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.
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
# TriCore™ 32-bit Unified Processor
## DSP Optimization Guide Part 2: Routines

**Table of Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>8</td>
</tr>
<tr>
<td>1.1</td>
<td>Information Table</td>
<td>9</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Number of Cycles</td>
<td>9</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Arithmetic Methods</td>
<td>13</td>
</tr>
<tr>
<td>1.2</td>
<td>Routine Organization</td>
<td>14</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Equation</td>
<td>14</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Pseudo Code</td>
<td>15</td>
</tr>
<tr>
<td>1.2.3</td>
<td>Pipe Resource Table</td>
<td>15</td>
</tr>
<tr>
<td>1.2.4</td>
<td>Assembly Code</td>
<td>16</td>
</tr>
<tr>
<td>1.2.5</td>
<td>Register Diagram</td>
<td>16</td>
</tr>
<tr>
<td>1.2.6</td>
<td>Notation</td>
<td>17</td>
</tr>
<tr>
<td>1.3</td>
<td>How to Test a DSP Routine</td>
<td>18</td>
</tr>
<tr>
<td>1.3.1</td>
<td>The Golden Models</td>
<td>18</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Generators</td>
<td>19</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Transcendental</td>
<td>19</td>
</tr>
<tr>
<td>1.3.4</td>
<td>Scalars</td>
<td>20</td>
</tr>
<tr>
<td>1.3.5</td>
<td>Vectors</td>
<td>20</td>
</tr>
<tr>
<td>1.3.6</td>
<td>Filters</td>
<td>21</td>
</tr>
<tr>
<td>1.3.7</td>
<td>Transforms</td>
<td>21</td>
</tr>
<tr>
<td>1.4</td>
<td>Measuring Cycles</td>
<td>22</td>
</tr>
<tr>
<td>1.4.1</td>
<td>How to Count Cycles</td>
<td>22</td>
</tr>
<tr>
<td>1.4.1.1</td>
<td>Counting Cycles for a Routine without Loops</td>
<td>23</td>
</tr>
<tr>
<td>1.4.1.2</td>
<td>Counting Cycles for a Routine with Loops</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Generator</td>
<td>28</td>
</tr>
<tr>
<td>2.1</td>
<td>Complex Wave Generation</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>Transcendental Functions</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>Square Root (by Newton-Raphson)</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Square Root (Taylor)</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Inverse (y=1/x)</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>Natural Logarithm (y= ln(x))</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Exponential (y=ex)</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Sine (y=sin(x)), range [-Pi/2, Pi/2]</td>
<td>37</td>
</tr>
<tr>
<td>3.7</td>
<td>Sine (y=sin(x)), range [-Pi, Pi]</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>Scalars</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>16-bit signed Multiplication</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>32-bit signed Multiplication</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>32-bit signed Multiplication (Result on 64-bit)</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>‘C’ Integer Multiplication</td>
<td>44</td>
</tr>
<tr>
<td>4.5</td>
<td>16-bit Update</td>
<td>45</td>
</tr>
<tr>
<td>4.6</td>
<td>32-bit Update</td>
<td>46</td>
</tr>
<tr>
<td>4.7</td>
<td>2nd Order Difference Equation (16-bit)</td>
<td>47</td>
</tr>
</tbody>
</table>
# Table of Contents

## 4 Routines

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8</td>
<td>2nd Order Difference Equation (32-bit)</td>
<td>48</td>
</tr>
<tr>
<td>4.9</td>
<td>Complex Multiplication</td>
<td>49</td>
</tr>
<tr>
<td>4.10</td>
<td>Complex Multiplication (Packed)</td>
<td>51</td>
</tr>
<tr>
<td>4.11</td>
<td>Complex Update</td>
<td>52</td>
</tr>
<tr>
<td>4.12</td>
<td>Complex Update (Packed)</td>
<td>54</td>
</tr>
</tbody>
</table>

## 5 Vectors

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Vector Sum</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Vector Multiplication</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Vector Pre-emphasis</td>
<td>59</td>
</tr>
<tr>
<td>5.4</td>
<td>Vector Square Difference</td>
<td>60</td>
</tr>
<tr>
<td>5.5</td>
<td>Vector Complex Multiplication</td>
<td>62</td>
</tr>
<tr>
<td>5.6</td>
<td>Vector Complex Multiplication (Packed)</td>
<td>64</td>
</tr>
<tr>
<td>5.7</td>
<td>Vector Complex Multiplication (Unrolled)</td>
<td>66</td>
</tr>
<tr>
<td>5.8</td>
<td>Color Space Conversion</td>
<td>68</td>
</tr>
<tr>
<td>5.9</td>
<td>Vector Scaling</td>
<td>71</td>
</tr>
<tr>
<td>5.10</td>
<td>Vector Normalization</td>
<td>73</td>
</tr>
</tbody>
</table>

## 6 Filters

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Dot Product</td>
<td>78</td>
</tr>
<tr>
<td>6.2</td>
<td>Magnitude Square</td>
<td>79</td>
</tr>
<tr>
<td>6.3</td>
<td>Vector Quantization</td>
<td>80</td>
</tr>
<tr>
<td>6.4</td>
<td>First Order FIR</td>
<td>81</td>
</tr>
<tr>
<td>6.5</td>
<td>Second Order FIR</td>
<td>82</td>
</tr>
<tr>
<td>6.6</td>
<td>FIR</td>
<td>84</td>
</tr>
<tr>
<td>6.7</td>
<td>Block FIR</td>
<td>86</td>
</tr>
<tr>
<td>6.8</td>
<td>Auto-Correlation</td>
<td>88</td>
</tr>
<tr>
<td>6.9</td>
<td>Complex FIR</td>
<td>90</td>
</tr>
<tr>
<td>6.10</td>
<td>First Order IIR</td>
<td>92</td>
</tr>
<tr>
<td>6.11</td>
<td>Second Order IIR</td>
<td>94</td>
</tr>
<tr>
<td>6.12</td>
<td>BIQUAD 4 Coefficients</td>
<td>96</td>
</tr>
<tr>
<td>6.13</td>
<td>N-stage BIQUAD 4 Coefficients</td>
<td>98</td>
</tr>
<tr>
<td>6.14</td>
<td>N-stage BIQUAD 5 Coefficients</td>
<td>100</td>
</tr>
<tr>
<td>6.15</td>
<td>Lattice Filter</td>
<td>102</td>
</tr>
<tr>
<td>6.16</td>
<td>Leaky LMS (Update Only)</td>
<td>104</td>
</tr>
<tr>
<td>6.17</td>
<td>Delayed LMS</td>
<td>106</td>
</tr>
<tr>
<td>6.18</td>
<td>Delayed LMS – 32-bit Coefficients</td>
<td>108</td>
</tr>
<tr>
<td>6.19</td>
<td>Delayed LMS – Complex</td>
<td>110</td>
</tr>
</tbody>
</table>

## 7 Transforms

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Real Butterfly – DIT – Radix 2</td>
<td>116</td>
</tr>
<tr>
<td>7.2</td>
<td>Real Butterfly – DIF – Radix 2</td>
<td>118</td>
</tr>
<tr>
<td>7.3</td>
<td>Complex Butterfly – DIT – Radix 2</td>
<td>120</td>
</tr>
<tr>
<td>7.4</td>
<td>Complex Butterfly – DIT – Radix 2 – with shift</td>
<td>123</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5 Complex Butterfly – DIF – Radix 2</td>
<td>126</td>
</tr>
<tr>
<td>8 Appendices</td>
<td>129</td>
</tr>
<tr>
<td>8.1 Tools</td>
<td>129</td>
</tr>
<tr>
<td>8.2 TriBoard Project Cycles Count</td>
<td>129</td>
</tr>
<tr>
<td>8.2.1 Steps to Run the Project</td>
<td>129</td>
</tr>
<tr>
<td>9 Glossary</td>
<td>131</td>
</tr>
</tbody>
</table>
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
- Transcendental Functions
- Scalars
- Vectors
- Filters
- Transforms

The following table identifies the characteristics of these routine types:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generator</td>
<td>single</td>
<td>1 series of ( n ) elements</td>
</tr>
<tr>
<td>Transcendental</td>
<td>single</td>
<td>single</td>
</tr>
<tr>
<td>Scalar</td>
<td>single</td>
<td>single</td>
</tr>
<tr>
<td>Vector (^1)</td>
<td>1 series of ( n ) elements</td>
<td>1 series of ( n ) elements</td>
</tr>
<tr>
<td>Filter</td>
<td>1 series of ( n ) elements</td>
<td>single</td>
</tr>
<tr>
<td>Transform</td>
<td>multiple</td>
<td>multiple</td>
</tr>
<tr>
<td></td>
<td>1 series of ( n ) elements</td>
<td>1 series of ( n ) elements</td>
</tr>
</tbody>
</table>

\(^1\) 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

\[
z = 0;
\text{for } (i=0; i<N; i++) \ z += X[i];
\]

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*

```assembly
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 ; (4)
ld.w  d0, [Xptr] ; (5)
mul  d0, d0, #5 ; (6)
sub  d0, d0, #1 ; (7)
st.w [Xptr+4], d0 ; (8)
```

A cycle is saved with Load and Store scheduling:

```
ld.w  d0, [Xptr] ; (1)
mul  d0, d0, #5 ; (2)
ld.w  d1, [Xptr]+4 ; (3)
sub  d0, d0, #1 ; (4)
st.w [Xptr+4], d0 ; (5)
mul  d1, d1, #5 ; (6)
sub  d1, d1, #1 ; (7)
st.w [Xptr+4], d1 ; (8)
```
Data Memory Interleaving
Here, data are mixed with other types of data in memory.

Example:

\[
\begin{array}{c|c}
X0 \text{ at } 0xD0000000 & X0 \text{ at } 0xD0000000 \\
X1 \text{ at } 0xD0000002 & Xr0 \text{ at } 0xD0000002 \\
Xr0 \text{ at } 0xD0000004 & \text{instead of:} \quad X1 \text{ at } 0xD0000004 \\
Xr1 \text{ at } 0xD0000006 & Xr1 \text{ at } 0xD0000006 \\
\end{array}
\]

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:

\[
\begin{align*}
\text{ld.w } & \quad d0, [Xptr+4] \quad ; \quad \text{ld } X0 \\
\text{ld.w } & \quad d1, [Xptr+4] \quad ; \quad \text{ld } X1 \\
\end{align*}
\]

It can be written as:

\[
\begin{align*}
\text{ld.d } & \quad e0, [Xptr+8] \quad ; \quad \text{ld } X0 \ X1 \\
\end{align*}
\]

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:

<table>
<thead>
<tr>
<th>Title</th>
<th>Equation</th>
<th>Pseudo Code</th>
<th>Pipe Resource Table</th>
<th>Assembly Code</th>
<th>Memory Organization</th>
<th>Number of Cycles</th>
<th>Register Diagram</th>
</tr>
</thead>
</table>
| 5.4 Vector Square Difference | \( Z_n = (X_n - Y_n)^2 \) | \( n = 0..N-1 \) | \( IP = 2 \) (1 sub, 1 mul) \( LD/ST = 3 \) (read \( sX, sY \), write \( sZ \)) | \begin{align*}
\text{lea} & \quad LC, [N/2 - 1] \quad ;(1) \quad \text{get loop number} \\
\text{ld.d} & \quad \text{asseXY}, [\text{Xptr}+8] \quad ;(2) \quad \text{X0 X1 Y0 Y1} \\
\text{sqdloop:} & \\
\text{subs.h} & \quad \text{ssTmp}, \text{ssX}, \text{ssY} \quad ;(1) \quad \text{X1 - Y1 X0 Y0} \\
\text{ld.d} & \quad \text{asseXY}, [\text{Xptr}+8] \quad ;(2) \quad \text{X2 X3 Y2 Y3} \\
\text{mulr.h} & \quad \text{ssZ}, \text{ssTmp}, \text{ssTmp} \quad ul,#1 \quad ;(2,3) \quad (X1 - Y1)^2 \quad (X0 - Y0)^2 \\
\text{st.w} & \quad [\text{Zptr}+4], \text{ssZ} \\
\text{loop} & \quad \text{LC}, \text{sqdloop} \\
\end{align*} | \begin{align*}
Xaddr: & \quad sX0 \\
Xaddr +2: & \quad sX1 \\
Xaddr +4: & \quad sX2 \\
Xaddr +6: & \quad sY1 \\
Xaddr +8: & \quad sY2 \\
Xaddr +10: & \quad \text{etc...} \\
\text{Example:} & \quad \text{416 cycles} \\
\text{Instruction} & \quad d0 \quad d1 \quad d3 / d2 \quad \text{Load/Store} \\
\text{ld x0 y0 x1 x0} & \quad y1 y0 x1 x0 \\
\end{align*} |

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

<table>
<thead>
<tr>
<th>Integer Processing</th>
<th>LD / ST = 2 (READ IX, write IZ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of IP, MAC Instructions</td>
<td>Number of LD / ST Instructions</td>
</tr>
</tbody>
</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.

