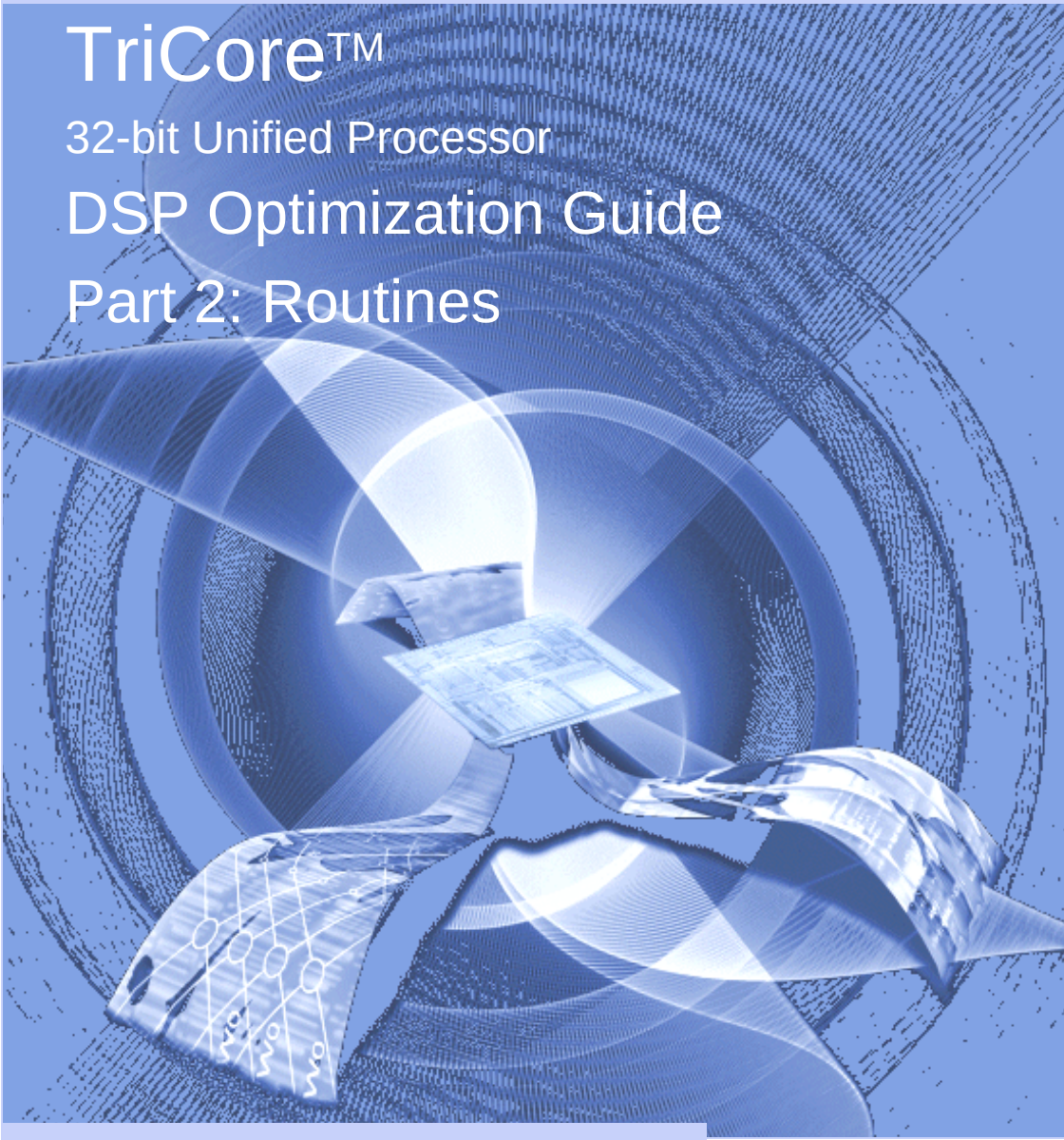
User Guide, v1.6.4, January 2003

TriCore™

32-bit Unified Processor

DSP Optimization Guide

Part 2: Routines



IP Cores

Infineon technologies

Never stop thinking.

Edition 2003-01

Published by Infineon Technologies AG,
St.-Martin-Strasse 53,
D-81541 München, Germany

Attention please!

The information herein is given to describe certain components and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Infineon Technologies is an approved CECC manufacturer.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office in Germany or our Infineon Technologies Representatives worldwide (see 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.

TriCore™

32-bit Unified Processor

DSP Optimization Guide

Part 2: Routines

DSP Optimization Guide

Revision History:	**2003-01**	v1.6.4
Previous Version:	v1.6.3	
Page	Subjects (major changes since last revision)	
All	Revised following internal review	

We Listen to Your Comments

Is there any information within this document that you feel is wrong, unclear or missing? Your feedback will help us to continuously improve the quality of this document. Please send your comments (referencing this document) to:

ipdoc@infineon.com

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

1 Introduction

This second part of the TriCore DSP Optimization guide contains short routines which, from the machine's perspective, offer a high degree of optimization.

Note: Machine (or assembly) perspective means instead of the algorithm perspective. There is always more gain to be made at the algorithm level, than at the machine (assembly implementation) level.

The time-pressures for a 'real-world' project situation coupled with basic DSP features, will often add cycles to a project routine, but there are ways in which some routines can be further optimized. Aside from the ingenuity of the DSP programmer themselves, these include for example:

- Using more unrolling and/or software pipelining (although there is then the potential drawback of making the code less readable and increasing the code size).
- Using a memory model where coefficients and data are not separated in 2 memory spaces, allowing many algorithms to take advantage of interleaving data and coefficients.

The chapters of this second part of the TriCore DSP Optimization guide are divided into the following function types:

- **Generator**
- **Scalars**
- **Filters**
- **Transcendental Functions**
- **Vectors**
- **Transforms**

The following table identifies the characteristics of these routine types:

	Input	**Output**	**Processing**
Generator	single	1 series of *n* elements	Iterative processing
Transcendental	single	single	Iterative processing
Scalar	single	single	simple equation
Vector [1)]	1 series of *n* elements	1 series of *n* elements	Identical to all *n* elements
Filter	1 series of *n* elements	single	
Transform	multiple	multiple	
	1 series of *n* elements	1 series of *n* elements	Not always identical to *n* elements

1) Includes a matrix operation

Most routines are implemented as memory to memory, but some routines can be:

- Register to register
- Memory to memory with initialization of pointers
- Full context switching (identical to a library call)

It is very simple to use one type of model or another.

In this document, each routine is presented with a short summary and all routines of the same type are grouped in an **Information Table** (a summary of the routines critical characteristics), at the start of each chapter.

1.1 Information Table

Each section begins with an information table which gives:

- Number of cycles
- Code size
- Optimization techniques (used in the assembly code)
- Arithmetic method used

1.1.1 Number of Cycles

The number of cycles is counted as follows:

- **Software Pipelining**
- **Loop Unrolling**
- **Packed Operation**
- **Load / Store Scheduling**
- **Data Memory Interleaving**
- **Packed Load / Store**

Software Pipelining

Software pipelining means starting an equation before the previous equation has finished. This is achieved with knowledge of the TriCore pipelining rules.

Example: Implementation of a C square difference

```
int sum = 0;
for (i=0; i<N; i++)
          {
          a = X[i];
          b = Y[I];
          c = a - b;
          sum = sum + c*c;
          }
```

A naïve implementation would be:

```
mov       d0,#0                    ; prolog
sumloop:                           ; loop
          ld.w     d1,[Xptr+]4     ; (1)       ; ld X0
          ld.w     d2,[Yptr+]4     ; (2)       ; ld Y0
          sub      d3,d1,d2        ; (3)       ;
          madd     d0,d0,d3,d3     ; (4,5,6)   ;
loop      LC,sumloop
st.w      sumAddr,d0               ; epilog
```

This implementation is very expensive in terms of cycles, because the loop begins with two LS (Load/Store) instructions followed by one IP (Integer Processing) instruction, and one MAC 32*32.

The number of cycles can easily be decreased by using a different instruction order:

- IP followed by LS, MAC 32*32 followed by LS

```
mov       d0,#0                    ; prolog
ld.w      d1,[Xptr+]4              ; ld X0
ld.w      d2,[Yptr+]4              ; ld Y0
sumloop:                           ; loop
          sub         d3,d1,d2     ; (1)       ;
          ld.w        d1,[Xptr+]4  ; ||        ; ld X1 for next pass
          madd        d0,d0,d3,d3  ; (2)       ;
          ld.w        d2,[Yptr+]4  ; ||        ; ld Y1 for next pass
loop      LC,sumloop
st.w      sumAddr,d0               ; epilog
```

Now the number of cycles is reduced to 2 cycles per loop, compared with the previous 6 cycles per loop. This can be described in C as:

```
int sum = 0;
a = X[0];
b = Y[0];
for (i=1; i<N; i++)
          {
              c = a - b;
              a = X[i];
              sum += c*c;
              b = Y[I];
          }
```

Loop Unrolling

The equation is written twice or more, inside a loop. This technique is usually used with software pipelining.

Example: Implementation of a C array sum

```
z =0;
for (i=0; i<N; i++) z += X[i];
```

In TriCore assembly language this becomes:

```
mov d0,#0                               ; prolog
vsumloop:                               ; loop
          ld.w          d1,[Xptr+]4     ; (1)     ; ld X0
          add           d0,d0,d1        ; (2)     ; z
loop      LC,vsumloop
st.w      Zaddr,d0                      ; epilog
```

Here the Load and Add operations are performed in 2 cycles for a single element of the array. This can be improved by computing the addition of two elements at a time, in the loop:

```
mov       d0,#0                         ; prolog
ld.w      d1,[Xptr+]4                   ; ld X0
vsumloop:                               ; loop
          add           d0,d0,d1        ; (1)     ; z
          ld.w          d1,[Xptr+]4     ; ||      ; ld X1
          add           d0,d0,d1        ; (2)     ; z
          ld.w          d1,[Xptr+]4     ; ||      ; ld X2
loop      LC,vsumloop
st.w      Zaddr,d0                      ; epilog
```

Adding two elements now takes only 2 cycles instead of 4. This can also be written in C:

```
z = 0;
for (i=0; i<N/2; i++)
          {
           z+= X[2*i];
           z+= Z[2*i+1];
          }
```

Packed Operation

With packed operation, two different data are packed in the same register.

Example: Load of two 16-bits values in a register

```
ld.w ssX,[Xaddr]            ; ld X0 X1
```

Load / Store Scheduling

In Load/Store scheduling, the Load and Store instructions are reorganized to reduce the number of cycles.

Example: Transform routine

X0	→	X0’
X1	→	X1’

This transform takes 6 cycles:

```
ld.w       d0,[Xptr]        ; (1)
mul        d0,d0,#5         ; (2)
sub        d0,d0,#1         ; (3)
st.w       [Xptr+]4,d0      ; ||
ld.w       d0,[Xptr]        ; (4)
mul        d0,d0,#5         ; (5)
sub        d0,d0,#1         ; (6)
st.w       [Xptr+]4,d0      ; ||
```

A cycle is saved with Load and Store scheduling:

```
ld.w       d0,[Xptr]        ; (1)
mul        d0,d0,#5         ; (2)
ld.w       d1,[Xptr]+4      ; ||
sub        d0,d0,#1         ; (3)
st.w       [Xptr+]4,d0      ; ||
mul        d1,d1,#5         ; (4)
sub        d1,d1,#1         ; (5)
st.w       [Xptr+]4,d1      ; ||
```

Data Memory Interleaving

Here, data are mixed with other types of data in memory.

Example:

Xi0 at 0xD0000000 Xi1 at 0xD0000002 Xr0 at 0xD0000004 Xr1 at 0xD0000006	instead of:	Xi0 at 0xD0000000 Xr0 at 0xD0000002 Xi1 at 0xD0000004 Xr1 at 0xD0000006

Packed Load / Store

With Packed Load/Store at least two data are loaded or stored in the same instruction.

Example: Load two 32-bits values

Instead of:

```
ld.w       d0,[Xptr+]4        ; ld X0
ld.w       d1,[Xptr+]4        ; ld X1
```

It can be written as:

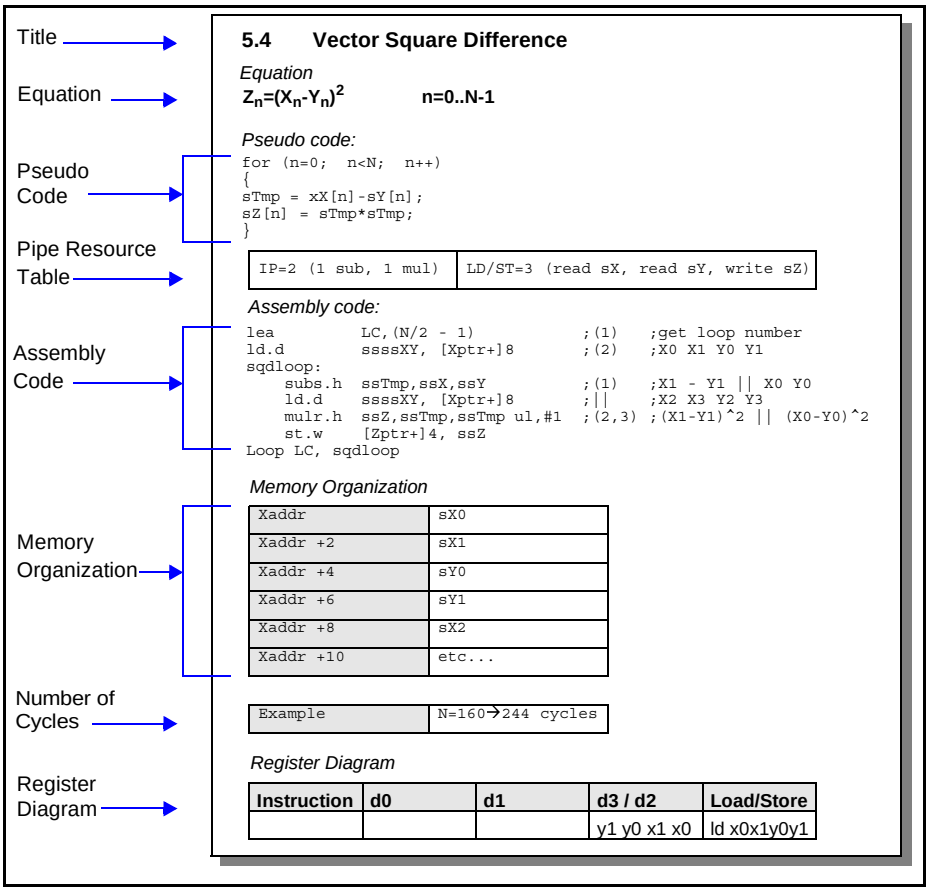
```
ld.d       e0,[Xptr+]8        ; ld X0 X1
```

1.1.2 Arithmetic Methods

- Saturation
  – at least one instruction is used with saturation.
- Rounding
  – at least one instruction is used with rounding.

1.2 Routine Organization

A typical routine page in this document is divided into the following different sections:



The topics shown in the figure above are described in the following sub-sections:

1.2.1 Equation

The Equation section gives the generic equation of the algorithm. When two (or more) variables are written back, there are several equations.

- *n* the variable is an array or a vector
- *r* the value is real
- *i* the value is imaginary

1.2.2 Pseudo Code

Pseudo Code provides an exact description of the equation(s). The ‘Pseudo C code’ has several advantages compared to C code:

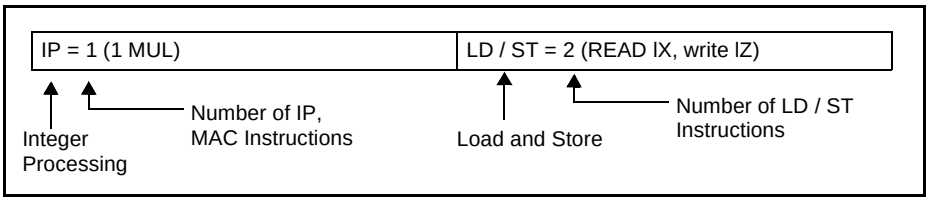
- There is no strict syntax requirement.
  The classic example is typecasting in C, which sometimes gives unreadable code.
- It may use ‘non C’ constructs. It is very difficult for example, to express 64-bit quantities (not a standard) and circular addressing in C.

The Pseudo code also determines the number of IP (Integer Processing) and LS (Load/ Store) instructions, as well as the memory bandwidth (2, 16-bit values do not give the same bandwidth as 2, 64-bit values). This is important to remember, as the Optimization can not perform better than implementing the minimum number of operations.

Pseudo Code Implementation

Pseudo code implementation is used to make the link between the description of specifications (in Pseudo code) and the implementation (in Assembly code). It explains how the specification Pseudo-code lines are actually computed in the Assembly code, and is very useful when Packed Techniques and Software Pipelining are used.

1.2.3 Pipe Resource Table



The Pipe Resource table acts as a ‘sanity check’, which can help in implementing the routine. For example, there is no need to optimize 2 instructions on the IP side of a routine if the bottleneck is due to 5 instructions on the LS side.

Note: As the pipe resource table is based on the Pseudo code, it will not show the number of IP and LS instructions inside the routine’s loop, because this is dependent on all of the Optimization techniques used/applied.

1.2.4 Assembly Code

The Assembly code given in this document is the actual TriCore Assembly code.

Where 'N' appears in the loop counter, it refers to the number of points given by the Equation or Pseudo code sections. Variable names (rather than registers) are used for easy readability. It should be noted that the pointers are not defined as these are generally declared in a global file.

Cycles are indicated in the Assembly code comments: (1), (2), (3). The total number of cycles is summarized in a table following the code. The time taken to enter and leave a loop is only indicated in the Information table.

1.2.5 Register Diagram

The Register diagram is an aid to visualizing the TriCore pipeline model. It also acts as a working sheet to optimize the algorithm.

The Register diagram is made up of the following fields:

- Instruction
  - Contains processing instructions (add, sub, madd, shifts, logic operation)
- d0…d7
  - The first eight registers are shown. The value in the register is the value at the end of the instruction. This can be confusing when a register is being loaded from memory at the same time as it is used in the instruction. The value is loaded after the calculation.
- Load / Store
  - Indicates when data is being read from memory or being stored back to memory.
- Bold borders
  - Indicates which instructions are in the loop.

1.2.6 Notation

Certain names and abbreviations are used to make the routines easier to understand. This notation is very useful when attempting to optimize a routine.

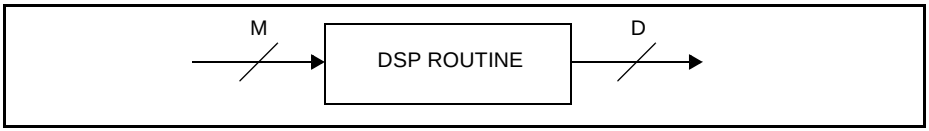
Abbreviation	Description
s	Short (16-bit value)
ss	Two short values are in a 32-bit register
ssss	Four short values are in a 64-bit register
l	Long (32-bit value)
ll	Long-long (2 long in a 64-bit register)
ll	Long-long (64-bit value)
||	Instruction executed in parallel
(1)	Cycle number
Xptr	Pointer for X values
Yptr	Pointer for Y values
Kptr	Pointer for K values
Zptr	Pointer for Z values
Vptr	Pointer for V values
Wptr	Pointer for W values
XBptr	Pointer for X values used for circular addressing (ex:a2/a3)
Xaddr	Address of X values
Kaddr	Address of K values
Yaddr	Address of Y values
Zaddr	Address of Z values
LC	Address register (usually a5) used as a loop counter

1.3 How to Test a DSP Routine

All of the routines described in this optimization guide are implemented in TriCore assembly code. The TriCore assembly code becomes, in effect, the reference code. The question that is then raised is how do we know it works? In other words, how do you test an optimized DSP routine? These questions are addressed in this section.

1.3.1 The Golden Models

For every type of routine there is a different input and output format. This, together with the way in which they use the memory, means that there is a specific way to test them. The essential idea is to see the routine as a 'black box':



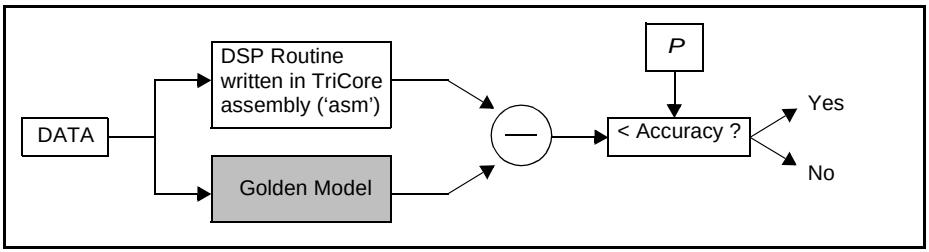
The 'Golden model' is used as the reference and is written in C.

When the data type is integer it gives a bit-exact result, which can be used to perform a direct comparison with the TriCore assembly implementation results.

When the data type is Floating-Point, the comparison cannot be exact. However the approximation of the result is sufficient to verify the TriCore implementation.

The best test process should have 3 steps:

- Compute the DSP routine and the Golden model on the same data buffer.
- Store the two results.
- Compute the difference between the Golden model and the TriCore assembly for each value, and keep the greatest difference. If this value is less than the maximum allowed error *P*, the test succeeds. If it exceeds the maximum allowed error then the implementation is either wrong or not accurate enough.
- If more precision is required, the comparison should be performed on a spreadsheet.



Test processes will be different for each kind of DSP algorithm, because of the different kind of input / output formats and different memory implementation.

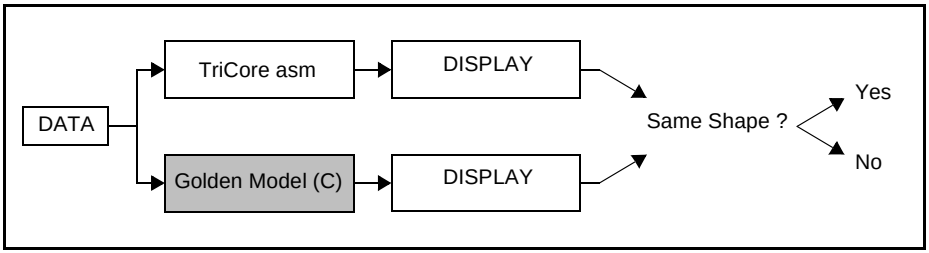
To help the programmer testing the routines, a project using Tasking EDE can be created. The project space should contain 6 projects, one for each type of routine:

- **Generators**
- **Scalars**
- **Filters**
- **Transcendental**
- **Vectors**
- **Transforms**

1.3.2 Generators

There are 5 assembly files (*.src) organized in 5 directories. The 5 generators are called by the C program (‘generators.c’) and are compared against the Golden models.

