



# Implementing and Optimizing an FIR Filter for the S5 Engine

Application Note

Version 1.2

**Confidential & Proprietary**

---

Last modified: 08/16/2005

© 2004 Stretch, Inc. All rights reserved. The Stretch logo, Stretch, and Extending the Possibilities are trademarks of Stretch, Inc. All other trademarks and brand names are the properties of their respective owners.

This preliminary publication is provided “AS IS.” Stretch, Inc. (hereafter “Stretch”) DOES NOT MAKE ANY WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF TITLE, NONINFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Information in this document is provided solely to enable system and software developers to use Stretch S5000 processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder. Stretch does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Part #: AN-0001-0001-001

# Contents

## About This Application Note

### Chapter 1 Implementing and Optimizing Applications

1.1	Finite Impulse Response (FIR) Filter Application . . . . .	1-1
1.2	Implement the FIR Filter in C . . . . .	1-3
1.3	Analyze Profile Output . . . . .	1-4
1.3.1	Comparison of FIR filter cycle counts . . . . .	1-4
1.4	Rewrite the Hot Spot for the ISEF. . . . .	1-5
1.5	Creating Extension Instructions for the FIR Filters . . . . .	1-6
1.5.1	Using Eight Multipliers on the ISEF . . . . .	1-6
1.5.2	Using 16 Multipliers in the ISEF . . . . .	1-11
1.5.3	Using 32 Multipliers in the ISEF . . . . .	1-13

### Chapter 2 Conventional Optimizations

2.1	Modifying the C Implementation . . . . .	2-1
2.1.1	Analyzing Profile Information for the Rewritten Application . . . . .	2-4
2.2	Improving Performance with Loop Optimizations . . . . .	2-5
2.3	Optimizing Between Loop Iterations . . . . .	2-5
2.4	Optimizing with Manual Loop Unrolling . . . . .	2-8
2.5	Recap of Optimization Effort. . . . .	2-11

### Appendix A Code Examples

A.1	README . . . . .	A-1
A.2	Makefile . . . . .	A-2
A.3	firMain.c . . . . .	A-5
A.4	fir0a.c . . . . .	A-11
A.5	fir8.xc . . . . .	A-12
A.6	fir8a.c . . . . .	A-13
A.7	fir8b.c . . . . .	A-15
A.8	fir8c.c . . . . .	A-17
A.9	fir16.xc . . . . .	A-21
A.10	fir32.xc . . . . .	A-22
A.11	firx.h . . . . .	A-25



# Figures

Figure 1-1	Graphical representation of an FIR filter . . . . .	1-2
Figure 1-2	FIR graph showing 8 multipliers . . . . .	1-7
Figure 1-3	Excerpt from the Resource file fir8.xr . . . . .	1-10
Figure 1-4	FIR graph showing 16 multipliers . . . . .	1-12
Figure 1-5	FIR graph showing 32 multipliers . . . . .	1-14
Figure 2-1	C implementation modified to call the Extension Instruction. . . . .	2-2



# Tables

Table 1-1	Unaccelerated C implementation performance statistics . . . . .	1-4
Table 2-1	Accelerated implementation performance statistics (8 multipliers) . . .	2-4
Table 2-2	Loop-optimized performance statistics . . . . .	2-7
Table 2-3	Manually unrolled loop performance statistics . . . . .	2-11
Table 2-4	Recap of optimization improvements . . . . .	2-12



# About This Application Note

The S5 Engine puts you in control of optimizing your applications. You can do a little or a lot, it is entirely up to you how much is enough (within the limits of the hardware, of course).

This application note demonstrates how to implement and optimize an FIR filter for the S5 Engine.

Chapter 1 introduces you to the development steps you should use for any application implementation for the S5 Engine, it then shows you how the application can be written in straight C, then it discusses some quick and easy optimizations written in Stretch C.

Chapter 2 shows how easily the original C implementation can be modified to use the Stretch C optimizations created in Chapter 1, and how you can further optimize the application by unrolling loops.

Appendix A lists all the code referenced in this document. The code is also included as part of the tools distribution in the `/Examples/kernels` directory.

The beauty of this approach to developing and optimizing applications is that *you* decide how much optimization is enough; you choose which method best suits *your* requirements.



# Chapter 1

# Implementing and Optimizing Applications

The six basic development steps for implementing any application on the S5 Engine are:

1. Define the application.
2. Write the application in C/C++.
3. Compile and link the application source files using the Stretch C Compiler (scc).
4. Run and profile the application.
5. Determine which parts of the application need to be accelerated based on the profile data.
6. Rewrite the computationally intensive code (hot spots) for the S5 Engine's Instruction Set Extension Fabric (ISEF), and repeat steps 2 through 5 until you are satisfied with the application's computation performance.

This application note shows you how to use steps 2 through 6 to implement a Finite Impulse Response (FIR) filter on the S5 Engine.

---

## 1.1 Finite Impulse Response (FIR) Filter Application

The FIR filter contains computational hot spots that lend themselves well to the type of optimization for which the S5 Engine was designed.

It can be implemented with the following equation:

$$y(n) = \sum_{t=0}^{t=T-1} ((h(t) * x)(n-t)) \text{ for } n = 0, \dots, N-1$$

Where

x(n) is the input signal

y(n) is the output signal

h(t) is the FIR filter coefficients

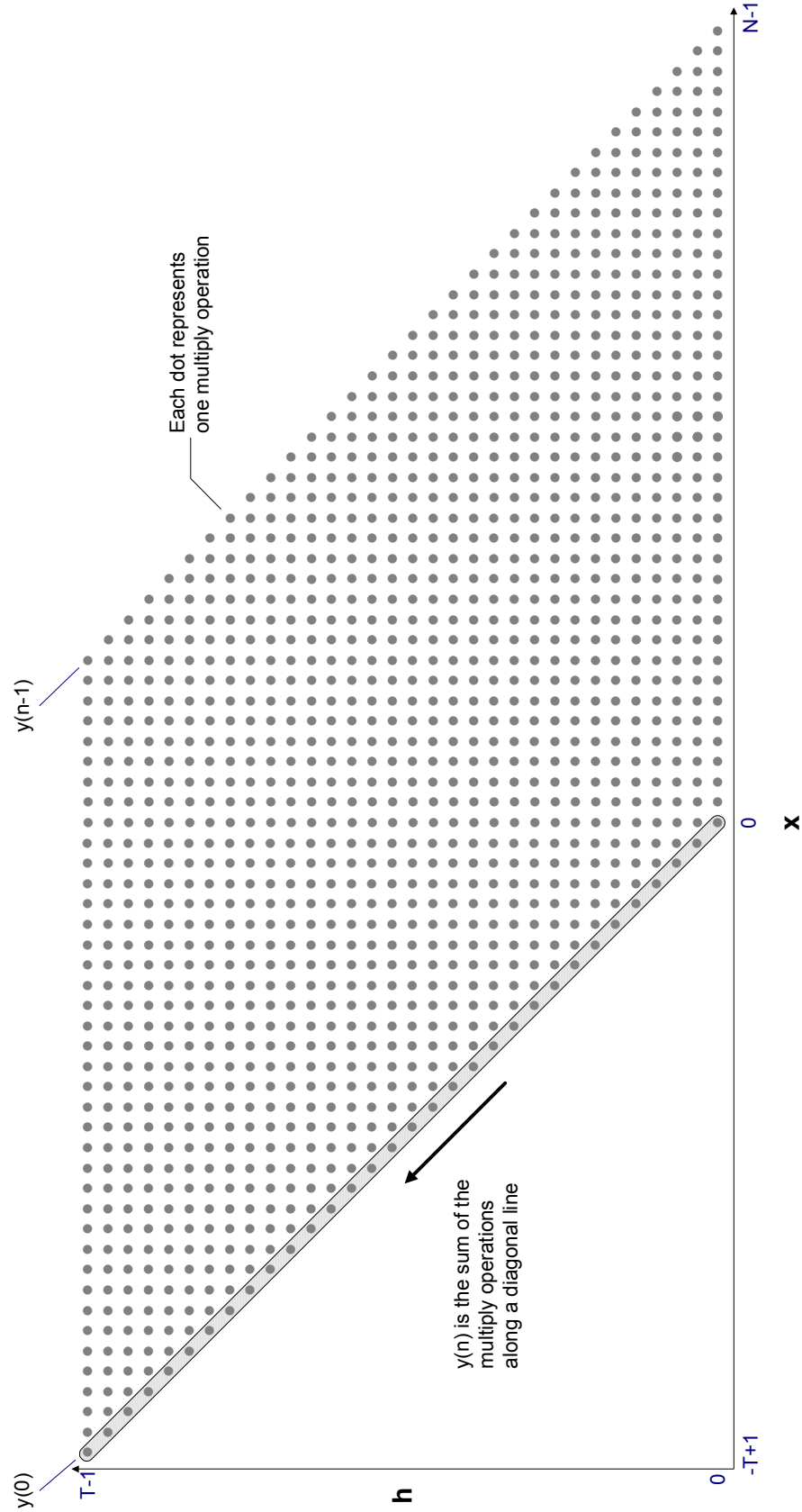


Figure 1-1 Graphical representation of an FIR filter



Figure 1-1 presents a useful way of visualizing the FIR operation. Every dot in this grid represents a multiply of a coefficient  $h[t]$  and a data point  $x[n-t]$ . Each diagonal strip groups all the products that must be added to obtain the result,  $y[n]$ .

In a traditional implementation, like the straight C implementation of the FIR filter, on a processor with a single multiplier, all the computations (multiplies and adds) within a strip must be completed before the adjacent strip can be computed.