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

![Diagram of Generators]

### 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.

![Diagram of Transcendental]

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 (.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:

```assembly
;####### 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:

```assembly
;####### 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
.endsm
;--- include routine here
1d.q d1,Xaddr
1d.q d2,Kaddr
mulr.q d15,d1 u,d2 u, #1
st.q Zaddr,d15
;--- end routine
.dup NUM
add d0,d0,d0
.endsm
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"
;j--------------------
;vector complex multiplication (5cycles in the loop)
;j--------------------
.def 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:
1d.w ssK, [Kptr]+4 ; (1) ; load k
1d.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 1u, #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:

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

Assembly code:

```asm
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 $N = 100 \rightarrow 305$
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d3</th>
<th>d5</th>
<th>d4</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>kikr</td>
<td>ld kikr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td>x y</td>
<td>ld x y</td>
</tr>
<tr>
<td>temp,xy,k ll,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madds.r.h</td>
<td></td>
<td></td>
<td>y=x*ki</td>
<td></td>
</tr>
<tr>
<td>xy,temp,xy,k uu,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y=y+y*kr</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st x y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>st x y</td>
<td></td>
</tr>
</tbody>
</table>
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

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

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

\[ Y_0 = 1.1033 - 0.6666\times X \]
\[ Y_{n+1} = Y_n \times (1.5 - (X^2) \times Y_n^2) \quad \text{where } n = 0, 1, 2 \]

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

**Pseudo code (continued):**
\[ sK = 2 \times sX; \]
\[ sY = 1.1033 - 0.6666 \times sX; \]
for \( n = 0; n<3; n++ \) \( sY = sY \times (1.5 - sK \times sY \times sY); \)
\( sZ = sY \times (2 \times sX - 1) + sY; \)

**Assembly code:**

```assembly
movh d0, #0x469c ; (1) ; 1.1033 in 2q14
lea a4, #0x5553 ; (2) ; 0.6666 in 1q15
ld.q sX, [a3+]; || ; load sX in 1q15
movh d9, #0x9000 ; (3) ; d9 = -1 in1q15
movh d8, #0x1000 ; (4) ; d8 = 0.5 in3q13
mov dK, sX ; (5) ; sX in 2q14 (1q15->2q14->*2)
msubr.q sY, d0, sX u, d1 u, #0 ; (6) ; Y0=q14

sqrloop:
mulr.q tmp, sY u, sY u, #1 ; (1,2) ; Y0^2 in q14
sh tmp, tmp, #1 ; (3) ; result in2q14
msubrs.q tmp, d8, tmp u, sK u, #1 ; (4,5) ; 0.5-sK*Y0^2 in 3q13
shas tmp, tmp, #1 ; (6) ; result in 2q14
sh d1, sY, #1 ; (7) ; Y in 3q13
maddrs.s sY, d1, sY u, tmp u, #1 ; (8,9) ; Y0+Y0 in 3q13
shas sY, sY, #1 ; (10) ; Y1 in 2q14
loop a4, sqrloop

adds d6, sX, d9 ; (7) ; 2*sX-1 in 1q15
mulr.q d0, sY u, d6 u, #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)
```

**IP= 4 (3 mul, 1 sub) | LD/ST= 2 (load sK, store sY)**
3.2 Square Root (Taylor)

Equations:
Input: [0.5, 1) in 2Q14 (X should be normalized to 0.5..1)
Output: 1Q15
\[ y^{0.5} = 1 + x - 0.5(x^2) + 0.25(x^3) - 0.625(x^4) + 0.875(x^5) \]

Pseudo code:
\[ sX[0] = (sY - 1)/2; \]
\[ sX[n] = sX[n-1] * sX[n-1]; \]
\[ eY = eY + sX[2*n] * sX[2*n] + sX[2*n+1] * sX[2*n+1]; \]

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
addi d1,d1,#0xc000 ; (5) ; x1=y/2-0.5
st.q [a3+2],d1 ; ||
movh d0,#0x8000 ; (6) ; x0
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,d2u1,#1 ; (11,12) ; y=y+x4k4+x5k5
ld.d e2,[-a2+8] ; || ; k3k2k1k0
maddms.h e0,e0,d2u1,#1 ; (13,14) ; y=y+x2k2+x3k3
ld.w d5,[-a3]4 ; || ; x1k0
maddms.h e0,e0,d3u1,#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.

$$Y_{k+1} = 2\times Y_k (1 - (X/2)\times Y_k)$$

$X, Y$ are 16-bit values

*Pseudo code:*

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

<table>
<thead>
<tr>
<th>IP</th>
<th>LD/ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

*Assembly code:*

```plaintext
lea a2, coef_inv ; (1)
lea a4, 0x02 ; (2)
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:
  msbrs.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
```

User Guide 34 v1.6.4, 2003-01
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.

Output: in 1Q15

Pseudo code:

\[
\text{for}(i=0; i<4; i++) \ sY *= sX + sK[n]
\]

\[
\begin{array}{c|c}
\text{IP= 1 (1 madd)} & \text{LD/ST= 1 (1 ld sK)} \\
\end{array}
\]

Assembly code:

\[
\begin{align*}
\text{lea a2, coef_log} & \quad \text{; (1)} \\
\text{lea a3, 0x03} & \quad \text{; (2)} \quad \text{; initialize the counter} \\
\text{ldq d4, [a4+2]} & \quad \text{; (3)} \quad \text{; load the number to log in 2Q14} \\
\text{movh d5, #0x4000} & \quad \text{; (4)} \quad \text{; 12Q14} \\
\text{ldq d2, [a2+2]} & \quad \text{; (5)} \quad \text{; load k5} \\
\text{sub d4, d4, d5} & \quad \text{; (5)} \quad \text{; z = x - 1} \\
\text{ldq d3, [a2+2]} & \quad \text{; (6)} \quad \text{; load k4} \\
\text{sh d4, d4, #1} & \quad \text{; (6)} \quad \text{; result in 1q15} \\
\text{i1op:} & \\
\text{maddrq d2, d3, d2u, d4u, #1} & \quad \text{; (1,2)} \quad \text{;} \\
\text{ldq d3, [a2+2]} & \quad \text{; (5)} \quad \text{; } \{(k5*z+k4)*z+k3\}z+k2 \text{z+k1} \\
\text{mulr q d6, d2u, d4u, #1} & \quad \text{; (7,8)} \quad \text{; } \{(k5*z+k4)*z+k3\}z+k2 \text{z+k1} \text{z} \\
\text{coef_log: .half 0x0404, 0xeef8, 0x2491, 0xc149, 0x7fe3 ; in 1Q15}
\end{align*}
\]
3.5 Exponential \((y=e^x)\)

Equations:

\[ Y = K_1 \cdot X + K_2 \cdot X^2 + K_3 \cdot X^3 + K_4 \cdot X^4 + K_5 \cdot X^5 + K_6 \cdot X^6 + K_7 \cdot X^7 \]

\(Y, X, K\) are 16-bit values

Input: \([0..1)\) in 1Q15

Output: in 3Q13

Pseudo code:

\[
\text{for}(i=0; i<6; i++) \; \text{sY} *= \text{sX} + \text{sK}[n]
\]

<table>
<thead>
<tr>
<th>(a_{\text{IP}}= 1) (1 \text{ madd})</th>
<th>(a_{\text{LD/ST}}= 1) (1 \text{ ld sK})</th>
</tr>
</thead>
</table>

Assembly code:

```assembly
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
    loop a3, ilop ; || ; load next coeff 2q14
mulr.q d6, d2u, d4u, #1 ; (7,8) ; (((((k7*z+k6)z+k5)z+k4)z+k3)z+k2)z+k1
    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 [-π/2, π/2)

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

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]
```

\[ \text{IP= 1 (1 madd) } \]
\[ \text{LD/ST= 1 (1 ld sK) } \]
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
1Q30
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 0xffffffff, 0x000000c7, 0xfffffe5fe6, 0x00444444, 0x20000000, 0x02000000; 1Q31
and 3Q29
```

User Guide 38 v1.6.4, 2003-01
3.7 Sine (y = \text{sin}(x)), range [-\pi, \pi)

Equations:
\[ \text{Sin}(x) = k_1 x + k_2 x^3 + k_3 x^5 + k_4 x^7 + k_5 x^9 + ... \]

Input: [-1, 1) in 1Q15
Output: [-1, 1) in 1Q31

By using the previous sine, we can get the result for the range [-\pi, \pi] with:
- \text{Sin}(\pi-x) = \text{Sin}(x)

Pseudo code:

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

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

Change of variable:
- If 0x4000 < x < 0xbfff then \(x' = (0x7fff - x) << 1\)
- If 0xc000 < x < 0x3fff then \(x' = x << 1\)
- \(y' = \text{Sine}(x')\)
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,d4,#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
0xffffffffca,0x000005c7,0xffffe5fe6,0x00444444,0xfaaaaaab,0x20000000;1Q31 and 3Q29
## 4 Scalars

<table>
<thead>
<tr>
<th>Name</th>
<th>Cycles</th>
<th>Code Size</th>
<th>Optimization Techniques</th>
<th>Arithmetic Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit signed Multiplication</td>
<td>3,4</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit signed Multiplication</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit signed Multiplication 32-bit result</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit signed Multiplication 64-bit result</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>“C” integer multiplication</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit update</td>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit update</td>
<td>6,5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2nd order diff. equation (16-bit)</td>
<td>4</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2nd order diff. equation (32-bit)</td>
<td>4</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex multiplication</td>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex multiplication (packed)</td>
<td>5</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex update</td>
<td>7</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex update (packed)</td>
<td>6</td>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.1 16-bit signed Multiplication

**Pseudo code:**

\[ sZ = sX \times sK; \]

<table>
<thead>
<tr>
<th>IP= 1 (1 mul)</th>
<th>LD/ST= 3 (read lX, read lK, write lZ)</th>
</tr>
</thead>
</table>

**Assembly code:**

\[
\begin{align*}
; & \text{result = 16-bit} \\
ld.q & \text{ sX, Xaddr } \quad ; (1) \\
ld.q & \text{ sK, Kaddr } \quad ; (2) \\
mul.q & \text{ lZ, sX u, sK u, #1 } \quad ; (3) \quad ; \text{left justified} \\
st.q & \text{ Zaddr, sZ } \quad ; || \quad ; \text{store 16-bit upper} \\

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

; & \text{same with result = rounded 16-bit} \\
ld.q & \text{ sX, Xaddr } \quad ; (1) \\
ld.q & \text{ sK, Kaddr } \quad ; (2) \\
mulr.q & \text{ sZ, sX u, sK u, #1 } \quad ; (3,4) \quad ; \text{left justified} \\
st.q & \text{ Zaddr, sZ } \quad ; || \quad ; \text{store 16-bit upper}
\end{align*}
\]
4.2 32-bit signed Multiplication

**Pseudo code:**

\[ lZ = lX \times lK; \]

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

**Assembly code:**

```assembly
; 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:**

\[ lLZ = lX \times lK; \]

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

**Assembly code:**

```assembly
; 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):

\[
\text{\texttt{lZ = lX*lK;}}
\]

<table>
<thead>
<tr>
<th>IP</th>
<th>LD/ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (1 mul)</td>
<td>3 (read lX, read lK, write lZ)</td>
</tr>
</tbody>
</table>

_Assembly code:_

;  lX, lK, lZ 32-bit signed
\[
\begin{align*}
\text{ld.w} & \quad \text{lX, Xaddr} \\
\text{ld.w} & \quad \text{lK, Kaddr} \\
\text{mul} & \quad \text{lZ, lX, lK} \\
\text{st.w} & \quad \text{Zaddr, lZ}
\end{align*}
\]

;  lX, lK, lZ 32-bit unsigned
\[
\begin{align*}
\text{ld.w} & \quad \text{lX, Xaddr} \\
\text{ld.w} & \quad \text{lK, Kaddr} \\
\text{mul} & \quad \text{lZ, lX, lK} \\
\text{st.w} & \quad \text{Zaddr, lZ}
\end{align*}
\]

; unsigned multiplication gives the same result as a signed multiplication. That is why there is no MUL.U instruction.

; lX, lK, lZ 32-bit signed (saturated result)
\[
\begin{align*}
\text{ld.w} & \quad \text{lX, Xaddr} \\
\text{ld.w} & \quad \text{lK, Kaddr} \\
\text{mul.s} & \quad \text{lZ, lX, lK} \\
\text{st.w} & \quad \text{Zaddr, lZ}
\end{align*}
\]

; lX, lK, lZ 32-bit unsigned (saturated result)
\[
\begin{align*}
\text{ld.w} & \quad \text{lX, Xaddr} \\
\text{ld.w} & \quad \text{lK, Kaddr} \\
\text{mul.s.u} & \quad \text{lZ, lX, lK} \\
\text{st.w} & \quad \text{Zaddr, lZ}
\end{align*}
\]
4.5 16-bit Update

**Pseudo code:**

\[ sZ = sY \times sX + sK; \]

<table>
<thead>
<tr>
<th>IP</th>
<th>LD/ST= 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 madd</td>
<td>(read sX, read sK, read sY, write sZ)</td>
</tr>
</tbody>
</table>

**Assembly code:**

\[
\begin{align*}
; \text{result} &= 32\text{-bit} \\
\text{ld.q} &\ sX,Xaddr \quad ; (1) \\
\text{ld.q} &\ sK,Kaddr \quad ; (2) \\
\text{ld.q} &\ sY,Yaddr \quad ; (3) \\
\text{madds.q} &\ 12,sY,sX \, u,sK \, u,\#1 \quad ; (4,5) \\
\text{st.q} &\ 2addr,sZ \\
\end{align*}
\]

\[
\begin{align*}
; \text{same with result} &= \text{rounded 16\text{-bit}} \\
\text{ld.q} &\ sX,Xaddr \quad ; (1) \\
\text{ld.q} &\ sK,Kaddr \quad ; (2) \\
\text{ld.q} &\ sY,Yaddr \quad ; (3) \\
\text{maddrs.q} &\ sZ,sY,sX \, u,sK \, u,\#1 \quad ; (4,5) \\
\text{st.q} &\ 2addr,sZ \\
\end{align*}
\]

The pseudo code \( sZ = sY \times sX + sK \) would have been written \( sZ = sY + (\text{long}) (sX \times sK) \), since the result of a 16\times16-bit multiplication is a 32-bit result in hardware. \( sZ = sY + (\text{long}) (sX \times sK) \) is actually equivalent to \( sZ = sY + (\text{short}) (sX \times sK) \). 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\times16-bit multiplication is a short value. The pseudo code is:

\[ sZ = sY \times sX + sK \]
4.6 32-bit Update

Pseudo code:
\[ Y += X \times K; \]

| IP = 1 (1 madd) | LD/ST= 4 (read X, read K, read Y, write Y) |

Assembly code:

; 32*32-bit multiplication, result = 32-bit
ld.w lX,Xaddr ; (1) ;
lw lK,Kaddr ; (2) ;
lw 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) ;
lw lK,Kaddr ; (2) ;
lw 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) ;
lw lK,Kaddr ; (2) ;
lw 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:**

\[
sY = sX - 2*sX1 + sX2;
\]

