled input is used for polynomial calculation.The calculated output is scaled down by 1/sqrt(2^n) to get the actual output.The obtained result is in 1Q15 format.

Sqrt — Square root (cont'd)

Pseudo code

```
{
   DataS x;        //input value
   DataS C;       // Coeffcient
   DataS y;        //output value
   DataS *scale factor; //scaling


  if(x>=0.5 && x<1)
       {
         y=((((c^5*x+c^4)x+c^3)x+c^2)x+c)x;  //compute Square root
         y=y+0.2075806;                      //compute Square root
       }
  if(x>=0 && x<0.5)
       {
          x=x*scale factor;
          y=((((c^5*x+c^4)x+c^3)x+c^2)x+c)x;  //compute Square root
          y=y+0.2075806;
          y=y/scale factor;
       }
```

Assumption
- Input is positive

Techniques
- Use of CoMUL instructions

Register Usage
- From .c file to .asm file:
  Defined by the compiler
- From .asm file to .c file
  R4 = y

Memory Note

Example *C166Lib\Keil\Examples\Math\ Squart.c*

Cycle Count

Register Initialization	1
checking if x<=0	3
checking if x>0.5	3N
compute square root	17
Multiplication with scalar	5

Sqrt — Square root (cont'd)

Return	1
Total	27+3N Example: N=4 cycle=39

Code Size

Register Initialization	2	bytes
checking if x<0	8	bytes
checking if x>0.5	12	bytes
compute square root	66	bytes
Multiplication with scalar	14	bytes
Return	2	bytes
Total	**104**	**bytes**

4.10 Statistical Functions

The following statistical functions are described.

- Correlation

  Cross-correlation

  Autocorrelation
- Convolution

4.10.1 Correlation

4.10.1.1 Definitions of Correlation

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 they are identical by 180 phase shift between them, then the 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.

Suppose that N1 and N2 represent the size of input signals x1 and x2, respectively, and N=N1+N2 and $N1 \geq N2$. Extending x1 to N-point vector through adding N2-points of zero at the beginning, and x2 to N-point vector through adding N1-points of zero in the end, we can define the discrete cross-correlation as follows.

Raw Cross-correlation:

$$r(j) = \sum_{i=0}^{N1+j-1} x1(i)x2(i+j) \quad , \quad -N2+1 \leq j \leq N1-1 \qquad [4.62]$$

Biased Cross-correlation:

$$r(j) = \frac{1}{N1} \times \sum_{i=0}^{N1+j-1} x1(i)x2(i+j) \quad , \quad -N2+1 \leq j \leq N1-1 \qquad [4.63]$$

Unbiased Cross-correlation:

$$r(j) = \frac{1}{N-\mathrm{abs}(j)} \times \sum_{i=0}^{N1+j-1} x1(i)x2(i+j) \quad , \quad -N2+1 \leq j \leq N1-1 \qquad [4.64]$$

The above definitions contain the full-length cross-correlation of the real input signals x1 and x2 with N points output, which consists of N2 points of the negative-side and N1 points of the positive-side.

If the input vectors x1 and x2 are same and equal to x with the size of N, we have the following definitions of the positive-side of the autocorrelation.

Raw Autocorrelation:

$$r(j) = \sum_{i=0}^{N-j-1} x(i)x(i+j) \qquad \text{for } j = 0 \text{ to } Nr \le N-1 \qquad [4.65]$$

Biased Autocorrelation:

$$r(j) = \frac{1}{N} \times \sum_{i=0}^{N-j-1} x(i)x(i+j) \qquad \text{for } j = 0 \text{ to } Nr \le N-1 \qquad [4.66]$$

Unbiased Autocorrelation:

$$r(j) = \frac{1}{N-j} \times \sum_{i=0}^{N-j-1} x(i)x(i+j) \qquad \text{for } j = 0 \text{ to } Nr \le N-1, \qquad [4.67]$$

where j is the lag value, as it indicates the shift/lag considered for the r(j) autocorrelation. Nr is the correlation length and it determines how much data is used for each autocorrelation result. Note that the full-length autocorrelation of vector x will have 2N-1 points with even symmetry around the lag 0 point r(0). The above definitions define only the positive half for memory and computational savings.

4.10.1.2 Implementation Note

Directly implementing the cross-correlation according to definitions in **Equation [4.62]** , **Equation [4.63]** and **Equation [4.64]** has difficulty due to the negative index. To make the algorithms realizable in assembly we need to rewrite the definitions.

Raw Cross-correlation:

The negative-side can be rewriten as with positive index

$$r(j) = \sum_{i=0}^{j} x1(i)x2(N2-j-1+i) \quad , \qquad 0 \le j \le N2-1 \qquad [4.68]$$

and the positive-side as

$$r(j+N2) = \sum_{i=0}^{N2-1} x1(i+j)x2(i) \quad , \qquad 0 \le j \le N1-N2-1 \qquad [4.69]$$

$$r(j+N2) = \sum_{i=0}^{N1-j-1} x1(i)x2(N2-j-1+i) \quad , \qquad N1-N2 < j \le N1-1 \quad . \qquad [4.70]$$

Biased Cross-correlation:

The negative-side:

$$r(j) = \frac{1}{N1} \times \sum_{i=0}^{j} x1(i)x2(N2-j-1+i) \quad , \qquad 0 \le j \le N2-1 \qquad [4.71]$$

The positive-side:

$$r(j+N2) = \frac{1}{N1} \times \sum_{i=0}^{N2-1} x1(i+j)x2(i) \quad , \qquad 0 \le j \le N1-N2-1 \qquad [4.72]$$

$$r(j+N2) = \frac{1}{N1} \times \sum_{i=0}^{N1-j-1} x1(i)x2(N2-j-1+i) \quad , \qquad N1-N2 < j \le N1-1 \qquad [4.73]$$

Unbiased Cross-correlation:

The negative-side:

$$r(j) = \frac{1}{abs(j+1)} \times \sum_{i=0}^{j} x1(i)x2(N2-j-1+i) \quad , \qquad 0 \le j \le N2-1 \qquad [4.74]$$

The positive-side:

$$r(j+N2) = \frac{1}{N2} \times \sum_{i=0}^{N2-1} x1(i+j)x2(i) \qquad , \qquad 0 \le j \le N1-N2-1 \qquad [4.75]$$

$$r(j+N2) = \frac{1}{N1-abs(j)} \times \sum_{i=0}^{N1-j-1} x1(i)x2(N2-j-1+i) \ , \ N1-N2 < j \le N1-1 \qquad [4.76]$$

4.10.2 Implementation Description

The following routines are described.