---

## 1.2 Implement the FIR Filter in C

A C implementation of the FIR filter can be written as follows:

The `fir()` function

```
void fir(short *X, short *H, short *Y, int N, int T)
{
    int n, t, acc;
    short *x, *h;

    /* Filter Input */
    for (n = 0; n < N; n++) {
        x = X;
        h = H;

        acc = (*x--) * (*h++);

        for(t = 1; t < T; t++) {
            acc += (*x--) * (*h++);
        }

        *Y = acc >> 14;
        X++;
        Y++;
    }
}
```

In the preceding code

`X` is a pointer for the Input Signal array, which points at the location representing discrete time index 0. It is a 16-bit quantity.

`H` is a pointer for the Filter Coefficient array, which points at the 0th coefficient. It is a 16-bit quantity.

`Y` is the pointer for the Output Signal array, which points at the 0th output sample location. It is a 16-bit quantity.

`N` represents the number of input–output signals.

`T` represents the FIR Filter Taps.



**NOTE:** The complete source listing for all code used in this document is in Appendix A as well as in the `/Examples/kernels` distribution directory; the following code is included here only for reference.

**TIP~** You can use the files in the `/Examples/kernels` distribution directory to compile, run, and profile the FIR implementation presented in this document.

## 1.3 Analyze Profile Output

After compiling, linking, running, and profiling the C implementation you can analyze the output from the profiler to find the hot spots in your code. The output from the profiler produces a summary of performance statistics collected by the simulator during execution. Table 1-1 is an excerpt of the performance statistics for the unaccelerated C implementation of the FIR implementation showing those functions that use the highest percentage of cycles.

The file `fir0a.exe.prof` in the `/Examples/kernels` distribution directory contains the complete profiler output for the preceding code.

Table 1-1 Unaccelerated C implementation performance statistics

	% cumulative cycles	self cycles	calls	self cycles/call	total cycles/call	name
64.34	136153.00	136153.00	5	27230.60	27230.60	fir
29.94	199501.00	63348.00				ResetH
3.54	206986.00	7485.00	1	7485.00	146984.38	main
0.35	207721.00	735.00	1	735.00	2786.38	_vfprintf_r
0.26	208273.00	552.00	1	552.00	728.50	_malloc_r
0.14	208569.00	296.00	1	296.00	296.00	__sinit
0.13	208846.00	277.00	1	277.00	610.88	__sfvwrite

### 1.3.1 Comparison of FIR filter cycle counts

As you can see in Table 1-1, the `fir()` module takes 64.34% of the total cycles, using 27230.6 cycles per call. Based on this number, we can conclude easily that the `fir()` function is an excellent candidate for execution on the ISEF, which should boost the performance of the FIR implementation on the S5 Engine.



## 1.4 Rewrite the Hot Spot for the ISEF

On the S5 Engine, there is a lot of flexibility that allows us to exploit the pattern of computations for the FIR, and accordingly map them to Extension Instructions, so that we can achieve maximum performance gain. But before we jump into this analysis, following are a few important features of the S5 Engine to remember when writing programs for it:

- All data access to and from the ISEFs are through register files located in the Extension Unit. There are two banks of register files (A and B). Each bank is 16 deep, and each register is 128 bits wide. Thus, in total we have 32 128-bit wide registers (WRs).
- Each ISEF has three read and two write ports. Any given instruction can perform a maximum of three 128-bit reads (384-bits), and two 128-bit writes (256-bits).
- The Stretch™ processor and the ISEF run off the same clock source. The timing between them is skewed by the issue rate (see the *SCP Architecture Reference* for details on issue rate). What this means is that we can perform 128-bit registers loads and stores in between consecutive invocations of Extension Instructions without under-utilizing the ISEF. In addition, because Extension Instruction execution is pipelined, there is no need to wait for an Extension Instruction to finish before the next Extension Instruction is issued. Thus, if an Extension Instruction does not require the output generated by the previous instruction, Extension Instructions can be issued at the issue rate. Eventually, however, some processor cycles will be spent waiting for the final results from the ISEF execution, which is dependent on the ISEF latency times the issue rate.
- The Stretch processor can read or write to or from the register files for memory-to-ISEF data transfers. Stretch processor instructions include support for 8-, 16-, 32-, 64-, and 128-bit loads and stores for the WRs with immediate, indexed, circular, or bit-reversed addressing. There is also support for bit and byte puts and gets to or from the WRs, and a 128-bit move instruction between WR registers. For anything less than 128-bit loads or stores, the upper bits are zero-padded.
- As noted in the Section 1.4.8, “The S5 Engine Instruction Set Extension Fabric Capability and Capacity” of the *SCP Architecture Manual*, an Extension Instruction using many AU or MU elements leaves fewer ISEF resources for other Extension Instructions in the same ISEF configuration.



We can now show how to exploit the symmetry of the FIR computations to create variations of Extension Instructions that will increase the performance gain. We also leverage the flexibility provided by the ISEF in creating different instructions that maximize the performance.

---

## 1.5 Creating Extension Instructions for the FIR Filters

The following sections show how we can vary the amount of ISEF resources we use to improve the performance of the FIR filter. We will first use eight multipliers within the ISEF to accelerate the inner loop, then 16 multipliers, and finally, 32 multipliers. Each variant provides increased performance at the cost of increasingly more, but not excessive, ISEF resources.

### 1.5.1 Using Eight Multipliers on the ISEF

In our example, the input signal and the filter coefficients are 16-bit quantities. Because we have the ability to pass three 128-bit registers simultaneously into the ISEF, we can pack eight input samples into one 128-bit wide register and eight coefficients into another 128-bit wide register. These wide registers are then used to input data into the ISEF instructions. Based on this, we see that if we can program the ISEF to perform eight multiplies every instruction, we would only require  $T/8$  inner loop computations, which should give us an 8x performance boost. Figure 1-2 on page 1-7 illustrates one approach that can be used to do the FIR computations in blocks of eight.

In this approach, we still use the strip concept introduced in Figure 1-1, but we perform eight multiplies and adds simultaneously. Each highlighted group of dots represents one Extension Instruction. Each instruction multiplies eight data values by eight coefficients, and accumulates all the products. The resulting partial sum is saved in an ISEF local state variable so that subsequent instructions can add their partial sums to it.





The Extension Instruction (`firFunc`) consists of two instructions: `FIR_MUL` and `FIR_MAC`. These instructions are invoked from the main thread using the following function call notation:

```
FIR_MUL(x, h, y);
FIR_MAC(x, h, y);
```

Where `x` and `h` are inputs to the instruction, and `y` is the output from the instruction. All three must be declared as wide registers, in the main program thread, using the `WR` type (refer to the *SCP Architecture Reference* for information on declaring types for Extension Instructions).

The first eight inner loop computations use the `FIR_MUL` instruction to initialize the state variable (`acc`) with the sum of the eight products. All subsequent iterations use `FIR_MAC` to add the resulting sum to `acc`. The output is always written out, irrespective of the computation's stage, to avoid using another instruction. It is up to the main thread to determine when the result from the instruction is of interest. For our 64-tap filter implementation, we call `FIR_MUL` once and `FIR_MAC` seven times for every outer loop iteration, so the output from `FIR_MUL` and the first six invocations of `FIR_MAC` is ignored.

**NOTE:** The complete source listing for the Extension Instruction is in Appendix A as well as in the file `fir8.xc` in the `/Examples/kernels` distribution directory; the following code segment is only for reference.

The Extension Instruction that defines `FIR_MUL` and `FIR_MAC` can be written as follows:

```
A ——— #include <stretch.h>
B ——— static se_sint<32> acc;
          /* Performs 8 parallel MAC */
          SE_FUNC void D ———
C ——— firFunc(SE_INST FIR_MUL, SE_INST FIR_MAC, WR X, WR H, WR *Y)
          {
F ———     se_sint<16> x, h
          se_sint<32> sum;
          int i;
          G ———
          sum = 0;
          J ——— for(i = 0; i < 128; i += 16) {
                    h = H(i + 15, i);
                    x = X(127-i, 112-i);
                    sum += x * h;
          }
K ———     acc = FIR_MUL ? sum : se_sint<32>(sum + acc);
L ———     *Y = acc >> 14;
          }
```



**The parts of the preceding Extension Instruction are:**

- A** This header file defines arbitrary-sized integer types and the usual arithmetic operations on them. It must be included in all source files that use Stretch-defined declarations and data types.
- B** This variable is a resource that is preserved between instruction invocations. As declared here, `acc` uses 32 bits of the local state resource. It may be implicitly used by either Extension Instruction.
- C** The ISEF function is declared as type `SE_FUNC void` and named `firFunc`. All ISEF functions must be declared as type `SE_FUNC void`.
- D** Parameters of type `SE_INST` specify instruction names. The names are `FIR_MUL` and `FIR_MAC`. When the application uses the instruction, these names do not appear.
- E** The Extension Instructions `FIR_MUL` and `FIR_MAC` are defined to have three arguments: `H`, `X` and `Y`. They are declared as type `WR`, which means that they are wide registers. In this example, `H` and `X` are inputs, and `Y` is an output. The asterisk (\*) before `Y` means that `Y` is an output.  
  