<table>
<thead>
<tr>
<th>IP= 3</th>
<th>LD/ST= 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 add, 1 sub, 1mul)</td>
<td>(read sX, read sX1, read sX2, write sY)</td>
</tr>
</tbody>
</table>

**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:**

<table>
<thead>
<tr>
<th>[Xptr] --&gt;</th>
<th>Xaddr</th>
<th>Xaddr + 2</th>
<th>Xaddr + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sX</td>
<td>sX1</td>
<td>sX2</td>
</tr>
</tbody>
</table>
4.8 2nd Order Difference Equation (32-bit)

**Pseudo code:**

\[ Y = X - 2 \times X_1 + X_2; \]

<table>
<thead>
<tr>
<th>IP= 3</th>
<th>LD/ST= 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 add, 1 sub, 1mul)</td>
<td>(read 1X, read 1X1, read 1X2, write 1Y)</td>
</tr>
</tbody>
</table>

**Assembly code:**

1. \( \text{ld.d } X1/1X, [Xptr] \)  
   \( \text{ld.d } X1, X1, #1 \)  
   \( \text{sub } Y, X1X, X11 \)  
   \( \text{ld.w } X2, [Xptr] + 8 \)  
   \( \text{add } Y, Y, X2 \)  
   \( \text{st.w } Yaddr, Y \)

**Memory organization:**

```
[Xptr]  Xaddr  Xaddr + 4  Xaddr + 8
        IX          IX1           IX2
```

\( \text{ld.d } X1/1X, [Xptr] \)  
\( \text{ld.d } X1, X1, #1 \)  
\( \text{sh } X11, X12, #1 \)  
\( \text{sub } Y, X1X, X11 \)  
\( \text{ld.w } X2, [Xptr] + 8 \)  
\( \text{add } Y, Y, X2 \)  
\( \text{st.w } Yaddr, Y \)
4.9 Complex Multiplication

Equations:
\[ Y_r = X_r \cdot K_r - X_i \cdot K_i \]
\[ Y_i = X_r \cdot K_i + X_i \cdot K_r \]

Pseudo code:
\[
\begin{align*}
\text{sYr} &= \text{sXr}\cdot\text{sKr} - \text{sXi}\cdot\text{sKi}; \\
\text{sYi} &= \text{sXr}\cdot\text{sKi} + \text{sXi}\cdot\text{sKr}; \\
\end{align*}
\]

Assembly code:
\[
\begin{align*}
\text{mov} &~ d6,\#0 ; (1) ; \\
\text{ld.w} &~ \text{ssK},[\text{Xptr}]+4 ; || ; \text{Ki} \text{ Kr} \\
\text{mov} &~ d7,\#0 ; (2) ; \\
\text{ld.w} &~ \text{ssX},[\text{Xptr}]+4 ; || ; \text{Xi} \text{ Xr} \\
\text{msubadm.h} &~ e0,e6,\text{ssK},\text{ssX} ul,#1 ; (3) ; \text{Yr} = \text{Xr}\cdot\text{Kr} - \text{Xi}\cdot\text{Ki} \\
\text{mulm.h} &~ e2,\text{ssK},\text{ssX} lu,#1 ; (4) ; \text{Yi} = \text{Xr}\cdot\text{Ki} + \text{Xi}\cdot\text{Kr} \\
\text{st.h} &~ [\text{Yptr}]+2,\text{sYr} ; || ; \text{store Yr} \\
\text{st.h} &~ [\text{Yptr}]+2,\text{sYi} ; (5) ; \text{store Yi}
\end{align*}
\]

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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1/ d0</th>
<th>d3/ d2</th>
<th>d5</th>
<th>d4</th>
<th>d7/ d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>kikr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld krki</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld xxr</td>
</tr>
<tr>
<td>msubadm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e0,e6,ssK,ssX ul,#1</td>
<td>y_i = x_i * k_r - x_i * k_i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st y_r</td>
</tr>
<tr>
<td>mulm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e2,ssK,ssX lu,#1</td>
<td>y_i = x_i * k_r + x_i * k_r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st y_i</td>
</tr>
</tbody>
</table>
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:
\[ sY_r = sX_r*sK_r - sX_i*sK_i; \]
\[ sY_i = sX_r*sK_i + sX_i*sK_r; \]

Pseudo code implementation:
\[ sY_i = sX_r*sK_i; \quad sY_r = sX_r*sK_r; \]
\[ sY_i += sX_i*sK_r; \quad sY_r -= sX_i*sK_i; \]

Assembly code:
\[ \text{ld.w} \quad \text{ssK}, \lbrack \text{Kptr+}\rbrack 4; \quad (1) \quad K_i \quad K_r \]
\[ \text{ld.w} \quad \text{ssX}, \lbrack \text{Xptr+}\rbrack 4; \quad (2) \quad X_i \quad X_r \]
\[ \text{mulr.h} \quad \text{ssY}, \text{ssK}, \text{ssX} \quad 11, \#1; \quad (3) \quad (\text{read } sX_r, sX_i, sK_i, sK_r, \text{ write } sY_r, sY_i) \]
\[ \text{maddsurs.h} \quad \text{ssY}, \text{ssY}, \text{ssK}, \text{ssX} \quad 11, \#1; \quad (4,5) \quad (\text{read } sX_r||sX_i, sK_i||sK_r, \text{ write } sY_r||sY_i) \]
\[ \text{st.w} \quad \lbrack \text{Yptr+}\rbrack, \text{ssY}; \quad (6) \quad \text{store } sY_r||sY_i \]

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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d5</th>
<th>d4</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ki kr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld krki</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xi xr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld xrxi</td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td></td>
<td>yi = x<em>r</em>ki</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsurs.h</td>
<td></td>
<td></td>
<td></td>
<td>yi += x<em>i</em>kr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st yiyr</td>
</tr>
</tbody>
</table>
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:
\[ sZr = sYr + sXr*sKr - sXi*sKi; \]
\[ sZi = sYi + sXr*sKi + sXi*sKr; \]

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] ; (6,7) ; Yi
st.h [Zptr+2],sZr ; (6,7) ; store Zr
maddm.h e0,e2,ssK,ssX 1u, #1 ; (6,7) ; Zl = Yi + Xr*Ki + Xi*Kr
st.h [Zptr+2],sZi ; (6,7) ; store Zl
```

The MSUBADM.H instruction is used because the memory organization is imaginary || real. The equation is \( z_r = y_r - (x_i*k_i - x_r*k_r) \), which is equivalent to \( z_r = y_r + x_r*k_r - x_i*k_i \).
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1/ d0</th>
<th>d2</th>
<th>d3</th>
<th>d5</th>
<th>d4</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>kikr</td>
<td></td>
<td></td>
<td></td>
<td>Id kikr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>xixr</td>
<td></td>
<td></td>
<td>Id xixr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yr</td>
<td></td>
<td>Id yr</td>
</tr>
<tr>
<td>msubadm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e0,e2,ssK,ssX ul,#1</td>
<td>zr = yr + xr<em>kr - xi</em>ki</td>
<td>yi</td>
<td></td>
<td></td>
<td></td>
<td>Id yi</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e0,e2,ssK,ssX lu,#1</td>
<td>zi = yi + xr<em>ki + xi</em>kr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Id yi</td>
</tr>
</tbody>
</table>

- **maddm.h**
  - Description: Adds the product of two vectors with a constant value.
  - Example: `zi = yi + xr*ki + xi*kr`
4.12 Complex Update (Packed)

Equations:
\[ Z_r = Y_r + X_r \cdot K_r - X_i \cdot K_i \]
\[ Z_i = Y_i + X_r \cdot K_i + X_i \cdot K_r \]

Pseudo code:
\[
\begin{align*}
  sZr &= sYr + sXr \cdot sKr - sXi \cdot sKi; \\
  sZi &= sYi + sXr \cdot sKi + sXi \cdot sKr;
\end{align*}
\]

<table>
<thead>
<tr>
<th>IP= 4 {3 madd, 1 msub}</th>
<th>LD/ST= 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>(read sXr, sXi, sKr, sKi, sYr, sYi, write sZr, sZi) equivalent to (packed format)= 4</td>
<td></td>
</tr>
<tr>
<td>(read sXr</td>
<td></td>
</tr>
</tbody>
</table>

Pseudo code implementation:
\[
\begin{align*}
  sZi &= sYi + sXr \cdot sKi; \\
  sZr &= sYr + sXr \cdot sKr; \\
  sZi &= sXr \cdot sKi; \\
  sZr &= sXr \cdot sKr;
\end{align*}
\]

Assembly code:
\[
\begin{align*}
  \text{ld.w } ssK, [Kptr+\uparrow4] &; (1) \text{ Ki Kr} \\
  \text{ld.w } ssX, [Xptr+\uparrow4] &; (2) \text{ Xi Xr} \\
  \text{maddrs.h ssZ,ssY,ssK,ssX ll,#1} &; (4) \text{ Zi = Yi+Xr*ki || Zr = Yr+Xr*Kr} \\
  \text{maddsurh.ssZ,ssZ,ssK,ssX uu,#1} &; (5,6) \text{ Zi +=Xi*kr || Zr -= Xi*Ki} \\
  \text{st.w } [Zptr+\uparrow4,ssZ] &; \text{ store Zi Zr}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d2</th>
<th>d4</th>
<th>d5</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>kikr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>id kikr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xixr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>id xixr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yiyr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>id yiyr</td>
</tr>
<tr>
<td>maddrs.h ssZ,ssY,ssK,ssX ll,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ziyi=yi+xr*ki</td>
</tr>
<tr>
<td>maddsur.h ssZ,ssZ,ssK,ssX uu,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ziyi=zi+xi*kr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>zizr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st zizr</td>
</tr>
</tbody>
</table>
## Vectors

<table>
<thead>
<tr>
<th>Name</th>
<th>Cycles</th>
<th>Code Size 1)</th>
<th>Optimization Techniques</th>
<th>Arithmetic Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software Pipelining</td>
<td>Load/Store Scheduling</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Loop Unrolling</td>
<td>Data Memory Interleaving</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Packed Operation</td>
<td>Packed Load/Store</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Saturation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rounding</td>
</tr>
<tr>
<td>Vector sum</td>
<td>$(3N/4 + 2) + 3$</td>
<td>32</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Vector multiplication</td>
<td>$(3N/4 + 2) + 3$</td>
<td>32</td>
<td>✓ ✓ ✓ ✓ - ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Vector pre-emphasis</td>
<td>$(3N/4 + 2) + 3$</td>
<td>40</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Vector square difference</td>
<td>$(3N/2 + 2) + 2$</td>
<td>24</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Vector complex multiplication</td>
<td>$(5N + 2) + 3$</td>
<td>28</td>
<td>- - ✓ ✓ - ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Vector complex multiplication (packed)</td>
<td>$(4N + 2) + 3$</td>
<td>24</td>
<td>- - ✓ ✓ - ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Vector complex multiplication (unroll)</td>
<td>$(2N + 2) + 4$</td>
<td>42</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Color space conversion</td>
<td>11</td>
<td>64</td>
<td>✓ ✓ ✓ ✓ - ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Vector scaling</td>
<td>$(2N/2 + 2) + 2$</td>
<td>28, 20</td>
<td>- - - - - ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Vector normalization</td>
<td>$(2N/2 + 2) + 7$</td>
<td>54</td>
<td>- - ✓ ✓ - ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

1) Code Size is in Bytes
5.1 Vector Sum

Equation:

\[ Z_n = V_n + W_n \quad n = 0..N-1 \]

Pseudo code:

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

<table>
<thead>
<tr>
<th>IP</th>
<th>LD/ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Assembly code:

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

Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d5 / d4</th>
<th>d7 / d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld v0v1</td>
<td>v1 v0 _ _</td>
<td></td>
<td></td>
<td>ld w0v1w2w3</td>
</tr>
<tr>
<td>adds.h</td>
<td>d1= v1+w1</td>
<td></td>
<td>v0+w0</td>
<td>v5 v4 v3 v2</td>
</tr>
<tr>
<td>ssZ1,ssV2,ssW1</td>
<td>d0= v1+w1</td>
<td></td>
<td>v0+w0</td>
<td></td>
</tr>
<tr>
<td>adds.h</td>
<td>d1= v3+w3</td>
<td></td>
<td>v2+w2</td>
<td>w7 w6 w5 w4</td>
</tr>
<tr>
<td>ssZ2,ssV1,ssW2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d1= z3 z2</td>
<td>d0= z1 z0</td>
<td></td>
<td></td>
<td>st z0z1z2z3</td>
</tr>
</tbody>
</table>

Example

\( N = 64 \Rightarrow 53 \) cycles
5.2 Vector Multiplication

Equation:
\[ Z_n = V_n \times W_n \quad n = 0..N-1 \]

Pseudo code:
for (n=0; n<N; n++)  sZ[n] = sV[n] \times sW[n];

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

Register diagram:

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

Example  
N = 64  \rightarrow  53 cycles
5.3 Vector Pre-emphasis

Equation:
\[ Z_n = V_n + X_n \cdot K \quad n = 0..N-1 \quad K = -28180 \]

Pseudo code:
for (n=0; n<N; n++)  \( sZ[n] = sV[n] + sX[n] \cdot sK \);

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 \( N = 160 \rightarrow 125 \) cycles

Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1/ d0</th>
<th>d3/ d2</th>
<th>d5/ d4</th>
<th>d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>maddrs.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssZ0,ssV0,ssX0,d6 ul,#1</td>
<td>( z1 )</td>
<td>( z0 )</td>
<td>( x5 ) x4 x3 x2</td>
<td>( x )</td>
<td>( x2 )x3x4x5</td>
</tr>
<tr>
<td>maddrs.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssZ1,ssV1,ssX1,d6 ul,#1</td>
<td>( z3 ) ( z2 ) ( z1 ) ( z0 )</td>
<td>( v7 ) v6 v5 v4</td>
<td>( v )</td>
<td>( v4 )v5v6v7</td>
<td></td>
</tr>
</tbody>
</table>

User Guide 59 v1.6.4, 2003-01
5.4 Vector Square Difference

Equation:

\[ Z_n = (X_n - Y_n)^2 \quad n = 0..N-1 \]

Pseudo code:

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

Assembly code:

```assembly
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:

<table>
<thead>
<tr>
<th>Xaddr</th>
<th>sX0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xaddr + 2</td>
<td>sX1</td>
</tr>
<tr>
<td>Xaddr + 4</td>
<td>sY0</td>
</tr>
<tr>
<td>Xaddr + 6</td>
<td>sY1</td>
</tr>
<tr>
<td>Xaddr + 8</td>
<td>sX2</td>
</tr>
<tr>
<td>Xaddr + 10</td>
<td>etc...</td>
</tr>
</tbody>
</table>

Example

\[ N = 160 \Rightarrow 244 \text{ cycles} \]
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d1</th>
<th>d3 / d2</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>subs.h ssTmp, ssX, ssY</td>
<td></td>
<td>x1−y1</td>
<td>x0−y0</td>
<td>ld x0x1y0y1</td>
</tr>
<tr>
<td>mulr.h ssZ, ssTmp, ssTmp ul,#1</td>
<td>(x1−y1)^2</td>
<td>(x0−y0)^2</td>
<td></td>
<td>st z0z1</td>
</tr>
</tbody>
</table>
5.5 Vector Complex Multiplication

Equations:

\[\begin{align*}
Y_r[n] &= X_r[n]K_r[n] - X_i[n]K_i[n] \quad n = 0..N-1 \\
Y_i[n] &= X_r[n]K_i[n] + X_i[n]K_r[n]
\end{align*}\]

Pseudo code:

```c
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];
}
```

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 \(\rightarrow\) 325 cycles
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d3 / d2</th>
<th>d4</th>
<th>d5</th>
<th>d7/ d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d6=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d7=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ki kr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld kikr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xi xr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld xixr</td>
</tr>
<tr>
<td>msubadm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yr</td>
</tr>
<tr>
<td>e0,e6,ssK,ssX ul,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yi</td>
</tr>
<tr>
<td>e2,ssK,ssX lu,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st yr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st yi</td>
</tr>
</tbody>
</table>
5.6 Vector Complex Multiplication (Packed)

Equations:
\[ Y_r[n] = X_r[n] * K_r[n] - X_i[n] * K_i[n] \quad n = 0..N-1 \]
\[ Y_i[n] = X_r[n] * K_i[n] + X_i[n] * K_r[n] \]

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];
}
```

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,ssX,ssK ll,#1 ; (1)
    Yi = Xr*Ki || Yr = Xr*Kr
maddsurs.h ssY,ssY,ssK,ssX uu,#1 ; (2,3)
    Yi+=Xi*Kr || Yi-=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: \( N = 64 \Rightarrow 261 \text{ cycles} \)
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d5</th>
<th>d4</th>
<th>d7/ d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>kikr</td>
<td></td>
<td></td>
<td>ld kikr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>xixr</td>
<td></td>
<td>ld xixr</td>
</tr>
<tr>
<td>mulr.h ssY,ssK,ssX ll,#1</td>
<td></td>
<td>yi=xi*ki</td>
<td></td>
<td>yr=xi*kr</td>
<td></td>
</tr>
<tr>
<td>maddsurs.h ssY,ssY,ssK,ssX uu,#1</td>
<td></td>
<td>yi=yi+xi*kr</td>
<td></td>
<td>yr=yr-xi*ki kikr</td>
<td>ld kikr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>xixr</td>
<td></td>
<td>ld xixr</td>
</tr>
<tr>
<td></td>
<td>yiyr</td>
<td></td>
<td></td>
<td></td>
<td>st yiyr</td>
</tr>
</tbody>
</table>
5.7 Vector Complex Multiplication (Unrolled)

Equations:
\[
Y_{rn} = X_{rn}K_{rn} - X_{in}K_{in} \quad n = 0..N-1
\]
\[
Y_{in} = X_{rn}K_{in} + X_{in}K_{rn}
\]

Pseudo code:

```c
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];
}
```

Pseudo Code implementation:

```c
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+1] = sXr[2*n+1]*sKr[2*n+1];
    sYi[2*n+1] = sXr[2*n+1]*sKi[2*n+1];
}
```

Assembly code:

```assembly
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*Kr ||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 Yi1 Yr1
    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
```

IP= 4
LD/ST= 3 in packed format (read sXr || sXi, sKr || sKi, write sYr || sYi)
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d1</th>
<th>d5 d4</th>
<th>d7 d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td>ki0</td>
<td>kr0</td>
<td>ld k0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ki1</td>
<td>x1</td>
<td>ld x0 x1</td>
</tr>
<tr>
<td>maddh.s</td>
<td>y0 = xi0 * ki0</td>
<td></td>
<td>yr0 = xi0 * kr0</td>
<td>st y1</td>
<td></td>
</tr>
<tr>
<td>ssY0,ssK0,ssX0 u1</td>
<td></td>
<td></td>
<td>ki2</td>
<td>kr2</td>
<td>ld k1 k2</td>
</tr>
<tr>
<td>madder.s</td>
<td>y0 += xi0 * kr0</td>
<td></td>
<td>yr0 -= xi0 * ki0</td>
<td>ki1</td>
<td></td>
</tr>
<tr>
<td>ssY0,ssX0 u0 u1</td>
<td></td>
<td></td>
<td>yi0</td>
<td>yr1</td>
<td>st y0</td>
</tr>
<tr>
<td>madder.s</td>
<td>yi1 += xi1 * ki1</td>
<td></td>
<td>yr1 -= xi1 * kr1</td>
<td>x1</td>
<td></td>
</tr>
<tr>
<td>ssY1,ssK1,ssX1 u1</td>
<td></td>
<td></td>
<td>xi3</td>
<td>x2</td>
<td>ld x2 x3</td>
</tr>
</tbody>
</table>
5.8  Color Space Conversion

_Equation:_

\[
X_i = \sum A_{ij} \cdot R'_j + K_j \quad \text{for } i, j = 0..n
\]

<table>
<thead>
<tr>
<th>X</th>
<th>A</th>
<th>R'</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>0.257</td>
<td>0.504</td>
<td>0.098</td>
</tr>
<tr>
<td>Cr</td>
<td>0.439</td>
<td>-0.368</td>
<td>-0.071</td>
</tr>
<tr>
<td>Cb</td>
<td>-0.148</td>
<td>-0.291</td>
<td>0.439</td>
</tr>
</tbody>
</table>

* 0 for UPF format, 16 for CCIR 601.

_Pseudo code:_

\[
sY = 0.257 \cdot sR + 0.504 \cdot sG + 0.098 \cdot sB;
sCr = 0.439 \cdot sR - 0.368 \cdot sG - 0.071 \cdot sB + 0.5;\nsCb = -0.148 \cdot sR - 0.291 \cdot sG + 0.439 \cdot sB + 0.5;
\]

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

(Assembly code and Register diagram follow)
Assembly code:

```assembly
ldap 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

mullm h 11Y, d2, d4ul, #1 ; (3) ; 11Y = sR*sK00 + sG*sK01
maddlq 11Y, 11Y, d3l, d5l, #1 ; (4, 5) ; 11Y = 11Y + sB*sK02