- Raw autocorrelation
- Biased autocorrelation
- Unbiased autocorrelation
- Raw cross-correlation
- Biased cross-correlation
- Unbiased cross-correlation

Auto_raw Raw autocorrelation

Signature

```
void  Auto_raw(   DataS*    x,
                  DataS*    y,
                  DataS     N_x,
                  DataS     N_y
                )
```

Inputs	x	:	Pointer to input vector in 1Q15 format
	N_x	:	The length of input vector
	N_y (Nr)	:	The length of output vector
Output	y	:	Pointer to output vector in 1Q15 format
Return		:	

Implementation Description

The function performs the positive side of the raw autocorrelation function of real vector x according to the definition in Equation [4.65] . The arguments to the function are pointer to the input vector, pointer to output buffer to store autocorrelation result, size of input buffer and number of auto correlated outputs desired. The input values and output values are both in 16 bit fractional format.

Pseudo code

```
{
   DataS *x;         //Ptr to input vector
   DataS *y;         //Ptr to output vector
   DataS N_x;        //size of input
   DataS N_y;        //size of autocorrelation result
   DataS i,j;

   for(j=0; j<N_y; j++)
   {
      y[j] = 0;
      for(i=0; i<N_x-j; i++)
        y[j] = y[j] + x[i+j]*x[i];
   }
}
```

Assumption

- N_y<=N_x

Techniques

- Use of CoMUL instructions

Auto_raw **Raw autocorrelation** (cont'd)

Register Usage

- From .c file to .asm file:
  Defined by the compiler

Memory Note

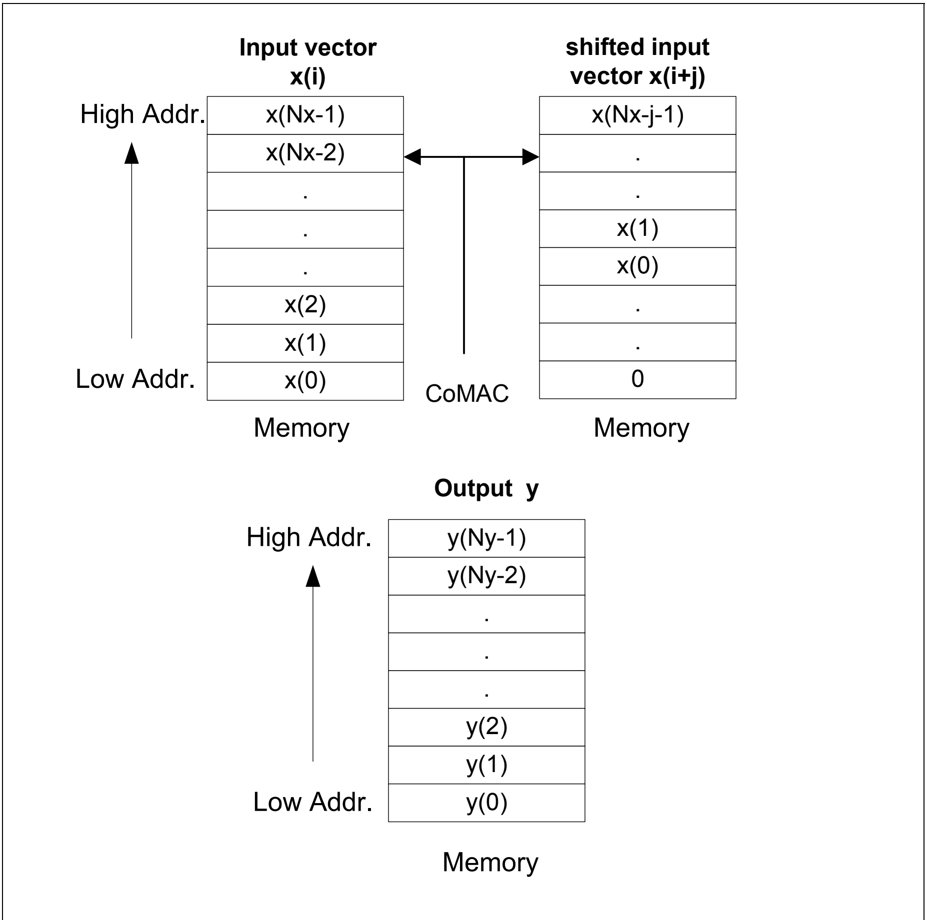

Figure 4-55 Raw autocorrelation

Auto_raw Raw autocorrelation (cont'd)

Example *C166Lib\Keil\Examples\Static_funcs\Auto_correlation\Autoc.c*

Cycle Count

Memory Initialization	4
Loop_cyc	$2 \cdot \sum_{j=0}^{Ny-1} (Nx - j + 8)$
Return	1
Total	**5 + loop_cyc**

Example:
Nx = 10, Ny = 3
cycle = 107

Code Size

Memory Initialization	8	bytes
Loop_cyc	46	bytes
Return	2	bytes
Total	**56**	**bytes**

Auto_bias Biased autocorrelation

Signature

```
void Auto_bias( DataS*   x,
                DataS*   y,
                DataS    N_x,
                DataS    N_y
              )
```

Inputs	x	:	Pointer to input vector in 1Q15 format
	N_x	:	The length of input vector
	N_y (Nr)	:	The length of output vector
Output	y	:	Pointer to output vector in 1Q15 format
Return		:	

Implementation Description

The function performs the positive side of the biased autocorrelation function of real vector x according to the definition in **Equation [4.66]**. The arguments to the function are pointer to the input vector, pointer to output buffer to store autocorrelation result, size of input buffer and number of auto correlated outputs desired. The input values and output values are both in 16 bit fractional format.