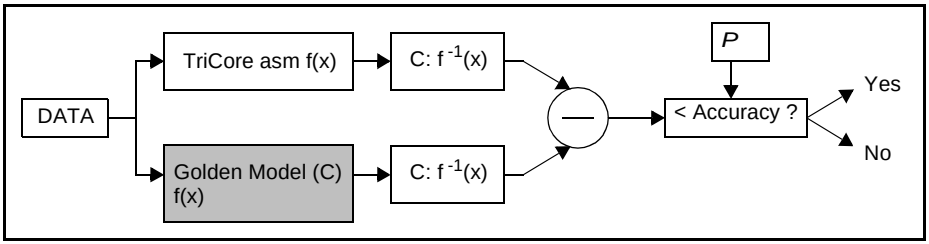
Only the complex wave generation is described in detail, as the other 4 generators are either directly written in C or are derived from the complex wave.



1.3.3 Transcendental

There are 6 transcendental assembly files (*.src), organized in 6 directories.

The two sine functions described in the manual are in the same file (‘testsin.src’), which also contains 8 sine versions representing different precision’s. All functions are called by the C program ‘trancendental.c’, and are compared against the Golden models.



Additionally, the inverse of each function is computed. This is easily achieved since a transcendental function such as y=f(x) will always have a corresponding $f^{-1}(x)$.

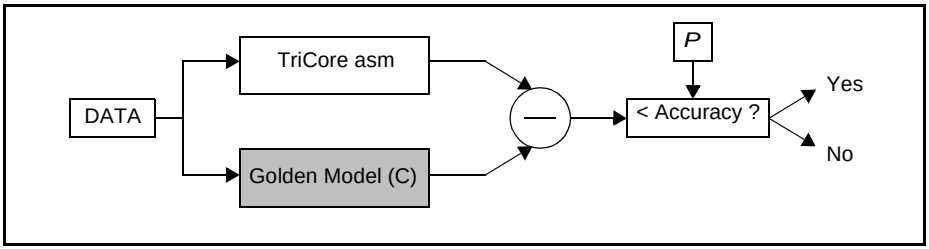
The principle advantage is that it is very fast to see mistakes. This is because the output ($f^{-1}(f(x))$ should be the same as the input.

For each routine there is an associated spreadsheet comparing precision.

1.3.4 Scalars

There are 12 scalar routines, organised as 11 scalar assembly files (8.src) in 8 directories. Two, 32-bit signed multiplications are in the same file.

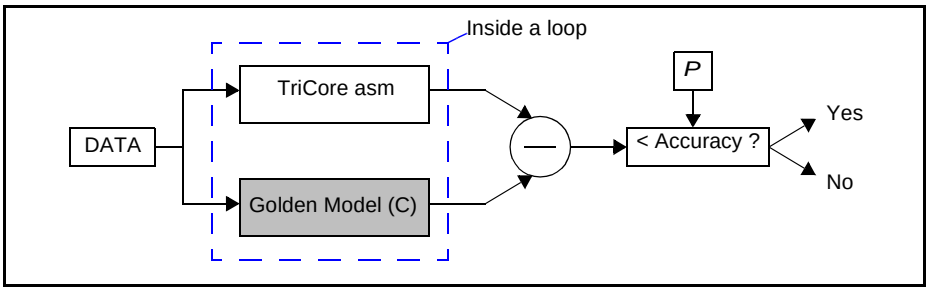
All functions are called by the C program ('scalars.c') and are compared against the Golden models.



In this instance the required accuracy is generally bit-exact, as Scalar routines are very easy to model in 32-bit integer C.

1.3.5 Vectors

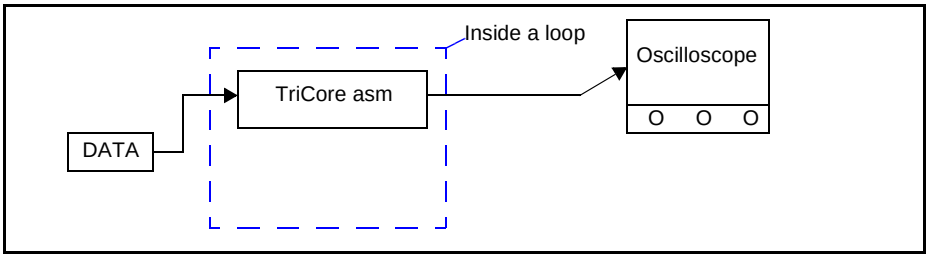
There are 10 vector assembly files (*.src), each in its own directory. All functions are called by the C program 'vectors.c', and are compared against the Golden models.



In this instance the required accuracy is generally bit-exact, as these routines are easy to model in 32-bit integer C.

1.3.6 Filters

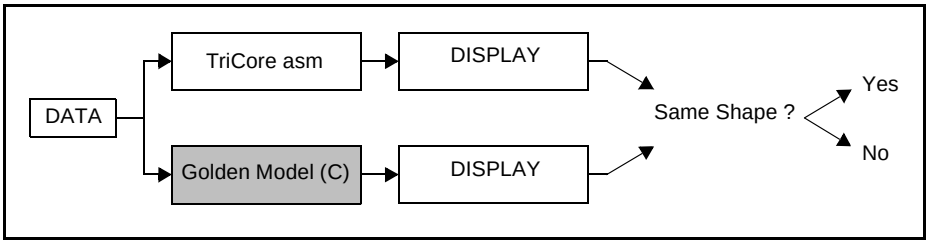
There are 19 filter assembly files (*.src), each in its own directory. The C program ‘filters.c’, calls all functions. There are no Golden models. Instead, the Tasking data analysis window ‘scope function’, can be used.



The routine is usually just the kernel of a filter, so it will be inside a loop and will require careful implementation in memory.

1.3.7 Transforms

There are 5 Transform assembly files (*.src), each in its own directory. All 5 routines are called by the C program ‘transform.c’, and compared against the Golden model.



1.4 Measuring Cycles

This section describes how to measure the number of cycles for a specific routine. As the test is dependent on the routine type (transcendentals, scalars, vectors, etc.), this section offers a universal method of testing.

Note: Some routines with loops require modifications. These are explained in this section.

A test should not change the code or the pointers. This means that it is important to be careful with the data memory mapping (When there is not enough space in the DMU, the test loop number should be decreased).

1.4.1 How to Count Cycles

The test code is a small program with the routine included. This code is called by an assembly function, CYCLE_COUNT().

The CYCLE_COUNT() function executes the code that starts in 0xD4000000 and returns an integer value, the number of cycles. The timer counter is read before the call, subtracted from the value after executing the test code, and then multiplied by two (because the timer counter is on the FPI clock, its speed is half of the CPU one). In the cycle test project this function is mapped in PMU, just after the test code.

Assembly code:

```
.sect "program.code"

CONST.A    .macro  reg,addr
           movh.a  reg,#((addr) + 0x8000 >> 16)
           lea reg,[reg]((((addr) + 0x8000) & 0xffff) - 0x8000)
.endm

CYCLE_COUNT:
CONST.A    a10,0xd4000000    ;load program location address
ld.w       d9,0xf0000310     ;load sys timer counter value before call
calli      a10               ;call the function
ld.w       d10,0xf0000310    ;load sys timer counter value after call
sub        d2,d10,d9         ;compute the difference
sh         d2,#1
ret
```

1.4.1.1 Counting Cycles for a Routine without Loops

The test program consists of a loop performed 1000 times with:

- 50 nop32
- The routine
- 50 add d0,d0,d0

To run the test:

1. Run an loop without the routine, execute and write down the cycle number N.
2. Include the routine inside the loop, execute and write down the cycle number M.
3. The number of cycles is (M-N)/1000.

The code is located in PMU (Rider-D: 0xD4000000 to 0xD4007fff), and the data should be in DMU (Rider-D: 0xD0000000 to 0xD0007fff).

Tasking macro operation .dup is used here for more clarity.

1. Run an empty Test Loop

```
Assembly code:
;####### DATA ###################################################
.sect "test_cycles.data"

;####### CODE ###################################################
.sect "test_cycles.code"

lea a5,999
. align 8
testloop:
  .dup 50
  nop32
  .endm
  ;--- include routine here
  ;--- end routine
  .dup 50
  add     d0,d0,d0
  .endm
  loop    a5,testloop
ret
```

2. Include the Routine in the Loop, and Data Memory

Assembly code:

```
;####### DATA ###################################################
.sect "test_cycles.data"

Xaddr: .half 0x1111
Kaddr: .half 0x2222
Zaddr: .half 0xDEAD

;####### CODE ###################################################
.sect "test_cycles.code"

NUM .set 50

lea       a5,999
 .align 8
superloop:
          .dup NUM
           nop32
          .endm
          ;--- include routine here
          ld.q              d1,Xaddr
          ld.q              d2,Kaddr
          mulr.q            d15,d1 u,d2 u, #1
          st.q              Zaddr,d15
          ;--- end routine
          .dup NUM
          add               d0,d0,d0
          .endm
loop      a5,superloop
ret
```

Note: Because of the addressing mode used in the routine, an initialization of pointers is usually required. This initialization should take place outside the loop (this should not be counted in the routine's number of cycles).

1.4.1.2 Counting Cycles for a Routine with Loops

Loops need to be aligned on an 8-byte boundary in order to be executed in the exact number of cycles predicted. Therefore to test the routine with a loop, alignment has to be carried out first, by including some nop16 and nop32 before the *test loop*.

The ld.d and st.d instructions also need special care. The pointer of the data loaded or stored needs to be aligned on an 8-byte boundary. Data memory and pointers' sometimes have to be changed to avoid a misalignment.

The test program consists of a jump loop, run 1000 times with:

- 50 nop32
- The pointer initialization
- The routine
- 50 add d0,d0,d0

To Run the Test:

1. Include the routine in the test loop and align data and loops.
2. Run a loop with the pointer initialization but without the routine (by commenting it out), execute and write down the cycle number N.
3. Remove the comments around the routine, execute and write down the cycle number M.
4. The number of cycles is (M-N)/1000.

Assembly code:

```
mov       d15,#999
testloop:

.dup 50
nop32
.endm

;--- include routine + init here ---
;--- end routine + init here -------

.dup 50
add       d0,d0,d0
.endm

jned      d15,#0, testloop
ret
```

Note: In this example the data register d15 is used for the jump. If this register is used in the routine, another one should be used (define macros area has to be checked).

Examples: Test of a routine with loop

Assembly code:

```
;######## CODE  ##############################################
.sect "test_cycles.code"
;-------------------------------------------------------------
;vector complex multiplication          (5cycles in the loop)
;-------------------------------------------------------------
.define    sYr        "d1"
.define    sYi        "d3"
.define    ssX        "d4"
.define    ssK        "d5"
.define    Xptr       "a2"
.define    Kptr       "a3"
.define    Yptr       "a4"
.define    LC         "a5"
.define    N          "8"

mov        d15,#999    ; testloop counter
nop32                  ; nops here to align label nmloop on a 8 bytes
                       ; boundary
nop16                  ;

testloop2:

.dup 50
nop32
.endm

;--- include routine + init here ---
lea        Xptr,buffin_vcplxmul1           ; Xptr
lea        Kptr,buffin_vcplxmul2           ; Kptr
lea        Yptr,buffout_vcplxmul           ; Yptr

lea        LC,(N-1)                        ; (1)   ; get loop number
movh       d6,#0                           ; (2)   ; clear 3rd source
mov        d7,#0                           ; (3)   ; clear 3rd source
nmloop:
           ld.w        ssK,[Kptr+]4        ; (1)   ; load k
           ld.w        ssX,[Xptr+]4        ; (2)   ; load x
           msubadm.h   e0,e6,ssK,ssX ul,#1 ; (3)   ; yr = xr*kr - xi*ki
           mulm.h      e2,ssK,ssX lu,#1    ; (4)   ; yi = xr*ki + xi*kr
           st.h        [Yptr+]2,sYr        ; ||    ; store yr
           st.h        [Yptr+]2,sYi        ; (5)   ; store yi
loop       LC,nmloop
;--- end routine + init here -------
```

```
.dup 50
add         d0,d0,d0
.endm

jned        d15,#0,testloop2
ret
```

Problems Related to Cycle Count

- ld.d and st.d:
  – Data pointer should be aligned on an 8-byte boundary.
    The memory mapping needs changes if the pointer is misaligned in the loop.
- Loop alignment:
  – Label should be aligned on an 8-byte boundary in PMU.
    Use nop32 and nop16 before the test loop to align the label.
- PMU cache (in rider-D: 0xD4000000 to 0xD4007FFF):
  – Because of the internal scratchpad, an address in the PMU cache could add one cycle.

Note: Warning for Rider-D

- Two loops inside each other will add more cycles, so this should be avoided
- Use 32-bit opcode inside a loop to maintain alignment

2 Generator

A Generator can be regarded as a moving vector (one or two dimensions), stored in memory, in a buffer. This window in memory is then displayed on a screen such as an oscilloscope.

2.1 Complex Wave Generation

Equation:

$X = X*K_r - Y*K_i$
$Y = X*K_i + Y*K_r$

Pseudo code:

```
for (n=0; n<N; n++)
{
sX = sX*sKr-sY*sKi;
sY = sX*sKi+sY*sKr;
}
```

IP= 4 (2 mul, 1 msub, 1 madd)	LD/ST= 1 in packed format (write sX || sY)

Assembly code:

```
lea       LC,(n-1)                          ; (1)    ; load loop counter
ld.w      k,[Kptr]                          ; (2)    ; ld rotation vector
ld.w      xy,startvect                      ; (3)    ; ld start vector
ldloop:
          mulr.h        temp,xy,k ll,#1     ; (1)    ; y' = y*b || x' = x*b
          st.w          [OUTptr+]4,xy       ; ||     ; st x1,y1 (next loop)
          maddsur.h     xy,temp,xy,k uu,#1  ; (2,3)  ; y' += x*k || x'-= y*k
loop      LC,ldloop
st.w      [OUTptr+]4,xy                     ; ||     ; st last x,y
```

Cycles	N = 100 → 305

Register diagram:

Instruction	d3	d5	d4	Load / Store
		kikr		ld kikr
			x y	ld x y
mulr.h temp,xy,k ll,#1	y=x*ki || x=x*kr			
maddsur.h xy,temp,xy,k uu,#1			y=y+y*kr || x=y-y*ki	st x y
				st x y

3 Transcendental Functions

Please note the following points on Transcendental functions:

- They are commonly used in domains other than DSP.
- They have more acute arithmetic problems (as opposed to signal processing problems).
- The same functions require different precision levels (application/programmer dependent)
- They do not easily lend themselves to multi-MAC operations, since they are inherently iterative (no parallelism) and they produce a single result.
- They can be implemented as a table look-up (space) or opposed to a series expansion (time).
- They can be implemented as a combination of space and time (partial look-up table and partial computation). This illustrates the first law of algorithms, that the space-time continuum is constant.

The table which follows summarises the Optimization techniques and Arithmetic methods that are applicable to the different types of Transcendental Functions.

Transcendental Functions Summary Table

Name	Cycles	Code Size [1]	Optimization Techniques						Arithmetic Methods	
			Software Pipelining	Loop Unrolling	Packed Operation	Load/Store Scheduling	Data Memory Interleaving	Packed Load/Store	Saturation	Rounding
Square Root (Newton-Raphson)	44	72	-	-	-	-	-	-	✓	✓
Square Root (Taylor)	26	82	-	-	-	✓	-	✓	✓	✓
Inverse	24	42	-	-	-	-	-	-	✓	✓
Natural Logarithm	16	46	-	-	-	-	-	-	-	✓
Exponential	22	40	-	-	-	-	-	-	-	✓
Sine [-PI/2,PI/2)	38	48	-	-	-	-	-	-	✓	-
Sine [-PI,PI)	42	68	-	-	-	-	-	-	✓	-

[1] Code Size is in Bytes

3.1 Square Root (by Newton-Raphson)

Equation:

Input: [0.25,1) in 1Q15 (X should be normalized to 0.25.. 1)

Output: 2Q14

$Y_0 = 1.1033 - 0.6666 * X$

$Y_{n+1} = Y_n * (1.5 - (X*2) * Y_n^2)$ **where n = 0,1,2**

Pseudo code:

```
// The loop calculates 1/(2*z)
sK = 2* sX;
sY = 1.1033 - 0.6666*sX;
for (n = 0; n<3; n++) sY = sY*(1.5- sK*sY*sY);
sZ = sY*(2*sX-1) + sY;
```

IP= 4 (3 mul, 1 sub)	LD/ST= 2 (load sK, store sY)

Assembly code:

```
movh     d0,#0x469c              ; (1)     ; 1.1033 in 2q14
lea      a4,2                    ; ||      ; initialization of counter
movh     d1,#0x5553              ; (2)     ; 0.6666 in 1q15
ld.q     sX,[a3+]                ; ||      ; load sX in 1q15
movh     d9,#0x8000              ; (3)     ; d9 = -1 in1q15
movh     d8,#0x1000              ; (4)     ; d8 = 0.5 in3q13
mov      sK,sX                   ; (5)     ; sK in 2q14(1q15->2q14->*2)
msubr.q  sY,d0,sX u,d1u,#0       ; (6)     ; Y02q14

sqrloop:
         mulr.q      tmp,sY u,sY u,#1      ; (1,2)     ;Y0^21q15
         sh          tmp,tmp,#-1           ; (3)       ;result in2q14
         msubrs.q    tmp,d8,tmp u,sK u,#1  ; (4,5)     ;0.5-sK*Y0^23q13
         shas        tmp,tmp,#1            ; (6)       ;result in 2q14
         sh          d1,sY,#-1             ; (7)       ;Y in 3q13
         maddrs.q    sY,d1,sY u,tmp u,#1   ; (8,9)
                                           ; Y0+Y0[0.5-sK*sY0^2]3q13
         shas        sY,sY,#1              ; (10)      ;Y1 in 2q14
loop     a4,sqrloop
adds     d6,sK,d9                ; (7)     ; 2*sX-1 in 1q15
mulr.q   d0,sY u,d6u,#1          ; (8,9)   ; (2*sX-1)/(2*sqrt(sX))
adds     d3,sY,d0                ; (10)    ; (2*sX-1)/(2*sqrt(sX))+1/
                                           ; (2*sqrt(sX)) = sqrt(sX)
```

3.2 Square Root (Taylor)

Equations:

Input: [0.5,1) in 2Q14 (X should be normalized to 0.5.. 1)

Output: 1Q15

x= (y – 1)/2

y^0.5 = 1 + x – 0.5*(x^2) + 0.5*(x^3) – 0.625*(x^4) + 0.875*(x^5)

Pseudo code:

```
sX[0] = (sY - 1)/2;
for(n=1;n<5;n++)       sX[n] = sX[0] * sX[n-1];
for(n=0;n<3;n++)       eY += sX[2*n] * sK[2*n] + sX[2*n+1] * sK[2*n+1];
```

IP= 1+2 (1 mul), (1 mul, 1 add)	LD/ST= 1 + 2 (st sX), (ld sX,sK)

Assembly code:

```
lea       a3,xsqrtvalue              ; (1)
lea       a4,3                       ; (2)      ;load loop counter
ld.q      d1,[a3]                    ; (3)      ;x
sh        d1,d1,#-1                  ; (4)      ;y/2
addih     d1,d1,#0xc000              ; (5)      ;x1=y/2-0.5
st.q      [a3+]2,d1                  ; ||
movh      d0,#0x8000                 ; (6)      ;x0
st.q      [a3+]2,d0                  ; ||
mov       d2,d1                      ; (7)
iloop:
          mulr.q     d2,d1u,d2u,#1   ; (1,2)    ;computation of x2,x3,x4,x5
          nop                        ; ||
          st.q       [a3+]2,d2       ; ||
loop      a4,iloop

lea       a2,ksqrtvalue              ; (8)
mov       d0,#0                      ; (9)      ;initialization of y(lower)
ld.w      d2,[a2+]4                  ; ||       ;k5k4
mov       d1,#0                      ; (10)     ;initialization of y(upper)
ld.d      e4,[-a3]8                  ; ||       ;x5x4x3x2

maddms.h  e0,e0,d5,d2ul,#1           ; (11,12)  ;y=y+x4k4+x5k5
ld.d      e2,[a2+]8                  ; ||       ;k3k2k1k0
maddms.h  e0,e0,d4,d2ul,#1           ; (13,14)  ;y=y+x2k2+x3k3
ld.w      d5,[-a3]4                  ; ||       ;x1x0
maddms.h  e0,e0,d5,d3ul,#1           ; (15,16)  ;y=y+x0k0+x1k1
st.h      YAddr,d1                   ; ||

ksqrtvalue:.half 0xb000,0x7000,0xc000,0x4000,0x7fff,0x8000 ;k5k4k3k2k1k0
xsqrtvalue:.half 0x7000,0x0000,0x0000,0x0000,0x0000,0x0000 ;x1x0x2x3x4x5
```

3.3 Inverse (y=1/x)

Equations:

Input: [+0.5..+1) in 2Q14.

Output: (+1..+2] in 2Q14.

$Y_{k+1} = 2*Y_k(1 – (X/2)*Y_k)$ X, Y are 16-bit values

Pseudo code:

```
sY = 1.457;
for(i=0;i<3;i++) sY = 2*sY*(1 - (sX/2)*sY)
```

IP= 4 (2 shift, 1 mul, 1 msub)	LD/ST= 0

Assembly code:

```
lea       a2,coef_inv                    ; (1)
lea       a4,0x02                        ; (2)
                                         ; number of iteration on the series
ld.q      d0,[a3+]                       ; (3)     ; load x    2q14
sh        d0,d0,#-1                      ; (4)     ; x/2    2q14
movh      d3,#0x2000                     ; (5)     ; 1 in    3q13
ld.q      d2,[a2]                        ; ||      ; y[0]= 1.457 in2q14
scond:
          msubrs.q    d4,d3,d0u,d2u,#1 ; (1,2)   ; temp = 1 - (x/2)*y 3q13
          sh          d4,d4,#1         ; (3)     ; temp = 1 - (x/2)*y 2q14
          mulr.q      d4,d4u,d2u,#1    ; (4,5)
                                         ; temp = y*(1 - (x/2)*y)3q13
          shas        d2,d4,#2         ; (6)     ; y = temp 2q14
loop      a4,scond

coef_inv: .half  0x5d3f                ;1.457
```

3.4 Natural Logarithm (y= ln(x))

Equations:

$Y = K_1*(X-1)+K_2*(X-1)^2+K_3*(X-1)^3+K_4*(X-1)^4+K_5*(X-1)^5$

Y, X, K are 16-bit values.

Input: [+1..+2) in 2Q14.