ld.d e4, [Kptr+6]   ; || ; sK10, sK11, sK12
maddm.h llCr, e0, d2, d4ul, #1 ; (6) ; llCr = sR*sK10 + sG*sK11 + 128
maddlq 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
maddlq 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, 0xdc1, 0x3831 ; ghiin Q15
rgbvalue: .half 0x4000, 0x3000, 0x2000 ; RGB are [0;+1] in Q14
```
### Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1/d0</th>
<th>d3/d2</th>
<th>d5/d4</th>
<th>d9/d8</th>
<th>d11/d10</th>
<th>d13/d12</th>
<th>Load/Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R'G'B'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Id R'G'B'</td>
</tr>
<tr>
<td>offset128</td>
<td>k00,</td>
<td>k01,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Id k00,</td>
</tr>
<tr>
<td></td>
<td>k02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k01, k02</td>
</tr>
<tr>
<td>mulm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y=R'*k00+G'*k01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lly,d2,d4u,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madd.q</td>
<td>k10,</td>
<td>k11,</td>
<td></td>
<td></td>
<td>Y = Y + B'*k02</td>
<td></td>
<td>Id k10,</td>
</tr>
<tr>
<td></td>
<td>k12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k11, k12</td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cr=R'*k10+G'*k11+128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lly,d3i,d5i,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madd.m</td>
<td>k20,</td>
<td>k21,</td>
<td></td>
<td></td>
<td>Cr=Cr+B'*k12</td>
<td></td>
<td>Id k20,</td>
</tr>
<tr>
<td></td>
<td>k22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k21, k22</td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>lly,e0,d2,d4u,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madd.q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cr=Cb+B'*k22</td>
<td></td>
<td>st Cr</td>
</tr>
<tr>
<td>lly,lly,d3i,d5i,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

User Guide 70 v1.6.4, 2003-01
5.9 Vector Scaling

Equation:
\[ Z_n = (X_n >> 3) << 2 \]
or \[ Z_n = (X_n >> \text{shift1}) << \text{shift2} \quad n = 0..N-1 \]

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

Assembly code:
\[
\begin{align*}
\text{mov} & \quad \text{Shift1},#-3 ; (1) \quad \text{load 1st shift value} \\
\text{lea} & \quad \text{LC},(N/2 - 1) ; || \quad \text{get loop number} \\
\text{mov} & \quad \text{Shift2},#2 ; (2) \quad \text{load 2nd shift value} \\
\text{ld.w} & \quad ssX,[Xptr+]4 ; || \quad X0 X1 \\
\text{isloop} : & \\
\text{sha.h} & \quad d1,ssX,Shift1 ; (1) \quad X0>>3, X1>>3 \\
\text{ld.w} & \quad ssX,[Xptr+]4 ; || \quad X2 X3 \\
\text{sha.h} & \quad ssZ,d1,Shift2 ; (2) \quad X0<<2, X1<<2 \\
\text{t.w} & \quad [Zptr+]4,ssZ ; || \quad \text{store Z0,Z1} \\
\text{loop} & \quad \text{LC, isloop} \\
\end{align*}
\]
Alternatively, the 2 shift values can be directly used
\[
\begin{align*}
\text{lea} & \quad \text{LC},(N/2 - 1) ; (1) \quad \text{get loop number} \\
\text{ld.w} & \quad ssX,[Xptr+]4 ; (2) \quad X0 X1 \\
\text{isloop} : & \\
\text{sha.h} & \quad d1,ssX,#-3 ; (1) \quad X0>>3, X1>>3 \\
\text{ld.w} & \quad ssX,[Xptr+]4 ; || \quad X2 X3 \\
\text{sha.h} & \quad ssZ,d1,#2 ; (2) \quad X0<<2, X1<<2 \\
\text{st.w} & \quad [Zptr+]4,ssZ ; || \quad \text{store Z0,Z1} \\
\text{loop} & \quad \text{LC, isloop} \\
\end{align*}
\]

Example
\[ N = 160 \Rightarrow 164 \text{ cycles} \]
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1/ d0</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d7/ d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shift1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Id shift1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shift2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Id shift2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x1x0</td>
<td></td>
<td></td>
<td></td>
<td>Id x0x1</td>
</tr>
<tr>
<td>sha.h d1,ssX,Shift1</td>
<td>x1&gt;&gt;3</td>
<td>x0&gt;&gt;3</td>
<td></td>
<td></td>
<td></td>
<td>Id x2x3</td>
</tr>
<tr>
<td>sha.h ssZ,d1,Shift2</td>
<td>x1&lt;&lt;2</td>
<td>x0&lt;&lt;2</td>
<td></td>
<td></td>
<td></td>
<td>St z0z1</td>
</tr>
</tbody>
</table>
5.10 Vector Normalization

Equations:
(1) \( \text{minex} = \text{minimum} (\text{minex}, \text{exponent} (X_n)) \quad n = 0..N-1 \)
(2) \( X_n = X_n \ll \text{minex} \quad n = 0..N-1 \)

Pseudo code Equation (1):
\[
s\text{Min} = 32;
\text{for } (n = 0; n < N; n++)
\{
    sZ = \text{exponent}(sX[n]);
    \text{if } (sZ < s\text{Min}) s\text{Min} = sZ; \text{ else } s\text{Min} = s\text{Min};
\}
\]

Pseudo code Equation (2):
\[
\text{for } (n = 0; n < N; n++) sX[n] = sX[n] \ll s\text{Min};
\]

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

LDP= 2 \quad (1 \text{ min}, 1 \text{ count leading sign})
LDP= 1 \quad (\text{read } sX, \text{ write } sX)
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.
<table>
<thead>
<tr>
<th>Name</th>
<th>Cycles</th>
<th>Code Size 1)</th>
<th>Optimization Techniques</th>
<th>Arithmetic Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software Pipelining</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Loop Unrolling</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Packed Operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Load/Store Scheduling</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Memory Interleaving</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Packed Load/Store</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Saturation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rounding</td>
<td></td>
</tr>
<tr>
<td>Dot product</td>
<td>$(2^*N/4)+6$</td>
<td>34</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Magnitude square</td>
<td>$(1^*N/2)+5$</td>
<td>16</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Vector quantization</td>
<td>$(3^*N/2)+6$</td>
<td>38</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>First order FIR</td>
<td>4</td>
<td>14</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Second order FIR</td>
<td>5</td>
<td>22</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FIR</td>
<td>$(2^*N/4 +2) +4$</td>
<td>34</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Block FIR</td>
<td>$(4^*(N/2) +2) +9$</td>
<td>66</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Auto-</td>
<td>$(3+3*(N/2-1)+2)*M/2+2+5$</td>
<td>70</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>correlation</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complex FIR</td>
<td>$(2^*N +2) +4$</td>
<td>32</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>First order IIR</td>
<td>5</td>
<td>24</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2nd order IIR</td>
<td>7</td>
<td>34</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Biquad 4</td>
<td>5</td>
<td>26</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>coefficients</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>N-stage Biquad</td>
<td>$(3^*(N-1) +2) +6$</td>
<td>48</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4 coefficients</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
### N-stage Biquad
5 coefficients

<table>
<thead>
<tr>
<th>Routine</th>
<th>Code Size</th>
<th>Checkmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-stage Biquad 5 coefficients</td>
<td>(5*(N-1) + 2) + 9</td>
<td>74</td>
</tr>
<tr>
<td>Lattice filter</td>
<td>(4*(N-2) + 2) + 10</td>
<td>54</td>
</tr>
<tr>
<td>Leaky LMS (update only)</td>
<td>(4*(N/4-1) + 2) + 9</td>
<td>70</td>
</tr>
<tr>
<td>Delayed LMS</td>
<td>(4*N/4 + 2) + 5</td>
<td>54</td>
</tr>
<tr>
<td>Delayed LMS – 32-bit coefficients</td>
<td>(4*(N/2-1) + 2) + 8</td>
<td>64</td>
</tr>
<tr>
<td>Delayed LMS – complex</td>
<td>(4*(N-1) + 2) + 9</td>
<td>60</td>
</tr>
</tbody>
</table>

| 1) Code Size is in Bytes |
6.1 Dot Product

Equation:
\[ Z = \sum (V_n \times W_n) \quad n = 0..N-1 \]

Pseudo code:
\[
\begin{align*}
sZ &= 0; \\
\text{for } (n = 0; n < N; n++) &\quad sZ += sV[n] \times sW[n];
\end{align*}
\]

Assembly code:
\[
\begin{align*}
&\text{lea } LC, (N/4 - 1) ; (1) ; \text{get loop number} \\
&\text{mov } d0,#0 ; (2) ; Z = 0 \text{ (lower)} \\
&\text{ld.d } ssssV,[Vptr]+8 ; \text{ dummy dummy } V0 \text{ V1} \\
&\text{mov } d1,#0 ; (3) ; Z = 0 \text{ (upper)} \\
&\text{ld.d } ssssW,[Wptr]+8 ; \text{ W0 W1 W2 W3} \\
&\text{dotloop:} \\
&\text{maddms.h } l1Z,l1Z,ssW0,ssV0 ul,#1 ; (1) ; Z +=V0*W0+V1*W1 \\
&\text{ld.d } ssssV,[Vptr]+8 ; \text{ V2 V3 V4 V5} \\
&\text{maddms.h } l1Z,l1Z,ssW1,ssV1 ul,#1 ; (2) ; Z +=V2*W2+V3*W3 \\
&\text{ld.d } ssssW,[Wptr]+8 ; \text{ W4 W5 W6 W7} \\
&\text{loop } LC,\text{dotloop} \\
&\text{st.h } [Zptr],sZ ; (4) ; \text{store } Z
\end{align*}
\]

Example
\[ N = 64 \rightarrow 38 \text{ cycles} \]

Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d3 / d2</th>
<th>d5 / d4</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>z(=0)</td>
<td></td>
<td>v1v0___</td>
<td></td>
<td>ld v0v1</td>
</tr>
<tr>
<td>z(=0)</td>
<td>w3w2w1w0</td>
<td></td>
<td></td>
<td>ld w0w1w2w3</td>
</tr>
<tr>
<td>maddms.h</td>
<td>z+v0w0+v1w1</td>
<td>v5v4v3v2</td>
<td></td>
<td>ld v2v3v4v5</td>
</tr>
<tr>
<td>maddms.h</td>
<td>z+v2w2+v3w3</td>
<td>w7w6w5w4</td>
<td></td>
<td>ld w4w5w6w7</td>
</tr>
</tbody>
</table>
6.2 Magnitude Square

Equation:
\[ Z = \sum (X_{rn}^2 + X_{in}^2) \quad n = 0..N-1 \]

Pseudo code:
\[
\begin{align*}
  & sZ = 0; \\
  & \text{for } (n=0; n<N; n++) \quad sZ += (sXr[n]*sXr[n] + sXi[n]*sXi[n]);
\end{align*}
\]

Assembly code:
\[
\begin{align*}
  & \text{lea } \text{LC, (N/2 - 1)}; \quad (1); \text{get loop number} \\
  & \text{ld.} \text{w ssX, [Xptr+4]}; \quad (2); \text{load Xr, Xi} \\
  & \text{msqloop:} \\
  & \quad \text{maddm.h llZ, llZ, ssX, ssX ul,#1}; \quad (1); \quad Z += (Xr^2 + Xi^2) \\
  & \quad \text{ld.} \text{w ssX, [Xptr+4]}; \quad ||; \text{load next Xr, Xi} \\
  & \quad \text{loop } \text{LC, msqloop}; \\
  & \quad \text{st.h } [\text{Zaddr}, sZ]; \quad (3); \text{store Z}
\end{align*}
\]

Memory organization:

<table>
<thead>
<tr>
<th>Xaddr</th>
<th>Xr0</th>
<th>Xaddr + 2</th>
<th>Xi0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xaddr + 4</td>
<td>Xr1</td>
<td>Xaddr + 6</td>
<td>Xi1</td>
</tr>
<tr>
<td>Xaddr + 8</td>
<td>Xr2</td>
<td>Xaddr + 10</td>
<td>etc..</td>
</tr>
</tbody>
</table>

Example: \[ N = 64 \rightarrow 37 \text{ cycles} \]

Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1/ d0</th>
<th>d2</th>
<th>d5 / d4</th>
<th>d7/ d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>maddm.h \llZ, llZ, ssX, ssX, #1</td>
<td>Z</td>
<td>xi1</td>
<td>xr1</td>
<td></td>
<td>ld xr1 xi1</td>
</tr>
</tbody>
</table>

ld xr0 xi0
6.3 Vector Quantization

Note: VALIDATED ON TC1 V1.3 SILICON.

Equation:
\[ Z = \sum (K_n - X_n)^2 \quad n = 0..N-1 \]

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

Assembly code:
```plaintext
lea LC,(N/2 - 1) ;(1) ; get loop number
mov d0,#0 ;(2) ; Z = 0(lower)
ld.w ssX,[Xptr]+4 ;|| ; sX0 sX1
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) ;
;jZ += ((K0-X0)^2+(K1-X1)^2)
ld.w ssK,[Kptr]+4 ;|| ; K2 K3
loop LC,quantloop
st.h [Zaddr],d1 ;(4) ; store sZ
```

Example

\[ N = 64 \Rightarrow 102 \text{ cycles} \]

Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d3 / d2</th>
<th>d5 / d4</th>
<th>d7 / d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>z = 0(lower)</td>
<td>x1 x0</td>
<td></td>
<td></td>
<td></td>
<td>load x0x1</td>
</tr>
<tr>
<td>z = 0 (upper)</td>
<td></td>
<td>k1 k0</td>
<td></td>
<td></td>
<td>load k0k1</td>
</tr>
<tr>
<td>subs.h ssTp,ssK,ssX</td>
<td>k1-x1</td>
<td></td>
<td>k0-x0</td>
<td>x3 x2</td>
<td>load x2x3</td>
</tr>
<tr>
<td>maddm.s ssTp,ssK,ssX</td>
<td>z += (k0-x0)^2 + (k1-x1)^2</td>
<td>k3 k2</td>
<td>load k2k3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.4 First Order FIR

Note: Validated on TriCore Rider D board.

Equation:

\[ Y_t = K_0 \cdot X_t + K_1 \cdot X_{t-1} \]

Pseudo code:

\[
\begin{align*}
Y &= sK0 \cdot sX0 + sK1 \cdot sX1; \\
X1 &= sX0;
\end{align*}
\]

Assembly code:

\[
\begin{align*}
ld.w & \quad ssK,[Xptr] \quad ;(1) \quad ; K0 \ K1 \\
ld.w & \quad ssX,[Xptr] \quad ;(2) \quad ; X0 \ X1 \\
mulm.h & \quad llY,ssK,ssX ul,#1 \quad ;(3,4) \quad ; Y = K0 \cdot X0 + K1 \cdot X1 \\
st.h & \quad [Xptr]+2,ssX \quad ;|| \quad ; X0 --> X1 \\
st.h & \quad [Yptr],d1 \quad ;|| \quad ; store \ Y
\end{align*}
\]

Memory organization:

<table>
<thead>
<tr>
<th>Entering</th>
<th>Leaving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xaddr</td>
<td>X0</td>
</tr>
<tr>
<td>Xaddr + 2</td>
<td>X0</td>
</tr>
</tbody>
</table>

Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d4</th>
<th>d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k1</td>
<td>k0</td>
<td></td>
<td>ld k1k0</td>
</tr>
<tr>
<td></td>
<td>x1</td>
<td>x0</td>
<td></td>
<td>ld x1x0</td>
</tr>
<tr>
<td>mulm.h</td>
<td>llY,ssK,ssX ul,#1</td>
<td>y = k0\cdot x + k1\cdot x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st y</td>
</tr>
</tbody>
</table>
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:
\[
\text{sY} = \text{sK0}\times\text{sX0} + \text{sK1}\times\text{sX1} + \text{sK2}\times\text{sX2} ;
\]
\[
\text{sX2} = \text{sX1};
\]
\[
\text{sX1} = \text{sX0};
\]

<table>
<thead>
<tr>
<th>IP= 2 (1 mul, 2 madd)</th>
<th>LD/ST= 9 (read sX0, sX1, sX2, sK0, sK1, sK2, write sY, write sX0 \rightarrow sX1, write sX1 \rightarrow sX2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld.d sssK,[Kptr]</td>
<td>; (1) ; K0 K1 K2</td>
</tr>
<tr>
<td>ld.d sssX,[Xptr]</td>
<td>; (2) ; X X1 X2</td>
</tr>
<tr>
<td>mulm.h llY,d4,d6 ul,#1</td>
<td>; (3) ; Y = K0<em>X + K1</em>X1</td>
</tr>
<tr>
<td>st.w [Xptr]+2,d6</td>
<td>; ; X0 --&gt; X1 X1 --&gt; X2</td>
</tr>
<tr>
<td>madd.q llY,llY,d7l,d7l,#1</td>
<td>; (4,5) ; Y = Y + K2*X2</td>
</tr>
<tr>
<td>st.h [Yptr],d1</td>
<td>; ; store Y</td>
</tr>
</tbody>
</table>

Assembly code:

Entering Leaving

<table>
<thead>
<tr>
<th>Xaddr</th>
<th>X0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Xaddr + 2</td>
<td>X1</td>
<td>X0</td>
</tr>
<tr>
<td>Xaddr + 4</td>
<td>X2</td>
<td>X1</td>
</tr>
</tbody>
</table>
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d5 / d4</th>
<th>d7 / d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>__ k2 k1 k0</td>
<td></td>
<td></td>
<td>___ k2 k1 k0</td>
<td>ld k0 k1 k2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_ x2 x1 x0</td>
<td>ld x0 x1 x2</td>
</tr>
<tr>
<td>mulm.h llY,d4,d6ul,#1</td>
<td>y = k0<em>x0 + k1</em>x1</td>
<td></td>
<td></td>
<td>st x0 x1</td>
</tr>
<tr>
<td>madd.q llY,llY,d5l,d7l,#1</td>
<td>y += k2*x2</td>
<td></td>
<td></td>
<td>st y</td>
</tr>
</tbody>
</table>
6.6 FIR

Equation:
\[ Y_t = \sum (X_{t-n} \times K_n) \quad n = 0..N-1 \]

Pseudo code:
\[
sY = 0;
\text{for } (n=0; n<N; n++) \quad sY += \text{circular}(sX[n]) \times sK[n];
\]

Assembly code:
\[
\text{mov.aa a2,Xptr} \quad ; \text{circular buffer initialization}
\text{CONST.A a3,}(2\times N)<<(16)
\text{lea } LC, (N/4 -1) \quad ; (1) \quad ; \text{get loop number}
\text{mov } d0,#0 \quad ; (2) \quad ; Y = 0 \text{ (lower)}
\text{ld.w ssK0,}[Kptr+]4 \quad ; || \quad ; K0 K1
\text{mov } d1,#0 \quad ; (3) \quad ; Y = 0 \text{ (upper)}
\text{ld.d ssssX,}[a2/a3+c]8 \quad ; || \quad ; X0 X1 X2 X3
\text{sfirloop:}
\text{maddm.h } llY, llY, ssX0, ssK0 ul,#1 \quad ; (1) \quad ; Y += X0 \times K0 + X1 \times K1
\text{ld.d ssssK,}[Kptr+]8 \quad ; || \quad ; K2 K3 K4 K5
\text{maddm.h } llY, llY, ssX1, ssK1 ul,#1 \quad ; (2) \quad ; Y += X2 \times K2 + X3 \times K3
\text{ld.d ssssX,}[a2/a3+c]8 \quad ; || \quad ; X4 X5 X6 X7
\text{loop } LC, sfirloop
\text{st.h } [Yptr],d1 \quad ; (4) \quad ; \text{store } Y
\]

Example: \[ N = 20 \rightarrow 16 \text{ cycles} \]
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d3 / d2</th>
<th>d5 / d4</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>k1k0 _ _</td>
<td>ld k0k1</td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td>x3x2 x1x0</td>
<td></td>
<td>ld x0x1x2x3</td>
</tr>
<tr>
<td>Y, Y, X0, X0 ul, #1</td>
<td>y += x0 * k0 + x1 * k1</td>
<td></td>
<td>k5k4 k3k2</td>
<td>ld k2k3k4k5</td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td>x7x6 x5x4</td>
<td></td>
<td>ld x4x5x6x7</td>
</tr>
<tr>
<td>Y, Y, X1, X1 ul, #1</td>
<td>y += x2 * k2 + x3 * k3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.7 Block FIR

Note: Validated on TriCore Rider D board.

Equation:
\[ Y_m = \sum_{n=0}^{N-1} X_{m-n} \cdot K_n \quad n = 0..N-1, m = 0..M-1 \]

Pseudo code:
for \( m = 0; m < M/4; m+4 \) 
{ 
    \( sY[m] = 0; \)
    for \( n = 0; n < N; n++ \) 
    \( sY[m] += sX[m+n] \cdot sK[n]; \)
    \( sY[m+1] += sX[m+n+1] \cdot sK[n]; \)
    \( sY[m+2] += sX[m+n+2] \cdot sK[n]; \)
    \( sY[m+3] += sX[m+n+3] \cdot sK[n]; \)
}

Assembly code:
```
mov.a a2,Xptr ;circular buffer address initialization
CONST.A a3,(2*N)<<16
mov.a a14,Xptr ;circular buffer address initialization
CONST.A a15,(2*N)<<16
lea LC,(N/2 - 1) ; (1) ; get loop number
mul 11Y0,d13,#0 ; (2) ; y0 = 0
mul 11Y2,d13,#0 ; (3) ; y2 = 0
ld.q d13,[a14/a15+c]2 ; || ; dummy load
mul 11Y1,d13,#0 ; (4) ; y1 = 0
ld.w ssK,[Kptr+]4 ; || ; k1k0
mul 11Y3,d13,#0 ; (5) ; y3 = 0
ld.d sssX,[a2/a3+c]4 ; || ; x3x2x1x0
blkfir:
madm.h 11Y0,11Y1,0d,ssK ul,#1 ; (1) ; y0+=x0*k0+x1*k1
madm.h 11Y2,11Y3,0d,ssK ul,#1 ; (2) ; y2+=x2*k0+x3*k1
ld.d sssX1,[a14/a15+c]4 ; || ; x4x3x2x1
madm.h 11Y1,11Y2,0d,ssK ul,#1 ; (3) ; y1+=x1*k0+x2*k1
ld.d sssX,[a2/a3+c]4 ; || ; x5x4x3x2
madm.h 11Y3,11Y1,0d,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
```

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

N = 20 → 51 cycles

**Register diagram:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1/d0</th>
<th>d3/d2</th>
<th>d5/d4</th>
<th>d7/d6</th>
<th>d9/d8</th>
<th>d11/d10</th>
<th>d12</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>y0 = 0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y2 = 0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y1 = 0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k1k0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>ld k1k0</td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lY0,lY0,d8,ssK ul,#1</td>
<td>y0+= x0<em>k0+ x1</em>k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lY2,lY2,d9,ssK ul,#1</td>
<td>y2+= x2<em>k0 +x3</em>k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lY1,lY1,d10,ssK ul,#1</td>
<td>y1+= x1<em>k0 +x2</em>k1</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>x4x3x2x1</td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lY3,lY3,d11,ssK ul,#1</td>
<td>y3+= x3<em>k0 +x4</em>k1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>x5x4x3x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

$N = 20 \rightarrow 51$ cycles
6.8 Auto-Correlation

Note: Validated on TriCore Rider D board.

Equation:

\[ Z_m = \sum_{n=0}^{N-1} X_n \times X_{n+m} \quad n = 0..N-1 \quad m = 0..M-1; \]

Pseudo code:

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

Assembly code:

```assembly
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,[Xptr+4] ; || ; x3 x2
    maddm.h llZe,llZe,ssX1,ssX3 ul,#1 ; (5) ; 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,[Xoptr+4] ; || ; x3 x2
    ld.w ssX3,[Xoptr+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
```

[IP= 1 (1 madd)  LD/ST= 3 (read sX, s Xn+m, write lZ) ]
maddm.h llZo, llZo, ssX1, ssX2 ul,#1
   ; (6) ;
st.w  [2ptr+], d1 ; || ; store z_even
add.a a9, #4 ; (7) ; adjust pointer position
st.w  [2ptr+], 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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d3 / d2</th>
<th>d5</th>
<th>d4</th>
<th>d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>z_even = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(lower)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z_even = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(upper)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z_odd = 0</td>
<td></td>
<td>x1x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(lower)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z_odd = 0</td>
<td></td>
<td>x1x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(upper)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h llZe, llZe, X1, X3 ul,#1</td>
<td>z_even += x0<em>x0+x1</em>x1</td>
<td>x2x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h llZo, llZo, X1, X2 ul,#1</td>
<td>z_odd += x0<em>x1+x1</em>x2</td>
<td>x3x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h llZe, llZe, X1, X3 ul,#1</td>
<td>z_even += x2<em>x2+x3</em>x3</td>
<td>x4x3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h llZo, llZo, X1, X2 ul,#1</td>
<td>z_odd += x0<em>x1+x1</em>x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

User Guide 89 v1.6.4, 2003-01
6.9 Complex FIR

Note: Validated on TriCore Rider D board.

Equations:
\[ Y_{rt} = \sum (X_{rt-n}K_{rn} - X_{it-n}K_{in}) \quad n = 0..N-1 \]
\[ Y_{it} = \sum (X_{it-n}K_{rn} + X_{rt-n}K_{in}) \quad n = 0..N-1 \]

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

Pseudo code implementation:
```pseudo
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:

<table>
<thead>
<tr>
<th>Xaddr</th>
<th>sXr0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xaddr + 2</td>
<td>sXi0</td>
</tr>
<tr>
<td>Xaddr + 4</td>
<td>sXr1</td>
</tr>
<tr>
<td>Xaddr + 6</td>
<td>sXi1</td>
</tr>
<tr>
<td>Xaddr + 8</td>
<td>sXr2</td>
</tr>
<tr>
<td>Xaddr + 10</td>
<td>etc..</td>
</tr>
<tr>
<td>Kaddr</td>
<td>sKi0</td>
</tr>
<tr>
<td>Kaddr + 2</td>
<td>sKr0</td>
</tr>
<tr>
<td>Kaddr + 4</td>
<td>sKi1</td>
</tr>
<tr>
<td>Kaddr + 6</td>
<td>sKr1</td>
</tr>
<tr>
<td>Kaddr + 8</td>
<td>sKi2</td>
</tr>
<tr>
<td>Kaddr + 10</td>
<td>etc..</td>
</tr>
</tbody>
</table>

Example: \( N = 20 \rightarrow 46 \text{ cycles} \)

Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d4</th>
<th>d5</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y = 0</td>
<td>xi0</td>
<td>xr0</td>
<td>ld xi0xr0</td>
</tr>
<tr>
<td>maddsrsh</td>
<td>yi+=xi\times ki</td>
<td>ki0</td>
<td></td>
<td>ld ki0</td>
</tr>
<tr>
<td>ssY,ssY,ssX,ssK uu,#1</td>
<td>yr=xi\times ki</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td>yi+=xi\times kr</td>
<td></td>
<td></td>
<td>ld xi1xr1</td>
</tr>
<tr>
<td>ssY,ssY,ssX,ssK ll,#1</td>
<td>yr=xi\times kr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.10 First Order IIR

Note: Validated on TriCore Rider D board.

Equation:
\[ Y_t = B \cdot Y_{t-1} + K \cdot X_t \]

where:
\[ B = (1 - K) \]

Pseudo code:
\[
\begin{align*}
\text{sY0} &= \text{sY1} - \text{sY1} \cdot \text{sK} + \text{sX0} \cdot \text{sK}; \\
\text{sY1} &= \text{sY0};
\end{align*}
\]

<table>
<thead>
<tr>
<th>IP</th>
<th>LD/ST</th>
<th>(read sX, sY1, sK, write sY0 (\rightarrow) sY1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>(1 madd, 1 msub)</td>
</tr>
</tbody>
</table>

Assembly code:
\[
\begin{align*}
\text{ld.q sK, [Kptr] ; (1) ; K} \\
\text{ld.q sX, [Xptr+\text{2}] ; (2) ; X0} \\
\text{ld.q sY, [Yptr] ; (3) ; Y1} \\
\text{msubs.q sY, sY, sK u, sY u, \#1 ; (4) ; Y0 = Y1-Y1*K} \\
\text{madds.q sY, sY, sK u, sX u, \#1 ; (5, 6) ; Y0 += X0*K} \\
\text{st.q [Yptr], sY} ; \text{|| ; store Y0}
\end{align*}
\]

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

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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>D3</th>
<th>d5</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld k</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld x0 y1</td>
</tr>
<tr>
<td>msubrs.h</td>
<td></td>
<td></td>
<td></td>
<td>y0 = y1-y1*k</td>
</tr>
<tr>
<td>sY,ssXY,ssXY,sK uu,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td></td>
<td></td>
<td></td>
<td>y0 += x0*k</td>
</tr>
<tr>
<td>sY,ssXY,sK uu,#1</td>
<td></td>
<td></td>
<td></td>
<td>st y</td>
</tr>
</tbody>
</table>
6.11 Second Order IIR

*Note: Validated on TriCore Rider D board.*

**Equation:**

\[ Y_t = K_0 X_t + K_1 X_{t-1} + K_2 X_{t-2} + B_1 Y_{t-1} + B_2 Y_{t-2} \]

where:

B1, B2 are negative

**Pseudo code:**

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

**Assembly code:**

```plaintext
ld.d ssssK,[Kptr]  ; (1)  ; K2 K1 K0
ld.w ssB,[Bptr]    ; (2)  ; B1 B0
ld.d ssssX,[Xptr+]2; (3)  ; X3 X2 X1
mulm.h e0,d2,d6 ul,#1; (4)  ; Y0 = K1*X1 + K0*X0
ld.w sssX,[Xptr+]2; || ; X3 X2 X1
madds.q e0,e0,ssX l,ssX l,#1; (5)  ; Y0 += X2*X2
maddm.h e0,e0,ssB,ssY,ssY ul,#1; (6,7)  ; Y0 += B1*Y1 + B2*Y2
st.h [a2/a3+c]0,d1  ; ||  ; store Y0
```

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

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

\[
y_2 = k_1 \times x_1 + k_0 \times x_0
\]

\[
y_2 = y + x \times k_2
\]

\[
y_2 = y_2 + b_1 \times y_1 + b_2 \times y_2
\]
6.12 BIQUAD 4 Coefficients

Note: Validated on TriCore Rider D board.

Equations:
\[ W_t = X_t + B_1 W_{t-1} + B_2 W_{t-2} \]
\[ Y_t = W_t + K_1 W_{t-1} + K_2 W_{t-2}; \]

Pseudo code:
\[
\begin{align*}
    sW0 &= sX0 + sB1*sW1 + sB2*sW2; \\
    sY0 &= sW0 + sK1*sW1 + sK2*sW2; \\
    sW2 &= sW1; \\
    sW1 &= sW0;
\end{align*}
\]

\begin{tabular}{|l|l|}
\hline
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) \\
\hline
\end{tabular}

Assembly code:
\[
\begin{align*}
    \text{ld.h} & \quad d0, [Xptr] \quad ; \quad ;x0-->d0 \\
    \text{mov} & \quad d1, #0 \quad ; (1) \quad ; y0 = 0 \\
    \text{ld.w} & \quad ssB, [BKptr+]4 \quad ; || \quad ; b2 b1 \\
    \text{ld.w} & \quad ssW, [a14/a15+c]2 \quad ; || \quad ; w2 w1 \\
    \text{maddm.h} & \quad llW, llY, ssW, ssB ul,#1 \quad ; (3) \quad ; w0=x0+w1*b1+w2*b2 \\
    \text{ld.w} & \quad ssK, [BKptr+]4 \quad ; || \quad ; k2 k1 \\
    \text{maddm.h} & \quad llY, llW, ssW, ssK ul,#1 \quad ; (4,5) \quad ; y0=w0+w1*k1+w2*k2 \\
    \text{st.h} & \quad [a14/a15+c]0,d7 \quad ; || \quad ; w0--> w1 \\
    \text{st.h} & \quad [Yptr], d1 \quad ; || \quad ; store y0
\end{align*}
\]
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d2</th>
<th>d4</th>
<th>d5</th>
<th>d7 / d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 0 ) (upper)</td>
<td></td>
<td>b2b1</td>
<td></td>
<td></td>
<td></td>
<td>ld b2b1</td>
</tr>
<tr>
<td>( x = 0 ) (lower)</td>
<td></td>
<td>w2w1</td>
<td></td>
<td></td>
<td></td>
<td>ld w2w1</td>
</tr>
<tr>
<td>\texttt{maddm.h} \texttt{IlW,IlY,ssW,ssB ul,#1}</td>
<td></td>
<td></td>
<td></td>
<td>k2k1</td>
<td></td>
<td>ld k2k1</td>
</tr>
<tr>
<td>( y_0 = w_0 + w_1 * b_1 + w_2 * b_2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>w1w0</td>
<td>st w0</td>
</tr>
<tr>
<td>\texttt{maddm.h} \texttt{IlY,IlW,ssW,ssK ul,#1}</td>
<td>y0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st y0</td>
</tr>
</tbody>
</table>
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:

```c
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;
}
```

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:

```assembly
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 ; (5) ; y0 = w0+w1*k1+w2*k2
    st.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+c]2, d1 ; || ; store y0
```

Example

| 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.