Pseudo code

```
{
   DataS *x;         //Ptr to input vector
   DataS *y;         //Ptr to output vector
   DataS N_x;        //size of input
   DataS N_y;        //size of autocorrelation result
   DataS i,j;

   for(j=0; j<N_y; j++)
   {
      y[j] = 0;
      for(i=0; i<N_x-j; i++)
        y[j] = (y[j] + x[i+j]*x[i])/N_x;
   }
}
```

Assumption
- N_y<=N_x

Techniques
- Use of CoMUL instructions

Auto_bias **Biased autocorrelation** (cont'd)

Register Usage

- From .c file to .asm file:
  Defined by the compiler

Memory Note

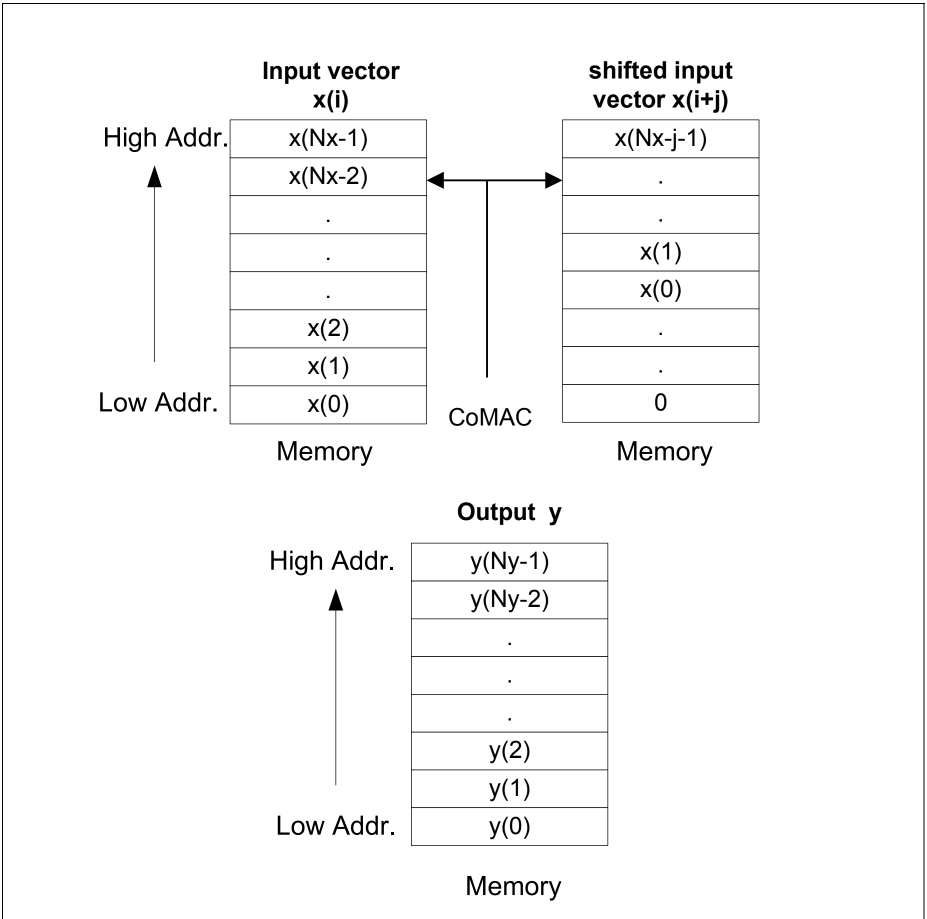

Figure 4-56 Biased autocorrelation

Auto_bias **Biased autocorrelation** (cont'd)

Example *C166Lib\Keil\Examples\Static_funcs\ Auto_correlation\Autoc.c*

Cycle Count

Memory Initialization	4
Loop_cyc	$2 \cdot \sum_{j=0}^{Ny-1} (Nx - j + 17)$
Return	1
Total	**5 + loop_cyc**

Example:
Nx = 10, Ny = 3
cycle = 161

Code Size

Memory Initialization	8	bytes
Loop_cyc	54	bytes
Return	2	bytes
Total	**64**	**bytes**

Auto_unbias Unbiased autocorrelation

Signature

```
void  Auto_unbias(  DataS*    x,
                    DataS*    y,
                    DataS     N_x,
                    DataS     N_y
                 )
```

Inputs	x	:	Pointer to input vector in 1Q15 format
	N_x	:	The length of input vector
	N_y (Nr)	:	The length of output vector
Output	y	:	Pointer to output vector in 1Q15 format
Return		:	

Implementation Description

The function performs the positive side of the unbiased autocorrelation function of real vector x according to the definition in **Equation [4.67]**. The arguments to the function are pointer to the input vector, pointer to output buffer to store autocorrelation result, size of input buffer and number of auto correlated outputs desired. The input values and output values are both in 16 bit fractional format.