In some cases, we can use `Y` as an input also. In this case, we would still declare `Y` in this fashion if the result were to be written out using `Y`.
- F** Variables of type `se_sint<n>` are arbitrary-width data types used to define variables `<n>` bits wide. The term `se_sint<>` is used for signed quantities, and the term `se_uint<>` is used for unsigned quantities. Because the width of these data types is arbitrary, `<n>` can be any value required for your application, within the physical limitations of the ISEF, of course.
- G** Resource used only during instruction computation. This variable is allocated when the instruction is invoked and is not preserved between invocations.
- H** Stretch C supports bit extraction within a certain range. Here, the 16-bit `h` variable is loaded with 16-bits from the `H` wide-register. The bit range is specified by the loop counter `i`. When `i = 0`, `h` is loaded with least-significant (0...15) 16-bits from `H`.
- I** Support for normal C operators (+, -, \*, <<, >>, <, =, >, etc.)
- J** Because 16-bits are read from wide-registers `H` and `X` per iteration, we execute the loop eight (128-bits/16-bits) times. In addition, because the inputs to the eight multiplications (`x * h`) are independent, `scc` will recognize this and execute the eight iterations in parallel. That is, the 16 16-bit extraction (eight for `h` and eight for `x`), and eight multiplication operations will all execute in parallel resulting in only one execute stage.



**The parts of the preceding Extension Instruction are:**

- K** Decide whether to initialize the state resource (*acc*) with the partial sum, or to add to the state resource for the next set of computations. For the first eight calculations, *FIR\_MUL* is invoked, so the partial sum is stored in the state resource. Subsequent sets of eight calculations are computed using *FIR\_MAC*, which adds the new partial sum to the state resource.
- L** The accumulated values are scaled to 16 bits and written to the output wide register.

The number of cycles, without considering cache misses or memory access latencies, we should expect from this implementation can be computed as follows:

$$cycles = N * \left\{ issue\ rate * \frac{T}{8} \right\} + (issue\ rate * (ISEF\ latency + 2))$$

*issue rate* is the difference between the processor clock and the ISEF clock, which is three for this example. *ISEF latency + 2* is the number of ISEF cycles plus two cycles for reading and writing the wide registers.

The *ISEF latency* is obtained from the user report generated by *scc* when it compiles the Extension Instructions. Figure 1-3 is an excerpt from the report for the preceding code. The line “Output: variable ‘Y’ written to WRA on ISEF write port 0 at end of cycle 5” tells you what the ISEF latency is.

Figure 1-3 Excerpt from the Resource file *fir8.xr*

```
// Total Number of configurations: 1
// Configuration 'fir8', implementing instructions FIR_MUL,FIR_MAC:
// INSTRUCTION=firFunc:FIR_MUL Opcode: 0: Assembly 'se_fir_mul X:in, H:in, Y:out'
//   Input : variable 'X' read from WRA on ISEF read port 2 at beginning of cycle 1
//   Input : variable 'H' read from WRA on ISEF read port 1 at beginning of cycle 1
//   Output: variable 'Y' written to WRA on ISEF write port 0 at end of cycle 5
//   ISEF Local State variable 'acc' (32 bits) written at end of cycle 4
// INSTRUCTION=firFunc:FIR_MAC Opcode: 1: Assembly 'se_fir_mac X:in, H:in, Y:out'
//   Input : variable 'X' read from WRA on ISEF read port 2 at beginning of cycle 1
//   Input : variable 'H' read from WRA on ISEF read port 1 at beginning of cycle 1
//   Output: variable 'Y' written to WRA on ISEF write port 0 at end of cycle 5
//   ISEF Local State variable 'acc' (32 bits) read at beginning of cycle 4
//   ISEF Local State variable 'acc' (32 bits) written at end of cycle 4 (write Read = 0)
// Class Assignments for configuration 'fir8'
// INSTRUCTION      Read Class  Write Class
//                   0 1 2 3      0 1 2 3
// firFunc:FIR_MUL
// firFunc:FIR_MAC
// Class 0: 0 stall(s)
// Class 1: 0 stall(s)
// Class 2: 0 stall(s)
// Class 3: 0 stall(s)
// Computational Resources
// Arithmetic bits.....224
// Logic bits.....0
// MUX bits.....32
// Register bits.....0
// Pipeline bits.....6
// AU total.....262 out of 4096
// Multiply bits.....2048
// MU total.....2048 out of 8192
// Extension registers.....32 out of 4096
// Extension registers are allocated from the available pool of AUs and
// included in the AU total above.
```

**ISEF  
Latency**



Assuming  $N = 80$ ,  $T = 64$  for the FIR filter, and with *issue rate* = 3, and *ISEF latency* = 5, the expected cycle count for an Extension Instruction that uses eight multipliers to compute the inner loop is 1941. When compared to the straight C implementation, this is a performance gain of 15x.

## 1.5.2 Using 16 Multipliers in the ISEF

In the performance enhancements described in the preceding section, we used eight multipliers to improve the performance of the FIR implementation. Because the ISEF has the flexibility to let you choose its execution behavior, and staying within the physical limitations of the ISEF, we can revise the instruction. This maximizes the computations in the ISEF per instruction invocation and, at the same time, reduces the number of Extension Instructions invoked. From the previous analysis, it would be beneficial to

- Minimize the instruction invocations to eliminate the need to worry about scheduling the Extension Instructions.
- Increase the computation load of a given Extension Instruction to reduce the number of instruction invocations.

In this and the next section we explore these possibilities, and show how to increase performance by using some of the suggested techniques.

If you were to use a more of the ISEF's resources to implement a 16-multiplier version of the Extension Instructions, you could potentially improve performance. Of course, this is strictly dependent on the algorithm being implemented. Figure 1-4 on page 1-12 illustrates this concept graphically.

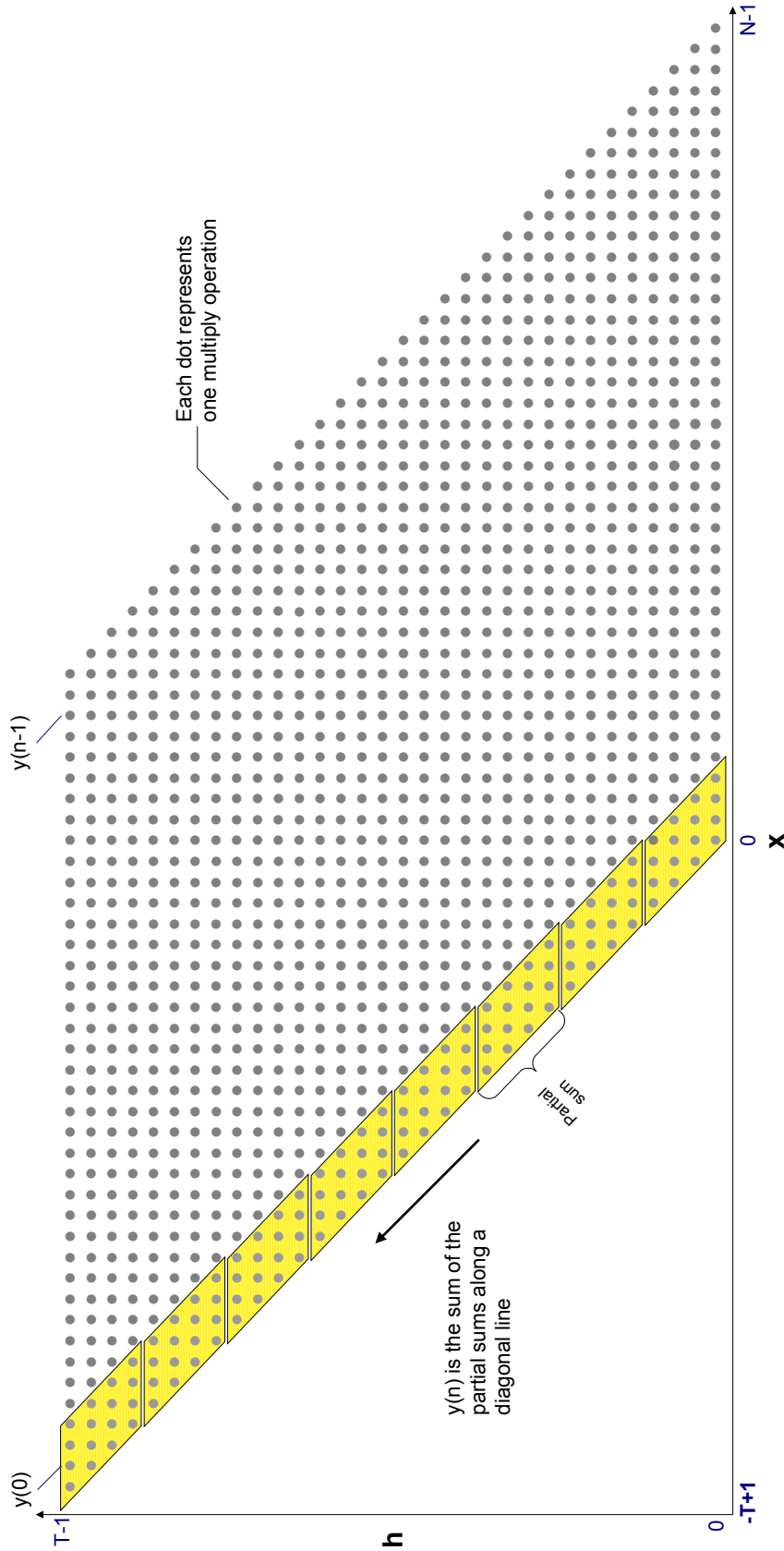


Figure 1-4 FIR graph showing 16 multipliers



Figure 1-4 implies that we can take advantage of the fact that we can implement 16 multipliers on the ISEF to operate on the 4 x 4 blocks outlined in the figure. In addition, we can easily achieve this while remaining well within the ISEF's available 384-bit input bandwidth.