Ouput: in 1Q15

Pseudo code:

```
for(i=0;i<4;i++) sY *= sX + sK[n]
```

IP= 1 (1 madd)	LD/ST= 1 (1 ld sK)

Assembly code:

```
lea       a2,coef_log             ; (1)
                                  ; load the address of the first coeff k5
lea       a3,0x03                 ; (2)    ;initialize the counter
ld.q      d4,[a4+]2               ; (3)    ;load the number to log in 2q14
movh      d5,#0x4000              ; (4)    ;12q14
ld.q      d2,[a2+]2               ; ||     ;load k5
sub       d4,d4,d5                ; (5)    ;z = x - 1
ld.q      d3,[a2+]2               ; ||     ;load k4
sh        d4,d4,#1                ; (6)    ;result in 1q15
ilop:
          maddr.q       d2,d3,d2u,d4u,#1  ;(1,2)    ;
          ld.q          d3,[a2+]2         ;||       ;
loop      a3,ilop                 ; ||     ;(((k5*z+k4)*z+k3)z+k2)z+k1
mulr.q    d6,d2u,d4u,#1           ; (7,8)  ;((((k5*z+k4)z+k3)z+k2)z+k1)z

coef_log: .half 0x0404,0xeef8,0x2491,0xc149,0x7fe3  ;in 1Q15
```

3.5 Exponential ($y=e^X$)

Equations:

$$Y = K_1*X+K_2*X^2+K_3*X^3+K_4*X^4+K_5*X^5+K_6*X^6+K_7*X^7$$

Y, X, K are 16-bit values
Input: [0..1) in 1Q15
Output: in 3Q13

Pseudo code:

```
for(i=0;i<6;i++) sY *= sX + sK[n]
```

IP= 1 (1 madd)	LD/ST= 1 (1 ld sK)

Assembly code:

```
lea     a2,coef_exp         ; (1)   ;load address of the first coeff k7
lea     a3,0x05             ; (2)   ;initialize the counter
ld.q    d4,[a4+]2           ; (4)   ;load the number we would like the exp
                                    ;in 1q15
ld.q    d2,[a2+]2           ; (5)   ;load k7  2q14
ld.q    d3,[a2+]2           ; (6)   ;load k6  2q14
ilop:
        maddr.q  d2,d3,d2u,d4u,#1 ; (1,2)  ;2q14
        ld.q     d3,[a2+]2        ; ||     ;load next coef 2q14
loop    a3,ilop             ; ||
                            ; (((((k7*z+k6)z+k5)z+k4)z+k3)z+k2)z+k1
mulr.q  d6,d2u,d4u,#1       ; (7,8)
                            ; ((((((k7*z+k6)z+k5)z+k4)z+k3)z+k2)z+k1)z
                            ; 2q14
addih   d6,d6,#0x4001       ; (9)   ;add 1 to result
sh      d6,#-1              ; (10)  ;result in 3Q13

coef_exp:.half 0x0003,0x0016,0x0088,0x02aa,0x0aaa,0x2000,0x4000;in Q14
```

3.6 Sine (y =sin(x)), range [-Pi/2, Pi/2)

Equations:

$Sin(x)=k_1*x+k_2*x^3+k_3*x^5+k_4*x^7+k_5*x^9+...$

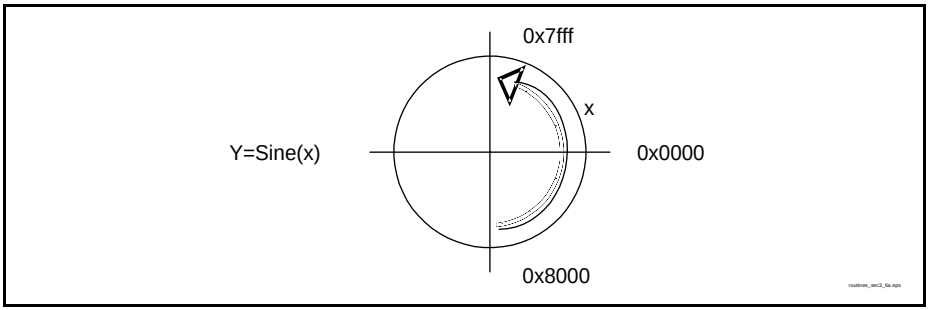
Input: [-1,1) in 1Q15

Output: [-1,1) in 1Q31

This can also be written as:

$Sin(x)=((((k_5*x^2+k_4)*x^2+k_4)*x^2+k_3)*x^2+k_2)*x^2+k_1)*x$

This series is valid for $x \in [-Pi/2, Pi/2]$, so the input between [-1,+1) is scaled to the range [-Pi/2, Pi/2) and gives a result between –1 and 1.



Pseudo code:

```
for(i=0;i<5;i++) lY *= lX + lK[n]
```

IP= 1 (1 madd)	LD/ST= 1 (1 ld sK)

Assembly code:

```
lea     a2,coef_sin         ; (1)       ;load the address of the first coeff
lea     a3,0x04             ; (2)       ;initialize the counter
ld.q    d4,[a4+]2           ; (3)       ;load number we would like the sine
1Q15
ld.w    d2,LX1              ; (4)       ;load the factor of norm
                            ; (PI/2)  2Q30
mul.q   d4,d4,d2,#1         ; (5,6,7)  ;x = x*a  2q30
mul.q   d1,d4,d4,#1         ; (8,9)             ;z = x*x, 3q29
ld.w    d2,[a2+]4           ; ||        ;load k5  1q31
ld.w    d8,[a2+]4           ; (10)      ;load k4  1q31
lloop:
        madds.q     d2,d8,d2,d1,#1  ; (1,2,3) ;give the result in 3q29
        sh          d2,d2,#2        ; (4)     ;1q31
        ld.w        d8,[a2+]4       ; ||      ;1q31
loop    a3,lloop            ; ||        ;(((k5*z+k4)*z+k3)z+k2)z+k1
mul.q   d6,d2,d4,#1         ; (11,12)  ;((((k5*z+k4)*z+k3)z+k2)z+k1)x 2q30
shas    d6,d6,#1            ; (13)      ;1q31

LX1:  .word 0x6487ED51;PI/2 in Q30

coef_sin:.word
0xffffffca,0x000005c7,0xfffe5fe6,0x00444444,0xfaaaaaab,0x20000000;1Q31
and 3Q29
```

3.7 Sine (y =sin(x)), range [-Pi, Pi)

Equations:

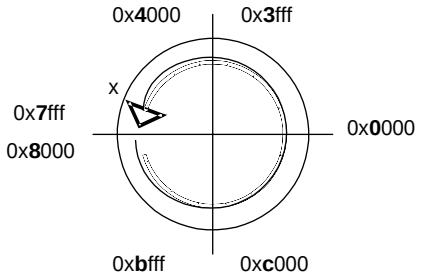
$Sin(x)=k_1*x+k_2*x^3+k_3*x^5+k_4*x^7+k_5*x^9+\ldots$

Input: [-1,1) in 1Q15

Output: [-1,1) in 1Q31

By using the previous sine, we can get the result for the range [-Pi, Pi] with:

- Sin(Pi-x) = Sin(x)



Change of variable:

If 0x4000<x<0xbfff f Then x'=(0x7fff-x)<<1

If 0xc000<x<0x3ff Then x'=x<<1

Y=Sine(x')

Pseudo code:

```
for(i=0;i<5;i++) lY *= lX + lK[n]
```

IP= 1 (1 madd)	LD/ST= 1 (1 ld sK)

Assembly code:

```
lea       a2,coef_sin0          ; (1)        ; load address of first coeff
lea       a3,0x04               ; (2)        ; initialize the counter
ld.q      d4,[a4+]2             ; (3)        ; load number we would like
                                               ; the sine 1Q15

;change of variable
movh      d9,#0x8000            ; (4)        ; -1
xor.t     d2,d4:31,d4:30        ; (5)        ; 0x4000<x<0xbfff ?
jz        d2,lab                ; (6,7)      ; if not, go to lab
add       d4,d9,d4              ; (8)        ; else x = x-1
rsub      d4,d4,#0              ; (9)        ; x = -(x-1) = 1-x
lab:     sh d4,#1               ; (10)       ; x'=x<<1
                                               ; end change of variable

ld.w      d2,LX1                ; (11)
                                ; load the factor of norm (PI/2)  2Q30
mul.q     d4,d4,d2,#1           ; (12,13,14) ; x = x*a 2q30
mul.q     d1,d4,d4,#1           ; (15,16)    ; z = x*x, 3q29
ld.w      d2,[a2+]4             ; ||         ; load k51q31
ld.w      d8,[a2+]4             ; (17)       ; load k41q31
llloop:
          madds.q    d2,d8,d2,d1,#1          ; (1,2,3)
                                               ; give the result in 3q29
          sh         d2,d2,#2   ; (4)        ; 1q31
          ld.w       d8,[a2+]4  ; ||         ; 1q31
loop      a3,llloop             ; ||         ; (((k5*z+k4)*z+k3)z+k2)z+k1
mul.q     d6,d2,d4,#1           ; (18,19)
                                ; ((((k5*z+k4)*z+k3)z+k2)z+k1)x 2q30
shas      d6,d6,#1              ; (20)       ; 1q31

LX1:  .word 0x6487ED51;PI/2 in Q30

coef_sin0:.word
0xffffffca,0x000005c7,0xfffe5fe6,0x00444444,0xfaaaaaab,0x20000000;1Q31
and 3Q29
```

4 Scalars

Name	Cycles	Code Size	Optimization Techniques						Arithmetic Methods	
		Bytes	Software Pipelining	Loop Unrolling	Packed Operation	Load/Store Scheduling	Data Memory Interleaving	Packed Load/Store	Saturation	Rounding
16-bit signed Multiplication	3,4	16	-	-	-	-	-	-	-	✓
32-bit signed Multiplication 32-bit result	5	16	-	-	-	-	-	-	-	-
32-bit signed Multiplication 64-bit result	5	16	-	-	-	-	-	-	-	-
“C” integer multiplication	5	16	-	-	-	-	-	-	-	-
16-bit update	5	20	-	-	-	-	-	-	✓	✓
32-bit update	6,5	20	-	-	-	-	-	-	✓	-
2nd order diff. equation (16-bit)	4	24	-	-	✓	-	-	✓	-	-
2nd order diff. equation (32-bit)	4	24	-	-	✓	-	-	✓	-	-
Complex multiplication	5	20	-	-	-	-	-	-	-	-
Complex multiplication (packed)	5	14	-	-	✓	-	-	✓	✓	✓
Complex update	7	24	-	-	-	-	-	-	-	-
Complex update (packed)	6	18	-	-	✓	-	-	✓	✓	✓

4.1 16-bit signed Multiplication

Pseudo code:

```
sZ = sX*sK;
```

IP= 1 (1 mul)	LD/ST= 3 (read lX, read lK, write lZ)

Assembly code:

```
;result = 16-bit
ld.q      sX, Xaddr              ; (1)    ;
ld.q      sK, Kaddr              ; (2)    ;
mul.q     lZ,sX u,sK u,#1        ; (3)    ; left justified
st.q      Zaddr,sZ               ; ||     ; store 16-bit upper

;same with result = 32-bit
ld.q      sX, Xaddr              ; (1)    ;
ld.q      sK, Kaddr              ; (2)    ;
mul.q     lZ,sX u,sK u,#1        ; (3)    ; left justified
st.w      Zaddr,lZ               ; ||     ; store 32-bit

;same with result = rounded 16-bit
ld.q      sX, Xaddr              ; (1)    ;
ld.q      sK, Kaddr              ; (2)    ;
mulr.q    sZ,sX u,sK u,#1        ; (3,4)  ; left justified
st.q      Zaddr,sZ               ; ||     ; store 16-bit upper
```

4.2 32-bit signed Multiplication

Pseudo code:

```
lZ = lX*lK;
```

IP= 1 (1 mul)	LD/ST= 3 (read lX, read lK, write lZ)

Assembly code:

```
; lX,lK,lZ  32-bit signed
ld.w      lX, Xaddr              ; (1)      ;
ld.w      lK, Kaddr              ; (2)      ;
mul.q     lZ,lX,lK,#1            ; (3,4,5)  ;
st.w      Zaddr,lZ               ; ||       ;
```

4.3 32-bit signed Multiplication (Result on 64-bit)

Pseudo code:

llZ = lX*lK; all signed

IP= 1 (1 mul)	LD/ST= 3 (read lX, read lK, write llZ)

Assembly code:

```
; lX,lK 32-bit signed, llZ 64-bit signed
ld.w      lX, Xaddr              ; (1)      ;
ld.w      lK, Kaddr              ; (2)      ;
mul.q     llZ,lX,lK,#1           ; (3,4,5)  ;
st.d      Zaddr,llZ              ; ||       ; same number of cycles
```

4.4 'C' Integer Multiplication

Pseudo code:

This multiplication takes the lower part of the result (MUL), whereas previous multiplications take the upper part (MUL.Q).

```
lZ = lX*lK;
```

IP= 1 (1 mul)	LD/ST= 3 (read lX, read lK, write lZ)

Assembly code:

```
; lX,lK,lZ  32-bit signed
ld.w       lX, Xaddr              ; (1)      ;
ld.w       lK, Kaddr              ; (2)      ;
mul        lZ,lX,lK               ; (3,4,5)  ;
st.w       Zaddr,lZ               ; ||       ;

; lX,lK,lZ  32-bit unsigned
ld.w       lX, Xaddr              ; (1)      ;
ld.w       lK, Kaddr              ; (2)      ;
mul        lZ,lX,lK               ; (3,4,5)  ;
; unsigned multiplication gives the same result
; as a signed multiplication. That is why there is no MUL.U instruction.
st.w       Zaddr,lZ               ; ||       ;

; lX,lK,lZ  32-bit signed (saturated result)
ld.w       lX, Xaddr              ; (1)      ;
ld.w       lK, Kaddr              ; (2)      ;
muls       lZ,lX,lK               ; (3,4,5)  ;
st.w       Zaddr,lZ               ; ||       ;

; lX,lK,lZ  32-bit unsigned (saturated result)
ld.w       lX, Xaddr              ; (1)      ;
ld.w       lK, Kaddr              ; (2)      ;
muls.u     lZ,lX,lK               ; (3,4,5)  ;
st.w       Zaddr,lZ               ; ||       ;
```

4.5 16-bit Update

Pseudo code:

```
sZ = sY + sX*sK;
```

IP = 1 (1 madd)	LD/ST= 4 (read sX, read sK, read sY, write sZ)

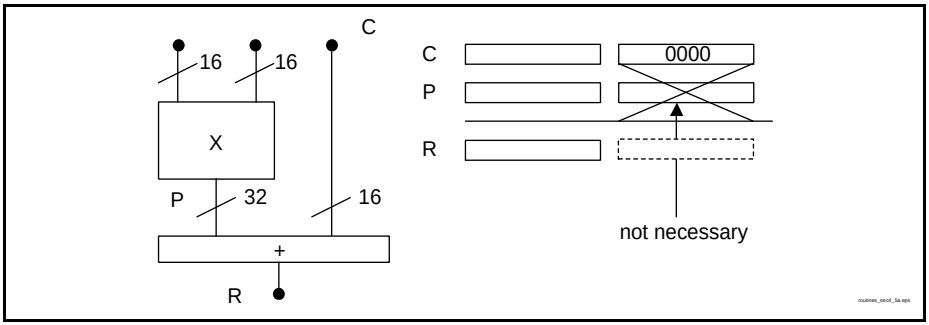
Assembly code:

```
;result = 32-bit
ld.q      sX,Xaddr                ; (1)      ;
ld.q      sK,Kaddr                ; (2)      ;
ld.q      sY,Yaddr                ; (3)      ;
madds.q   lZ,sY,sX u,sK u,#1      ; (4,5)    ;
st.q      Zaddr,sZ                ; ||       ;

;same with result = rounded 16-bit
ld.q      sX,Xaddr                ; (1)      ;
ld.q      sK,Kaddr                ; (2)      ;
ld.q      sY,Yaddr                ; (3)      ;
maddrs.q  sZ,sY,sX u,sK u,#1      ; (4,5)    ;
st.q      Zaddr,sZ                ; ||       ;
```

The pseudo code sZ = sY + sX*sK would have been written sZ = sY + (long) (sX*sK), since the result of a 16*16-bit multiplication is a 32-bit result in hardware.

sZ = sY + (long) (sX*sK) is actually equivalent to sZ = sY + (short) (sX*sK). The proof can be seen in the following figure:



Since the lower part of C is only zeros, it will not change the lower part of P. We can therefore say that the result of the 16*16-bit multiplication is a short value. The pseudo code is:

```
sZ = sY + sX*sK
```

4.6 32-bit Update

Pseudo code:

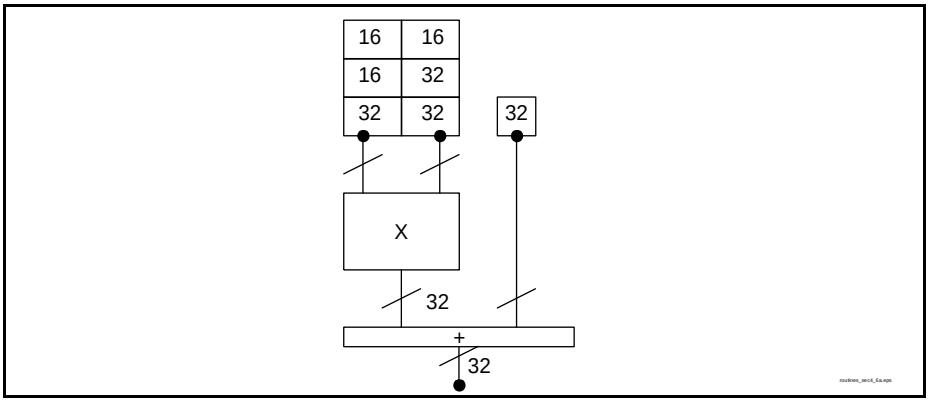
```
lY += lX*lK;
```

IP = 1 (1 madd)	LD/ST= 4 (read lX, read lK, read lY, write lY)

Assembly code:

```
;32*32-bit multiplication, result = 32-bit
ld.w      lX,Xaddr                ; (1)      ;
ld.w      lK,Kaddr                ; (2)      ;
ld.w      lY,Yaddr                ; (3)      ;
madds.q   lY,lY,lX,lK,#1          ; (4,5,6)  ;
st.w      Yaddr,lY                ; ||       ;
```

A 32-bit update can not only be executed with a 32*32-bit multiplication, but also with a 16*32-bit multiplication and a 16*16-bit multiplication.




```
;16*32-bit multiplication, result = 32-bit
ld.q      sX,Xaddr                ; (1)      ;
ld.w      lK,Kaddr                ; (2)      ;
ld.w      lY,Yaddr                ; (3)      ;
madds.q   lY,lY,lK,sX u,#1        ; (4,5)    ;
st.w      Yaddr,lY                ; ||       ;
```

```
;16*16-bit multiplication, result = 32-bit
ld.q      sX,Xaddr                ; (1)      ;
ld.w      lK,Kaddr                ; (2)      ;
ld.w      lY,Yaddr                ; (3)      ;
madds.q   lY,lY,sK u,sX u,#1      ; (4,5)    ;
st.w      Yaddr,lY                ; ||       ;
```

4.7 2nd Order Difference Equation (16-bit)

Pseudo code:

```
sY = sX - 2*sX1 + sX2;
```

IP= 3 (1 add, 1 sub, 1mul)	LD/ST= 4 (read sX, read sX1, read sX2, write sY)

Optimization note:

sXX is 32-bit register holding the 2 16-bit variables sX, sX1

Assembly code:

```
ld.w      sXX, [Xptr]          ; (1)      ; X1 ||  X
sh        sX1,sXX,#-15         ; (2)      ; 0000 || X1*2
sub       sY,sXX,sX1           ; (3)      ; Y = X - 2*X1
ld.h      sX2, [Xptr]+4        ; ||       ;
add       sY,sY,sX2            ; (4)      ; Y = X - 2*X1 + X2
st.h      Yaddr,sY             ; ||       ;
```

Memory organization:

[Xptr] -->

Xaddr	sX
Xaddr + 2	sX1
Xaddr + 4	sX2

4.8 2nd Order Difference Equation (32-bit)

Pseudo code:

```
lY = lX - 2*lX1 + lX2;
```

IP= 3 (1 add, 1 sub, 1mul)	LD/ST= 4 (read lX, read lX1, read lX2, write lY)

Assembly code:

```
ld.d    lX1/lX, [Xptr]          ; (1)     ; lX1 || lX
sh      lX1,lX1,#1              ; (2)     ; lX1*2
sub     lY,lXX,lX1              ; (3)     ; lX - 2*lX1
ld.w    lX2, [Xptr]+8           ; ||      ;
add     lY,lY,lX2               ; (4)     ; lX - 2*lX1 + lX2
st.w    Yaddr,lY                ; ||      ;
```

Memory organization:

[Xptr] -->

Xaddr	lX
Xaddr + 4	lX1
Xaddr + 8	lX2

4.9 Complex Multiplication

Equations:

$Y_r = X_r*K_r - X_i*K_i$

$Y_i = X_r*K_i + X_i*K_r$

Pseudo code:

```
sYr = sXr*sKr – sXi*sKi;
sYi = sXr*sKi + sXi*sKr;
```

IP= 4 (2 mul, 1 madd, 1msub)	LD/ST= 6 (read sXr, sXi, sKr, sKi, write sYr, sYi) is equivalent to (packed format) = 3 (read sXr||sXi, sKr||sKi, write sYr||sYi)

Assembly code:

```
mov       d6,#0                  ; (1)      ;
ld.w      ssK,[Kptr+]4           ; ||       ; Ki Kr
mov       d7,#0                  ; (2)      ;
ld.w      ssX,[Xptr+]4           ; ||       ; Xi Xr

msubadm.h e0,e6,ssK,ssX ul,#1    ; (3)      ; Yr = Xr*Kr - Xi*Ki
mulm.h    e2,ssK,ssX lu,#1       ; (4)      ; Yi = Xr*Ki + Xi*Kr
st.h      [Yptr+]2,sYr           ; ||       ; store Yr
st.h      [Yptr+]2,sYi           ; (5)      ; store Yi
```

Note: TriCore MULM.H does not have a direct subtraction, only addition. A MADDSUM.H instruction is used to get the subtraction, with the 3rd source register set to 0 (e6). The 2 results are not packed in one register, and so two stores are required.