Pseudo code

```
{
   DataS *x;         //Ptr to input vector
   DataS *y;         //Ptr to output vector
   DataS N_x;        //size of input
   DataS N_y;        //size of autocorrelation result
   DataS i,j;

   for(j=0; j<N_y; j++)
   {
      y[j] = 0;
      for(i=0; i<N_x-j; i++)
        y[j] = (y[j] + x[i+j]*x[i])/(N_x-j);
   }
}
```

Assumption

- N_y<=N_x

Techniques

- Use of CoMUL instructions

Auto_unbias Unbiased autocorrelation (cont'd)

Register Usage

- From .c file to .asm file:
  Defined by the compiler

Memory Note

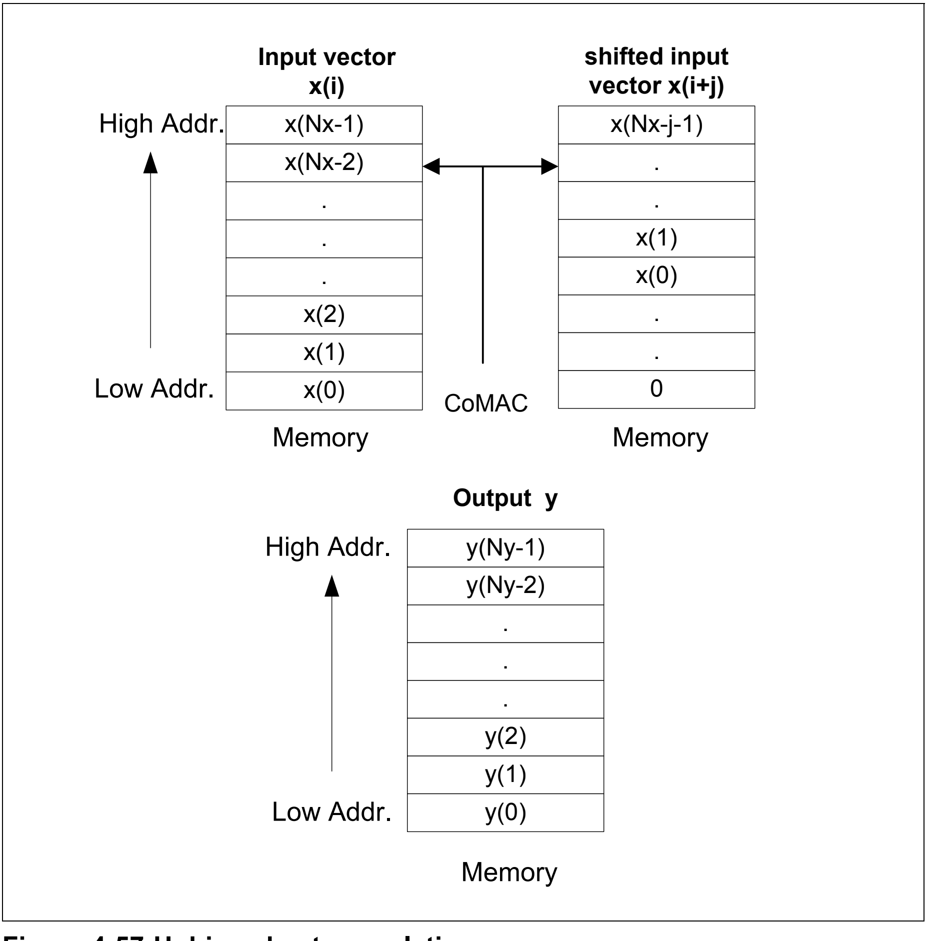

Figure 4-57 Unbiased autocorrelation

Auto_unbias **Unbiased autocorrelation** (cont'd)

Example *C166Lib\Keil\Examples\Static_funcs\ Auto_correlation\Autoc.c*

Cycle Count

Memory Initialization	4
Loop_cyc	$2 \cdot \sum_{j=0}^{Ny-1} (Nx - j + 18)$
Return	1
Total	**5+ loop_cyc**

Example:
Nx = 10, Ny = 3
cycle = 167

Code Size

Memory Initialization	8	bytes
Loop_cyc	56	bytes
Return	2	bytes
Total	**66**	**bytes**

Cross_raw Raw cross-correlation

Signature void Cross_raw(DataS* x1,
DataS* x2
DataS* y,
DataS N_x1,
DataS N_x2
)

Inputs	x1	:	Pointer to first input vector in 1Q15 format
	x2	:	Pointer to second input vector in 1Q15 format
	N_x1 (N1)	:	The length of the first input vector
	N_x2 (N2)	:	The length of the second input vector
Output	y	:	Pointer to output vector in 1Q15 format
Return		:	

Implementation Description

The function performs the full-length raw cross-correlation function of real vector x1 and x2 according to **Equation [4.68]**, **Equation [4.69]** and **Equation [4.70]**. The arguments to the function are pointers to the input vectors, pointer to output buffer to store autocorrelation result and sizes of input buffers. The input values and output values are both in 16 bit fractional format.

Cross_raw Raw cross-correlation (cont'd)

Pseudo code

```
{
   DataS *x1;       //Ptr to the first input vector
   DataS *x2;       //Ptr to the second input vector
   DataS *y;        //Ptr to output vector
   DataS N1;      //size of the first input
   DataS N2;      //size of the second input
   DataS i,j;

   //negative side
   for(j=0; j<N2; j++)
   {
      y[j] = 0;
      for(i=0; i<j; i++)
        y[j] = y[j] + x1[i]*x2[N2-j-1+i];
   }

   //positive side
   if(0<=j<=(N1-N2-1))
   {
     y[j+N2] = 0;
     for(i=0; i<N2; i++)
        y[j+N2] = y[j+N2] + x1[i+j]*x2[i];
   }
   else if((N1-N2)<=j<=(N1-1))
   {
     y[j+N2] = 0;
     for(i=0; i<N1-j; i++)
        y[j+N2] = y[j+N2] + x1[i]*x2[N2-j-1+i];
   }

}
```

Assumption
- N2<=N1
- x1 must be stored in DPRAM area.