We could compute four partial sums for four outputs every Extension Instruction. This would require us to load four filter coefficients and seven data (input) samples. In each instruction execution, we compute partial sums of the four outputs; subsequent instructions add to these partial sums, thus accumulating the products until the final result is achieved. In the last instruction, the individual results can be computed and packed into one wide register for output to the processor. In addition, we would now need to preserve four 32-bit accumulator values to the local state instead of just one.

**NOTE:** The complete code for this modification is in Appendix A and in the file `fir16.xc` in the `/Examples/kernels` distribution directory.

By using this Extension Instruction, as we are computing four outputs per outer loop iteration, we are reducing the outer loop by four. We are using four coefficients per Extension Instructions, so to cycle through the complete set of filter coefficients, we need to invoke T/4 Extension Instructions (T is the filter length).

The theoretical minimum number of cycles we should expect from this implementation can be computed as follows:

$$cycles = \frac{N}{4} * \left\{ issue\ rate * \frac{T}{4} \right\} + (issue\ rate * (ISEF\ latency + 2))$$

Using  $N = 80$ ,  $T = 64$ ,  $issue\ rate = 3$ , and  $ISEF\ latency = 5$  as before, the expected cycle count for an Extension Instruction that uses 16 multipliers to compute the inner loop is 981. When compared to the straight C implementation, this is a theoretical performance gain of 30x.

### 1.5.3 Using 32 Multipliers in the ISEF

By using additional multiplier resources in the ISEF, we can even further improve the performance because there is potential to compute eight partial sums (outputs) every ISEF invocation. Figure 1-5 on page 1-14 illustrates this concept graphically.

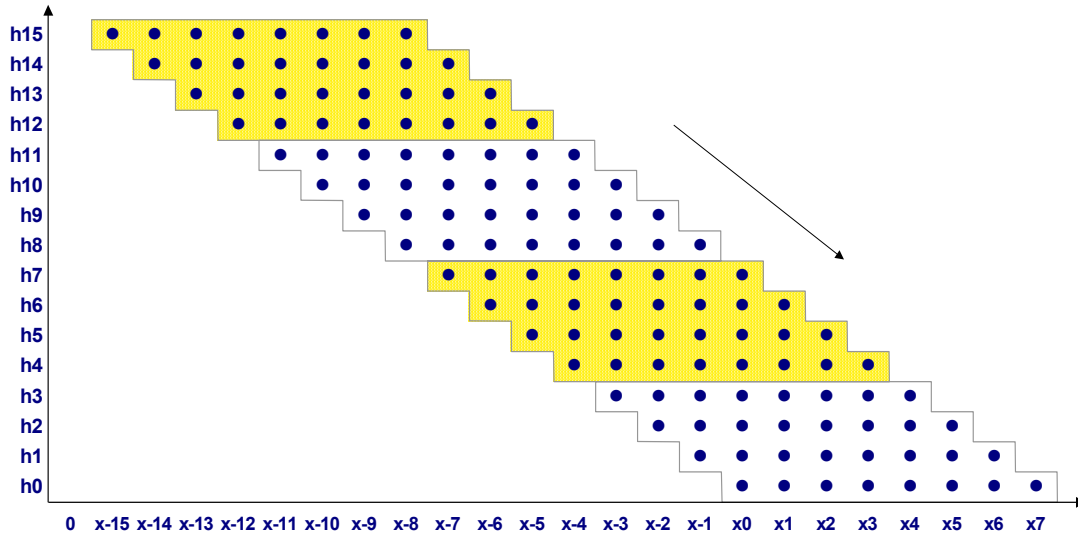
We would require four coefficients and 11 data samples to compute eight partial sums per Extension Instruction. Eight of the data samples are kept in ISEF state from one instruction to the next

The `FIR_INIT_MUL` instruction is used first to zero the sums. Its inputs are the first eight data points, four filter values, and four additional data points.



Then the FIR\_MAC instruction is used repeatedly. Its inputs are from state, the last eight data points; from WRs, four new data points and four new filter values. Eleven data points are used; the twelfth point is simply held and shifted down for the next call.

Figure 1-5 FIR graph showing 32 multipliers



**NOTE:** The complete code for this modification is in Appendix A and in the file `fir32.xc` in the `/Examples/kernels` distribution directory.

The theoretical minimum number of cycles expected from this implementation can be computed as follows:

$$cycles = \frac{N}{8} * \left\{ issue\ rate * \frac{T}{4} \right\} + \{ issue\ rate * (ISEF\ latency + 2) \}$$

Using  $N = 80$  points,  $T = 64$  coefficients,  $issue\ rate = 3$ , and  $ISEF\ latency = 5$  we get:

$$\begin{aligned} &= 10 * (3 * 16) + (3 * 7) \\ &= 501\ cycles \end{aligned}$$

When compared to the straight C implementation, there is a theoretical performance gain of approximately 58x.

## Chapter 2

# Conventional Optimizations

This chapter we show you, using the eight multiplier Extension Instruction, how to rewrite the FIR application to invoke the Extension Instructions. We also show you several optimization steps that could be applied to the C code with changing the Extension Instruction. These step show how an implementation can be completely optimized by manipulating the C source without having to write any assembly code or change the Extension Instruction.

In Section 2.1 we discuss the changes made to the C implementation to incorporate the Extension Instruction. We also show you the profiler output and the performance improvement achieved after changing the C source and invoking the Extension Instruction.

In Section 2.2 and 2.3 we show how to reduce cycle further by hand-optimizing the loop execution.

Finally, Section 2.4 we show how unrolling loops by hand lets us achieve the expected results discussed in Chapter 1.

---

## 2.1 Modifying the C Implementation

The modified C implementation that calls the eight-multiplier version of the Extension Instruction can be written as shown in Figure 2-1 on page 2-2.



Figure 2-1 C implementation modified to call the Extension Instruction

```

A ——— #include "fir8.h"

        #define      ST_DECR      1
        #define      ST_INCR      0

        void fir(short *X, short *H, short *Y, short N, short T)
        {
B ———   int n, t, t8;
          WR x, h, y;
C ———   t8 = T/8;
D ———   WRPUTINIT(ST_INCR, Y) ;

          for (n = 0; n < N; n++) {
E ———   WRGETOINIT(ST_INCR, H) ;
          X++ ; ————— F
          WRGET1INIT(ST_DECR, X) ;

          WRGETOI( &h, 16 );
          WRGET1I( &x, 16);
          FIR_MUL(x, h, &y); ————— H
          |
          for (t = 1; t < t8; t++) {
J ———   WRGETOI(&h, 16);
          WRGET1I(&x, 16);
          FIR_MAC(x, h, &y); ————— I
          }

K ———   WRPUTI(y, 2) ;
        }

L ———   WRPUTFLUSH0() ;
          WRPUTFLUSH1() ;
        }
  
```

**The parts of the preceding code are:**

- A** Typically you would also include the Stretch system file `stretch.h`, but because it is automatically included in the header file generated by `scc`, we don't include it here.
- B** Wide Register allocations for the Input, Coefficients and Output samples.
- C** Inner loop counter (divide by 8, as we perform 8 computations per ISEF invocation).
- D** The `WRPUTINIT()` intrinsic defines the usage of the `WRPUTINIT` instruction. Here we initialize the output (write) byte stream mechanism with the memory address (`Y` in this case) where the data is to be written, and the direction indicator (increment in this case).

**The parts of the preceding code are:**

- E** Similar to `WRPUTINIT`. Here we initialize the input (read) byte stream mechanism using the `WRGETINIT( )` intrinsic. In addition, we use `WRGET0INIT( )` for the `H` buffer and `WRGET1INIT( )` for the `X` buffer. This implies that `H` and `X` are using two different read byte stream mechanisms. In total we can independently read data from three different memory locations.

In the C implementation, we incremented the `H` pointer while the `X` pointer was decremented during the inner loop calculation. Thus, we use the `ST_INCR` (increment) option for the buffer pointer that reads from the coefficient array, and the `ST_DECR` (decrement) option for the buffer pointer that reads from the input sample array.

- F** The `X++` pointer is incremented before we initialize the read byte stream mechanism that is going to bring in data from the `X` memory address. We use the decrement direction for data access, so we need to initialize the read mechanism with one address *above* the address from which we want to start reading to ensure that we get the data from the correct initial position. We need to do this because the read mechanism is defined to adjust the address from which it is to read *before* reading the data).

- G** The `WRGET` instructions provide mechanisms to load 1–16 bytes from an unaligned memory location.

Here we fetch eight 16-bit quantities from the `H` and `X` array in memory. The eight 16-bit values of both the arrays are transferred from memory to their respective wide registers in a single cycle.

After the `GET` mechanism is initialized, the subsequent `GET` instructions become single-cycle instructions.

The short data points in Wide Register `x` are ordered with the lowest-address data in the lowest-order bits, the same as in Register `h`. That is, although the successive `GETs` for `x` decrement the pointer, the data retrieved in one `GET` remains in the usual little endian order.

- H** Invoke the first Extension Instruction, which performs the first eight multiply–accumulates and initializes the accumulator (internal to the Extension Instruction). This is equivalent to the C implementation, where we performed the computation of the 0th input sample and filter coefficient to initialize the accumulator.
- I** Invoke the Extension Instructions using function call notation. The function names are the same as the instruction names defined in the `ISEF` function. The calling module supplies all the required I/O arguments when invoking these instructions.



**The parts of the preceding code are:**

- J** Compute the rest of the inner loop calculations in blocks of eight. The `FIR_MAC` instruction performs the multiplications of the input samples and filter coefficients, and adds the products to the accumulator to keep a running history of the accumulated sum.
- K** The outputs `acc` are now ready for a Wide Register-to-Memory transfer. The `FIR_MAC` instruction was writing into `acc` for every invocation, but it is up to the main thread to decide when the output is legal.