Register diagram:

Instruction	d1/ d0	d3/ d2	d5	d4	d7/ d6	Load / Store
					0	
			kikr			ld krki
					0	
				xixr		ld xrxi
msubadm.h e0,e6,ssK,ssX ul,#1	$y_r=x_r*k_r-x_i*ki$					st yr
mulm.h e2,ssK,ssX lu,#1		$y_i=x_r*k_i+x_i*k_r$				st yi

4.10 Complex Multiplication (Packed)

Equations:

$Y_r = X_r*K_r - X_i*K_i$

$Y_i = X_r*K_i + X_i*K_r$

Pseudo code:

```
sYr = sXr*sKr – sXi*sKi;
sYi = sXr*sKi – sXi*sKr;
```

IP= 4 (2 mul, 1 madd, 1 msub)	LD/ST= 6 (read sXr, sXi, sKr, sKi, write sYr, sYi) is equivalent to (packed format) = 3 (read sXr||sXi, sKr||sKi, write sYr||sYi)

Pseudo code implementation:

```
sYi = sXr*sKi;  sYr = sXr*sKr;
sYi += sXi*sKr;  sYr -= sXi*sKi;
```

Assembly code:

```
ld.w        ssK,[Kptr+]4            ; (1)     ; Ki Kr
ld.w        ssX,[Xptr+]4            ; (2)     ; Xi Xr
mulr.h      ssY,ssK,ssX ll,#1       ; (3)     ; Yi =Xr*Ki || Yr = Xr*Kr
maddsurs.h  ssY,ssY,ssK,ssX uu, #1  ; (4,5)   ; Yi +=Xi*Kr || Yr -= Xi*Ki
st.w        [Yptr+],ssY             ; ||      ; store Yi Yr
```

Note: In this example we save one store because the computed results are packed in one register. Rounding is used to pack them.

Register diagram:

Instruction	d0	d5	d4	Load / Store
		ki kr		ld krki
			xi xr	ld xrxi
mulr.h ssY,ssK,ssX ll,#1	yi = xr*ki ||yr = xr*kr			
maddsurs.h ssY,ssY,ssK,ssX uu, #1	yi += xi*kr ||yr -= xi*ki			st yiyr

4.11 Complex Update

Equations:

$Z_r = Y_r + X_r*K_r - X_i*K_i$

$Z_i = Y_i + X_r*K_i + X_i*K_r$

Pseudo code:

```
sZr = sYr + sXr*sKr - sXi*sKi;
sZi = sYi + sXr*sKi + sXi*sKr;
```

IP= 4 (3 madd, 1 msub)	LD/ST= 8 (read sXr, sXi, sKr, sKi, sYr, sYi, write sZr, sZi) equivalent to (packed format)= 4 (read sXr|| sXi, sKr || sKi, sYr || sYi, write sZr || sZi)

Assembly code:

```
movh       d2,#0                  ; (1)      ;
ld.w       ssK,[Kptr+]4           ; ||       ; Ki Kr
ld.w       ssX,[Xptr+]4           ; (2)      ; Xi Xr
ld.h       sYr,[Yptr+]2           ; (3)      ; Yr
msubadm.h  e0,e2,ssK,ssX ul, #1   ; (4,5)    ; Zr = Yr + Xr*Kr - Xi*Ki
ld.h       sYi,[Yptr+]2           ; ||       ; Yi
st.h       [Zptr+]2,sZr           ; ||       ; store Zr
maddm.h    e0,e2,ssK,ssX lu,#1    ; (6,7)    ; Zi = Yi + Xr*Ki + Xi*Kr
st.h       [Zptr+]2,sZi           ; ||       ; store Zi
```

The MSUBADM.H instruction is used because the memory organization is imaginary || real. The equation is zr = yr – (xi*ki - xr*kr), which is equivalent to zr = yr + xr*kr - xi*ki.

Register diagram:

Instruction	d1/ d0	d2	d3	d5	d4	Load / Store
		0		kikr		ld kikr
					xixr	ld xixr
			yr			ld yr
msubadm.h e0,e2,ssK,ssX ul,#1	zr = yr + xr*kr - xi*ki		yi			ld yi
						st zr
maddm.h e0,e2,ssK,ssX lu,#1	zi = yi + xr*ki + xi*kr					
						st zi

4.12 Complex Update (Packed)

Equations:

$Z_r = Y_r + X_r*K_r - X_i*K_i$

$Z_i = Y_i + X_r*K_i + X_i*K_r$

Pseudo code:

```
sZr = sYr + sXr*sKr - sXi*sKi;
sZi = sYi + sXr*sKi + sXi*sKr;
```

IP= 4 (3 madd, 1 msub)	LD/ST= 8 (read sXr, sXi, sKr, sKi, sYr, sYi, write sZr, sZi) equivalent to (packed format)= 4 (read sXr|| sXi, sKr || sKi, sYr || sYi, write sZr || sZi)

Pseudo code implementation:

```
sZi  = sYi + sXr * sKi; sZr   = sYr + sXr * sKr;
sZi += sXi * sKr; sZr -= sXi * sKi;
```

Assembly code:

```
ld.w        ssK,[Kptr+]4           ; (1)   ; Ki Kr
ld.w        ssX,[Xptr+]4           ; (2)   ; Xi Xr
ld.w        ssY,[Yptr+]4           ; (3)   ; Yi Yr
maddrs.h    ssZ,ssY,ssK,ssX ll,#1 ; (4)   ; Zi = Yi+Xr*ki || Zr = Yr+Xr*Kr
maddsurs.h ssZ,ssZ,ssK,ssX uu,#1 ; (5,6) ; Zi +=Xi*kr || Zr -= Xi*Ki
st.w        [Zptr+]4,ssZ           ; ||    ; store Zi Zr
```

Note: Rounding is used to pack the 2 results in 1 register.They can be stored in one instruction.

Register diagram:

Instruction	d0	d2	d4	d5	Load / Store
				kikr	ld kikr
			xixr		ld xixr
		yiyr			ld yiyr
maddrs.h ssZ,ssY,ssK,ssX ll,#1	zi=yi+xr*ki ||zr=yr+xr*kr				
maddsurs.h ssZ,ssZ,ssK,ssX uu,#1	zi=zi+xi*kr || zr=zr-xi*ki	yiyr			ld yiyr
	zizr				st zizr

5 Vectors

Name	Cycles	Code Size [1]	Optimization Techniques						Arithmetic Methods	
			Software Pipelining	Loop Unrolling	Packed Operation	Load/store Scheduling	Data memory interleaving	Packed Load/store	Saturation	Rounding
Vector sum	(3*N/4 +2)+ 3	32	✓	✓	✓	✓	-	✓	✓	-
Vector multiplication	(3*N/4 +2) +3	32	✓	✓	✓	✓	-	✓	-	-
Vector pre-emphasis	(3*N/4 +2) +3	40	✓	✓	✓	✓	-	✓	✓	✓
Vector square difference	(3*N/2 +2) +2	24	✓	-	✓	✓	✓	✓	✓	✓
Vector complex multiplication	(5*N +2) +3	28	-	-	✓	-	-	✓	-	-
Vector complex multiplication (packed)	(4*N +2) +3	24	-	-	✓	-	-	✓	✓	✓
Vector complex multiplication (unroll)	(2*N +2) +4	42	✓	✓	✓	-	✓	✓	✓	✓
Color space conversion	11	64	✓	✓	✓	-	✓	✓	-	-
Vector scaling	(2*N/2 +2) +2	28, 20	-	-	-	-	-	✓	✓	✓
Vector normalization	(2*N/2 +2) + (2*N/2 +2) +7	54	-	-	✓	-	-	✓	-	-

[1] Code Size is in Bytes

5.1 Vector Sum

Equation:

$Z_n = V_n + W_n$ n = 0..N-1

Pseudo code:

```
for (n=0; n<N; n++) sZ[n] = sV[n] + sW[n];
```

IP= 1 (1 add)	LD/ST= 3 (read sV, read sW, write sZ)

Assembly code:

```
lea       LC,(N/4-1)                    ; (1)    ; get loop number
ld.w      ssV0,[Vptr+]4                 ; (2)    ; V0 V1
ld.d      ssssW,[Wptr+]8                ; (3)    ; W0 W1 W2 W3
vadloop:
          adds.h        ssZ0,ssV0,ssW0  ; (1)    ; V1+W1 || V0+W0
          ld.d          ssssV,[Vptr+]8  ; ||     ; V2 V3 V4 V5
          adds.h        ssZ1,ssV1,ssW1  ; (2)    ; V3+W3 || V2+W2
          ld.d          ssssW,[Wptr+]8  ; ||     ; W4 W5 W6 W7
          st.d          [Zptr+]8,ssssZ  ; (3)    ; store Z0 Z1 Z2 Z3
loop      LC,vadloop
```

Example	N = 64 ➔ 53 cycles

Register diagram:

Instruction	**d1 / d0**	**d5 / d4**	**d7/ d6**	**Load / Store**
		v1 v0 _ _		ld v0v1
			w3 w2 w1 w0	ld w0w1w2w3
adds.h ssZ1,ssV2,ssW1	d1= ---------- d0= v1+w1 || v0+w0	v5 v4 v3 v2		ld v2v3v4v5
adds.h ssZ2,ssV1,ssW2	d1= v3+w3 || v2+w2		w7 w6 w5 w4	ld w4w5w6w7
	d1= z3 z2 d0= z1 z0			st z0z1z2z3

5.2 Vector Multiplication

Equation:

$Z_n = V_n * W_n$ n = 0..N-1

Pseudo code:

```
for (n=0; n<N; n++)  sZ[n] = sV[n] * sW[n];
```

IP= 1 (1 mul)	LD/ST= 3 (read sV, read sW, write sZ)

Assembly code:

```
lea       LC,(N/4 - 1)            ; (1)    ; get loop number
ld.w      ssV0,[Vptr+]4           ; (2)    ; V0 V1
ld.d      ssssW,[Wptr+]8          ; (3)    ; W0 W1 W2 W3

vect2loop:
          mulr.h   ssZ0,ssV0,ssW0 ul,#1    ; (1)    ; V1*W1 || V0*sW0
          ld.d     ssssV,[Vptr+]8          ; ||     ; V2 V3 V4 V5
          mulr.h   ssZ1,ssV1,ssW1 ul,#1    ; (2,3)  ; V3*W3 || V2*W2
          ld.d     ssssW,[Wptr+]8          ; ||     ; W4 W5 W6 W7
          st.d     [Zptr+]8,ssssZ          ; ||     ; store Z0 Z1 Z2 Z3
loop      LC,vect2loop
```

Example	N = 64 ➔ 53 cycles

Register diagram:

Instruction	d1/ d0	d5 / d4	d7/ d6	Load / Store
		v1 v0 _ _		ld v0v1
			w3 w2 w1 w0	ld w0w1w2w3
mulr.h ssZ0,ssV0,ssW0 ul,#1	_ _ z1 z0	v5 v4 v3 v2		ld v2v3v4v5
mulr.h ssZ1,ssV1,ssW1 ul,#1	z3 z2 z1 z0		w7 w6 w5 w4	ld w4w5w6w7
	z3 z2 z1 z0			st z0z1z2z3

5.3 Vector Pre-emphasis

Equation:

$Z_n = V_n + X_n*K$ n = 0..N-1 K = -28180

Pseudo code:

```
for (n=0; n<N; n++)  sZ[n] = sV[n] + sX[n]*sK;
```

IP= 1 (1 madd)	LD/ST= 3 (read sX, read sV, write sZ)

Assembly code:

```
lea     LC,(N/4 -1)                 ; (1)   ;get loop number
mov.u   d6,#0x91ec                  ; (2)   ;K=-28180
ld.w    ssX0,[Xptr+]4               ; ||    ;X0 X1
addih   d6,d6,#0x91ec               ; (3)   ;K || K
ld.d    ssssV,[Vptr+]8              ; ||    ;V0 V1 V2 V3

preloop:
        maddrs.h    ssZ0,ssV0,ssX0,d6 ul,#1 ;(1); Z1=V1+X1*K ||Z0=V0+X0*K
        ld.d        ssssX,[Xptr+]8  ; ||    ;X2 X3 X4 X5
        maddrs.h    ssZ1,ssV1,ssX1,d6 ul,#1 ;(2,3)
                                    ;Z3=V3+X3*K ||Z2=V2+X2*K
        ld.d        ssssV,[Vptr+]8  ; ||    ;V4 V5 V6 V7
        st.d        [Zptr+]8,ssssZ  ; ||    ;store Z0 Z1 Z2 Z3
loop    LC,preloop
```

Example	N = 160 ➔ 125 cycles

Register diagram:

Instruction	d1/ d0	d3 / d2	d5 / d4	d6	Load / Store
		x1x0 _ _		k	ld x0x1
			v3 v2 v1 v0	k||k	ld v0v1v2v3
maddrs.h ssZ0,ssV0,ssX0,d6 ul,#1	_ _ z1 z0	x5 x4 x3 x2			ld x2x3x4x5
maddrs.h ssZ1,ssV1,ssX1,d6 ul,#1	z3 z2 z1 z0		v7 v6 v5 v4		ld v4v5v6v7
					st z0z1z2z3

5.4 Vector Square Difference

Equation:

$Z_n = (X_n - Y_n)^2$ $n = 0..N-1$

Pseudo code:

```
for (n=0; n<N; n++)
{
sTmp = sX[n]-sY[n];
sZ[n] = sTmp*sTmp;
}
```

IP= 2 (1 sub, 1 mul)	LD/ST= 3 (read sX, read sY, write sZ)

Assembly code:

```
lea       LC, (N/2 - 1)                ; (1) ; get loop number
ld.d      ssssXY, [Xptr+]8             ; (2) ; X0 X1 Y0 Y1
sqdloop:
          subs.h    ssTmp,ssX,ssY      ; (1) ; X1 - Y1 || X0 - Y0
          ld.d      ssssXY, [Xptr+]8   ; ||  ; X2 X3 Y2 Y3
          mulr.h    ssZ, ssTmp,ssTmp ul,#1   ; (2,3)
                                       ; (X1-Y1)^2 || (X0-Y0)^2
          st.w      [Zptr+]4,ssZ       ; ||  ; store Z0 Z1
loop LC, sqdloop
```

Memory organization:

Xaddr	sX0
Xaddr + 2	sX1
Xaddr + 4	sY0
Xaddr + 6	sY1
Xaddr + 8	sX2
Xaddr + 10	etc…

Example	N = 160 ➔ 244 cycles

Register diagram:

Instruction	d0	d1	d3 / d2	Load / Store
			y1 y0 x1 x0	ld x0x1y0y1
subs.h ssTmp,ssX,ssY		x1–y1 || x0–y0	y3 y2 x3 x2	ld x2x3y2y3
mulr.h ssZ, ssTmp,ssTmp ul,#1	(x1–y1)^2 || (x0–y0)^2			
				st z0z1

5.5 Vector Complex Multiplication

Equations:

$Yr_n = Xr_n*Kr_n - Xi_n*Ki_n$ $\quad n = 0..N-1$

$Yi_n = Xr_n*Ki_n + Xi_n*Kr_n$

Pseudo code:

```
for (n=0; n<N; n++)
{
sYr[n] = sXr[n]*sKr[n]-sXi[n]*sKi[n];
sYi[n] = sXr[n]*sKi[n]+sXi[n]*sKr[n];
}
```

IP= 4 (2 mul, 1 msub, 1 madd)	LD/ST= 3 in packed format (read sXr || sXi, read sKr|| sKi, write sYr || sYi)

Assembly code:

```
lea     LC, (N - 1)                     ; (1)   ; get loop number
mov     d6,#0                           ; (2)   ; clear 3rd source
mov     d7,#0                           ; (3)   ; clear 3rd source
nmloop:
        ld.w        ssK,[Kptr+]4        ; (1)   ; load K
        ld.w        ssX,[Xptr+]4        ; (2)   ; load X
        msubadm.h   e0,e6,ssK,ssX ul,#1 ; (3)   ; Yr= Xr*Kr-Xi*Ki
        mulm.h      e2,ssK,ssX lu,#1    ; (4)   ; Yi= Xr*Ki+Xi*Kr
        st.h        [Yptr+]2,sYr        ; ||    ; store Yr
        st.h        [Yptr+]2,sYi        ; (5)   ; store Yi
loop    LC,nmloop
```

Example	N = 64 ➔ 325 cycles

Register diagram:

Instruction	d1 / d0	d3 / d2	d4	d5	d7/ d6	Load / Store
					d6=0	
					d7=0	
				ki kr		ld kikr
			xi xr			ld xixr
msubadm.h e0,e6,ssK,ssX ul,#1	yr					
mulm.h e2,ssK,ssX lu,#1		yi				st yr
						st yi

5.6 Vector Complex Multiplication (Packed)

Equations:

$Yr_n = Xr_n*Kr_n - Xi_n*Ki_n$ $\quad n = 0..N-1$

$Yi_n = Xr_n*Ki_n + Xi_n*Kr_n$

Pseudo code:

```
for (n=0; n<N; n++)
{
sYr[n] = sXr[n]*sKr[n]-sXi[n]*sKi[n];
sYi[n] = sXr[n]*sKi[n]+sXi[n]*sKr[n];
}
```

IP= 4 (2 mul, 1 msub, 1 madd)	LD/ST= 3 in packed format (read sXr || sXi, sKr|| sKi, write sYr || sYi)

Pseudo code implementation:

```
for (n=0; n<N; n++)
{
sYr[n]  = sXr[n]*sKr[n]; sYi[n]  = sXr[n]*sKi[n];
sYr[n] -= sXi[n]*sKi[n]; sYi[n] += sXi[n]*sKr[n];
}
```

Assembly code:

```
lea     LC,(N - 1)                  ; (1)     ; get loop number
ld.w    ssK,[Kptr+]4                ; (2)     ; Ki Kr
ld.w    ssX,[Xptr+]4                ; (3)     ; Xi Xr
nloop:
        mulr.h      ssY,ssK,ssX ll,#1       ; (1)
                                    ; Yi =Xr*Ki || Yr  = Xr*Kr
        maddsurs.h  ssY,ssY,ssK,ssX uu,#1   ; (2,3)
                                    ; Yi+=Xi*Kr || Yr -= Xi*Ki
        ld.w        ssK,[Kptr+]4    ; ||      ; Ki1 Kr1
        ld.w        ssX,[Xptr+]4    ; ||      ; Xi1 Xr1
        st.w        [Yptr+]4,ssY    ; (4)     ; store Yi Yr
loop    LC,nloop
```

Example	N = 64 ➔ 261 cycles

Register diagram:

Instruction	d0	d5	d4	d7/ d6	Load / Store
		kikr			ld kikr
			xixr		ld xixr
mulr.h ssY,ssK,ssX ll,#1	yi=xr*ki || yr=xr*kr				
maddsurs.h ssY,ssY,ssK,ssX uu,#1	yi=yi+xi*kr||yr=yr-xi*ki	kikr			ld kikr
			xixr		ld xixr
	yiyr				st yiyr

5.7 Vector Complex Multiplication (Unrolled)

Equations:

$Yr_n = Xr_n*Kr_n - Xi_n*Ki_n$ $\quad n = 0..N-1$

$Yi_n = Xr_n*Ki_n + Xi_n*Kr_n$

Pseudo code:

```
for (n=0; n<N; n++)
{
sYr[n] = sXr[n]*sKr[n]-sXi[n]*sKi[n];
sYi[n] = sXr[n]*sKi[n]+sXi[n]*sKr[n];
}
```

IP= 4 (2 mul, 1 msub, 1 madd)	LD/ST= 3 in packed format (read sXr || sXi, sKr|| sKi, write sYr || sYi)

Pseudo Code implementation:

```
for (n=0; n<N/2-1; n++)
{
 sYr[2*n]     = sXr[2*n]*sKr[2*n]; sYi[2*n]   = sXr[2*n]*sKi[2*n];
 sYr[2*n]    -= sXi[2*n]*sKi[2*n]; sYi[2*n] += sXi[2*n]*sKr[2*n];
 sYr[2*n+1]  = sXr[2*n+1]*sKr[2*n+1];sYi[2*n+1] =  sXr[2*n+1]*sKi[2*n+1];
 sYr[2*n+1] -= sXi[2*n+1]*sKi[2*n+1];sYi[2*n+1] += sXi[2*n+1]*sKr[2*n+1];
}
```

Assembly code:

```
lea      LC,(N/2-1)                          ;(1)  ;get loop number
ld.w     ssK0,[Kptr+]4                       ;(2)  ;Ki0 Kr0
ld.d     ssssX,[Xptr+]8                      ;(3)  ;Xi1 Xr1 Xi0 Xr0
cxloop:
         mulr.h      ssY0,ssK0,ssX0 ll,#1;(1)  ;Yi0 =Xr*Ki||Yr0=Xr*Kr
         st.w        [Yptr+]4,ssY1       ;||   ;store former Yi1 Yr1
         maddsurs.h  ssY0,ssY0,ssK0,ssX0 uu,#1 ;(2)
                                         ;Yi0 +=Xi*sKr ||Yr0 -= Xi*Ki
         ld.d        ssssK,[Kptr+]8      ;||   ;Ki2 Kr2  Ki1 Kr1
         mulr.h      ssY1,ssK1,ssX1 ll,#1;(3)  ;Yi1 =Xr*Ki ||Yr1=Xr*Kr
         st.w        [Yptr+]4,ssY0       ;||   ;store Yi0 Yr0
         maddsurs.h  ssY1,ssY1,ssK1,ssX1 uu,#1 ;(4)
                                         ;Yi1 +=Xi*Kr ||Yr1 -= Xi*Ki
         ld.d        ssssX,[Xptr+]8      ;||   ;Xi3 Xr3 Xi2 Xr2
loop     LC,cxloop
st.w     [Yptr],ssY1 ; (4)                   ;store last Yi1 Yr1
```

Register diagram:

Instruction	d0	d1	d5 d4	d7 d6	Load / Store
				ki0 kr0 x x	ld k0
			xi1 xr1 xi0 xr0		ld x0 x1
mulr.h ssY0,ssK0,ssX0 ll,#1	yi0=xr0*ki0 || yr0=xr0*kr0				st y1
maddsurs.h ssY0,ssY0,ssK0,ssX0uu,#1	yi0 +=xi0*kr0 || yr0 -= xi0*ki0			ki2 kr2 ki1 ki1	ld k1 k2
mulr.h ssY1,ssK1,ssX1 ll,#1		yi1=xr1*ki1 || yr1=xr1*kr1			st y0
maddsurs.h ssY1,ssY1,ssK1,ssX1uu,#1		yi1+=xi1*kr1 || yr1 -= xi1*ki1	xi3 xr3 xi2 xr2		ld x2 x3
					st y1

5.8 Color Space Conversion

Equation:

X		A				R'		K
Y	=	0.257	0.504	0.098	*	R	+	0*
Cr		0.439	-0.368	-0.071		G		0.5
Cb		-0.148	-0.291	0.439		B		0.5

* 0 for UPF format, 16 for CCIR 601.

$$X_i = \sum A_{ij} * R'_j + K_j \qquad \text{for } i, j = 0..n$$

Pseudo code:

```
sY  = 0.257*sR + 0.504*sG + 0.098*sB;
sCr  = 0.439*sR - 0.368*sG - 0.071*sB + 0.5;
sCb = -0.148*sR - 0.291*sG + 0.439*sB + 0.5;
```

RGB belongs to [0; +1[([0; 256[) so YCrCb will be in [0; +1[([0; 256[).