Example

\( N = 13 \Rightarrow 44 \text{ cycles} \)
Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d2</th>
<th>d4</th>
<th>d5</th>
<th>d7 / d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0 (upper)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld b2b1</td>
</tr>
<tr>
<td>x = 0 (lower)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld w2w1</td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld k2k1</td>
</tr>
<tr>
<td>lIy, lIw, ssW, ssB ul,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lIy, lIw, ssW, ssK ul,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y0 =</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w0 + w1 * k1 + w2 * k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st w0</td>
</tr>
<tr>
<td>w1w0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld w1w0</td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lIy, lIw, ssW, ssB ul,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lIy, lIw, ssW, ssK ul,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y0 =</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w0 + w1 * k1 + w2 * k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st w0</td>
</tr>
<tr>
<td>y0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st y0</td>
</tr>
</tbody>
</table>
### 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;
}
```

**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         ; y0 = 0
ld.w ssB, [BKptr + 14] ; || ; b2 b1
mov d0, #0 ; 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 ; extract w0
lea LC, (N - 2) ; 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
```

| IP= 5 (5 madd) | LD/ST= 9 (read sW1, sW2, read sK0, sK1, sK2, read sB1, write sW0 --> sW1, write sW1 --> sW2) |
**Example**  
N = 13  \( \rightarrow \) 71 cycles

### Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>D2</th>
<th>d3</th>
<th>d5 / d4</th>
<th>d7 / d6</th>
<th>d8</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = 0 (lower)</td>
<td>b2b1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld b2b1</td>
</tr>
<tr>
<td>y = 0 (upper)</td>
<td>W2w1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld w2w1</td>
</tr>
<tr>
<td>maddm.h lIw,lIY,ssW,ssB ul,#1</td>
<td>k2k1k0</td>
<td>w0=y0+w1* b1+w2*b2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld k2k1k0</td>
</tr>
<tr>
<td>dextr d8,d7,d6,#16</td>
<td>w0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>muls.m lIY,ssW,ssK2 ul,#1</td>
<td>y0=w1<em>k1+ w2</em>k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madds.q lIY,lIY,d8u,ssK u,#1</td>
<td>w0=k0*w0</td>
<td>w0</td>
<td>st w0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddm.h lIw,lIY,ssW,ssB ul,#1</td>
<td>w0=y0+w1* b1+w2*b2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dextr d8,d7,d6,#16</td>
<td>w0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>muls.m lIY,ssW,ssK2 ul,#1</td>
<td>y0=w1<em>k1+ w2</em>k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madds.q lIY,lIY,d8u,ssK u,#1</td>
<td>w0=k0*w0</td>
<td>w0</td>
<td>st w0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st y0</td>
</tr>
</tbody>
</table>
6.15 Lattice Filter

Equations:
\[ Z_n = Z_{n-1} - X_n \cdot K_{n-1} \quad n = 1..N-1 \]
\[ T_{n-1} = X_n + Z_n \cdot K_{n-1} \]

Pseudo code:
for \( n = 1; n < N; n++ \) {
    \[ sZ[n] = sZ[n-1] - sX[n] \cdot sK[n-1]; \]
    \[ sT[n-1] = sX[n] + sZ[n] \cdot sK[n-1]; \]
}

Example \( N = 10 \)  
\[ 44 \text{ cycles} \]

Assembly code:
lea LC, (N - 3) ; (1) ; get loop number
ld.q sX, [Xptr+12] ; (2) ; x9
ld.q sK, [Kptr+12] ; (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+12] ; || ; x8
ld.q sK, [Kptr+12] ; || ; 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+12] ; || ; x7
ld.q sK, [Kptr+12] ; || ; 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)
### Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d1</th>
<th>d4</th>
<th>d5</th>
<th>d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x9</td>
<td></td>
<td></td>
<td></td>
<td>ld x9</td>
</tr>
<tr>
<td></td>
<td>k10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld k10</td>
</tr>
<tr>
<td>msubs.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>id x8</td>
</tr>
<tr>
<td>sZ,sY,sX,sK ul,#1</td>
<td>z9 = y - x9*k10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>id x9</td>
</tr>
<tr>
<td>msubs.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>id x9</td>
</tr>
<tr>
<td>sZ,sZ,sX,sK ul,#1</td>
<td>z8 = z9 - x8*k9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>id x8</td>
</tr>
<tr>
<td>sT,sX,sZ,sK ul,#1</td>
<td>t9 = x8 + z8*k9</td>
<td>x8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>id k9</td>
</tr>
<tr>
<td>msubs.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st t10</td>
</tr>
<tr>
<td>sZ,sZ,sX,sK ul,#1</td>
<td>z7 = z8 - x7*k8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st t0</td>
</tr>
<tr>
<td>sT,sX,sZ,sK ul,#1</td>
<td>t1 = x0 + z0*k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st t1</td>
</tr>
</tbody>
</table>
6.16 Leaky LMS (Update Only)

Note: Validated on TriCore Rider D board.

Equation:
\[
K_{t,n} = K_{t-1,n} * B + X_{t-n} * u * \text{Err}_{t-1} \quad n = 0..N-1
\]

Pseudo code:
for \( n = 0 ; n < N ; n++ \) \( sK[n] = sK[n] * sB + \text{circular}(sX[n]) * sErr ; \)

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

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

| N = 20 | 27 cycles |

Register representation:

<table>
<thead>
<tr>
<th>X input</th>
<th>sK1</th>
<th>sK0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant input</td>
<td>sB</td>
<td></td>
</tr>
<tr>
<td>Result of the instruction</td>
<td>sK1*sB</td>
<td>sK0*sB</td>
</tr>
</tbody>
</table>

Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d1</th>
<th>d3</th>
<th>d5</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k1k0</td>
<td></td>
<td></td>
<td></td>
<td>Id k0k1</td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td>x3</td>
<td>x2</td>
<td></td>
</tr>
<tr>
<td>ssKK0,ssK0,ssB</td>
<td></td>
<td>k1= k1*B</td>
<td></td>
<td>k0= k0*B</td>
<td>x0x1x2x3</td>
</tr>
<tr>
<td>maddrs.h</td>
<td>k5</td>
<td>k4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssKK0,ssK0,ssX0, sErr</td>
<td>k1+=x1*Err</td>
<td></td>
<td>k0 +=x0*Err</td>
<td>k2k3k4k5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x3</td>
<td>x2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x1</td>
<td>x0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x7</td>
<td>x6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x5</td>
<td>x4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st k0k1</td>
</tr>
<tr>
<td>ssKK1,ssK1,ssB</td>
<td></td>
<td>k3= k3*B</td>
<td></td>
<td>k2= k2*B</td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td>k5</td>
<td>k4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssKK1,ssKK1,ssX1, sErr</td>
<td>k3+=x3*Err</td>
<td></td>
<td>k2+=x2*Err</td>
<td>k4x5x6x7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x3</td>
<td>x2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x1</td>
<td>x0</td>
<td></td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssKK0,ssK0,ssB</td>
<td></td>
<td>k5= k5*B</td>
<td></td>
<td>k4= k4*B</td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td>k5</td>
<td>k4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssKK0,ssK0,ssX0, sErr</td>
<td>k3+=x3*Err</td>
<td></td>
<td>k2+=x2*Err</td>
<td>k4x5x6x7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssKK1,ssK1,ssB</td>
<td></td>
<td>k3= k3*B</td>
<td></td>
<td>k2= k2*B</td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssKK1,ssKK1,ssX1, sErr</td>
<td>k3+=x3*Err</td>
<td></td>
<td>k2+=x2*Err</td>
<td>k4x5x6x7</td>
<td></td>
</tr>
</tbody>
</table>
6.17 Delayed LMS

**Note:** Validated on TriCore Rider D board.

**Equations:**

\[ Y_t = \sum X_{t-n}K_{t-1,n} \quad n = 0..N-1 \]

\[ K_{t,n} = K_{t-1,n} + X_{t-n}u_{err,t-1} \quad n = 0..N-1 \]

**Pseudo code:**

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

**Assembly code:**

```plaintext
lea    LC,(N/4 - 1) ; (1) ; get loop number
ld.h   sErr,[Raddr] ; (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   sssX,[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)
    ld.d   sssX,[a14/a15+c]8 ; || ; X4 X5 X6 X7
    maddrs.h d7,ssK1,ssX1,sErr ll,#1 ; (3) ; Y += X2*K2+X3*K3
    maddm.h llY,llY,ssX1,ssK1 ul,#1 ; (4)
    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**

| Example | N = 20 | 27 cycles |
Register representation:

<table>
<thead>
<tr>
<th>X input</th>
<th>sX1</th>
<th>sX0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error input</td>
<td>sErr</td>
<td></td>
</tr>
<tr>
<td>Result of the instruction</td>
<td>sX1*sErr</td>
<td>sX0*sErr</td>
</tr>
</tbody>
</table>

Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d3 / d2</th>
<th>d5 / d4</th>
<th>d7 / d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = 0 (d1)</td>
<td>y = 0 (d0)</td>
<td>x3x2</td>
<td>k1k0</td>
<td>k3k2</td>
<td>x0x1x2x3</td>
</tr>
<tr>
<td>maddm.h</td>
<td>y += x0<em>k0 + x1</em>k1</td>
<td>k3k2</td>
<td>k1k0</td>
<td>--------</td>
<td>dummy store</td>
</tr>
<tr>
<td>maddrs.h</td>
<td>d6,ssK0,ssX0,d8ul,#1</td>
<td>k5k4</td>
<td>k3k2</td>
<td>k1 += x1*err</td>
<td></td>
</tr>
<tr>
<td>maddm.h</td>
<td>y += x2<em>k2 + x3</em>k3</td>
<td>x7x6</td>
<td>x5x4</td>
<td>--------</td>
<td>ld k2k3k4k5</td>
</tr>
<tr>
<td>maddrs.h</td>
<td>d7,ssK1,ssX1,d8ul,#1</td>
<td>k3 += x3*err</td>
<td></td>
<td>k2 += x2*err</td>
<td></td>
</tr>
<tr>
<td>next iteration</td>
<td>st</td>
<td>k0k1k2k3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.18 Delayed LMS – 32-bit Coefficients

Note: Validated on TriCore Rider D board.

Equations:

\[ Y_t = \sum X_{t-n} K_{t-1,n} \quad n = 0..N-1 \]

\[ K_{t,n} = K_{t-1,n} + X_{t-n} u^* err_{t-1} \quad n = 0..N-1 \]

Pseudo code:

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

Assembly code:

```assembly
lea LC, (N/2 - 2) ; (1) get loop number
ld.h sErr, [Raddr] ; (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, l1K, 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, l1K, 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 |

IP = 2 (2 madd) LD/ST= 3 (read sX, read lK, write lK)
## Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d1 / d0</th>
<th>d3 / d2</th>
<th>d5 / d4</th>
<th>d7 / d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = 0</td>
<td></td>
<td></td>
<td>k1k0</td>
<td></td>
<td>ld k1k0</td>
</tr>
<tr>
<td>(lower)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y = 0</td>
<td></td>
<td>x1x0</td>
<td></td>
<td></td>
<td>ld x1x0</td>
</tr>
<tr>
<td>(upper)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madds.h l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IY,IY,ssX,IK0 ul,#1</td>
<td>y += x0*k0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madds.h l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IY,IY,ssX,IK1 uu,#1</td>
<td>y += x1*k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madds.h l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e6,IIK,ssX,sErr ll,#1</td>
<td>y += x0*k0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madds.h l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IY,IY,ssX,IK0 ul,#1</td>
<td>y += x2*k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>k5k4</td>
<td></td>
<td>st k1k0</td>
</tr>
<tr>
<td>madds.h l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IIY,IIY,ssX,IK1 uu,#1</td>
<td>y = y+x_n*k_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madds.h l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| e6,IIK,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:
\[ Y_{rt} = \sum X_{rt,n} \cdot K_{rt-1,n} - X_{it,n} \cdot K_{lt-1,n} \]
\[ Y_{it} = \sum X_{rt,n} \cdot K_{lt-1,n} + X_{it,n} \cdot K_{rt-1,n} \]
\[ K_{rt,n} = K_{rt-1,n} + X_{rt,n} \cdot u \cdot E_{rt-1} - X_{it,n} \cdot u \cdot E_{it-1} \]
\[ K_{lt,n} = K_{lt-1,n} + X_{rt,n} \cdot u \cdot E_{it-1} + X_{it,n} \cdot u \cdot E_{rt-1} \quad n = 0..N-1 \]

Pseudo code:
```cpp
sYr = 0;
sYi = 0;
for (n = 0; n<N; n++)
{
  sYr += circular(sXr[n] * sKr[n]) - circular(sXi[n] * sKi[n]);
  sYi += circular(sXr[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, 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] ; || ; kilkr1
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, 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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d1</th>
<th>d4</th>
<th>d5</th>
<th>d6</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>yi = 0</td>
<td></td>
<td>yr = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xixr0</td>
<td></td>
<td></td>
<td></td>
<td>Id xixr0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Id k0kr0</td>
</tr>
<tr>
<td>maddsurs.h</td>
<td>yi+=xr0*ki0</td>
<td></td>
<td>yr-=xi0*ki0</td>
<td>ki0kr0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssY,ssY,ssX,ssK uu,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsurs.h</td>
<td>yi+=xi0*kr0</td>
<td></td>
<td>yr+=xr0*kr0</td>
<td>ki1kr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssK1,ssK,ssX,ssE uu,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td>yi+=xi0*kr1</td>
<td></td>
<td>yr+=xr1*kr1</td>
<td>xixr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssY,ssY,ssX,ssK ll,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td>ki0+=xr0*ei</td>
<td></td>
<td>kr0+=xi0*ei</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssK1,ssK,ssX,ssE ll,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsurs.h</td>
<td>yi+=xr1*ki1</td>
<td></td>
<td>yr-=xi1*ki1</td>
<td>ki0kr0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssY,ssY,ssX,ssK uu,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsurs.h</td>
<td>ki1+=xr1*ei</td>
<td></td>
<td>kr1+=xi1*ei</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssK1,ssK,ssX,ssE uu,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td>yi+=xin*krn</td>
<td></td>
<td>yr+=xrn*krn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssY,ssY,ssX,ssK ll,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td>kin+=xin*er</td>
<td></td>
<td>krn+=xrn*er</td>
<td>yryi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssK1,ssK,ssX,ssE ll,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddrs.h</td>
<td>kinkrn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssK1,ssK,ssX,ssE ll,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)
<table>
<thead>
<tr>
<th>Name</th>
<th>Cycles</th>
<th>Code Size 1)</th>
<th>Optimization Techniques</th>
<th>Arithmetic Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software Pipelining</td>
<td></td>
</tr>
<tr>
<td>Real butterfly – DIT2) radix 2</td>
<td>$(5*N/2 +2) +3$</td>
<td>32</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Loop Unrolling</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Packed Operation</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Load/Store Scheduling</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Memory Interleaving</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Packed Load/Store</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Saturation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rounding</td>
<td></td>
</tr>
<tr>
<td>Real butterfly – DIF3) radix 2</td>
<td>$(5*N/2 +2) +3$</td>
<td>36</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Complex butterfly – DIT radix 2</td>
<td>$(9*N/2 +2) +3$</td>
<td>74</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Complex butterfly – DIF radix 2</td>
<td>$(11*N/2 +2) +3$</td>
<td>82</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>– with shift</td>
<td></td>
<td></td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Complex butterfly – DIF radix 2</td>
<td>$(9*N/2 +2) +3$</td>
<td>76</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

1) Code size is in Bytes
2) DIT = Decimation In Time
3) DIF = Decimation In Frequency
7.1 Real Butterfly – DIT – Radix 2

Equations:
\[ X_n' = X_n + Y_n \cdot K_n \]
\[ Y_n' = X_n - Y_n \cdot K_n \]
\( n = 0..N-1 \)

Pseudo code:
```c
for (n = 0; n<N; n++) {
    sXX[n] = sX[n] + sY[n] \cdot sK[n];
    sYY[n] = sX[n] - sY[n] \cdot sK[n];
}
```

Pseudo code implementation:
```c
ssP1  = ssY1 * ssK1;  ssP0  = ssY0 * ssK0;
ssXX1 = ssX1 + ssP1;  ssXX0 = ssX0 + ssP0;
ssYY1 = ssX1 - ssP1;  ssYY0 = ssX0 - ssP0;
```

Assembly code:
```assembly
lea LC,(N/2 – 1)          ; (1) ;get loop number
ld.w ssY,[Yptr+]4         ; (2) ;y1y0
ld.w ssK,[Xptr+]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,[Xptr+]4        ; (5) ;k3 k2
```

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

### Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d6</th>
<th>d8</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>y1</td>
<td>y0</td>
<td></td>
<td></td>
<td>ld y1y0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k1</td>
<td>k0</td>
<td>ld k1k0</td>
</tr>
<tr>
<td>mulr.h ssP.ssY.ssKul,#1</td>
<td></td>
<td></td>
<td>p1=y1*k1</td>
<td></td>
<td>p0=y0*k0</td>
<td>x1</td>
<td>x0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y3</td>
<td>y2</td>
<td></td>
<td></td>
<td>ld y3y2</td>
</tr>
<tr>
<td>add.h ssXX.ssX.ssP</td>
<td></td>
<td></td>
<td>x'1=x1+p1</td>
<td></td>
<td>x'0=x0+p0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub.h ssYY.ssX.ssP</td>
<td></td>
<td></td>
<td>y'1=x1-p1</td>
<td></td>
<td>y'0=x0-p0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k3</td>
<td>k2</td>
<td>ld k3k2</td>
</tr>
</tbody>
</table>
7.2 Real Butterfly – DIF – Radix 2

Equations:
\[
X_n' = X_n + Y_n \\
Y_n' = X_n*K_n - Y_n*K_n \\
\text{n=0..N-1}
\]

Pseudo code:

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

Pseudo code implementation (loop only):