We could have rewritten the `FIR_MAC` not to write out the output every time it was called, and then defined another instruction that can be invoked after all the computations to retrieve the output from the ISEF local state (where the accumulated results are stored), but this would have added an extra Extension Instruction.

- L** The `WRPUTFLUSH` instructions flush the output byte stream. They are required when using `WRPUT` instructions to write data to memory.

## 2.1.1 Analyzing Profile Information for the Rewritten Application

Table 2-1 on page 2-4 is an excerpt of the performance statistics for the eight-multiplier accelerated implementation of the FIR application produced by the profiler showing those functions that use the highest percentage of cycles.

Table 2-1 Accelerated implementation performance statistics (8 multipliers)

%	cumulative cycles	self cycles	calls	self cycles/call	total cycles/call	name
62.90	63348.00	63348.00				ResetH
25.15	88673.00	25325.00	5	5065.00	5065.00	fir
7.43	96158.00	7485.00	1	7485.00	36144.00	main
0.73	96893.00	735.00	1	735.00	2766.0	_vfprintf_r
0.55	97445.00	552.00	1	552.00	718.00	_malloc_r
0.29	97740.00	295.00	1	295.00	295.00	__sinit
0.28	98017.00	277.00	1	277.00	606.00	__sfwwrite

The `fir()` function now takes 25.15% of the total cycles, using 5065.00 cycles per call. Using the ISEF to accelerate the `fir()` inner loop provided a significant drop in cycles—5065.00 cycles compared with 27230.60 cycles for the unaccelerated C implementation. This is more than a 5x improvement over the straight C implementation with absolutely no low-level assembly coding or significant modification to the structure of the source files.



---

## 2.2 Improving Performance with Loop Optimizations

The goal of loop optimization is to remove wasted cycles, and to use every cycle per iteration. Wasted cycles arise because, after invoking the last `FIR_MAC` instruction for a given outer loop iteration, the processor must wait for  $issue\ rate * ISEF\ latency$  cycles before the results of the ISEF are of interest to the processor. Thus, in a loop optimization scheme, we try to utilize these wasted cycles by starting the computations for the next output while waiting for the previous output. By design, the ISEF execution is pipelined, so the Extension Instruction invocations need not wait for the previous Extension Instruction to complete.

For our example, we see that for an output, we need to issue `FIR_MUL` and `FIR_MAC` seven times. Thus, the total processor cycles required to perform this task would be  $issue\ rate * number\ of\ ISEF\ instructions$ . If the value  $issue\ rate * ISEF\ latency$  is less than the cycles required to issue the ISEF instruction, we can take advantage of this. While we are waiting for a given output, we could start issuing the ISEF instructions for the next output. Thus, as soon as we are done issuing the ISEF instruction, given the fact that the processor cycles required to complete the issuing is greater than the processor cycles required to wait for the output, we are guaranteed that upon finishing the ISEF invocations for the current samples, the output from previous sample will be available for the processor to use.

---

## 2.3 Optimizing Between Loop Iterations

Given that the inner loop calculation for a particular outer loop iteration is independent of the next or previous outer loop iteration, we could try to offset the waiting for the result by issuing a new outer loop iteration. That is, after the last `FIR_MAC` is issued for outer loop iteration 0, we could start the inner loop calculations for outer loop iteration 1 without waiting for the results for outer loop iteration 0 to complete. Because the last `FIR_MAC` takes roughly 21 processor cycles before providing the result, we can issue the next set of Extension Instructions for the next iteration. We are going to execute eight Extension Instructions per outer loop iteration, so it will take roughly  $8 * 3 = 24$  processor cycles for the entire inner loop calculation. Therefore, while the last `FIR_MAC` for iteration 0 is being computed on the ISEF, we will have already



issued Extension Instructions for outer loop iteration 1. At the end of the inner loop calculations for outer loop iteration 1, the result for outer loop 0 (iteration 0) would be available for us to write to memory without any delay. Similarly, while the ISEF is working on instructions for iteration 1, we could start issuing Extension Instructions for iteration 2, and so on. Thus, by pipelining the inner loop calculations in the outer loop, we can remove the extra 18 processor cycles incurred every iteration. At the end of the computation, for the result of the last outer loop iteration, we have to incur the latency because there isn't anything else with which to fill the cycles, but this only happens once per `fir()` invocation. In the overall scheme of things, this overhead would be negligible.

The following section of code (from `fir8b.c`) shows the modified `fir()` for the preceding scenario.

```
/* Include the Stretch Instruction Specific Header */
#include "fir8.h"

#define ST_DECR 1 /* Decrement Indicator */
#define ST_INCR 0 /* Increment Indicator */

/* define macro for the FIR ISEF instruction invocations */
#define FIR(H, X, h, x, t8, y) \
{ \
    int t8m1 = (t8)-1; \
    \
    WRGET0INIT(ST_INCR, (H)) ; \
    (X)++ ; \
    WRGET1INIT(ST_DECR, (X)) ; \
    \
    WRGET0I( &(h), 8 * sizeof(short) ); \
    WRGET1I( &(x), 8 * sizeof(short) ); \
    FIR_MUL( (x), (h), &(y) ); \
    \
    for (t = 1; t < (t8m1); t++) \
    { \
        WRGET0I( &(h), 16 ); \
        WRGET1I( &(x), 16 ); \
        FIR_MAC( (x), (h), &(y) ); \
    } \
    WRGET0I( &(h), 16 ); \
    WRGET1I( &(x), 16 ); \
    FIR_MAC( (x), (h), &(y) ); \
}

/*
 * - FIR using 8 multipliers in ISEF
 * - Loop optimized
 */
void fir(short *X, short *H, short *Y, short N, short T)
{
    int n, t, t8 ;
    WR x, h, y1, y2, y3, y4;

    t8 = T/8 ;

    WRPUTINIT(ST_INCR, Y) ; /* init output stream */
}
```



```

FIR (H, X, h, x, t8, y1) ;    /* x * h + y => y1 */
/* loop ((N/2)-1) times */
n = 0;
do
{
    FIR (H, X, h, x, t8, y2) ;    /* x * h + y => y2 */
    WRPUTI(y1, 2) ;                /* put (y1) result */

    FIR (H, X, h, x, t8, y1) ;    /* x * h + y => y1 */
    WRPUTI(y2, 2) ;                /* put (y2) result */
} while ( ++n < ((N>>1)-1) );
FIR (H, X, h, x, t8, y2) ;    /* x * h + y => y2 */
WRPUTI(y1, 2) ;                /* put (y1) result */
WRPUTI(y2, 2) ;                /* put (y2) result */

WRPUTFLUSH0() ;                /* flush output stream */
WRPUTFLUSH1() ;                /* flush output stream */
}

```

Table 2-2 is an excerpt of the performance statistics for the loop-optimized code showing those functions that use the highest percentage of cycles.

Table 2-2 Loop-optimized performance statistics

%	cumulative cycles	self cycles	calls	self cycles/call	total cycles/call	name
69.36	63348.00	63348.00				ResetH
17.46	79293.00	15945.00	5	3189.00	3189.00	fir
8.20	86778.00	7485.00	1	7485.00	26754.75	main
0.80	87513.00	735.00	1	735.00	2756.75	_vfprintf_r
0.60	88065.00	552.00	1	552.00	727.00	_malloc_r
0.32	88360.00	295.00	1	295.00	295.00	__sinit
0.30	88637.00	277.00	1	277.00	599.75	__sfwwrite

With the changes we just made, the `fir()` function now takes 17.46% of the total cycles, using 3189.00 cycles per call. The changes to the processor code that schedules the Extension Instructions has improved the performance by 1876cycles (5065 – 3189 = 1876). Thus, with minimal effort to change the processor code, we are able to achieve more than a 8x improvement when compared to 27230.60 cycles obtained when running the `fir()` in straight C.



## 2.4 Optimizing with Manual Loop Unrolling

Although we achieved some cycle reductions with loop optimization, we can improve performance further by manually unrolling the inner loop. In the inner loop calculation, we are keeping the inner loop counter as a variable quantity. With this approach, the compiler cannot estimate the number of times the loop will be executed, so we would expect to see branch instructions within the inner loop calculations.

Given that the branch flushes the processor pipeline, they adversely affect the scheduling of the Extension Instructions. This adds extra cycles because the processor must stall if it has already missed the processor cycle on which it can issue an Extension Instruction. There isn't much the compiler can do about this, as the FIR macro is written for a generic number of taps. That is, the example assumes that the same code can be used for a different number of filter taps or input-output samples. Thus, as the compiler expects this to change at runtime, there is not much it can do to unroll this loop.

Because we want to reduce as many cycles as we can for our implementation, we need to start looking at customizing the implementation using the properties of the current algorithm. One thing that can be done readily is to manually unroll the inner loop scheduling of the Extension Instructions. Because we are implementing a 64-tap filter with each instruction doing eight iterations, we only need to write eight Extension Instructions per output. In addition, we can further optimize by making sure the wide registers required for a Extension Instruction are loaded well in advance of the need to use them to ensure that we avoid any load penalties. In addition, because we have 64 16-bit coefficients, we can pre-load eight wide register (128 bits each) before we do any calculations. Thus, while issuing the Extension Instructions, we do not need to worry about also loading the coefficients.