(*Assembly code and Register diagram follow*)

Assembly code:

```
.define    RGBptr  "a0"
.define    OFFSET128  "0x2000"
.define    llY  "e8"
.define    llCr  "e10"
.define    llCb  "e12"

lea        RGBptr,rgbvalue
lea        Kptr,kmatvalue
lea        a4,stvalue

mov        d0,#0                    ; (1)   ;
ld.d       e2,[RGBptr+]6            ; ||    ; sRsGsB
mov.u      d1,#OFFSET128            ; (2)   ; in Q14 with 16-bits of sign
ld.d       e4,[Kptr+]6              ; ||    ; sK00,sK01,sK02

mulm.h     llY,d2,d4ul,#1           ; (3)   ; llY = sR*sK00 + sG*sK01
madd.q     llY,llY,d3l,d5l,#1       ; (4,5) ; llY = llY + sB*sK02

ld.d       e4,[Kptr+]6              ; ||    ; sK10,sK11,sK12
maddm.h    llCr,e0,d2,d4ul,#1       ; (6)   ; llCr=sR*sK10+sG*sK11+128
madd.q     llCr,llCr,d3l,d5l,#1     ; (7,8) ; llCr=llCr+sB*sK12

ld.d       e4,[Kptr+]6              ; ||    ; sK20,sK21,sK22
maddm.h    llCb,e0,d2,d4ul,#1       ; (9)   ; llCb=sR*sK20+sG*sK21+128
st.h       [a4+]2,d9                ; ||    ; store sY
madd.q     llCb,llCb,d3l,d5l,#1     ; (10)  ; llCb=llCb+sB*sK22
st.h       [a4+]2,d11               ; ||    ; store sCr
st.h       [a4+]2,d13               ; (11)  ; store sCb

kmatvalue:.half   0x20e6,0x4803,0x0c83       ; abcin Q15
          .half   0x3831,0xd0e5,0xf6e9       ; defin Q15
          .half   0xed0e,0xdcc1,0x3831       ; ghiin Q15
rgbvalue: .half   0x4000,0x3000,0x2000       ; RGB are [0;+1[ in Q14
```

Register diagram:

Instruction	d1/d0	d3/d2	d5/ d4	d9/d8	d11/d10	d13/ d12	Load/ Store
	0	R'G'B'					ld R'G'B'
	offset128		k00, k01, k02				ld k00, k01, k02
mulm.h llY,d2,d4ul,#1				Y=R*k00 +G*k01			
madd.q llY,llY,d3l,d5l,#1			k10, k11, k12	Y = Y + B*k02			ld k10, k11, k12
maddm.h llCr,e0,d2,d4ul,#1					Cr=R*k10+ G*k11+128		
madd.q llCr,llCr,d3l,d5l,#1			k20, k21, k22		Cr=Cr+B*k12		ld k20, k21, k22
maddm.h llCb,e0,d2,d4ul,#1				Y		Cb= R*k20+ G*k21+ 128	st Y
madd.q llCb,llCb,d3l,d5l,#1					Cr	Cb= Cb+B* k22	st Cr
						Cb	st Cb

5.9 Vector Scaling

Equation:

$Z_n = (X_n >> 3) << 2$

or $Z_n = (X_n >> shift1) << shift2$ n = 0..N-1

Pseudo code:

for (n=0; n<N; n++)sZ[n] = (sX[n] >>3) << 2;

IP= 2 (2 shifts)	LD/ST= 2 (read sX, write sZ)

Assembly code:

```
mov     Shift1,#-3                  ; (1)  ; load 1st shift value
lea     LC,(N/2 - 1)                ; ||   ; get loop number
mov     Shift2,#2                   ; (2)  ; load 2nd shift value
ld.w    ssX,[Xptr+]4                ; ||   ; X0 X1
isloop:
        sha.h       d1,ssX,Shift1   ; (1)  ; X0>>3, X1>>3
        ld.w        ssX,[Xptr+]4    ; ||   ; X2 X3
        sha.h       ssZ,d1,Shift2   ; (2)  ; X0<<2, X1<<2
        t.w         [Zptr+]4,ssZ    ; ||   ; store Z0,Z1
loop    LC,isloop
```

Alternatively, the 2 shift values can be directly used

```
lea     LC,(N/2 - 1)                ; (1)  ; get loop number
ld.w    ssX,[Xptr+]4                ; (2)  ; X0 X1
isloop:
        sha.h       d1,ssX,#-3      ; (1)  ; X0>>3, X1>>3
        ld.w        ssX,[Xptr+]4    ; ||   ; X2 X3
        sha.h       ssZ,d1,#2       ; (2)  ; X0<<2, X1<<2
        st.w        [Zptr+]4,ssZ    ; ||   ; store Z0,Z1
loop    LC,isloop
```

Example	N = 160 ➔ 164 cycles

Register diagram:

Instruction	d1/ d0	d2	d3	d4	d7/ d6	Load / Store
			Shift1			ld shift1
				Shift2		ld shift2
		x1x0				ld x0x1
sha.h d1,ssX,Shift1	x1>>3 || x0>>3	x3x2				ld x2x3
sha.h ssZ,d1,Shift2	x1<<2 || x0<<2					st z0z1

5.10 Vector Normalization

Equations:

(1) minex = minimum (minex,exponent (X_n)) n = 0..N-1

(2) X_n = X_n << minex n = 0..N-1

Pseudo code Equation (1):

```
sMin = 32;
for (n = 0; n<N; n++)
          {
          sZ = exponent(sX[n]);
          if (sZ < sMin) sMin = sZ; else sMin = sMin;
          }
```

IP= 2 (1 min, 1 count leading sign)	LD/ST= 1 (read sX)

Pseudo code Equation (2):

```
for (n = 0; n<N; n++)  sX[n] = sX[n] << sMin;
```

IP= 1 (1 shift)	LD/ST= 2 (read sX, write sX)

Assembly code:

```
movh     d4,#16                 ; (1)       ; d4 upper = max
lea      LC,(N/2 - 1)           ; ||        ; get loop number
addi     d4,d4,#16              ; (2)       ; d4 lower = max
ld.w     ssX,[Xptr+]4           ; ||        ; X0 X1
bexloop:
         cls.h     ssZ,ssX      ; (1)       ; Z1 =exp.(X1)||Z0 =exp.(X0)
         min.h     d4,d4,ssZ    ; (2)
                                ; Min1=min(Z1,Min1)|| Min0=min(Z0,Min0)
         ld.w      ssX,[Xptr+]4 ; ||        ; X2 X3
loop     LC,bexloop
sh       d1,d4,#-16             ; (3)       ; d1 = Min1
extr.u   d3,d4,#0,#16           ; (4)       ; d0 = Min0
min.h    d3,d1,d3               ; (5)       ; Min = min(Min1,Min0)
ld.w     ssX,[X1ptr]            ; (6)       ; X0 X1
lea      LC, (N/2 - 1)          ; (7)       ; get loop number
normloop:
         sh.h      ssX,ssX,d3   ; (1)       ; X0 << Min || X1 << Min
         st.w      [X1ptr+]4,ssX ; ||       ; store normalized X0, X1
         ld.w      ssX,[X1ptr]  ; (2)       ; load unormalized X2, X3
loop     LC,normloop
```

Finding minimum exponent loop:

Example	N = 160 ➔ 169 cycles

Normalization loop:

Example	N = 160 ➔ 162 cycles

Total:

Example	N = 160 ➔ 331 cycles

6 Filters

If all the possible data types are considered, the list of Filter routines can be very large:

- 16-bit, 32-bit, mixed 16-bit/32-bit
- Complex 16-bit, complex 32-bit, complex mixed 16-bit/32-bit
- Result accumulated on 32bit or > 32bit
- Saturated data type

Most common cases are covered, with the aim of providing a maximum of diversity.

The first 3 instances are not filter routines, but Vector to Scalar operations. These have a lot in common with Filters since the result is accumulated. The only difference between a FIR (Finite Impulse Response) filter and a Dot Product for example, is that a FIR is likely to use Circular Addressing. It therefore makes sense to place these together, as they will be optimized in the same way.

The next 5 are FIR filter routines, beginning with the more trivial (non-looped) cases and ending with the complex FIR routine. This routine can be extremely complicated to optimize, especially since TriCore can perform it in 2 cycles per complex tap. Between these 2 extreme are the standard *n*-tap FIR (0.5 cycle per tap) and the Block FIR.

Auto-correlation appears next to the Block FIR, because from an implementation point of view, these are nearly identical. They belong to a famous class of algorithms, known as BLOCK algorithms. Those 2 examples should be sufficient to develop other routines.

The IIR filter routines logically come after the FIR. These routines all contain feedback terms, which always makes the implementation more difficult than for FIR. The first 3 IIR routines are non-looped cases (1, 2 and 4 coefficients). This is followed by the most common N-stage biquad (in 2 types, of 4 or 5 coefficients) and ends with the Lattice IIR. The Lattice IIR routine is always extremely complicated to optimize with pipeline MAC, which is the case with TriCore.

The last group are LMS routines, or more precisely, adaptive FIR filters using the Least Mean Square (LMS) method to update the coefficients. There are dozens of adaptive methods and the delayed LMS is used because it is fast.

Leaky LMS and the 3 standard cases (16-bit, 32-bit & complex) have also been addressed. In many ways the LMS can be seen as the standard algorithm by which to judge the power of an architecture, and TriCore gives:

- 1 cycle per tap (16-bit coefficients)
- 1 cycle per tap (32-bit coefficients)
- 2 cycles per complex tap (16-bit coefficients)

These results are extremely good, and few DSPs can reach these figures.

Name	Cycles	Code Size [1]	Optimization Techniques						Arithmetic Methods	
			Software Pipelining	Loop Unrolling	Packed Operation	Load/Store Scheduling	Data Memory Interleaving	Packed Load/Store	Saturation	Rounding
Dot product	(2*N/4)+6	34	✓	-	✓	-	-	✓	✓	-
Magnitude square	(1*N/2)+5	16	✓	-	✓	-	-	✓	-	-
Vector quantization	(3*N/2)+6	38	-	-	✓	-	-	✓	-	-
First order FIR	4	14	-	-	-	-	-	✓	-	-
Second order FIR	5	22	-	-	-	-	-	✓	-	-
FIR	(2*N/4 +2) +4	34	✓	✓	✓	-	-	✓	-	-
Block FIR	(4*(N/2) +2) +9	66	✓	-	✓	✓	-	✓	-	-
Auto-correlation	((11+3*(N/2- 1) + 2) *M/2+2)+5	70	✓	-	✓	✓	-	✓	-	-
Complex FIR	(2*N +2) +4	32	✓	-	✓	✓	-	✓	✓	✓
First order IIR	5	24	-	-	✓	-	-	✓	✓	✓
2nd order IIR	7	34	✓	-	✓	-	-	✓	✓	-
Biquad 4 coefficients	5	26	✓	-	✓	-	-	✓	-	-
N-stage Biquad 4 coefficients	(3*(N-1) +2) +6	48	✓	✓	✓	✓	-	✓	-	-

N-stage Biquad 5 coefficients	(5*(N-1) +2) +9	74	✓	✓	✓	✓	-	✓	✓	-
Lattice filter	(4*(N-2) +2) +10	54	✓	✓	-	✓	-	-	✓	✓
Leaky LMS (update only)	(4*(N/4-1) +2) +9	70	✓	✓	✓	✓	-	✓	✓	✓
Delayed LMS	(4*N/4 +2) +5	54	✓	-	✓	✓	-	✓	✓	✓
Delayed LMS – 32-bit coefficients	(4*(N/2 –1) +2) + 8	64	✓	-	✓	✓	-	✓	✓	-
Delayed LMS – complex	(4*(N-1) +2) +9	60	✓	✓	✓	✓	-	✓	✓	✓

1) Code Size is in Bytes

6.1 Dot Product

Equation:

$Z = \sum (V_n * W_n)$ $n = 0..N-1$

Pseudo code:

```
sZ = 0;
for (n = 0; n < N; n++) sZ += sV[n] * sW[n];
```

IP= 1 (1 madd)	LD/ST= 2 (read sV, read sW)

Assembly code:

```
lea     LC, (N/4 -1)                    ; (1) ; get loop number
mov     d0,#0                           ; (2) ; Z =0 (lower)
ld.d    ssssV,[Vptr+]8                  ; ||  ; dummy dummy V0 V1
mov     d1,#0                           ; (3) ; Z =0 (upper)
ld.d    ssssW,[Wptr+]8                  ; ||  ; W0 W1 W2 W3
dotloop:
        maddms.h    llZ,llZ,ssW0,ssV0 ul,#1 ; (1)   ; Z +=V0*W0+V1*W1
        ld.d        ssssV,[Vptr+]8    ; ||  ; V2 V3 V4 V5
        maddms.h    llZ,llZ,ssW1,ssV1 ul,#1 ; (2)   ; Z +=V2*W2+V3*W3
        ld.d        ssssW,[Wptr+]8    ; ||  ; W4 W5 W6 W7
loop    LC,dotloop
st.h    [Zptr],sZ                       ; (4) ; store Z
```

Example	N = 64 ➔ 38 cycles

Register diagram:

Instruction	d1 / d0	d3 / d2	d5 / d4	Load / Store
	z(=0)		v1v0 _ _	ld v0v1
	z(=0)	w3w2w1w0		ld w0w1w2w3
maddms.h llZ,llZ,lW0,lV0ul,#1	z+v0w0+v1w1		v5v4v3v2	ld v2v3v4v5
maddms.h llZ,llZ,lW1,lV1ul,#1	z+v2w2+v3w3	w7w6w5w4		ld w4w5w6w7

6.2 Magnitude Square

Equation:

$Z = \Sigma (Xr_n^2 + Xi_n^2)$ $n = 0..N-1$

Pseudo code:

```
sZ = 0;
for (n=0; n<N; n++)  sZ += (sXr[n]*sXr[n] + sXi[n]*sXi[n]);
```

IP= 2 (2 madd)	LD/ST= 2 (read sXr, read sXi)

Assembly code:

```
lea      LC, (N/2 - 1)                  ; (1)  ; get loop number
ld.w     ssX, [Xptr+]4                  ; (2)  ; load Xr, Xi
msqloop:
         maddm.h     llZ,llZ,ssX,ssX ul,#1    ; (1); Z +=(Xr^2 + Xi^2)
         ld.w        ssX, [Xptr+]4      ; ||   ; load next Xr,Xi
loop     LC, msqloop                    ;
st.h     [Zaddr],sZ                     ; (3)  ; store Z
```

Memory organization:

Xaddr	Xr0	**Xaddr + 2**	Xi0
Xaddr + 4	Xr1	**Xaddr + 6**	Xi1
Xaddr + 8	Xr2	**Xaddr +10**	etc..

Example	N = 64 ➔ 37 cycles

Register diagram:

Instruction	**d1/ d0**	**d2**	**d5 / d4**	**d7/ d6**	**Load / Store**
	0	xi0 xr0			ld xr0 xi0
maddm.h llZ,llZ,ssX,ssX,#1	Z	xi1 xr1			ld xr1 xi1

6.3 Vector Quantization

Note: VALIDATED ON TC1 V1.3 SILICON.

Equation:

$Z = \Sigma (K_n - X_n)^2 \qquad n = 0..N-1$

Pseudo code:

```
sZ = 0;
for (n=0; n<N; n++) sZ += (sK[n]-sX[n])^2;
```

IP= 2 (1 sub, 1 madd)	LD/ST= 2 (read sX, read sK)

Assembly code:

```
lea      LC,(N/2 - 1)                    ;(1)    ; get loop number
mov      d0,#0                           ;(2)    ; Z = 0(lower)
ld.w     ssX,[Xptr+]4                    ;||     ; sX0sX1
mov      d1,#0                           ;(3)    ; Z = 0 (upper)
ld.w     ssK,[Kptr+]4                    ;||     ; K0 K1
quantloop:
         subs.h      ssTp,ssK,ssX        ;(1)    ; K1-X1|| K0-X0
         ld.w        ssX,[Xptr+]4        ;||     ; X2 X3
         maddm.h     llZ,llZ,ssTp,ssTp ul,#1  ; (2,3)
                                         ;Z + = ((K0-X0)^2+(K1-X1)^2)
         ld.w        ssK,[Kptr+]4        ;||     ; K2 K3
loop     LC,quantloop
st.h     [Zaddr],d1                      ;(4)    ; store sZ
```

Example	N = 64 ➔ 102 cycles

Register diagram:

Instruction	d1 / d0	d3 / d2	d5 / d4	d7 / d6	Load / Store
	z = 0(lower)		x1 x0		load x0x1
	z = 0 (upper)			k1 k0	load k0k1
subs.h ssTp,ssK,ssX		k1-x1 || k0-x0	x3 x2		load x2x3
maddms.h llZ,llZ,ssTp, ssTp ul, #1	z += (k0-x0)^2 + (k1-x1)^2)			k3 k2	load k2k3

6.4 First Order FIR

Note: Validated on TriCore Rider D board.

Equation:

$Y_t = K_0 * X_t + K_1 * X_{t-1}$

Pseudo code:

```
sY  = sK0*sX0 + sK1*sX1;
sX1 = sX0;
```

IP= 2 (1 mul, 1 madd)	LD/ST= 6 (read sX0, sX1, sK0, sK1, write sY, write sX0→ sX1)

Assembly code:

```
ld.w      ssK,[Kptr]              ; (1)      ; K0 K1
ld.w      ssX,[Xptr]              ; (2)      ; X0 X1
mulm.h    llY,ssK,ssX ul,#1       ; (3,4)    ; Y = K0*X0 + K1*X1
st.h      [Xptr]+2,ssX            ; ||       ; X0 --> X1
st.h      [Yptr],d1               ; ||       ; store Y
```

Memory organization:

	Entering	Leaving
Xaddr	X0	X0
Xaddr + 2	X1	X0

Register diagram:

Instruction	d1 / d0	d4	d6	Load / Store
		k1 k0		ld k1k0
			x1 x0	ld x1x0
mulm.h llY,ssK,ssX ul,#1	y = k0*x + k1*x1			
				st y

6.5 Second Order FIR

Note: Validated on TriCore Rider D board.

Equation:

$Y_t = K_0*X_t + K_1*X_{t-1} + K_2*X_{t-2}$

Pseudo code:

```
sY  = sK0*sX0 + sK1*sX1 + sK2*sX2 ;
sX2 = sX1;
sX1 = sX0;
```

IP= 2 (1 mul, 2 madd)	LD/ST= 9 (read sX0, sX1, sX2, sK0, sK1, sK2, write sY, write sX0 → sX1, write sX1 → sX2)

Assembly code:

```
ld.d      sssK,[Kptr]            ; (1)      ; K0 K1 K2
ld.d      sssX,[Xptr]            ; (2)      ; X X1 X2
mulm.h    llY,d4,d6 ul,#1        ; (3)      ; Y = K0*X + K1*X1
st.w      [Xptr]+2,d6            ; ||       ; X0 --> X1  X1 --> X2
madd.q    llY,llY,d5l,d7l,#1     ; (4,5)    ; Y  = Y + K2*X2
st.h      [Yptr],d1              ; ||       ; store Y
```

Memory organization:

	Entering
Xaddr	X0
Xaddr + 2	X1
Xaddr + 4	X2

Leaving
X0
X0
X1

Register diagram:

Instruction	d1 / d0	d5 / d4	d7 / d6	Load / Store
		_ k2 k1 k0		ld k0 k1 k2
			_ x2 x1 x0	ld x0 x1 x2
mulm.h llY,d4,d6ul,#1	y = k0*x0 + k1*x1			st x0 x1
madd.q llY,llY,d5l,d7l,#1	y += k2*x2			st y

6.6 FIR

Equation:

$Y_t = \sum (X_{t-n} * K_n)$ **n = 0..N-1**

Pseudo code:

```
sY = 0;
for (n=0; n<N; n++) sY += circular(sX[n])*sK[n];
```

IP= 1 (1 madd)	LD/ST= 2 (read sX, read sK)

Assembly code:

```
mov.aa   a2,Xptr                          ; circular buffer initialization
CONST.A  a3,(2*N)<<16

lea      LC,(N/4 -1)                      ; (1)   ; get loop number

mov      d0,#0                            ; (2)   ; Y = 0 (lower)
ld.w     ssK0,[Kptr+]4                    ; ||    ; K0 K1
mov      d1,#0                            ; (3)   ; Y = 0 (upper)
ld.d     ssssX,[a2/a3+c]8                 ; ||    ; X0 X1 X2 X3

sfirloop:
         maddm.h      llY,llY,ssX0,ssK0 ul,#1 ; (1) ; Y +=X0*K0+X1*K1
         ld.d         ssssK,[Kptr+]8          ; ||  ; K2 K3 K4 K5
         maddm.h      llY,llY,ssX1,ssK1 ul,#1 ; (2) ; Y +=X2*K2+X3*K3
         ld.d         ssssX,[a2/a3+c]8        ; ||  ; X4 X5 X6 X7
loop     LC,sfirloop
st.h     [Yptr],d1                        ; (4)   ; store Y
```

Example	N = 20 ➔ 16 cycles

Register diagram:

Instruction	d1 / d0	d3 / d2	d5 / d4	Load / Store
			k1k0 _ _	ld k0k1
		x3x2 x1x0		ld x0x1x2x3
maddm.h llY,llY,ssX0,ssK0 ul,#1	y +=x0*k0+x1*k1		k5k4 k3k2	ld k2k3k4k5
maddm.h llY,llY,ssX1,ssK1 ul,#1	y +=x2*k2+x3*k3	x7x6 x5x4		ld x4x5x6x7

6.7 Block FIR

Note: Validated on TriCore Rider D board.