Techniques
- Use of CoMUL instructions

Register Usage
- From .c file to .asm file:
  Defined by the compiler

Cross_raw **Raw cross-correlation** (cont'd)

Memory Note

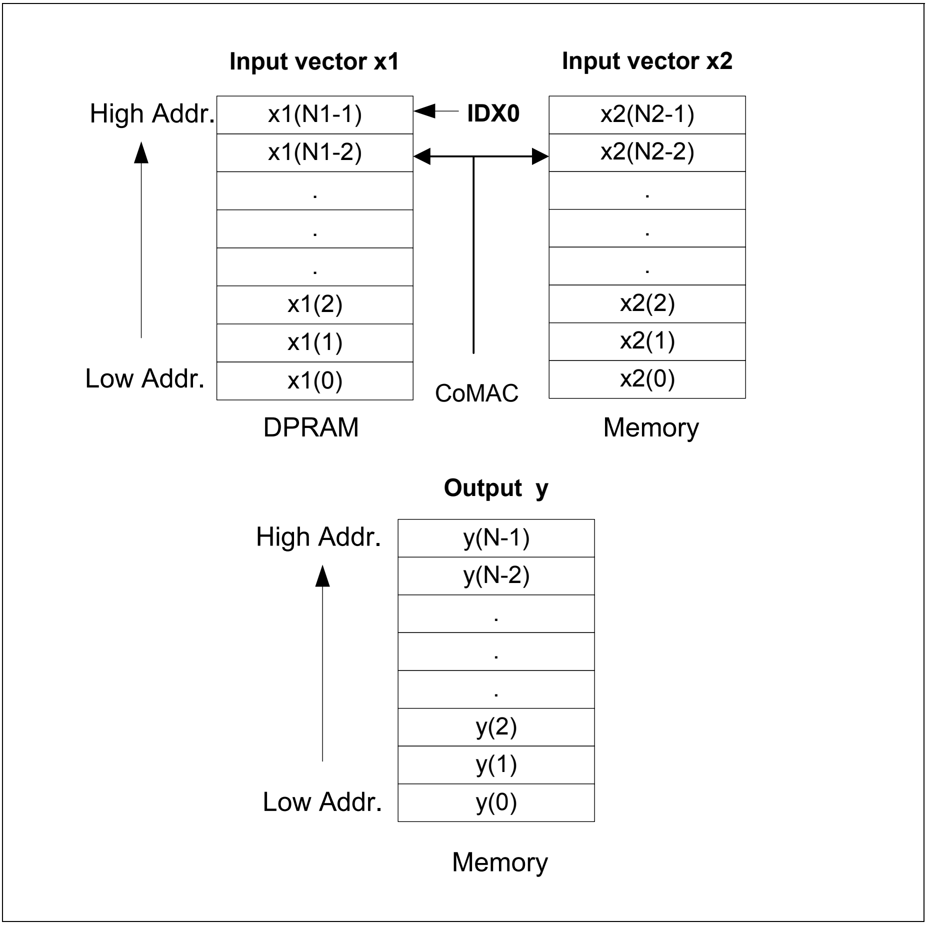

Figure 4-58 Raw cross-correlation

Example *C166Lib\Keil\Examples\Static_funcs\ Cross_correlation\Cross.c*

Cross_raw Raw cross-correlation (cont'd)

Cycle Count

Memory Initialization	4
Neg_cyc	$\sum_{j=0}^{N2-1} (j+18)$
Positive-side:	
Intialization	3
Pos_cyc1	(N1-N2)*(N2+19)
Pos_cyc2	$\sum_{j=0}^{N2-1} (N1-j+19)$
Return	1
Total	**8+Neg_cyc+Pos_cyc1+Pos_cyc2**

Example:
N1 = 3, N2 = 3
cycle = 128

Code Size

Memory Initialization	8	bytes
Neg_cyc	50	bytes
Positive-side:		
Intialization	6	bytes
Pos_cyc1	36	bytes
Pos_cyc2	44	bytes
Return	2	bytes
Total	**146**	**bytes**

Cross_bias Biased cross-correlation

Signature	void Cross_bias(DataS* x1, DataS* x2 DataS* y, DataS N_x1, DataS N_x2)		
Inputs	x1	:	Pointer to first input vector in 1Q15 format
	x2	:	Pointer to second input vector in 1Q15 format
	N_x1 (N1)	:	The length of the first input vector
	N_x2 (N2)	:	The length of the second input vector
Output	y	:	Pointer to output vector in 1Q15 format
Return		:	
Implementation Description	The function performs the full-length biased cross-correlation function of real vector x1 and x2 according to **Equation [4.71]**,**Equation [4.72]** and **Equation [4.73]**. The arguments to the function are pointers to the input vectors, pointer to output buffer to store autocorrelation result and sizes of input buffers. The input values and output values are both in 16 bit fractional format.		

Cross_bias   Biased cross-correlation (cont'd)

Pseudo code

```
{
   DataS *x1;         //Ptr to the first input vector
   DataS *x2;         //Ptr to the second input vector
   DataS *y;          //Ptr to output vector
   DataS N1;          //size of the first input
   DataS N2;          //size of the second input
   DataS i,j;

   //negative side
   for(j=0; j<N2; j++)
   {
      y[j] = 0;
      for(i=0; i<j; i++)
        y[j] = (y[j] + x1[i]*x2[N2-j-1+i])/N1;
   }

   //positive side
   if(0<=j<=(N1-N2-1))
   {
     y[j+N2] = 0;
     for(i=0; i<N2; i++)
        y[j+N2] = (y[j+N2] + x1[i+j]*x2[i])/N1;
   }
   else if((N1-N2)<=j<=(N1-1))
   {
     y[j+N2] = 0;
     for(i=0; i<N1-j; i++)
        y[j+N2] = (y[j+N2] + x1[i]*x2[N2-j-1+i])/N1;
   }

}
```

Assumption
- N2<=N1
- x1 must be stored in DPRAM area.