The following section of code (from `fir8c.c`) shows the `fir()` module modified to incorporate manual loop unrolling. It assumes that the data buffer aligns to 16-byte boundaries.

```
/* Include the Stretch Instruction Specific Header */
#include "fir8.h"

#define ST_DECR 1 /* Decrement Indicator */
#define ST_INCR 0 /* Increment Indicator */

#define FIR(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, X) \
{ \
    WRGETOI( &(h1), 8 * sizeof(short) ); \
```



```

    WRGET1I( &(x1), 16 ) ;           \
    X++ ;                           \
    WRGET0I( &(h2), 16 );           \
    WRGET1I( &(x2), 16 ) ;         \
    FIR_MUL( (x1), (h1), &(y1) );\
                                     \
    WRGET0I( &(h3), 16 );           \
    WRGET1I( &(x1), 16 ) ;         \
    FIR_MAC( (x2), (h2), &(y1) );\
                                     \
    WRGET0I( &(h4), 16 );           \
    WRGET1I( &(x2), 16 ) ;         \
    FIR_MAC( (x1), (h3), &(y1) );\
                                     \
    WRGET0I( &(h5), 16 );           \
    WRGET1I( &(x1), 16 ) ;         \
    FIR_MAC( (x2), (h4), &(y1) );\
                                     \
    WRGET0I( &(h6), 16 );           \
    WRGET1I( &(x2), 16 ) ;         \
    FIR_MAC( (x1), (h5), &(y1) );\
                                     \
    WRGET0I( &(h7), 16 );           \
    WRGET1I( &(x1), 16 ) ;         \
    FIR_MAC( (x2), (h6), &(y1) );\
                                     \
    WRGET0I( &(h8), 16 );           \
    WRGET1I( &(x2), 16 ) ;         \
    FIR_MAC( (x1), (h7), &(y1) );\
                                     \
    WRGET1INIT(ST_DECR, X) ;       \
    FIR_MAC( (x2), (h8), &(y1) );\
}
#define FIR1(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) \
{
    WRGET1I( &(x1), 16 ) ;           \
    FIR_MUL( (x1), (h1), &(y2) );\
                                     \
    WRGET1I( &(x1), 16 ) ;           \
    FIR_MAC( (x1), (h2), &(y2) );\
                                     \
    WRGET1I( &(x1), 16 ) ;           \
    FIR_MAC( (x1), (h3), &(y2) );\
                                     \
    WRGET1I( &(x1), 16 ) ;           \
    FIR_MAC( (x1), (h4), &(y2) );\
                                     \
    WRGET1I( &(x1), 16 ) ;           \
    X++ ;                             \
    FIR_MAC( (x1), (h5), &(y2) );\
                                     \
    WRGET1I( &(x1), 16 ) ;           \
    WRGET1I( &(x2), 16 ) ;           \
    FIR_MAC( (x1), (h6), &(y2) );\
                                     \
    WRGET1I( &(x1), 16 ) ;           \
    WRGET1INIT0(ST_DECR, X) ;       \
}

```



```
    FIR_MAC( (x2), (h7), &(y2) );\
    WRGET1INIT1() ;           \
    WRPUTI(y1, 2) ;           \
    FIR_MAC( (x1), (h8), &(y2) );\
}
#define FIR2(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) \
{
    WRGET1I( &(x1), 16 ) ;    \
    FIR_MUL( (x1), (h1), &(y1) );\
    WRGET1I( &(x1), 16 ) ;    \
    FIR_MAC( (x1), (h2), &(y1) );\
    WRGET1I( &(x1), 16 ) ;    \
    FIR_MAC( (x1), (h3), &(y1) );\
    WRGET1I( &(x1), 16 ) ;    \
    FIR_MAC( (x1), (h4), &(y1) );\
    WRGET1I( &(x1), 16 ) ;    \
    X++ ;                      \
    FIR_MAC( (x1), (h5), &(y1) );\
    WRGET1I( &(x1), 16 ) ;    \
    WRGET1I( &(x2), 16 ) ;    \
    FIR_MAC( (x1), (h6), &(y1) );\
    WRGET1I( &(x1), 16 ) ;    \
    WRGET1INIT0(ST_DECR, X) ;  \
    FIR_MAC( (x2), (h7), &(y1) );\
    WRGET1INIT1() ;           \
    WRPUTI(y2, 2) ;           \
    FIR_MAC( (x1), (h8), &(y1) );\
}
/*
 * - FIR using 8 multipliers in ISEF
 * - Loop optimized / Hand unrolled
 */
void fir(short *X, short *H, short *Y, short N, short T)
{
    int n, t, t8 ;
    WR h1, h2, h3, h4, h5, h6, h7, h8 ;
    WR x1, x2;
    WR y1;
    WR y2;
    // (these alternative "register" declarations make no difference:)
    // register WR y1 SE_REG("wra1") ;
    // register WR y2 SE_REG("wra2") ;

    WRPUTINIT(ST_INCR, Y) ;      /* init output stream */
    WRGET0INIT(ST_INCR, H) ;    /* init coefficient stream */
    X++ ;
    WRGET1INIT(ST_DECR, X) ;    /* init input stream */

    /* compute Y[0] in y1 */

```



```

FIR(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, X) ;
/* loop ((N/2)-1) times */
for (n = 0; n < ((N>>1)-1); n++)
{
    /* FIR1 writes previous output (y1) and computes current output (y2) */
    FIR1(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) ;

    /* FIR1 writes previous output (y2) and computes current output (y1) */
    FIR2(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) ;
}
/* compute Y[N-1] in y2 and write Y[N-2] from y1 */
FIR1(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) ;
WRPUTI(y2, 2) ; /* write U[N-1] */
WRPUTFLUSH0() ; /* flush output stream */
WRPUTFLUSH1() ; /* flush output stream */
}

```

Table 2-3 is an excerpt of the performance statistics for the manually unrolled code showing those functions that use the highest percentage of cycles.

Table 2-3 Manually unrolled loop performance statistics

%	cumulative cycles	self cycles	calls	self cycles/call	total cycles/call	name
71.22	63348.00	63348.00				ResetH
15.24	76905.00	13557.00	5	2711.40	2711.40	fir
8.42	84390.00	7485.00	1	7485.00	24366.75	main
0.83	85125.00	735.00	1	735.00	2756.75	_vfprintf_r
0.62	85677.00	552.00	1	552.00	727.00	_malloc_r
0.33	85972.00	295.00	1	295.00	295.00	__sinit
0.31	86249.00	277.00	1	277.00	599.75	__sfvwrite

The `fir()` function now takes only 15.24% of the total cycles, using just 2711.40 cycles per call. With the hand-optimized processor code, we achieved nearly a 10x improvement when compared to 27230.60 cycles obtained when running the `fir()` in straight C.

## 2.5 Recap of Optimization Effort

Table 2-4 recaps what we have done, starting with the straight C implementation. We separated the inner loop into Extension Instructions, which gave us significant improvement; other improvements were achieved by simply manipulating the C code that schedules the Extension Instructions without changing the Extension Instruction itself.



Table 2-4 Recap of optimization improvements

Source	Optimization	% of Cycles	# of Cycles	Performance Improvement over Straight C Code
<i>fir0a</i>	none	64.34	27230.60	–
<i>fir8a</i>	Separated inner loop into Extension Instruction	25.15	5065.00	>5x
<i>fir8b</i>	Shuffled Extension Instruction schedule	17.46	3189.00	>8x
<i>fir8c</i>	Manually unrolled loops	15.24	2711.40	>10x
<i>fir16</i>	Used 16 multipliers	6.96	1128	>24x
<i>fir32</i>	Used 32 multipliers	4.36	687	>39x

# Appendix A      **Code Examples**

This appendix contains the code you need to be able to compile, run, and profile the FIR application. You can find electronic versions of each of these files in the Examples distribution directory.

---

## **A.1      README**

The README file gives you a brief description of Examples distribution directory and the files it contains.

### FILES:

Makefile	build rules
firMain.c	main program and test data
fir0a.c	Generic FIR
fir32.xc	FIR instruction using 32 multipliers
fir32a.c	accelerated FIR using 32 multipliers, not loop optimized
fir32b.c	accelerated FIR using 32 multipliers, loop optimized
fir32c.c	accelerated FIR using 32 multipliers, loop optimized and hand unrolled
fir16.xc	FIR instruction using 16 multipliers
fir16a.c	accelerated FIR using 16 multipliers, not loop optimized
fir16b.c	accelerated FIR using 16 multipliers, loop optimized
fir16c.c	accelerated FIR using 16 multipliers, loop optimized and hand unrolled
fir8.xc	FIR instruction using 8 multipliers
ir8a.c	accelerated FIR using 8 multipliers, not loop optimized
fir8b.c	accelerated FIR using 8 multipliers, loop optimized
fir8c.c	accelerated FIR using 8 multipliers, loop optimized and hand unrolled

This directory contains examples for

- various implementations of FIR using Stretch SCP
- how to compile for Stretch SCP
- how to run Stretch SCP simulator
- how to profile application code

For implementation without using Stretch SCP ISEF:

```
make MUL=0 build run profile
```



For implementation using <N> multipliers without loop optimization:  
 make MUL=<N> VER=a build run profile

For implementation using <N> multipliers with loop optimization:  
 make MUL=<N> VER=b build run profile

For implementation using <N> multipliers with loop optimized and hand unrolled:  
 make MUL=<N> VER=c build run profile

Where <N> = 8, 16, or 32

To run all the versions:  
 make run-all

To profile all the versions:  
 make profile-all

The "profile" sub-directory contains the golden profiler outputs generated by the tools with the release version of the example. These files could be used as a reference for proper installation / execution of the tools.