Equation:

$Y_m = \sum X_{m-n} * K_n$ n = 0..N-1, m = 0..M-1

Pseudo code:

```
for (m = 0; m <M/4; m+4)
{
sY[m] = 0;
for (n = 0; n<N; n++)
sY[m]   += sX[m+n]*sK[n];
sY[m+1] += sX[m+n+1]*sK[n];
sY[m+2] += sX[m+n+2]*sK[n];
sY[m+3] += sX[m+n+3]*sK[n];
}
```

IP= 4 (4 madd)	LD/ST=5 (read sXm, sXm+1, sXm+2, sXm+3, sK)

Assembly code:

```
mov.aa   a2,Xptr                  ;circular buffer address initialization
CONST.A  a3,(2*N)<<16
mov.aa   a14,Xptr                 ;circular buffer address initialization
CONST.A  a15,(2*N)<<16

lea      LC,(N/2 - 1)             ; (1)      ; get loop number

mul      llY0,d13,#0              ; (2)      ; y0 = 0
mul      llY2,d13,#0              ; (3)      ; y2 = 0
ld.q     d13,[a14/a15+c]2         ; ||       ; dummy load
mul      llY1,d13,#0              ; (4)      ; y1 = 0
ld.w     ssK,[Kptr+]4             ; ||       ; k1k0
mul      llY3,d13,#0              ; (5)      ; y3 = 0
ld.d     ssssX,[a2/a3+c]4         ; ||       ; x3x2x1x0

blkfir:
         maddm.h     llY0,llY0,d8,ssK ul,#1  ; (1) ; y0+=x0*k0+x1*k1
         maddm.h     llY2,llY2,d9,ssK ul,#1  ; (2) ; y2+=x2*k0+x3*k1
         ld.d        ssssX1,[a14/a15+c]4     ; ||  ; x4x3x2x1
         maddm.h     llY1,llY1,d10,ssK ul,#1 ; (3) ; y1+=x1*k0+x2*k1
         ld.d        ssssX,[a2/a3+c]4        ; ||  ; x5x4x3x2
         maddm.h     llY3,llY3,d11,ssK ul,#1 ; (4) ; y3+=x3*k0+x4*k1
         ld.w        ssK,[Kptr+]4            ; ||  ; k3k2
loop     LC,blkfir

st.h     [Yptr+]2,d1              ; (6)      ; store y0
st.h     [Yptr+]2,d3              ; (7)      ; store y2
st.h     [Yptr+]2,d5              ; (8)      ; store y1
st.h     [Yptr+]2,d7              ; (9)      ; store y3
```

Example	N = 20 ➔ 51 cycles

Register diagram:

Instruction	**d1/d0**	**d3/d2**	**d5/d4**	**d7/d6**	**d9/d8**	**d11/d10**	**d12**	**Load / Store**
	y0 = 0							
		y2 = 0						
			y1 = 0				k1k0	ld k1k0
				y3 = 0	x3x2x1x0			ld x3x2x1x0
maddm.h llY0,llY0,d8,ssKul,#1	y0+= x0*k0+ x1*k1							
maddm.h llY2,llY2,d9,ssKul,#1		y2+= x2*k0 +x3*k1				x4x3x2x1		ld x4x3x2x1
maddm.h llY1,llY1,d10,ssKul,#1			y1+= x1*k0 +x2*k1		x5x4x3x2			ld x5x4x3x2
maddm.h llY3,llY3,d11,ssKul,#1				y3+= x3*k0 +x4*k1			k3k2	ld k3k2
	y0							st y0
		y2						st y2
			y1					st y1
				y3				st y3

6.8 Auto-Correlation

Note: Validated on TriCore Rider D board.

Equation:

$Z_m = \sum X_n * X_{n+m}$ **n = 0..N-1 ; m=0..M-1;**

Pseudo code:

```
for (m=0; m<M; m++)
{
lZ[m] = 0;
for (n=0; n<N; n++) lZ += sX[n]*sX[n+m];
}
```

IP= 1 (1 madd)	LD/ST= 3 (read sX, s Xn+m, write lZ)

Assembly code:

```
mov.aa    a8,Xaddress            ; (1)     ; odd values
mov.aa    a9,Xaddress            ; (2)     ; even values
add.a     a8,#2                  ; (3)     ; adjust pointer position
lea       LCe,(M/2 - 1)          ; (4)     ; get external loop number
mov.aa    Xoptr,a8               ; (5)     ; odd pointer init
extautoloop:
          mov     d0,#0          ; (1)     ; z_even = 0
          lea     Xptr,Xaddress  ; ||      ; even values
          mov     d1,#0          ; (2)     ; z_even = 0
          lea     LCi,(N/2 - 2)  ; ||      ; get internal loop number
          mov     d2,#0          ; (3)     ; z_odd = 0
          ld.w    ssX1,[Xptr+]4  ; ||      ; x1 x0 (even values)
          mov     d3,#0          ; (4)     ; z_odd = 0
          ld.w    ssX3,[Xeptr+]4 ; ||      ; x1 x0

          maddm.h llZe,llZe,ssX1,ssX3 ul,#1
                                 ; (5)     ; z_even=z_even+x0*x0+x1*x1
          ld.w    ssX2,[Xoptr+]4 ; ||      ; x2 x1

          intautoloop:
                  maddm.h        llZo,llZo,ssX1,ssX2 ul,#1
                                          ; (1,2)  ; z_odd +=x0*x1+x1*x2
                  ld.w           ssX1,[Xptr+]4     ; || ; x3 x2
                  ld.w           ssX3,[Xeptr+]4    ; || ; x3 x2
                  maddm.h        llZe,llZe,ssX1,ssX3 ul,#1
                                          ; (3)    ; z_even +=x2*x2+x3*x3
                  ld.w           ssX2,[Xoptr+]4    ; || ; x4 x3
          loop LCi,intautoloop
```

```
            maddm.h llZo,llZo,ssX1,ssX2 ul,#1
                                    ; (6)     ;
            st.w    [Zptr+]4,d1     ; ||      ; store z_even
            add.a   a9,#4           ; (7)     ; adjust pointer position
            st.w    [Zptr+]4,d3     ; (8)     ; store z_odd
            mov.aa  Xeptr,a9        ; (9)     ; even pointer init
            add.a   a8,#4           ; (10)    ; adjust pointer position
            mov.aa  Xoptr,a8        ; (11)    ; odd pointer init
loop        LCe,extautoloop
```

Example	N=160, M = 10 ➔ 1247 cycles

Register diagram:

Instruction	**d1 / d0**	**d3 / d2**	**d5**	**d4**	**d6**	**Load / Store**
	z_even = 0 (lower)					
	z_even = 0 (upper)					
	z_odd = 0 (lower)			x1x0		ld x1x0
	z_odd = 0 (upper)				x1x0	ld x1x0
maddm.h llZe,llZe,lX1,lX3 ul,#1	z_even += x0*x0+x1*x1		x2x1			ld x2x1
maddm.h llZo,llZo,lX1,lX2 ul,#1		z_odd += x0*x1+x1*x2		x3x2		ld x3x2
					x3x2	ld x3x2
maddm.h llZe,llZe,lX1,lX3 ul,#1	z_even += x2*x2+x3*x3		x4x3			ld x4x3
maddm.h llZo,llZo,lX1,lX2 ul,#1		z_odd += x0*x1+x1*x2				st z_even
						st z_odd

6.9 Complex FIR

Note: Validated on TriCore Rider D board.

Equations:

$Yr_t = \sum (Xr_{t-n} * Kr_n - Xi_{t-n} * Ki_n)$ n = 0..N-1

$Yi_t = \sum (Xi_{t-n} * Kr_n + Xr_{t-n} * Ki_n)$ n = 0..N-1

Pseudo code:

```
sYr = 0;
sYi = 0;
for (n = 0; n<N; n++)
{
sYr += circular(sXr)*sKr - circular(sXi)*sKi;
sYi += circular(sXr)*sKi + circular(sXi)*sKr;
}
```

IP= 4 (3 madd, 1 msub)	LD/ST (in packed format) = 2 (read sXi || sXr, sKr || sKi)

Pseudo code implementation:

```
sYr = 0;
sYi = 0;
for (n = 0; n< N; n++)
{
sYi += sXr*sKi; sYr -= sXi*sKi;
sYi += sXi*sKr; sYr += sXr*sKr;
}
```

Assembly code:

```
lea       LC,(N - 1)                    ; (1)   ; get loop number
mov       ssY,#0                        ; (2)   ; y = 0
ld.w      ssX,[Xptr+]4                  ; ||    ; xi0 xr0
ld.q      ssK,[Kptr+]2                  ; (3)   ; ki0
cxloop:
          maddsurs.h  ssY,ssY,ssX,ssK uu,#1     ; (1)
                                        ; yi += xr0*ki0 ||  yr -= xi0*ki0
          ld.w        ssK,[Kptr+]4      ;||     ; ki1 kr0
          maddrs.h    ssY,ssY,ssX,ssK ll,#1     ; (2)
                                        ; yi += xi0*kr0 ||  yr += xr0*kr0
          ld.w        ssX,[a2/a3+c]4    ;||     ; xi1 xr1
loop      LC,cxloop
st.w      [Yptr],ssY                    ; (4)   ; store yiyr
```

Note: Warning! This algorithm only works when the coefficients are organised in the reverse order {imag,real} compared to data {real,imag} .

Memory organization:

Xaddr	sXr0
Xaddr + 2	sXi0
Xaddr + 4	sXr1
Xaddr + 6	sXi1
Xaddr + 8	sXr2
Xaddr + 10	etc..

Kaddr	sKi0
Kaddr + 2	sKr0
Kaddr + 4	sKi1
Kaddr + 6	sKr1
Kaddr + 8	sKi2
Kaddr + 10	etc..

Example	N = 20 ➔ 46 cycles

Register diagram:

Instruction	**d0**	**d4**	**d5**	**Load / Store**
	y = 0	xi0 xr0		ld xi0xr0
			ki0	ld ki0
maddsurs.h ssY,ssY,ssX,ssK uu,#1	yi+=xr*ki || yr-=xi*ki		ki1 kr0	ld ki1kr0
maddrs.h ssY,ssY,ssX,ssK ll,#1	yi+=xi*kr || yr+=xr*kr	xi1 xr1		ld xi1xr1

6.10 First Order IIR

Note: Validated on TriCore Rider D board.

Equation:

$Y_t = B*Y_{t-1} + K*X_t$

where:

B= (1- K)

Pseudo code:

```
sY0 = sY1 – sY1*sK + sX0*sK;
sY1 = sY0;
```

IP= 2 (1 madd, 1 msub)	LD/ST= 4 (read sX, sY1, sK, write sY0 → sY1)

Assembly code:

```
;sX and sY are in different registers
ld.q        sK,[Kptr]               ; (1)    ; K
ld.q        sX,[Xptr+]2             ; (2)    ; X0
ld.q        sY,[Yptr]               ; (3)    ; Y1
msubs.q     sY,sY,sK u,sY u,#1      ; (4)    ; Y0 = Y1-Y1*K
madds.q     sY,sY,sK u,sX u,#1      ; (5,6)  ; Y0 += X0*K
st.q        [Yptr],sY               ; ||     ; store Y0

;alternatively if sX0 and sY1 are in the same register
ld.q        sK,[Kptr]               ; (1)    ; K
ld.w        ssXY,[Xptr+]2           ; (2)    ; X0 || Y1
msubrs.h    sY,ssXY,ssXY,sK ul,#1   ; (3)    ; Y0 = Y1- Y1*K
maddrs.h    sY,sY,ssXY,sK uu,#1     ; (4,5)  ; Y0 += X0*K
st.q        [Yptr],sY               ; ||     ; store Y0
```

Note: Operates with dual MAC instruction but only 1 result is used.

Register diagram:

Instruction	d0	D4	d5	Load / Store
			k0 0	ld k
		x0 0		ld x0
	y0 0			ld y1
msubs.q sY,sY,sKu , sY u,#1	y1 = y0*(1-k)			
madds.q sY,sY,sK u,sX u,#1	y1 += x0*k			st y0

Instruction	d0	D3	d5	Load / Store
			k0 0	ld k
		x0||y1		ld x0 y1
msubrs.h sY,ssXY,ssXY,sK uu,#1	y0 = y1-y1*k			
maddrs.h sY,sY,ssXY,sK uu,#1	y0 += x0*k			st y

6.11 Second Order IIR

Note: Validated on TriCore Rider D board.

Equation:

$$Y_t = K_0*X_t + K1*X_{t-1} + K2*X_{t-2} + B1*Y_{t-1} + B2*Y_{t-2}$$

where:

B1, B2 are negative

Pseudo code:

```
sY0 = sK0*sX0 + sK1*sX1 + sK2*sX2 + sB1*sY1 + sB2*sY2;
sY2 = sY1;
sY1 = sY0;
```

IP=5 (1 mul, 4 madd)	LD/ST= 12 (read sX0, sX1, sX2, sK0, sK1, sK2, sB1, sB2, sY1, sY2, write sY0 --> sY1, write sY1 --> sY2)

Assembly code:

```
ld.d      ssssK,[Kptr]            ; (1)      ; K2 K1 K0
ld.w      ssB,[Bptr]              ; (2)      ; B1 B0
ld.d      ssssX,[Xptr+]2          ; (3)      ; X2 X1 X0

mulm.h    e0,d2,d6 ul,#1          ; (4)      ; Y0 = K1*X1 + K0*X0
ld.w      ssY,[a2/a3+c]2          ; ||       ; Y2 Y1
madds.q   e0,e0,ssK l,ssX l,#1    ; (5)      ; Y0 += K2*X2
ld.d      ssssX,[Xptr+]2          ; ||       ; X3 X2 X1
maddm.h   e0,e0,ssB,ssY ul,#1     ; (6,7)    ; Y0 += B1*Y1 + B2*Y2
st.h      [a2/a3+c]0,d1           ; ||       ; store Y0
```

Register diagram:

Instruction	d1 / d0	d3 / d2	d5 / d4	d7 / d6	d8	Load / Store
		k2k1k0				ld k0k1k2
			b1b0			ld b0b1
				x2x1x0		ld x0x1x2
mulm.h e0,d2,d6ul,#1	y2 = k1*x1 + k0*x0				y2y1	ld y1y2
madds.q e0,e0,ssK l,ssX l,#1	y2 = y + x*k2			x3x2x1		ld x1x2x3
maddm.h e0,e0,ssB,ssYul,#1	y2 = y2 + b1*y1 + b2*y2					st y0

6.12 BIQUAD 4 Coefficients

Note: Validated on TriCore Rider D board.

Equations:

$W_t = X_t + B1*W_{t-1} + B2*W_{t-2}$

$Y_t = W_t + K1*W_{t-1} + K2*W_{t-2};$

Pseudo code:

```
sW0 = sX0 + sB1*sW1 + sB2*sW2;
sY0 = sW0 + sK1*sW1 + sK2*sW2;
sW2 = sW1;
sW1 = sW0;
```

IP=4 (4 madd)	LD/ST= 10 (read sX0, read sW1, sW2, read sK1, sK2, read sB1, sB2, write sY0, write sW0 --> sW1, write sW1 --> sW2)

Assembly code:

```
ld.h      d0,[Xptr]              ;          ;x0-->d0
mov       d1,#0                  ; (1)      ; y0 = 0
ld.w      ssB,[BKptr+]4          ; ||       ; b2 b1
ld.w      ssW,[a14/a15+c]2       ; ||       ; w2 w1
maddm.h   llW,llY,ssW,ssB ul,#1  ; (3)      ; w0=x0+w1*b1+w2*b2
ld.w      ssK,[BKptr+]4          ; ||       ; k2 k1
maddm.h   llY,llW,ssW,ssK ul,#1  ; (4,5)    ; y0=w0+w1*k1+w2*k2
st.h      [a14/a15+c]0,d7        ; ||       ; w0--> w1
st.h      [Yptr],d1              ; ||       ; store y0
```

Register diagram:

Instruction	d1 / d0	d2	d4	d5	d7 / d6	Load / Store
	x = 0 (upper)		b2b1			ld b2b1
	x = 0 (lower)	w2w1				ld w2w1
maddm.h llW,llY,ssW,ssB ul,#1				k2k1	w0 = x+w1*b1+ w2*b2	ld k2k1
maddm.h llY,llW,ssW,ssK ul,#1	y0 = w0+w1*k1 +w2*k2				w1w0	st w0
	y0					st y0

6.13 N-stage BIQUAD 4 Coefficients

Note: Validated on TriCore Rider D board.

Equations:

$W_{0,n} = Y_{0,n-1} + B_{1,n}*W_{1,n} + B_{2,n}*W_{2,n}$

$Y_{0,n} = W_{0,n} + K_{1,n}*W_{1,n} + K_{2,n}*W_{2,n}$ $\quad n = 0..N-1$

Pseudo code:

```
for (n = 0; n< N; n++)
{
 sW0 = sY0 + sB1[n]*sW1[n] + sB2[n]*sW2[n];
 sY0 = sW0 + sK1[n]*sW1[n] + sK2[n]*sW2[n];
 sW2[n] = sW1[n];
 sW1[n] = sW0;
}
```

IP=4 (4 madd)	LD/ST= 8 (read sW1, sW2, sK1, sK2, sB1, sB2, write sW0 --> sW1, write sW1 --> sW2)

Note: sY0 can be kept in a register and does not need to be written back and re-read from memory between stages.

Assembly code:

```
ld.h     d0,[Xptr]                    ;       ; x0-->d0
mov      d1,#0                        ; (2)   ; y0 = 0
ld.w     ssB,[BKptr+]4                ; ||    ; b2 b1
;mov     d0,#0                        ; (3)   ; y0 = 0
ld.w     ssW,[a14/a15+c]2             ; ||    ; w2 w1
maddm.h  llW,llY,ssW,ssB ul,#1        ; (4)   ; w0=y0+w1*b1+w2*b2
ld.w     ssK,[BKptr+]4                ; ||    ; k2 k1
lea      LC,(N - 2)                   ; (1)   ; get loop number
bq4loop:
         maddm.h   llY,llW,ssW,ssK ul,#1      ; (1,2) ; y0=w0+w1*k1+w2*k2
         st.h      [a14/a15+c]0,d7    ; ||    ; store w0
         ld.w      ssW,[a14/a15+c]2   ; ||    ; w1 w0
         maddm.h   llW,llY,ssW,ssB ul,#1      ; (3)   ; w0=y0+w1*b1+w2*b2
loop     LC,bq4loop
maddm.h  llY,llW,ssW,ssK ul,#1        ; (5,6) ; y0=w0+w1*k1+w2*k2
st.h     [a14/a15+c]0,d7              ; ||    ; store w0
st.h     [Yptr+]2,d1                  ; ||    ; store y0
```

Example	N = 13 ➔ 44 cycles

Register diagram:

Instruction	d1 / d0	d2	d4	d5	d7 / d6	Load / Store
	x = 0 (upper)		b2b1			ld b2b1
	x = 0 (lower)	w2w1				ld w2w1
maddm.h llW,llY,ssW,ssB ul,#1				k2k1	w0= x0+w1*b1+ w2*b2	ld k2k1
maddm.h llY,llW,ssW,ssK ul,#1	y0= w0+w1*k1+ w2*k2				w0	st w0
		w1w0				ld w1w0
maddm.h llW,llY,ssW,ssB ul,#1					w0= y0+w1*b1+ w2*b2	
maddm.h llY,llW,ssW,ssK ul,#1	y0= w0+w1*k1+ w2*k2				w0	st w0
	y0					st y0

6.14 N-stage BIQUAD 5 Coefficients

Note: Validated on TriCore Rider D board.

Equations:

$W_{0,n} = Y_{0,n-1} + B_{1,n}*W_{1,n} + B_{2,n}*W_{2,n}$

$Y_{0,n} = K_{0,n}*W_{0,n} + K_{1,n}*W_{1,n} + K_{2,n}*W_{2n}$

Pseudo code:

```
for (n = 0; n<N; n++)
{
 sW0 = sY0 + sB1[n]*sW1[n] + sB2[n]*sW2[n];
 sY0 = sK0[n]*sW0 + sK1[n]*sW1[n] + sK2[n]*sW2[n];
 sW2[n] = sW1[n];
 sW1[n] = sW0;
}
```

IP= 5 (5 madd)	LD/ST= 9 (read sW1, sW2, read sK0, sK1, sK2, read sB1, write sW0 --> sW1, write sW1 --> sW2)

Note: sY0 can be kept in a register and does not need to be written back and re-read from memory between stages.

Assembly code:

```
ld.h      d0,[Xptr]                          ;        ; x0-->d0
mov       d1,#0                              ; (2)    ; y0 = 0
ld.w      ssB,[BKptr+]4                      ; ||     ; b2 b1
mov       d0,#0                              ; (3)    ; y0 = 0
ld.w      ssW,[a14/a15+c]2                   ; ||     ; w2 w1
maddm.h   llW,llY,ssW,ssB ul,#1              ; (4,5)  ; w0=y0+w1*b1+w2*b2
ld.d      ssssK,[BKptr]                      ; ||     ; k2 k1 k0 0
dextr     d8,d7,d6,#16                       ; (6)    ; extract w0
lea       LC,(N - 2)                         ; (1)    ; get loop number
bq5loop:
          mulm.h    llY,ssW,ssK2 ul,#1       ; (1)    ; y0=w1*k1+w2*k2
          st.q      [a14/a15+c]0,d8          ; ||     ; store w0
          madds.q   llY,llY,d8u,ssK u,#1     ; (2)    ; y0=y0+k0*w0
          ld.w      d2,[a14/a15+c]2          ; ||     ; w1 w0
          maddm.h   llW,llY,ssW,ssB ul,#1    ; (3,4)  ; w0=y0+w1*b1+w2*b2
          dextr     d8,d7,d6,#16             ; (5)    ; extract w0
loop      LC,bq5loop
mulm.h    llY,ssW,ssK2 ul,#1                 ; (7)    ; y0=w1*k1+w2*k2
madds.q   llY,llY,d8u,ssK u,#1               ; (8,9)  ; y0=y0+k0*w0
st.h      [a14/a15+c]0,d8                    ; ||     ; store w0
st.h      [Yptr+]2,d1                        ; ||     ; store y0
```

Example	N = 13 ➔ 71 cycles

Register diagram:

Instruction	d1 / d0	D2	d3	d5 / d4	d7 / d6	d8	Load / Store
	y = 0 (lower)		b2b1				ld b2b1
	y = 0 (upper)	W2w1					ld w2w1
maddm.h llW,llY,ssW,ssBul,#1				k2k1k0	w0=y0+w1*b1+w2*b2		ld k2k1k0
dextr d8,d7,d6,#16						w0	
mulm.h llY,ssW,ssK2 ul,#1	y0=w1*k1+w2*k2						
madds.q llY,llY,d8u,ssK u,#1	y0+=k0*w0					w0	st w0
		W1w0					ld w1w0
maddm.h llW,llY,ssW,ssBul,#1					w0=y0+w1*b1+w2*b2		
dextr d8,d7,d6,#16						w0	
mulm.h llY,ssW,ssK2 ul,#1	y0=w1*k1+w2*k2						
madds.q llY,llY,d8u,ssK u,#1	y0+=k0*w0					w0	st w0
	y0						st y0

6.15 Lattice Filter

Equations:

$Z_n = Z_{n-1} - X_n * K_{n-1}$ $\quad n = 1..N-1$

$T_{n-1} = X_n + Z_n * K_{n-1}$

Pseudo code:

```
for (n = 1; n<N; n++)
{
          sZ[n] = sZ[n-1] - sX[n]*sK[n-1];
          sT[n-1] = sX[n] + sZ[n]*sK[n-1];
}
```

IP = 2 (1 msub, 1 madd)	LD/ST= 5 (read sX, read sK, read sZ, write sZ, write sT)

Assembly code:

```
lea       LC,(N - 3)                    ; (1)    ; get loop number
ld.q      sX,[Xptr+]2                   ; (2)    ; x9
ld.q      sK,[Kptr+]2                   ; (3)    ; k10
ld.q      sY,[Yaddr]                    ; (4)    ; y (input)
msubrs.h  sZ,sY,sX,sK ul,#1             ; (5,6)  ; z9 = y - x9*k10
ld.q      sX,[Xptr+]2                   ; ||     ; x8
ld.q      sK,[Kptr+]2                   ; ||     ; k9
msubrs.h  sZ,sZ,sX,sK ul,#1             ; (7,8)  ; z8 = z9 - x8*k9
latloop:
          maddrs.h    sT,sX,sZ,sK ul,#1 ; (1,2)  ; t9 = x8 + z8*k9
          ld.q        sX,[Xptr+]2       ; ||     ; x7
          ld.q        sK,[Kptr+]2       ; ||     ; k8
          msubrs.h    sZ,sZ,sX,sK ul,#1 ; (3,4)  ; z7 = z8 - x7*k8
          st.q        [Xptr]-6,sT       ; ||     ; store t9
loop      LC,latloop
maddrs.h  sT,sX,sZ,sK ul,#1             ; (9,10) ; t1 = x0 + z0*k1
st.q      [Xptr]-4,sT                   ; ||     ; store t1
st.q      [Xptr]-2,sZ                   ; ||     ; store z0 (result)
```

Example	N = 10 ➔ 44 cycles

Register diagram:

Instruction	d0	d1	d4	d5	d6	Load / Store
			x9			ld x9
				k10		ld k10
					Y	ld y
msubrs.h sZ,sY,sX,sK ul,#1	z9 = y - x9*k10					ld x8
						ld x9
msubrs.h sZ,sZ,sX,sK ul,#1	z8 = z9 - x8*k9					
maddrs.h sT,sX,sZ,sK ul,#1		t9 = x8 + z8*k9	x8			ld x8
				k9		ld k9
msubrs.h sZ,sZ,sX,sK ul,#1	z7 = z8 – x7*k8					st t10
maddrs.h sT,sX,sZ,sK ul,#1		t1 = x0 + z0*k1				st t0
		t1				st t1

6.16 Leaky LMS (Update Only)

Note: Validated on TriCore Rider D board.