Techniques
- Use of CoMUL instructions

Register Usage
- From .c file to .asm file:
  Defined by the compiler

Cross_bias **Biased cross-correlation** (cont'd)

Memory Note

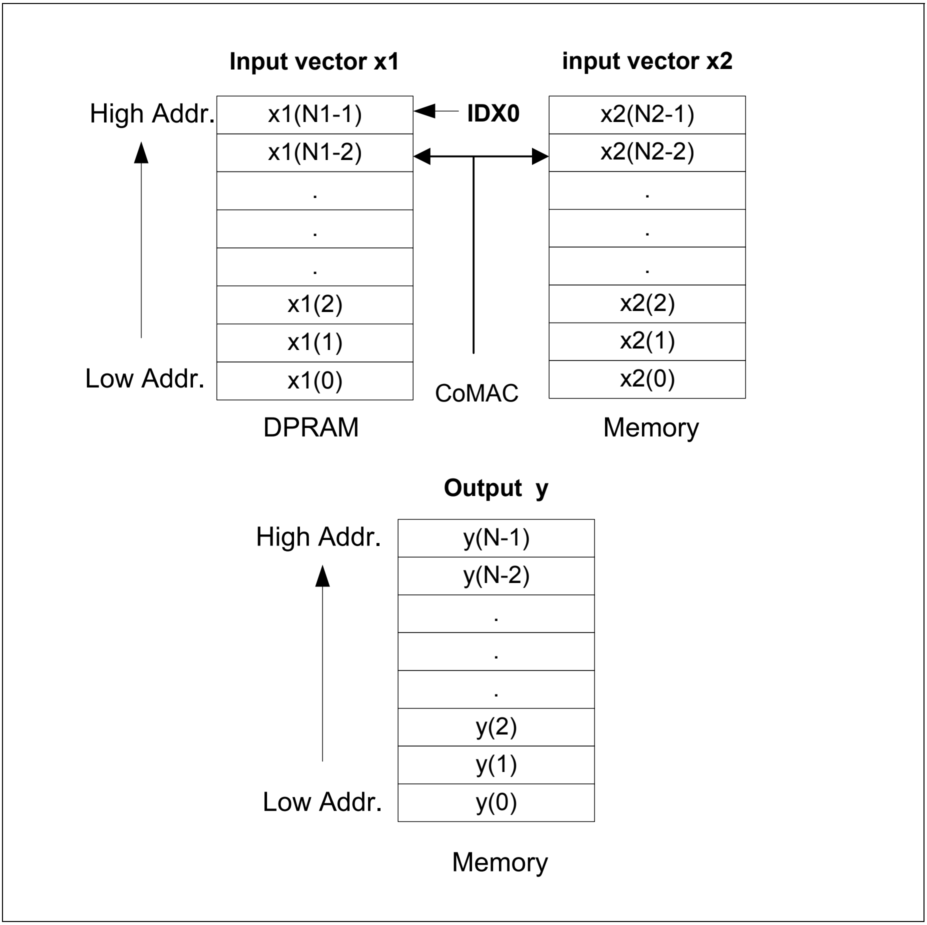

Figure 4-59 Biased cross-correlation

Example *C166Lib\Keil\Examples\Static_funcs\ Cross_correlation\Cross.c*

Cross_bias **Biased cross-correlation** (cont'd)

Cycle Count

Memory Initialization	4
Neg_cyc	$\sum_{j=0}^{N2-1}(j+36)$
Positive-side:	
Intialization	3
Pos_cyc1	(N1-N2)*(N2+37)
Pos_cyc2	$\sum_{j=0}^{N2-1}(N1-j+34)$
Return	1
Total	**8+Neg_cyc+Pos_cyc1+Pos_cyc2**

Example:
N1 = 3, N2 = 3
cycle = 156

Code Size

Memory Initialization	8	bytes
Neg_cyc	56	bytes
Positive-side:		
Intialization	6	bytes
Pos_cyc1	32	bytes
Pos_cyc2	52	bytes
Return	2	bytes
Total	**156**	**bytes**

Cross_unbias Unbiased cross-correlation

Signature

```
void  Cross_unbias( DataS*    x1,
                    DataS*    x2
                    DataS*    y,
                    DataS     N_x1,
                    DataS     N_x2
                  )
```

Inputs	x1	:	Pointer to first input vector in 1Q15 format
	x2	:	Pointer to second input vector in 1Q15 format
	N_x1 (N1)	:	The length of the first input vector
	N_x2 (N2)	:	The length of the second input vector
Output	y	:	Pointer to output vector in 1Q15 format
Return		:	

Implementation Description

The function performs the full-length unbiased cross-correlation function of real vector x1 and x2 according to **Equation [4.74]**,**Equation [4.75]** and **Equation [4.76]** . The arguments to the function are pointers to the input vectors, pointer to output buffer to store autocorrelation result and sizes of input buffers. The input values and output values are both in 16 bit fractional format.