```assembly
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 ssX,[Xptr+4] ; (3) ; k1 k0
    mulr.h ssYY,ssD,ssK u1,#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
```

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d2</th>
<th>d3 / d2</th>
<th>d4</th>
<th>d6</th>
<th>d8</th>
<th>Load / Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y1 y0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x1 x0</td>
</tr>
<tr>
<td>add.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st xx1 xx0</td>
</tr>
<tr>
<td>ssXX,ssX, ssY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld y1y0</td>
</tr>
<tr>
<td>sub.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld y3y2</td>
</tr>
<tr>
<td>ssD,ssX,ssY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k1 k0</td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld x1x0</td>
</tr>
<tr>
<td>ssYY,ssD,ssK ul,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ld x3x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>st yy1 yy0</td>
</tr>
</tbody>
</table>
7.3 Complex Butterfly – DIT – Radix 2

Equations:
\[ X'_r = X_r + (Y_r^*K_r - Y_i^*K_i) \]
\[ X'_i = X_i + (Y_r^*K_r + Y_i^*K_i) \]
\[ Y'_r = X_r - (Y_r^*K_r - Y_i^*K_i) \]
\[ Y'_i = X_i - (Y_r^*K_r + Y_i^*K_i) \]

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];
}

Pseudo code implementation (loop only):
sP10 = sYr0 * sKi0 ; sPr0 = sYr0 * sKr0;
sP10 = sP10 + sYi0 * sKr0 ; sPr0 = sPr0 - sYi0 * sKi0;
sP11 = sYr1 * sKi1 ; sPr1 = sYr1 * sKr1;
sP11 = sP11 + sYi1 * sKr1 ; sPr1 = sPr1 - sYi1 * sKi1;
sXXi0 = sXr0 + sP10 ; sXXr0 = sXr0 + sPr0;
sYYi0 = sXi0 - sP10 ; sYYr0 = sXr0 - sPr0;
sXXi1 = sXr1 + sP11 ; sXXr1 = sXr1 + sPr1;
sYYi1 = sXr1 - sP11 ; sYYr1 = sXr1 - sPr1;

IP=8 (4 madd, 4 msub)   LD/ST= 10 (read sXr, sXi, sYr, sYi, read sKr, sKi, write sXXr, sXXi, sYYr, sYYi)
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) ;yly0
ld.w  ssK1,[Kptr] ; (4) ;kikr0

cditloop:
    mulsr.h ssP1,ssK1,ssY1 ll,#1 ; (1) ;pi0=yr0*ki0 || pr0=yr0*kr0
    add.a  Xptr,Kptr,kindex ; || ;
    maddsr.h ssP1,ssP1,ssK1,ssY1 uu,#1
    ; (2)
    ; pi1=kr1*yi1 || pr1=yi1*kr1
    ld.w  ssK2,[Kptr] ; || ;kikr1
    mulsr.h ssP2,ssK2,ssY2 l1,#1 ; (3)
    ; pi1=yr1*ki1 || pr1=yr1*kr1
    maddsr.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  Xptr,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=xil+pi1 || xr'1=xrl+pr1
    std.  [Xptr]-8,e0 ; || ; store x'1x'0
    sub.h  ssYY2,ssX2,ssP2 ; (9)
    ; yi'1=xil-pi1 || yr'1=xrl-pr1
    std.  [Yptr]-16,e2 ; || ; store ylyy0

loop LC,cditloop
```
## Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d5 / d4</th>
<th>d7 / d6</th>
<th>d8</th>
<th>d9</th>
<th>d10</th>
<th>d11</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssP1, ssK1, ssY1 l, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsur.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssP1, ssP1, ssK1, ssY1 uu, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssP2, ssK2, ssY2 l, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsur.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssP2, ssP2, ssK2, ssY2 uu, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssXX1, ssX1, ssP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssYY1, ssX1, ssP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssXX2, ssX2, ssP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssYY2, ssX2, ssP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `yiyr1`<br>  `yiyr0`<br>  `kikr0`
- `pi0 || pr0`
- `kikr1`<br>  `pi0 || pr0`
- `pi1 || pr1`
- `yiyr3`<br>  `yiyr2`
7.4 Complex Butterfly – DIT – Radix 2 – with shift

Equations:

\[ X'_r = X_r + (Y_r^*K_r - Y_i^*K_i) \]
\[ X'_i = X_i + (Y_r^*K_r + Y_i^*K_i) \]
\[ Y'_r = X_r - (Y_r^*K_r - Y_i^*K_i) \]
\[ Y'_i = X_i - (Y_i^*K_r + Y_r^*K_i) \]

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];
}
```

Pseudo code implementation (loop only):

```
sP10 = sYr0 * sKi0 ; sPr0 = sYr0 * sKr0;
sp10 = sp10 + sYi0 * sKrl ; sp10 = sp10 - sYi0 * sKil;
sP11 = sYrl * sKil ; sp11 = sp11 + sYil * sKrl ;
sp11 = sp11 - sYil * sKil;
sXr0 >> 1 ; sXr0 >> 1;
sXXr0 = sXi0 + sP10 ; sXXr0 = sXr0 + sPr0;
sYYr0 = sXi0 - sP10 ; sYYr0 = sXr0 - sPr0;
sXr1 >> 1 ; sXr1 >> 1;
sXXr1 = sXi1 + sP11 ; sXXr1 = sXr1 + sPr1;
sYYr1 = sXi1 - sP11 ; sYYr1 = sXr1 - sPr1;
```

IP=8 (4 madd, 4 msub)   LD/ST= 10 (read sXr, sXi, sYr, sYi, read sKr, sKi, write sXXr, sXXi, sYYr, sYYi)
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) ; yly0
ld.w ssK1, [Xptr] ; (4) ; kikr0
cditsloop:

mulr.h ssP1, ssK1, ssY1 ll,#0 ; (1)
; pi0=yr0*ki0 || pr0=yr0*kr0
add.a Xptr, Kptr, kindex
; ||
maddsur.h ssP1, ssP1, ssY1 uu,#0 ; (2)
; pi0+=kr0*yi0 || pr0-=yi0*ki0
ld.w ssK2, [Xptr]
; || ; 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] ; || ; xixr0 xixr0
sha.h ssX1, ssX1, #1 ; (6) ; x0>>1
add.a Xptr, Kptr, kindex
; ||
add.h ssXX1, ssX1, ssP1
; (7)
; xi’0=xi0+pi0 || xr’0=xr0+pr0
ld.w ssK1, [Xptr]
; || ; kikr2
sub.h ssYY1, ssX1, ssP1
; (8)
; yi’0=xi0-pi0 || yr’0=xr0-pr0
ld.d e6, [Yptr+8] ; || ; yiyy2 yiyy2
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’lx’0
sub.h ssYY2, ssX2, ssP2
; (11)
; yi’1=xi1-pi1 || yr’1=xr1-pr1
st.d [Yptr]-16, e2
; || ; store yyyy0
loop LC, cditsloop
### Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d5 / d4</th>
<th>d7 / d6</th>
<th>d8</th>
<th>d9</th>
<th>d10</th>
<th>d11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yiyr1</td>
<td>yiyr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssP1,ssK1,ssY1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ll,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kikr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsur.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssP1,ssP1,ssK1,ssY1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uu,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kikr1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pi0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pi1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pr1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulr.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssP2,ssK2,ssY2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ll,#1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xixr1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xixr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pi1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pr1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sha.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssX1,ssX1,#-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0&gt;&gt;1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssXX1,ssX1,ssP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xi0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kikr2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssYY1,ssX1,ssP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yi0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yiyr3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yiyr2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sha.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssX2,ssX2,#-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1&gt;&gt;1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssXX2,ssX2,ssP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1'x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1'1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x'r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub.h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssYY2,ssX2,ssP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y1y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yiy1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yy1'1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y'r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

User Guide 125 v1.6.4, 2003-01
7.5 Complex Butterfly – DIF – Radix 2

Equations:

\[
\begin{align*}
X'_r &= X_r + Y_r \\
X'_i &= X_i + Y_i \\
Y'_r &= (X_r - Y_r)K_r - (X_i - Y_i)K_i \\
Y'_i &= (X_i - Y_i)K_r + (X_r - Y_i)K_i 
\end{align*}
\]

Pseudo code:

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

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:

\[
\begin{align*}
sXXi0 &= sXi0 + sYi0 \quad ; sXXr0 = sXr0 + sYr0; \\
sXXi1 &= sXi1 + sYi1 \quad ; sXXr1 = sXr1 + sYr1; \\
sDi0 &= sXi0 - sYi0 \quad ; sDr0 = sXr0 - sYr0; \\
sDi1 &= sXi1 - sYi1 \quad ; sDr1 = sXr1 - sYr1; \\
sYYi0 &= sDi0 \ast sKr0 \quad ; sYYr0 = sDr0 \ast sKr0; \\
sYYi1 &= sDi1 \ast sKr1 \quad ; sYYr1 = sDr1 \ast sKr1; \\
sYYi0 &= sYYi0 + sDr0 \ast sKi0 \quad ; sYYr0 = sYYr0 - sDi0 \ast sKi0; \\
sYYi1 &= sYYi1 + sDr1 \ast sKi1 \quad ; sYYr1 = sYYr1 - sDi1 \ast sKi1; \\
\end{align*}
\]
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); yiy0
ld d e4, [Xptr+]8 ; (3); xlx0
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=xil+yi1 || xr’1=xrl+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); dil=xil-yi1 || dr1=xrl-yr1
    ld w ssK2, [Kptr] ; || ; kikr1
    mulr h ssYY1, ssD1, ssK1 li, #1 ; (5)
      ; yi’0=di0*kr0 || yr’0=dr0*kr0
    st d [Xptr]-8, e0 ; || ; store x’lx’0
    maddsur h ssYY1, ssYY1, ssD1, ssK1 uu, #1; (6)
      ; yi’0+=dr0*ki0 || yr’0=di0*ki0
    mulr h ssYY2, ssD2, ssK2 li, #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 yylyy0
loop LC, cdifloop
```
### Register diagram:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>d0</th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d5 / d4</th>
<th>d7 / d6</th>
<th>d8</th>
<th>d9</th>
<th>d10</th>
<th>d11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yiyr0</td>
<td>yiyr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xixr0</td>
<td>xixr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.h ssXX1, ssX1, ssP1</td>
<td>xi0</td>
<td></td>
<td>xr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yiyr1</td>
<td>yiyr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xixr1</td>
<td>xixr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.h ssXX2, ssX2, ssP2</td>
<td>xi1</td>
<td></td>
<td>xr1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yiyr0</td>
<td>yiyr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xixr0</td>
<td>xixr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub.h ssYY1, ssX1, ssP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d0</td>
<td></td>
<td></td>
<td>dr0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yi0</td>
<td></td>
<td></td>
<td>yr0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xixr0</td>
<td>xixr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub.h ssYY2, ssX2, ssP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d1</td>
<td></td>
<td></td>
<td>dr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yi0</td>
<td></td>
<td></td>
<td>yr0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulr.h ssP1, ssK1, ssY1 ll, #1</td>
<td>x1</td>
<td></td>
<td>x1'0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y1</td>
<td></td>
<td></td>
<td>yr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsur.h ssP1, ssP1, ssK1, ssY1 uu, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsur.h ssP2, ssK2, ssY2 ll, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maddsur.h ssP2, ssP2, ssK2, ssY2 uu, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yiyr3</td>
<td>yiyr2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y1</td>
<td></td>
<td></td>
<td>yr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y1</td>
<td></td>
<td></td>
<td>yr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yiyr0</td>
<td>yiyr0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y1</td>
<td></td>
<td></td>
<td>yr1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

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

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’m 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

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