Equation:

$K_{t,n} = K_{t-1,n} * B + X_{t-n} * u * Err_{t-1}$ n = 0..N-1

Pseudo code:

```
for (n = 0; n<N; n++)  sK[n] = sK[n]*sB + circular(sX[n])*sErr;
```

IP = 2 (1 mul, 1 madd)	LD/ST= 3 (read sX, read sK, write sK)

Assembly code:

```
lea       LC,(N/4 - 2)                  ; (1)   ; get loop number
ld.h      sB,[Baddr]                    ; (2)   ; beta
ld.h      sErr,[Eaddr]                  ; (3)   ; error
ld.w      ssK0,[Kptr+]4                 ; (4)   ; K1 K0
mulr.h    ssKK0,ssK0,sB ll,#1           ; (5)   ; K1=K1*B|| K0 =K0*B
ld.d      ssssX,[a14/a15+c]8            ; ||    ; X3 X2 X1 X0
llmsloop:
          maddrs.h  ssKK0,ssKK0,ssX0,sErr ll,#1
                                        ; (1)   ; K1+=X1*Err
                                        ; ||    ; K0+=X0*Err
          ld.d      ssssK,[Kptr+]8      ; ||    ; K5 K4 K3 K2
          mulr.h    ssKK1,ssK1,sB ll,#1 ; (2)   ; K3= K3*B
                                        ; ||    ; K2= K2*B
          st.w      [Kptr]-12,ssKK0     ; ||    ; store K0 K1
          maddrs.h  ssKK1,ssKK1,ssX1,sErr ll,#1; (3)  ; K3+=X3*Err
                                        ; ||    ; K2+=X2*Err
          ld.d      ssssX,[a14/a15+c]8  ; ||    ; X7 X6 X5 X4
          mulr.h    ssKK1,ssK1,sB ll,#1 ; (4)   ; K5=K5*B
                                        ; ||    ; K4=K4*B
          st.w      [Kptr]-8,ssK1       ; ||    ; store K2 K3
loop      LC,llmsloop
                                        ; epilog
maddrs.h  ssKK0,ssKK0,ssX0,sErr ll,#1   ; (6)   ; K(n-2)+=X(n-2)*Err
                                        ; ||    ; K(n-3)+=X(n-3)*Err
ld.w      ssK1,[Kptr+]4                 ; ||    ; Kn K(n-1)
mulr.h    ssKK1,ssK1,sB ll,#1           ; (7)   ; Kn=Kn*B
                                        ; ||    ; K(n-1)= K(n-1)*B
maddrs.h  ssKK1,ssKK1,ssX1,sErr ll,#1   ; (8,9) ; Kn +=Xn*Err
                                        ; ||    ; K(n-1)+=X(n-1)*Err
st.d      [Kptr]-8,e0                   ; ||    ; Kn K(n-1) K(n-2) K(n-3)
```

Example	N = 20 ➔ 27 cycles

Register representation:

X input	sK1	sK0
Constant input		sB
Result of the instruction	sK1*sB	sK0*sB

Register diagram:

Instruction	**d0**	**d1**	**d3 d2**	**d5 d4**	**Load / Store**
			k1k0 — —		ld k0k1
mulr.h ssKK0,ssK0,sB ll,#1	k1= k1*B || k0= k0*B			x3 x2 x1 x0	ld x0x1x2x3
maddrs.h ssKK0,ssKK0,ssX0, sErr ll,#1	k1+=x1*Err||k0 +=x0*Err		k5 k4 k3 k2		ld k2k3k4k5
mulr.h ssKK1,ssK1,sB ll,#1		k3= k3*B || k2= k2*B			st k0k1
maddrs.h ssKK1,ssKK1,ssX1, sErr ll,#1		k3+=x3*Err|| k2+=x2*Err		x7 x6 x5 x4	ld x4x5x6x7
mulr.h ssKK0,ssK0,sB ll,#1	k5= k5*B || k4= k4*B				st k3k2
maddrs.h ssKK0,ssKK0,ssX0, sErr ll,#1					
mulr.h ssKK1,ssK1,sB ll,#1					
maddrs.h ssKK1,ssKK1,ssX1, sErr ll,#1					

6.17 Delayed LMS

Note: Validated on TriCore Rider D board.

Equations:

$Y_t = \sum X_{t-n} * K_{t-1,n}$ n = 0..N-1

$K_{t,n} = K_{t-1,n} + X_{t-n} * u * err_{t-1}$ n = 0..N-1

Pseudo code:

```
sY = 0;
for (n = 0; n<N; n++)
{
sY += circular(sX[n])*sK[n];
          sK[n] += circular(sX[n])*sErr;
}
```

IP = 2 (2 madd)	LD/ST = 3 (read sX, read sK, write sK)

Assembly code:

```
lea     LC,(N/4 - 1)                        ; (1) ; get loop number
ld.h    sErr,[Eaddr]                        ; (2) ; Err
mov     d0,#0                               ; (3) ; Y = 0 (lower)
ld.w    ssK1,[Kptr+]4                       ; ||  ; K0 K1
mov     d1,#0                               ; (4) ; Y = 0 (upper)
ld.d    ssssX,[a14/a15+c]8                  ; ||  ; X0 X1 X2 X3
dlmsloop:
        maddm.h   llY,llY,ssX0,ssK0 ul,#1   ; (1) ; Y += X0*K0+X1*K1
        st.d      [Kptr]-16,e6              ; ||  ; store (next loop)
                                            ; K0 K1 K2 K3
        maddrs.h  d6,ssK0,ssX0,sErr ll,#1   ; (2)
                                            ; K0 +=X*Err|| K1 +=X1*Err
        ld.d      ssssK,[Kptr+]8            ; ||  ; K2 K3 K4 K5
        maddm.h   llY,llY,ssX1,ssK1 ul,#1   ; (3) ; Y += sX2*K2+X3*K3
        ld.d      ssssX,[a14/a15+c]8        ; ||  ; X4 X5 X6 X7
        maddrs.h  d7,ssK1,ssX1,sErr ll,#1   ; (4)
                                            ; K2 +=X2*Err||K3 +=X3*Err
loop    LC,dlmsloop
st.d    [Kptr]-16,e6; ||                    ; store last 4 K
st.h    [Yptr+]2,d1; (5)                    ; store Y
```

Note: Warning! Dummy store on first iteration of the loop.

Example	N = 20 ➔ 27 cycles

Register representation:

X input	sX1	sX0
Error input		sErr
Result of the instruction	sX1*sErr	sX0*sErr

Register diagram:

Instruction	d1 / d0	d3 d2	d5 d4	d7 d6	Load / Store
	y = 0 (d1)		k1k0 ----		ld k0k1
	y = 0 (d0)	x3x2 x1x0			ld x0x1x2x3
maddm.h llY,llY,ssX0,ssK0 ul,#1	y += x0*k0 + x1*k1		k3k2 k1k0		dummy store
maddrs.h d6,ssK0,ssX0,d8ul,#1			k5k4 k3k2	---------- k1 += x1*err || k0 += x0*err	ld k2k3k4k5
maddm.h llY,llY,ssX1,ssK1 ul,#1	y + = x2*k2 + x3*k3	x7x6 x5x4			ld x4x5x6x7
maddrs.h d7,ssK1,ssX1,d8ul,#1				k3 += x3*err || k2 += x2*err k1 += x1*err || k0 += x0*err	
next iteration					st k0k1k2k3

6.18 Delayed LMS – 32-bit Coefficients

Note: Validated on TriCore Rider D board.

Equations:

$Y_t = \sum X_{t-n} * K_{t-1,n}$ n = 0..N-1

$K_{t,n} = K_{t-1,n} + X_{t-n} * u * err_{t-1}$ n = 0..N-1

Pseudo code:

```
sY = 0;
for (n = 0; n< N; n++)
{
sY += circular(sX[n])* up(lK[n]);
          lK[n] += circular(sX[n])* sErr;
}
```

IP = 2 (2 madd)	LD/ST= 3 (read sX, read lK, write lK)

Assembly code:

```
lea       LC,(N/2 - 2)                  ;(1)   ; get loop number
ld.h      sErr,[Eaddr]                  ;(2)   ; err
mov       d0,#0                         ;(3)   ; y = 0 (lower)
ld.d      llK,[Kptr+]8                  ;||    ; k0k1
mov       d1,#0                         ;(4)   ; y = 0 (upper)
ld.w      ssX,[a14/a15+c]4              ;||    ; x0x1
madds.h   llY,llY,ssX,lK0 ul,#1         ;(5)   ; y +=x0*k0
dlms32loop:
          madds.h llY,llY,ssX,lK1 uu,#1 ;(1)   ; y += x1*k1
          madds.h e6,llK,ssX,sErr ll,#1 ;(2,3)
                                        ;k0=k0+x0*err||k1=k1+x1*err
          ld.d    llK,[Kptr+]8          ;||    ; k3k2
          ld.w    ssX,[a14/a15+c]4      ;||    ; x2x3
          madds.h llY,llY,ssX,lK0 ul,#1 ;(4)   ; y=y+x2*k2
          st.d    [Kptr]-16,llK         ;||    ; store k0k1
loop      LC,dlms32loop
madds.h   llY,llY,ssX,lK1 uu,#1         ;(6)   ; y=y+xn*kn
madds.h   e6,llK,ssX,sErr ll,#1         ;(7,8); kn=kn+xn*err
                                        ;||    ; k(n-1)=k(n-1)+x(n-1)*err
st.h      [Yptr]2,d1                    ;||    ; store Y
st.d      [Kptr]-8,e6                   ;||    ; store k(n-1)kn
```

Example	N = 20 ➔ 46 cycles

Register diagram:

Instruction	d1 / d0	d3 / d2	d5 / d4	d7/ d6	Load / Store
	y = 0 (lower)		k1k0		ld k1k0
	y = 0 (upper)	x1x0			ld x1x0
madds.h llY,llY,ssX,lK0 ul,#1	y += x0*k0				
madds.h llY,llY,ssX,lK1 uu,#1	y += x1*k1				
madds.h e6,llK,ssX,sErr ll,#1			k3k2	k1 += x1*err || k0 += x0*err	ld k3k2
		x3x2			ld x3x2
madds.h llY,llY,ssX,lK0 ul,#1	y += x2*k2		k5k4		st k1k0
madds.h llY,llY,ssX,lK1 uu,#1	$y=y+x_n*k_n$				
madds.h e6,llK,ssX,sErr ll,#1				$k_n=k_n+x_n*err$ || $k_{(n-1)}=k_{(n-1)} + x_{(n-1)}*err$	st y st $k_{(n-1)}k_n$

6.19 Delayed LMS – Complex

Note: Validated on TriCore Rider D board.

Equations:

$Yr_t = \sum Xr_{t-n}*Kr_{t-1,n} - Xi_{t-n}*Ki_{t-1,n}$

$Yi_t = \sum Xr_{t-n}*Ki_{t-1,n} + Xi_{t-n}*Kr_{t-1,n}$

$Kr_{t,n} = Kr_{t-1,n} + Xr_{t-n}*u*Er_{t-1} - Xi_{t-n}*u*Ei_{t-1}$

$Ki_{t,n} = Ki_{t-1,n} + Xr_{t-n}*u*Ei_{t-1} + Xi_{t-n}*u*Er_{t-1}$ $\quad n = 0..N-1$

Pseudo code:

```
sYr = 0;
sYi = 0;
for (n = 0; n<N; n++)
{
sYr += circular(sXr[n]*sKr[n] – circular(sXi[n])*sKi[n];
sYi += circular(sXi[n]*sKr[n] + circular(sXr[n])*sKi[n];
sKr[n] += circular(sXr[n])*sEr - circular(sXi[n])*sEi;
sKi[n] += circular(sXr[n])*sEi + circular(sXi[n])*sEr;
}
```

IP=8 (6 madd, 2 msub)	LD/ST=6 (read sXr, read sXi, read sKr, read sKi, write sKr, write sKi)

Assembly code:

```
lea         LC,(N - 2)                    ; (1) ;get loop number
mov         ssY,#0                        ; (2) ;yi = 0 || yr = 0
ld.w        ssE,[Eaddr]                   ; ||  ;Ei || Er
ld.w        ssX,[Xptr+]4                  ; (3) ;xixr0
ld.w        ssK,[Kptr+]4                  ; (4) ;ki0kr0

maddsurs.h ssY,ssY,ssX,ssK uu,#1          ; (5) ;yi+=xr0*ki0 || yr-=xi0*ki0
maddsurs.h ssK0,ssK,ssX,ssE uu,#1         ; (6) ;ki0+=xr0*ei||kr0-=xi0*ei
dlmscpxloop:
            maddrs.h    ssY,ssY,ssX,ssK ll,#1
                                          ; (1) ;yi+=xi0*kr0
                                          ; ||  ;yr+=xr0*kr0
            ld.w        ssK,[Kptr+]4      ; ||  ;ki1kr1
            maddrs.h    ssK0,ssK0,ssX,ssE ll,#1
                                          ; (2) ;ki0+=xi0*er
                                          ; ||  ;kr0+=xr0*er
            ld.w        ssX,[a14/a15+c]4  ; ||  ;xixr1
            maddsurs.h ssY,ssY,ssX,ssK uu,#1
                                          ; (3) ;yi+=xr1*ki1 || yr-=xi1*ki1
            st.w        [Kptr]-8,ssK0     ; ||  ;store ki0kr0
            maddsurs.h ssK0,ssK,ssX,ssE uu,#1
                                          ; (4) ;ki1+=xr1*ei || kr1-=xi1*ei
loop        LC,dlmscpxloop

maddrs.h    ssY,ssY,ssX,ssK ll,#1         ; (7) ;yi+=xin*krn || yr+=xrn*krn
maddrs.h    ssK0,ssK0,ssX,ssE ll,#1       ; (8,9)
                                          ; kin+=xin*er || krn+=xrn*er
st.w        [Yptr+]4,ssY                  ; ||  ;store yryi
st.w        [Kptr]-4,ssK0                 ; ||  ;store kin krn
```

Example	N = 20 ➔ 87 cycles

Register diagram:

Instruction	d0	d1	d4	d5	d6	Load / Store
	yi = 0 || yr = 0				EiEr	ld EiEr
			xixr0			ld xixr0
				ki0kr0		ld ki0kr0
maddsurs.h ssY,ssY,ssX,ssK uu,#1	yi+=xr0*ki0 || yr-=xi0*ki0					
maddsurs.h ssK1,ssK,ssX,ssE uu,#1		ki0+=xr0*ei ||kr0-=xi0*ei				
maddrs.h ssY,ssY,ssX,ssK ll,#1	yi+=xi0*kr0 || yr+=xr0*kr0			ki1kr1		ld ki1kr1
maddrs.h ssK1,ssK,ssX,ssE ll,#1		ki0+=xi0*er ||kr0+=xr0*er	xixr1			ld xixr1
maddsurs.h ssY,ssY,ssX,ssK uu,#1	yi+=xr1*ki1 || yr-=xi1*ki1			ki0kr0		st ki0kr0
maddsurs.h ssK1,ssK,ssX,ssE uu,#1		ki1+=xr1*ei ||kr1-=xi1*ei				
maddrs.h ssY,ssY,ssX,ssK ll,#1	yi+=xin*krn || yr+=xrn*krn					
maddrs.h ssK1,ssK,ssX,ssE ll,#1		kin+=xin*er || krn+=xrn*er		yryi		st yryi
		kinkrn				st kinkrn

7 Transforms

The DSP functions classed as transforms are much more complete than simple kernel functions. They cover complete applications and can be sub-divided into a series of kernel routines. Taking the example of FFT, this is sub-divided into:

- Initialisation
- Bit reverse
- Pass loop (most external loop)
- Group loop (medium loop)
- Butterfly loop (inner loop)
- Squaring of results
- Normalisation, etc.

*Note: Each loop has a different importance. For a 256 pt FFT the pass loop will be carried out 8 times, whereas the butterfly loop in radix 2 will be done 8*128 times.*

The butterfly can be divided into various types:

- Radix 2 or radix 4 or radix 8 or radix N
- Real or complex data
- Real or complex coefficients
- 16-bit or 32-bit data
- 16-bit or 32-bit coefficients
- Decimation In Time (DIT) or Decimation In Frequency (DIF) butterfly

The butterfly can be implemented in several ways:

- With or without shift
  (shift is generally necessary above 32 points and application dependent)
- Use of block floating point
- Use of packed data (2 points at a time)
- Data incremented in power of 2, coefficients constants
- Data incremented, coefficients incremented
- Data incremented, coefficients incremented in power of 2
- Degenerated (case of first 3 passes in radix 8)

As demonstrated, the number of variables is large.

The most common cases of inner loops of radix 2 butterflies for FFTs are:

- **Real butterfly – DIT – radix 2**
- **Real butterfly – DIF – radix 2**
- **Complex butterfly – DIT – radix 2**
- **Complex butterfly – DIT – radix 2 – with shift**
- **Complex butterfly – DIF – radix 2**

Real butterfly – DIT – radix 2

A simple implementation, 2 points at a time. The coefficients and data are incremented by 2 points.

Conditions:

- Works for pass 1 to pass N-1; does not work for pass 0 (due to packing of data)
- Number of coefficients must be doubled (duplicated table)

Real butterfly – DIF – radix 2

As above, a simple implementation, 2 points at a time. The coefficients and data are incremented by 2 points.

Conditions:

- Works for pass 0 to pass N-2; does not work for pass N-1 (due to packing of data)
- Number of coefficients must be doubled (duplicated table).

Complex butterfly – DIT – radix 2

A straightforward implementation, 2 points at a time. The coefficients are incremented by an index value.

Conditions:

- Works for pass 1 to pass N-1; does not work for pass 0 (due to packing of data)

Complex butterfly – DIT – radix 2 – with shift

Same implementation as above, with shift.

Conditions:

- Works for pass 1 to pass N-1; does not work for pass 0 (due to packing of data)

Complex butterfly – DIF – radix 2

A straightforward implementation, 2 points at a time. The coefficients are incremented by an index value.