Cross_unbias Unbiased cross-correlation (cont'd)

Pseudo code

```
{
   DataS *x1;        //Ptr to the first input vector
   DataS *x2;        //Ptr to the second input vector
   DataS *y;         //Ptr to output vector
   DataS N1;         //size of the first input
   DataS N2;         //size of the second input
   DataS i,j;

   //negative side
   for(j=0; j<N2; j++)
   {
      y[j] = 0;
      for(i=0; i<j; i++)
        y[j] = (y[j] + x1[i]*x2[N2-j-1+i])/abs(j+1);
   }

   //positive side
   if(0<=j<=(N1-N2-1))
   {
     y[j+N2] = 0;
     for(i=0; i<N2; i++)
        y[j+N2] = (y[j+N2] + x1[i+j]*x2[i])/N2;
   }
   else if((N1-N2)<=j<=(N1-1))
   {
     y[j+N2] = 0;
     for(i=0; i<N1-j; i++)
        y[j+N2] = (y[j+N2] + x1[i]*x2[N2-j-1+i])/(N1-abs(j));
   }

}
```

Assumption
- N2<=N1
- x1 must be stored in DPRAM area.

Techniques
- Use of CoMUL instructions

Register Usage
- From .c file to .asm file:
  Defined by the compiler

Cross_unbias **Unbiased cross-correlation** (cont'd)

Memory Note

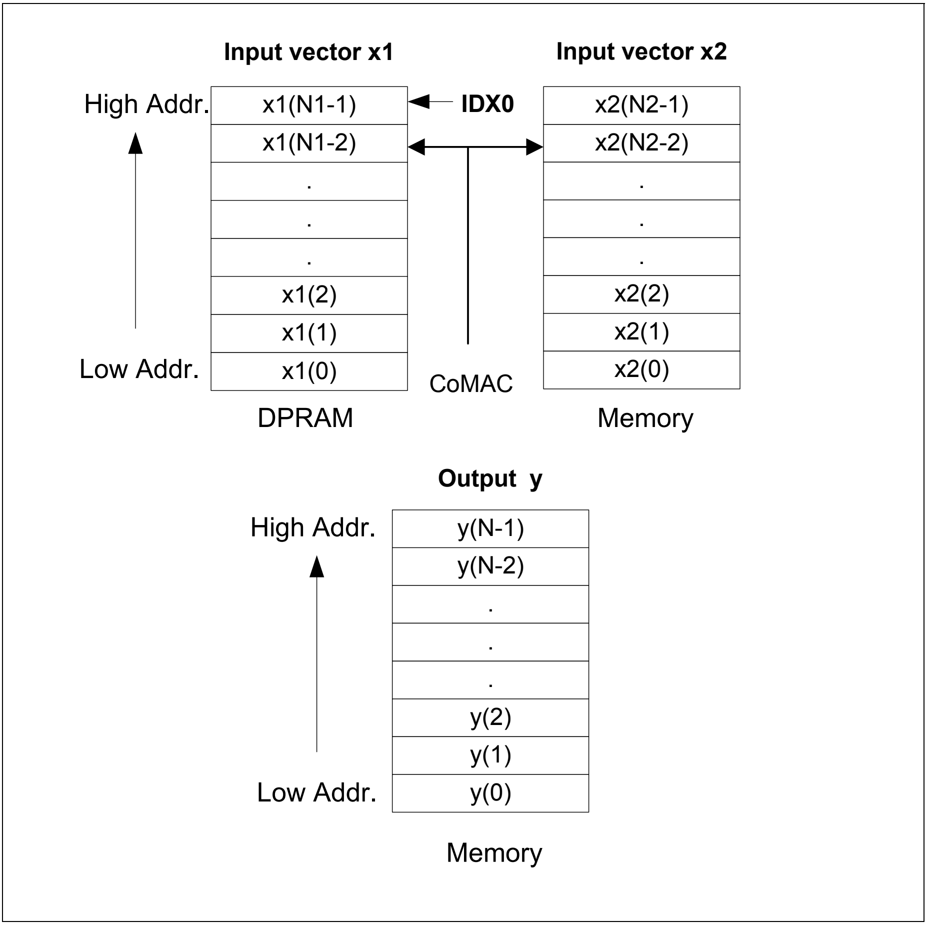

Figure 4-60 Unbiased cross-correlation

Example *C166Lib\Keil\Examples\Static_funcs\ Cross_correlation\Cross.c*

Cross_unbias Unbiased cross-correlation (cont'd)

Cycle Count

Memory Initialization	4
Neg_cyc	$\sum_{j=0}^{N2-1} (j+38)$
Positive-side:	
Intialization	3
Pos_cyc1	(N1-N2)*(N2+39)
Pos_cyc2	$\sum_{j=0}^{N2-1} (N1-j+41)$
Return	1
Total	**8+Neg_cyc+Pos_cyc1+Pos_cyc2**

Example:
N1 = 3, N2 = 3
cycle = 212

Code Size

Memory Initialization	8	bytes
Neg_cyc	60	bytes
Positive-side:		
Intialization	6	bytes
Pos_cyc1	36	bytes
Pos_cyc2	64	bytes
Return	2	bytes
Total	**176**	**bytes**

4.10.3 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

$$R(n) = \sum_{k=0}^{nX+nH-1} H(k)X(n-k) \quad [4.77]$$

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.

Convol Convolution

Signature

```
DataS Convol (DataS*  x,
              DataS* h,
              DataS* y,
              DataS* d_buffer,
              DataS   N_x,
              DataS   N_h
              )
```

Inputs	x	:	Pointer to input vector x in 1Q15 format
	h	:	Pointer to input vector h in 1Q15 format
	d_buffer	:	Pointer to delay buffer
	N_x	:	Size of x
	N_h	:	Size of h
Output	y	:	Convolution output in 1Q15 format
Return			

Implementation Description

The convolution of the two sequences x and h is done.The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM.The delay buffer is located in the DPRAM area.