---

## A.2 Makefile

```
#
#*****\
#*                                     *
#* Copyright 2003 Stretch, Inc. All rights reserved.           *
#*                                     *
#* THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF *
#* STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT *
#* THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.      *
#*                                     *
#*****/
#

GENLSP =

CFLAGS =

ifdef DEBUG
CFLAGS += -g -DDEBUG=$(DEBUG)
else
# CFLAGS += -O2
CFLAGS += -O3
endif

ifdef X86
```



```
CFLAGS += -ms5-native -DSTRETCH_NATIVE=1

else

ifeq ($(ROUTE),0)
ISS_ONLY = -stretch-nobits
endif

# parameter for the Development board (this is okay for s5620 too) ..
ifdef BOARD

CFLAGS += -ms5610
ifdef  OCD
CFLAGS += -mlsp=s56db-ddr
else
GENLSP = st-genlsp -f -ms5610 -e s56db-redboot redboot.lsp
CFLAGS += -mlsp=./redboot.lsp
endif

endif

endif

ifeq ($(OSTYPE),linux)
RM_RECURSE = -r
else
RM_RECURSE = /s
endif

CFLAGS += -DXC_NAME=fir$(MUL)

ifndef VER
VER = a
endif

ifndef MUL
MUL = 0
endif

RUN = st-run

ifneq ($(MUL),0)
SA = fir$(MUL).a
```



```
SH = fir$(MUL).h
endif

default:
    @echo "usage: make [MUL=0|8|16|32] [VER=a|b|c] [X86=1] [DEBUG=1]
asm|build|run|profile|trace"

asm: fir$(MUL)$(VER).s

build: fir$(MUL)$(VER).exe

run: fir$(MUL)$(VER).exe
    $(RUN) $^

profile: fir$(MUL)$(VER).exe
    $(RUN) --mem_model --summary --profile=gmon.out $^
    st-gprof $^ > $^.prof

trace:  fir$(MUL)$(VER).exe
    $(RUN) --trace=6 --mem_model --summary $^ > $^.tr

run-all:
    make clean
    make MUL=0 VER=a run
    make MUL=8  VER=a ROUTE=$(ROUTE) run
    make MUL=8  VER=b ROUTE=$(ROUTE) run
    make MUL=8  VER=c ROUTE=$(ROUTE) run
    make MUL=16 VER=a ROUTE=$(ROUTE) run
    make MUL=16 VER=b ROUTE=$(ROUTE) run
    make MUL=16 VER=c ROUTE=$(ROUTE) run
    make MUL=32 VER=a ROUTE=$(ROUTE) run
    make MUL=32 VER=b ROUTE=$(ROUTE) run
    make MUL=32 VER=c ROUTE=$(ROUTE) run

profile-all:
    make clean
    make MUL=0 VER=a profile
    make MUL=8 VER=a profile
    make MUL=8 VER=b profile
    make MUL=8 VER=c profile
    make MUL=16 VER=a profile
    make MUL=16 VER=b profile
    make MUL=16 VER=c profile
    make MUL=32 VER=a profile
```



```
make MUL=32 VER=b profile
make MUL=32 VER=c profile

fir$(MUL).h fir$(MUL).a: fir$(MUL).xc
    scc $(CFLAGS) -stretch-h fir$(MUL).h -o fir$(MUL).a $(ISS_ONLY) fir$(MUL).xc

firMain.o: firMain.c
    scc $(CFLAGS) -c -o $*.o $*.c

fir$(MUL)$ (VER).o: fir$(MUL)$ (VER).c $(SH)
    scc $(CFLAGS) -c -o $*.o $*.c

fir$(MUL)$ (VER).s: fir$(MUL)$ (VER).c $(SH)
    scc $(CFLAGS) -S $*.c

fir$(MUL)$ (VER).exe: fir$(MUL)$ (VER).o $(SA) firMain.o
    $(GENLSP)
    scc $(CFLAGS) $^ -o $@

clean:
    $(RM) $(RM_RECURSE) stretch-tdk
    $(RM) *.out *.prof *.exe *.s *.o fir8.h fir16.h fir32.h fir*.a
```

---

## A.3 firMain.c

```
/******\
 *
 * Copyright 2003 Stretch, Inc. All rights reserved.
 *
 * THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF
 * STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT
 * THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.
 *
 * THIS SOFTWARE IS CONSIDERED AS "REDISTRIBUTABLES" AND ITS USE IS
 * SUBJECT TO THE TERMS AND CONDITIONS OUTLINED IN STRETCH'S SOFTWARE
 * LICENSE AGREEMENT
 *
 \*****/

#defineFIR_FILTER_LEN      64
#defineFIR_FRAME_SIZE     80
```



```
#ifndef __STRETCH_NATIVE__
#ifndef __STRETCH_S5_ISS__
#include <s5000/sx-isef.h>
#endif
#endif
#include <s5000/sx-timer.h>

#include "firx.h"

#define FIR_NUM_ITERATIONS 5

short __attribute__ ( (aligned (16), section (".dram.data"))) )
golden_out[2*FIR_FRAME_SIZE] =
{
    /* golden output for the 1st frame */
    0, 0, 0, 0,          0, 0, 0, 0,
    0, 0, 0, 0,          0, 0, 0, 0,
    122, 160, 62, 20,    94, 265, 177, -81,
    -50, 274, 438, -108, -653, 181, 2671, 5598,
    8445, 10042, 9617, 8887, 8306, 8171, 7908, 6958,
    6281, 5528, 4903, 4235, 3212, 2577, 1770, 979,
    234, -673, -1261, -2014, -2634, -3116, -3751, -4073,
    -4571, -4908, -5058, -5325, -5238, -5319, -5269, -5079,
    -5070, -4728, -4530, -4187, -3722, -3466, -2945, -2661,
    -2307, -1855, -1614, -1150, -964, -737, -455, -420,

    /* golden output for rest of the frames */
    -161, -180, -180, -161, -420, -455, -737, -964,
    -1150, -1614, -1855, -2307, -2661, -2945, -3466, -3664,
    -4032, -4375, -4673, -5036, -4913, -5023, -5243, -5365,
    -5318, -4722, -4686, -4996, -4586, -2556, 1309, 5309,
    8584, 10036, 9463, 8737, 8284, 8167, 7805, 6859,
    6205, 5505, 4868, 4156, 3175, 2559, 1770, 979,
    234, -673, -1261, -2014, -2634, -3116, -3751, -4073,
    -4571, -4908, -5058, -5325, -5238, -5319, -5269, -5079,
    -5070, -4728, -4530, -4187, -3722, -3466, -2945, -2661,
    -2307, -1855, -1614, -1150, -964, -737, -455, -420
};

/* 16-Bit Interger Input Signal */
short __attribute__ ( (aligned (16), section (".dram.data"))) )
inpSignal[FIR_FRAME_SIZE] =
```



```
{
    10000, 3900, 7500, 7400, 3400, 9200, 2800, 6100, 5700, 1500,
    7000, 400, 3600, 3100, -1100, 4200, -2300, 800, 300, -3800,
    1600, -4700, -1400, -1700, -5700, 000, -6200, -2600, -2600,
    -6400, -300, -6300, -2500, -2300, -5900, 300, -5400, -1400, -1100,
    -4600, 1600, -4000, -100, 100, -3300, 2800, -3000, 800, 900,
    -2700, 3300, -2700, 900, 800, -3000, 2800, -3300, 100, -100,
    -4000, 1600, -4600, -1100, -1400, -5400, 300, -5900, -2300, -2500,
    -6300, -300, -6400, -2600, -2600, -6200, 000, -5700, -1700, -1400
    ,0
} ;

/*
 * 16-bit Integer Filter Coefficients:
 * in some versions of function fir(), the real filter must be aligned,
 * and there must be some zero pads at the front and back.
 */
short __attribute__ ( (aligned (16), section (".dram.data")) )
firCoeff[] =
{
    0, 0, 0, 0, 0, 0, 0, 0, // 16 pads: 13 unused, 3 required.
    0, 0, 0, 0, 0, 0, 0, 0,

    // Real filter starts here == H[0]; 16 zeroes first ..
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    200, 185, -119, -206, 119, 385, 18, -510,
    -225, 661, 639, -780, -1460, 876, 5120, 7290,
    7290, 5120, 876, -1460, -780, 639, 661, -225,
    -510, 18, 385, 119, -206, -119, 185, 200,
    0, 0, 0, 0, 0, 0, 0, 0, // H[48]: first trailing zero.
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0 // H[64 .. 67] pad for 32 MAC ISEF inst.
} ;

int error_count = 0;

static int
check(short actual[], short golden[])
{
    int i, anerr;

    for (i = 0; i < FIR_FRAME_SIZE && error_count < 15; i++)
```



```

    {
    anerr = actual[i] != golden[i];
        error_count += anerr;
    if ( anerr )
        printf( "\nWRONG[%d] = %d (gold %d)!\n", i, actual[i], golden[i] );
    }

    return error_count;
}

/*
 * Array for history and new input.
 * Points [0] through [FIR_FILTER_LEN-2] are history.
 * Point [FIR_FILTER_LEN-1] is the first data point. The first FIR output
 * is computed on the first (FIR_FILTER_LEN) elements of firInput:
 * pre-history and the first new data sample. Alignment is required
 * on this first block of data in some versions of function fir().
 *
 * (total array dimension is 4 extra, then forced to a multiple of 8)
 */
short __attribute__ ( (aligned (16), section (".dram.data"))) )
    firInput[ ( (FIR_FILTER_LEN-1 + FIR_FRAME_SIZE +4 +7) & ~7 ) ];

/*
 * Array for output (size forced to a multiple of 8) ..
 */
short __attribute__ ( (aligned (16), section (".dram.data"))) )
    firOutput[ (FIR_FRAME_SIZE +7) & ~7 ];

/*****\
    Prototypes
\*****/

void fir( short *inSignal, short filterCoeff[], short outSignal[],
        short frameSize, short filterLen ) ;

#ifndef __STRETCH_NATIVE__
#ifndef __STRETCH_S5_ISS__
/*
 * This is not necessary for this test to simply function correctly; without
 * this the ISEF configuration will load automatically upon using the first
 * instruction.
 *
 */