Conditions:

- Works for pass 0 to pass N-2; does not work for pass N-1 (due to packing of data)

Name	Cycles	Code Size [1]	Optimization Techniques						Arithmetic Methods	
			Software Pipelining	Loop Unrolling	Packed Operation	Load/Store Scheduling	Data Memory Interleaving	Packed Load/Store	Saturation	Rounding
Real butterfly – DIT[2] radix 2	(5*N/2 +2) +3	32	✓	-	✓	✓	-	✓	-	✓
Real butterfly – DIF[3] radix 2	(5*N/2 +2) +3	36	-	-	-	-	-	✓	✓	✓
Complex butterfly – DIT radix 2	(9*N/2 +2) +3	74	✓	-	✓	✓	-	✓	-	-
Complex butterfly – DIT radix 2 – with shift	(11*N/2 +2) +3	82	✓	-	✓	✓	-	✓	-	-
Complex butterfly – DIF radix 2	(9*N/2 +2) +3	76	✓	-	✓	✓	-	✓	-	-

1) Code size is in Bytes

2) DIT = Decimation In Time

3) DIF = Decimation In Frequency

7.1 Real Butterfly – DIT – Radix 2

Equations:

$X_n' = X_n + Y_n * K_n$

$Y_n' = X_n - Y_n * K_n$ n=0..N-1

Pseudo code:

```
for (n = 0; n<N; n++)
{
          sXX[n] = sX[n] + sY[n]*sK[n];
          sYY[n] = sX[n] - sY[n]*sK[n];
}
```

IP = 2 (1 madd, 1 msub)	LD/ST= 5 (read sX, read sK, read sY, write sXX, write sYY)

Pseudo code implementation:

```
ssP1  = ssY1 * ssK1; ssP0  = ssY0 * ssK0;
ssXX1 = ssX1 + ssP1; ssXX0 = ssX0 + ssP0;
ssYY1 = ssX1 - ssP1; ssYY0 = ssX0 - ssP0;
```

Assembly code:

```
lea       LC,(N/2 - 1)                  ; (1)   ;get loop number
ld.w      ssY,[Yptr+]4                  ; (2)   ;y1y0
ld.w      ssK,[Kptr+]4                  ; (3)   ;k1k0
rditloop:
          mulr.h    ssP,ssY,ssK ul,#1   ;(1,2)  ;p1=y1*k1 || p0=y0*k0
          ld.w      ssX,[Xptr+]4        ; ||    ;x1 x0
          ld.w      ssY,[Yptr+]4        ; ||    ;y3 y2
          add.h     ssXX,ssX,ssP        ; (3)   ;xx1=x1+p1 || xx0=x0+p0
          st.w      [Xptr]-4,ssXX       ; ||    ;store xx1 xx0
          sub.h     ssYY,ssX,ssP        ; (4)   ;yy1=x1-p1 || yy0=x0-p0
          st.w      [Yptr]-8,ssYY       ; ||    ;store yy1 yy0
          ld.w      ssK,[Kptr+]4        ; (5)   ;k3 k2
loop      LC,rditloop
```

Register diagram:

Instruction	d0	d2	d3	d4	d6	d8	Load / Store
				y1 y0			ld y1y0
						k1 k0	ld k1k0
mulr.h ssP,ssY,ssKul,#1			p1=y1*k1 || p0=y0*k0	x1 x0			ld x1x0
					y3 y2		ld y3y2
add.h ssXX,ssX,ssP	x'1=x1+p1 || x'0=x0+p0						st x'1x'0
sub.h ssYY,ssX,ssP		y'1=x1-p1 || y'0=x0-p0					st y'1y'0
						k3 k2	ld k3k2

7.2 Real Butterfly – DIF – Radix 2

Equations:

$X_n' = X_n + Y_n$

$Y_n' = X_n*K_n - Y_n*K_n$ n=0..N-1

Pseudo code:

```
for (n = 0; n<N; n++)
{
          sXX[n] = sX[n] + sY[n];
          sYY[n] = sX[n]*sK[n] - sY[n]*sK[n];
}
```

IP = 3 (1 add, 1 mul, 1 msub)	LD/ST= 5 (read sX, sK, sY, write sXX, sYY)

Pseudo code implementation (loop only):

```
ssXX1 = ssX1 + ssY1; ssXX0 = ssX0 + ssY0;
ssD1  = ssX1 - ssY1; ssD0  = ssX0 - ssY0;
ssYY1 = ssD1*ssK1;   ssYY0 = ssD0*ssK0;
```

Assembly code:

```
lea       LC,(N/2 - 1)                ; (1)    ; get loop number
ld.w      ssY,[Yptr+]4                ; (2)    ; y1y0
ld.w      ssX,[Xptr+]4                ; (3)    ; x1x0
rdifloop:
          add.h    ssXX,ssX,ssY       ; (1)    ; xx1=x1+y1 || xx0=x0+y0
          st.w     [Xptr]-4,ssXX      ; ||     ; store xx1 xx0
          sub.h    ssD,ssX,ssY        ; (2)    ; d1=x1-y1 || d0=x0-y0
          ld.w     ssY,[Yptr+]4       ; ||     ; y3 y2
          ld.w     ssK,[Kptr+]4       ; (3)    ; k1 k0
          mulr.h   ssYY,ssD,ssK ul,#1 ; (4,5)  ; yy1=d1*k1 || yy0=d0*k0
          ld.w     ssX,[Xptr+]4       ; ||     ; x3 x2
          st.w     [Yptr]-8,ssYY      ; ||     ; store yy1 yy0
loop      LC,rdifloop
```

Register diagram:

Instruction	d0	d2	d3 / d2	d4	d6	d8	Load / Store
					y1 y0		ld y1y0
				x1 x0			ld x1x0
add.h ssXX,ssX, ssY	xx1=x1+y1|| xx0=x0+y0						st xx1 xx0
sub.h ssD,ssX,ssY			d1=x1-y1 || d0=x0-y0		y3 y2		ld y3y2
						k1 k0	ld k1k0
mulr.h ssYY,ssD,ssK ul,#1		yy1=d1*k1|| yy0=d0*k0		x3 x2			ld x3x2
		yy1 yy0					st yy1 yy0

7.3 Complex Butterfly – DIT – Radix 2

Equations:

$X'_r = X_r + (Y_r*K_r - Y_i*K_i)$

$X'_i = X_i + (Y_i*K_r + Y_r*K_i)$

$Y'_r = X_r - (Y_r*K_r - Y_i*K_i)$

$Y'_i = X_i - (Y_i*K_r + Y_r*K_i)$

Pseudo code:

```
for (n = 0; n<N; n++)
{
          sXXr[n] = sXr[n] + sYr[n]*sKr[n] - sYi[n]*sKi[n];
          sXXi[n] = sXi[n] + sYi[n]*sKr[n] + sYr[n]*sKi[n];
          sYYr[n] = sXr[n] - sYr[n]*sKr[n] + sYi[n]*sKi[n];
          sYYi[n] = sXi[n] - sYi[n]*sKr[n] - sYr[n]*sKi[n];
}
```

IP=8 (4 madd, 4 msub)	LD/ST= 10 (read sXr, sXi, sYr, sYi, read sKr, sKi, write sXXr, sXXi, sYYr, sYYi)

Pseudo code implementation (loop only):

```
sPi0  = sYr0 * sKi0         ; sPr0  = sYr0 * sKr0;
sPi0  = sPi0 + sYi0 * sKr0  ; sPr0  = sPr0 - sYi0 * sKi0;
sPi1  = sYr1 * sKi1         ; sPr1  = sYr1 * sKr1;
sPi1  = sPi1 + sYi1 * sKr1  ; sPr1  = sPr1 - sYi1 * sKi1;
sXXi0 = sXi0 + sPi0         ; sXXr0 = sXr0 + sPr0;
sYYi0 = sXi0 - sPi0         ; sYYr0 = sXr0 - sPr0;
sXXi1 = sXi1 + sPi1         ; sXXr1 = sXr1 + sPr1;
sYYi1 = sXi1 - sPi1         ; sYYr1 = sXr1 - sPr1;
```

Assembly code:

```
lea    LC,(N/2 - 1)                        ; (1) ;get loop number
lea    kindex,4                            ; (2) ;index is pass dependent
ld.d   e6,[Yptr+]8                         ; (3) ;y1y0
ld.w   ssK1,[Kptr]                         ; (4) ;kikr0
cditloop:
       mulr.h     ssP1,ssK1,ssY1 ll,#1     ; (1) ;pi0=yr0*ki0 || pr0=yr0*kr0
       add.a      Kptr,Kptr,kindex         ; ||  ;
       maddsur.h  ssP1,ssP1,ssK1,ssY1 uu,#1
                                           ; (2)
                                           ; pi0+=kr0*yi0 || pr0-=yi0*ki0
       ld.w       ssK2,[Kptr]              ; ||  ;kikr1
       mulr.h     ssP2,ssK2,ssY2 ll,#1     ; (3)
                                           ; pi1=yr1*ki1 || pr1=yr1*kr1
       maddsur.h  ssP2,ssP2,ssK2,ssY2 uu,#1   ; (4,5)
                                           ; pi1+=kr1*yi1 ||pr1-=yi1*ki1
       ld.d       e4,[Xptr+]8              ; ||  ; xixr1 xixr0
       ld.d       e6,[Yptr+]8              ; ||  ; yiyr3 yiyr2
       add.h      ssXX1,ssX1,ssP1          ; (6)
                                           ; xi'0=xi0+pi0 || xr'0=xr0+pr0
       add.a      Kptr,Kptr,kindex         ; ||  ;
       sub.h      ssYY1,ssX1,ssP1          ; (7)
                                           ; yi'0=xi0-pi0 || yr'0=xr0-pr0
       ld.w       ssK1,[Kptr]              ; ||  ; kikr2
       add.h      ssXX2,ssX2,ssP2          ; (8)
                                           ; xi'1=xi1+pi1 || xr'1=xr1+pr1
       st.d       [Xptr]-8,e0              ; ||  ; store x'1x'0
       sub.h      ssYY2,ssX2,ssP2          ; (9)
                                           ; yi'1=xi1-pi1 || yr'1=xr1-pr1
       st.d       [Yptr]-16,e2             ; ||  ; store yy1yy0
loop LC,cditloop
```

Register diagram:

Instruction	d0	d1	d2	d3	d5 / d4	d7/ d6	d8	d9	d10	d11
						yiyr1 yiyr0				
							kikr0			
mulr.h ssP1,ssK1,ssY1 ll,#1									pi0 || pr0	
maddsur.h ssP1,ssP1,ssK1,ssY1 uu,#1								kikr1	pi0 || pr0	
mulr.h ssP2,ssK2,ssY2 ll,#1										pi1 || pr1
maddsur.h ssP2,ssP2,ssK2,ssY2 uu,#1					xixr1 xixr0					pi1 || pr1
						yiyr3 yiyr2				
add.h ssXX1,ssX1,ssP1	xi'0 || xr'0									
sub.h ssYY1,ssX1,ssP1			yi'0 || yr'0				kikr2			
add.h ssXX2,ssX2,ssP2	x'1x'0	xi'1 || xr'1								
sub.h ssYY2,ssX2,ssP2			yy1 yy0	yi'1 || yr'1						

7.4 Complex Butterfly – DIT – Radix 2 – with shift

Equations:

$X'_r = X_r + (Y_r*K_r - Y_i*K_i)$

$X'_i = X_i + (Y_i*K_r + Y_r*K_i)$

$Y'_r = X_r - (Y_r*K_r - Y_i*K_i)$

$Y'_i = X_i - (Y_i*K_r + Y_r*K_i)$

Pseudo code:

```
for (n = 0; n<N; n++)
{
          sXXr[n] = sXr[n] + sYr[n]*sKr[n] - sYi[n]*sKi[n];
          sXXi[n] = sXi[n] + sYi[n]*sKr[n] + sYr[n]*sKi[n];
          sYYr[n] = sXr[n] - sYr[n]*sKr[n] + sYi[n]*sKi[n];
          sYYi[n] = sXi[n] - sYi[n]*sKr[n] - sYr[n]*sKi[n];
}
```

IP=8 (4 madd, 4 msub)	LD/ST= 10 (read sXr, sXi, sYr, sYi, read sKr, sKi, write sXXr, sXXi, sYYr, sYYi)

Pseudo code implementation (loop only):

```
sPi0 = sYr0 * sKi0          ; sPr0 = sYr0 * sKr0;
sPi0 = sPi0 + sYi0 * sKr0   ; sPr0 = sPr0 - sYi0 * sKi0;
sPi1 = sYr1 * sKi1          ; sPr1 = sYr1 * sKr1;
sPi1 = sPi1 + sYi1 * sKr1   ; sPr1 = sPr1 - sYi1 * sKi1;
sXi0 >> 1                   ; sXr0 >> 1;
sXXi0 = sXi0 + sPi0         ; sXXr0 = sXr0 + sPr0;
sYYi0 = sXi0 - sPi0         ; sYYr0 = sXr0 - sPr0;
sXi1 >> 1                   ; sXr1 >> 1;
sXXi1 = sXi1 + sPi1         ; sXXr1 = sXr1 + sPr1;
sYYi1 = sXi1 - sPi1         ; sYYr1 = sXr1 - sPr1;
```

Assembly code:

```
lea    LC,(N/2 - 1)                        ; (1)   ; get loop number
lea    kindex,4                            ; (2)   ; index is pass dependent
ld.d   e6,[Yptr+]8                         ; (3)   ; y1y0
ld.w   ssK1,[Kptr]                         ; (4)   ; kikr0

cditsloop:
       mulr.h     ssP1,ssK1,ssY1 ll,#0     ; (1)
                                           ; pi0=yr0*ki0 || pr0=yr0*kr0
       add.a      Kptr,Kptr,kindex         ; ||    ;
       maddsur.h  ssP1,ssP1,ssK1,ssY1 uu,#0    ; (2)
                                           ; pi0+=kr0*yi0 || pr0-=yi0*ki0
       ld.w       ssK2,[Kptr]              ; ||    ; kikr1
       mulr.h     ssP2,ssK2,ssY2 ll,#0     ; (3)
                                           ; pi1=yr1*ki1 || pr1=yr1*kr1
       maddsur.h  ssP2,ssP2,ssK2,ssY2 uu,#0    ; (4,5)
                                           ; pi1+=kr1*yi1 || pr1-=yi1*ki1
       ld.d       e4,[Xptr+]8              ; ||    ; xixr1 xixr0
       sha.h      ssX1,ssX1,#-1            ; (6)   ; x0>>1
       add.a      Kptr,Kptr,kindex         ; ||    ;
       add.h      ssXX1,ssX1,ssP1          ; (7)
                                           ; xi'0=xi0+pi0 || xr'0=xr0+pr0
       ld.w       ssK1,[Kptr]              ; ||    ; kikr2
       sub.h      ssYY1,ssX1,ssP1                  ; (8)
                                           ; yi'0=xi0-pi0 || yr'0=xr0-pr0
       ld.d       e6,[Yptr+]8              ; ||    ; yiyr3 yiyr2
       sha.h      ssX2,ssX2,#-1            ; (9)   ; x1>>1
       add.h      ssXX2,ssX2,ssP2          ; (10)
                                           ; xi'1=xi1+pi1 || xr'1=xr1+pr1
       st.d       [Xptr]-8,e0              ; ||    ; store x'1x'0
       sub.h      ssYY2,ssX2,ssP2          ; (11)
                                           ; yi'1=xi1-pi1 || yr'1=xr1-pr1
       st.d       [Yptr]-16,e2             ; ||    ; store yy1yy0
loop LC,cditsloop
```

Register diagram:

Instruction	d0	d1	d2	d3	d5 / d4	d7 / d6	d8	d9	d10	d11
						yiyr1 yiyr0				
							kikr0			
mulr.h ssP1,ssK1,ssY1 ll,#1									pi0|| pr0	
maddsur.h ssP1,ssP1,ssK1,ssY1 uu,#1								kikr1	pi0|| pr0	
mulr.h ssP2,ssK2,ssY2 ll,#1										pi1|| pr1
maddsur.h ssP2,ssP2,ssK2,ssY2 uu,#1					xixr1 xixr0					pi1|| pr1
sha.h ssX1,ssX1,#-1				x0>>1						
add.h ssXX1,ssX1,ssP1	xi'0 || xr'0						kikr2			
sub.h ssYY1,ssX1,ssP1			yi'0|| yr'0			yiyr3 yiyr2				
sha.h ssX2,ssX2,#-1				x1>>1						
add.h ssXX2,ssX2,ssP2	x'1x'0	xi'1|| xr'1								
sub.h ssYY2,ssX2,ssP2			yy1y y0	yi'1||yr'1						

7.5 Complex Butterfly – DIF – Radix 2

Equations:

$X'_r = X_r + Y_r$

$X'_i = X_i + Y_i$

$Y'_r = (X_r - Y_r)*K_r - (X_i - Y_i)*K_i$

$Y'_i = (X_i - Y_i)*K_r + (X_r - Y_r)*K_i$

Pseudo code:

```
for (n = 0; n<N; n++)
{
          sXXr[n] = sXr[n] + sYr[n];
          sXXi[n] = sXi[n] + sYi[n];
          sYYr[n] = (sXr[n] - sYr[n])*sKr[n] - (sXi[n] - sYi[n])*sKi[n];
          sYYi[n] = (sXi[n] - sYi[n])*sKr[n] + (sXr[n] - sYr[n])*sKi[n];
}
```

IP=8 (2 add, 2 sub, 2 mul, 1 madd, 1 msub)	LD/ST= 10 (read sXr, sXi, sYr, sYi, read sKr, sKi, write sXXr, sXXi, sYYr, sYYi)

Pseudo code implementation:

```
sXXi0 = sXi0 + sYi0          ; sXXr0 = sXr0 + sYr0;
sXXi1 = sXi1 + sYi1          ; sXXr1 = sXr1 + sYr1;
sDi0  = sXi0 - sYi0          ; sDr0  = sXr0 - sYr0;
sDi1  = sXi1 - sYi1          ; sDr1  = sXr1 - sYr1;
sYYi0 = sDi0 * sKr0          ; sYYr0 = sDr0 * sKr0;
sYYi0 = sYYi0 + sDr0 * sKi0 ; sYYr0 = sYYr0 - sDi0 * sKi0;
sYYi1 = sDi1 * sKr1          ; sYYr1 = sDr1 * sKr1;
sYYi1 = sYYi1 + sDr1 * sKi1 ; sYYr1 = sYYr1 - sDi1 * sKi1;
```

Assembly code:

```
lea   LC,(N/2 - 1)                    ; (1) ; get loop number
lea   kindex,4                        ; (2) ; index is pass dependent
ld.d  e6,[Yptr+]8                     ; (2) ; y1y0
ld.d  e4,[Xptr+]8                     ; (3) ; x1x0
cdifloop:
      add.h      ssXX1,ssX1,ssY1      ; (1) ; xi'0=xi0+yi0 || xr'0=xr0+yr0
      add.a      Kptr,Kptr,kindex     ; ||  ;
      add.h      ssXX2,ssX2,ssY2      ; (2) ; xi'1=xi1+yi1 || xr'1=xr1+yr1
      ld.w       ssK1,[Kptr]          ; ||  ; kikr0
      sub.h      ssD1,ssX1,ssY1       ; (3) ; di0=xi0-yi0 || dr0=xr0-yr0
      add.a      Kptr,Kptr,kindex     ; ||  ;
      sub.h      ssD2,ssX2,ssY2       ; (4) ; di1=xi1-yi1 || dr1=xr1-yr1
      ld.w       ssK2,[Kptr]          ; ||  ; kikr1
      mulr.h     ssYY1,ssD1,ssK1 ll,#1     ; (5)
                                           ; yi'0=di0*kr0 || yr'0=dr0*kr0
      st.d       [Xptr]-8,e0          ; ||  ; store x'1x'0
      maddsur.h  ssYY1,ssYY1,ssD1,ssK1 uu,#1; (6)
                                           ; yi'0+=dr0*ki0||yr'0-=di0*ki0
      mulr.h     ssYY2,ssD2,ssK2 ll,#1     ; (7)
                                           ; yi'1=di1*kr1 || yr'1=dr1*kr1
      ld.d       e4,[Xptr+]8          ; ||  ; xixr3 xixr2
      maddsur.h  ssYY2,ssYY2,ssD2,ssK2 uu,#1
                                           ;(8,9)
                                           ; yi'1+=dr1*ki1||yr'1-=di1*ki1
      ld.d       e6,[Yptr+]8          ; ||  ; yiyr3 yiyr2
      st.d       [Yptr]-16,e2         ; ||  ; store yy1yy0
loop LC,cdifloop
```

Register diagram:

Instruction	d0	d1	d2	d3	d5 / d4	d7/ d6	d8	d9	d10	d11
						yiyr1 yiyr0				
					xixr1 xixr0					
add.h ssXX1,ssX1,ssP1	xi'0||xr'0									
add.h ssXX2,ssX2,ssP2		xi'1||xr'1					kikr0			
sub.h ssYY1,ssX1,ssP1			di0|| dr0							
sub.h ssYY2,ssX2,ssP2				di1| dr1			kikr1			
mulr.h ssP1,ssK1,ssY1 ll,#1	x'1x'0								yi0|| yr0	
maddsur.h ssP1,ssP1,ssK1,ssY 1 uu,#1									yi0|| yr0	
mulr.h ssP2,ssK2,ssY2 ll,#1					xixr3 xixr2					yi1|| yr1
maddsur.h ssP2,ssP2,ssK2,ssY 2 uu,#1						yiyr3 yiyr2				yi1|| yr1
			yy1 yy0							

8 Appendices

8.1 Tools

Several tools have been used to test the DSP routines shown in this Optimization guide:

- EDE Tasking 1.4 r1:
  Used to build the projects. One project was created for each DSP algorithm, and one project for the cycle count.
- CrossView debugger:
  Used to run the routine. The output is the screen (stdout) or a file.
- A spreadsheet such as Excel:
  Used to compare TriCore fixed-point precision (or approximation of the fixed-point DSP algorithms) and the floating point results.

8.2 TriBoard Project Cycles Count

The project will send the number of cycles of the routine under test by the serial port. The configuration used is indicated below:

Hardware	Software
TriBoard Rider-D w/MMU M3301. Power supply 12V/2A One Parallel cable female/female One serial cable male/male	Tasking EDE 1.4 r1 Crossview debugger 1.1 Windows Hyperterminal Excel

8.2.1 Steps to Run the Project

1. Connect the TriBoard with the computer

2. Launch the Hyperterminal

The settings should be:

Port used : COM1 (This is the com port where the serial cable connects the board to the computer)

Bits/Second : 9600

Data bits : 8

Parity : None

Stop bits : 1

Flow : Hardware

3. Connect the power supply. The board will respond with:

```
Hello World!

I'am the TriBoard with Rider-D
developed at
INFINEON Technologies AG in Munich
Department AI MC AE

St.-Martin-Str. 76
D-81541 Munich
Tel.:+49-89-234-0
Fax.:+49-89-234-81785

If you have questions to this board or to TriCore CPU,
see the manuals on the TriBoard CD.
Have fun working with me!

The CPU running at 160000000 Hz

The EBU running at 80000000 Hz

Checking On Board SDRAM: ok

running since: 00h:00min:02sec
```

4. Launch Tasking EDE and select the project space “Testing DSP”.
   Open the project “cycles count”, add the routine (see 1.3.2) and compile it.

5. Open Crossview debugger 1.1 and load the file cycle count.abs in the board.

6. Run the code. A message is sent by the board:

```
f CPU = 160000000 Hz
f EBU = 80000000 Hz
K = 1
N = 10
--------------------------------
number of cycle : 102018
--------------------------------
```

9 Glossary

Reference	Definition
asm	Assembly code
CPU	Central Processing Unit
DIF	Decimation In Frequency
DIT	Decimation In Time
DSP	Digital Signal Processing
FFT	Fast Fourier Transformations
FIR	Finite Impulse Response
FPI	Flexible Peripheral Interface
IIR	Infinite Impulse Response
IP	Integer Processing
LDMST	Load Modify Store instruction
LMS	Least Mean Square
LS	Load Store
LSB	Least Significant Bit
LT	Less Than - compare instruction
MAC	Multiply and Accumulate
MSB	Most Significant Bit
nop	No Operation
PMU	Program Memory Unit
TC	Abbreviation for TriCore (TC1 or TriCore v2.0, for example)

Infineon goes for Business Excellence

“Business excellence means intelligent approaches and clearly defined processes, which are both constantly under review and ultimately lead to good operating results.
Better operating results and business excellence mean less idleness and wastefulness for all of us, more professional success, more accurate information, a better overview and, thereby, less frustration and more satisfaction.”

Dr. Ulrich Schumacher

http://www.infineon.com

Published by Infineon Technologies AG