Pseudo code

```
{
   DataS *x;        //pointer to the input vector x
   DataS *h;        //pointer to the input vector h
   DataS i,j;
   DataS N_h,N_x;

 for(j=0 to N_h+N_x-1)
    {
      for(k=0 to N_h-1)
        {
           y(j)=y(j)+h(k)*x(j-k);
        }
     }
     return y;
}
```

Techniques Use of CoMAC instructions

Convol **Convolution** (cont'd)

Assumption Delay buffer must be located in DPRAM area(0xf200-0xfe00)

Register Usage
- From .c file to .asm file:
  Defined by the compiler

Memory Note

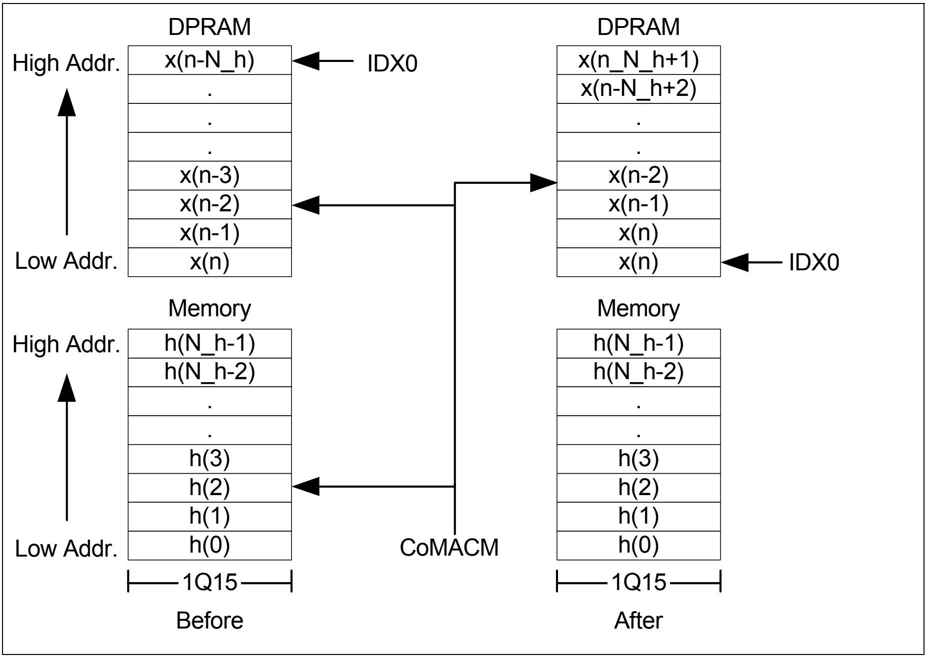

Figure 4-61 Convolution

Example *C166Lib\Keil\Examples\Static_funcs\Convolution\conv.c*

Cycle Count

Read parameters	4
Register Initialization	7
Set counters	5

Convol Convolution (cont'd)

Convolution	2N_h+8N_x-1
Write output	2
Return	1
Total	**2N_h+8N_x+18**

Example:
N_h=4 ,N_x=8
cycle:90

Code Size

Read parameters	8	bytes
Register Initialization	14	bytes
Set counters	10	bytes
Convolution	42	bytes
Write output	8	bytes
Return	2	bytes
Total	**84**	bytes

5 References

1. C166S V2, 16-bit Microcontroller, User Manual, V 1.7, Jan. 2001.
2. C166S V2,A DSP Library for C166S V2 Core,User's Manual,V1.1,Sept.2002.
3. XC161CJ, Peripheral Units, 16-Bit Single-Chip Microcontroller with C166S V2 Core, V1.1, Dec. 2000.
4. XC16Board, Hardware Manual, Board REV. 200, V1.0, Sept. 2001.
5. TriLib, A DSP Library for TriCore, User's Manual, V 1.2, Jan. 2001.
6. Application Note: ST10 DSP MAC Signal Processing Algorithms, 1998.
7. Application Note: Fast - Fourier - Transformation for C166 Microcontroller Family, 1997.

TMS320C54x DSPLIB User's Guide, Texas Instruments, Sept. 1998.

Total Quality Management

Qualität hat für uns eine umfassende Bedeutung. Wir wollen allen Ihren Ansprüchen in der bestmöglichen Weise gerecht werden. Es geht uns also nicht nur um die Produktqualität – unsere Anstrengungen gelten gleichermaßen der Lieferqualität und Logistik, dem Service und Support sowie allen sonstigen Beratungs- und Betreuungsleistungen.

Dazu gehört eine bestimmte Geisteshaltung unserer Mitarbeiter. Total Quality im Denken und Handeln gegenüber Kollegen, Lieferanten und Ihnen, unserem Kunden. Unsere Leitlinie ist jede Aufgabe mit „Null Fehlern“ zu lösen – in offener Sichtweise auch über den eigenen Arbeitsplatz hinaus – und uns ständig zu verbessern.

Unternehmensweit orientieren wir uns dabei auch an „top“ (Time Optimized Processes), um Ihnen durch größere Schnelligkeit den entscheidenden Wettbewerbsvorsprung zu verschaffen.

Geben Sie uns die Chance, hohe Leistung durch umfassende Qualität zu beweisen.

Wir werden Sie überzeugen.

Quality takes on an allencompassing significance at Semiconductor Group. For us it means living up to each and every one of your demands in the best possible way. So we are not only concerned with product quality. We direct our efforts equally at quality of supply and logistics, service and support, as well as all the other ways in which we advise and attend to you.

Part of this is the very special attitude of our staff. Total Quality in thought and deed, towards co-workers, suppliers and you, our customer. Our guideline is “do everything with zero defects”, in an open manner that is demonstrated beyond your immediate workplace, and to constantly improve.

Throughout the corporation we also think in terms of Time Optimized Processes (top), greater speed on our part to give you that decisive competitive edge.

Give us the chance to prove the best of performance through the best of quality – you will be convinced.

http://www.infineon.com

Published by Infineon Technologies AG