```



```
* BUT for an accurate profile of the main loop cycle timing, we want the ISEF
* loaded and ready before we begin the application. So we explicitly load
* the ISEF and then wait, enough time for the load to complete.
*/
int load_my_fig( sx_isef i_unit, char *string )
{
    int err;

    // printf( "\nLOADING ISEF %d cfg now: \"%s\".\n\n", i_unit, string );
    err = sx_isef_load_by_name_async( i_unit, string );

    // 200 microseconds is plenty of time for ISEF to load ..
    sx_delay_us(200);

    return err;
}
#endif
#endif

int
main( int argc, char *argv[] )
{
    int i, j, err;
    short *ldAddr, *stAddr ;

#ifdef __STRETCH_NATIVE__
#ifdef __STRETCH_S5_ISS__

    printf( "starting now.\n" );
    /* Load the ISEF */
    err = load_my_fig( sx_isef_a, isef_a_name );
    if (err)
    {
        printf( "\nERROR %d from Config Load ISEF 1.\n\n", err );
        while (1); // Spin in an endless loop so we can break here and debug
    }
    printf( "\n - - CONFIGURATION LOADED. - -\n" );

#endif
#endif

    #if 0/* This would be needed if there is another ISEF configuration: */
    err = load_my_fig( sx_isef_b, isef_b_name );
    if (err)
    {

```



```
        printf( "\nERROR %d from Config Load ISEF 2.\n\n", err );
        while (1); // Spin in an endless loop so we can break here and debug
    }
    printf( "\n - - CONFIGURATION B LOADED. - -\n" );
#endif

#endif

#endif

    error_count = 0;

    /* Zero the History Buffer
    */
    for(i = 0; i < FIR_FILTER_LEN-1; i++)
    {
        firInput[i] = 0 ;
    }

    // Update firInput[] with input samples
    for(j = 0; j < FIR_FRAME_SIZE; j++)
    {
        firInput[FIR_FILTER_LEN -1 + j] = inpSignal[j];
    }

    /* Main Loop */
    for (i = 0; i < FIR_NUM_ITERATIONS; i++)
    {
        fir( &firInput[FIR_FILTER_LEN-1], &firCoeff[16], firOutput,
            FIR_FRAME_SIZE, FIR_FILTER_LEN );

        // printf( " iter %d,\n", i+1 );
        if (err = check(firOutput, &golden_out[(i == 0) ? 0 : FIR_FRAME_SIZE]))
            break;

    /*
    * Update the history buffer for next frame processing
    * (moves (FIR_FILTER_LEN -1) samples to front)
    */
    for(j = 0; j < FIR_FILTER_LEN-1; j++)
    {
        firInput[j] = firInput[FIR_FRAME_SIZE + j] ;
    }
    }
```



```
    if (err)
printf("FAILED\n");
    else
printf("PASSED\n");

    return 0 ;
}
```

---

## A.4 fir0a.c

```
/*
 *
 * Copyright 2003 Stretch, Inc. All rights reserved.
 *
 * THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF
 * STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT
 * THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.
 *
 * THIS SOFTWARE IS CONSIDERED AS "REDISTRIBUTABLES" AND ITS USE IS
 * SUBJECT TO THE TERMS AND CONDITIONS OUTLINED IN STRETCH'S SOFTWARE
 * LICENSE AGREEMENT
 *
 */
*****\

/*
 * Generic FIR
 *   X - inputs
 *   H - coefficients
 *   Y - outputs
 *   N - Number of samples
 *   T - Number of tabs
 */
void fir(short *X, short *H, short *Y, int N, int T)
{
    int n, t, acc;
    short *x, *h;

    /* Filter Input */
    for (n = 0; n < N; n++) {
        x = X;
        h = H;
```



```

acc = (*x--) * (*h++);

for(t = 1; t < T; t++) {
    acc += (*x--) * (*h++);
}

*Y = acc >> 14;
X++;
Y++;
}
}

```

---

## A.5 fir8.xc

```

/*****\
*
* Copyright 2003 Stretch, Inc. All rights reserved.
*
* THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF
* STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT
* THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.
*
* THIS SOFTWARE IS CONSIDERED AS "REDISTRIBUTABLES" AND ITS USE IS
* SUBJECT TO THE TERMS AND CONDITIONS OUTLINED IN STRETCH'S SOFTWARE
* LICENSE AGREEMENT
*
\*****/

#include <stretch.h>

static se_sint<32> acc;      /* cost one 128-bit state resource */

/* Performs 8 parallel MAC */
SE_FUNC void
firFunc(SE_INST FIR_MUL, SE_INST FIR_MAC, WR X, WR H, WR *Y)
{
    se_sint<16> x, h ;
    se_sint<32> sum ;
    int i ;

    /*

```



```

    * The loop will be unrolled 8 times and the computations readjusted to
    * perform the 8 multiplications in parallel, followed by multiple stages
    * of addition using 7 32-bit adders.
    */
    sum = 0;
    for(i = 0; i < 128; i += 16) {
    h = H(i + 15, i);          /* no cost */
    x = X(127-i, 112-i);      /* no cost */
    sum += x * h ;           /* cost 16x16x8 MUs & 32x7 AUs */
    }

    /* cost 32 AUs */
    acc = FIR_MAC ? se_sint<32> (sum + acc) : sum;

    /* no cost */
    *Y = acc >> 14 ;
}

```

---

## A.6 fir8a.c

```

/*****\
*
* Copyright 2003 Stretch, Inc. All rights reserved.
*
* THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF
* STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT
* THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.
*
* THIS SOFTWARE IS CONSIDERED AS "REDISTRIBUTABLES" AND ITS USE IS
* SUBJECT TO THE TERMS AND CONDITIONS OUTLINED IN STRETCH'S SOFTWARE
* LICENSE AGREEMENT
*
\*****/

/* Include the Stretch Instruction Specific Header */
#include "fir8.h"

#define ST_DECR1    /* Decrement Indicator */
#define ST_INCR0    /* Increment Indicator */

char isef_a_name[] = "fir8";
// char isef_b_name[] = "fir8_2"; /* Use this if there is a 2nd ISEF config. */

```



```
/*
 * - FIR using 8 multipliers in ISEF
 * - Not loop optimized
 */
void fir(short *X, short *H, short *Y, short N, short T)
{
    int n, t, t8;
    WR x, h, y;

    t8 = T/8;

    WRPUTINIT(ST_INCR, Y) ;           /* init output stream */

    for (n = 0; n < N; n++)
    {
        WRGET0INIT(ST_INCR, H) ;      /* init coefficient stream */
        X++ ;                          /* adj input ptr for DECR usage */
        WRGET1INIT(ST_DECR, X) ;      /* init input stream */

        WRAGET0I( &h, 16 );           /* get 8 coefficients */
        WRAGET1I( &x, 16 );           /* get 8 inputs */
        FIR_MUL(x, h, &y);             /* x * h => y */

        for (t = 1; t < t8; t++)
        {
            WRAGET0I(&h, 16);
            WRAGET1I(&x, 16);
            FIR_MAC(x, h, &y);         /* x * h + y => y */
        }
        WRPUTI(y, 2) ;                 /* write result */
    }

    WRPUTFLUSH0() ;                    /* flush 0 */
    WRPUTFLUSH1() ;                    /* flush 1 */
}
```



## A.7 fir8b.c

```
/* *****\
 *
 * Copyright 2003 Stretch, Inc. All rights reserved.
 *
 * THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF
 * STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT
 * THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.
 *
 * THIS SOFTWARE IS CONSIDERED AS "REDISTRIBUTABLES" AND ITS USE IS
 * SUBJECT TO THE TERMS AND CONDITIONS OUTLINED IN STRETCH'S SOFTWARE
 * LICENSE AGREEMENT
 *
 * *****/

/* Include the Stretch Instruction Specific Header */
#include "fir8.h"

#define ST_DECR1 /* Decrement Indicator */
#define ST_INCR0 /* Increment Indicator */

char isef_a_name[] = "fir8";
// char isef_b_name[] = "fir8_2"; /* Use this if there is a 2nd ISEF config. */

/* define macro for the FIR ISEF instruction invocations */
#define FIR(H, X, h, x, t8, y) \
{ \
    int t8m1 = (t8)-1; \
    \
    WRGET0INIT(ST_INCR, (H)) ; \
    (X)++ ; \
    WRGET1INIT(ST_DECR, (X)) ; \
    \
    WRGET0I( &(h), 8 * sizeof(short) ); \
    WRGET1I( &(x), 8 * sizeof(short) ); \
    FIR_MUL( (x), (h), &(y) ); \
    \
    for (t = 1; t < (t8m1); t++) \
    { \

```



```
        WRGETOI( &(h), 16 );          \
        WRGET1I( &(x), 16 );         \
        FIR_MAC( (x), (h), &(y) );   \
    }                                  \
    WRGETOI( &(h), 16 );             \
    WRGET1I( &(x), 16 );             \
    FIR_MAC( (x), (h), &(y) );       \
}

/*
 * - FIR using 8 multipliers in ISEF
 * - Loop optimized
 */
void fir(short *X, short *H, short *Y, short N, short T)
{
    int n, t, t8 ;
    WR x, h, y1, y2, y3, y4;

    t8 = T/8 ;

    WRPUTINIT(ST_INCR, Y) ;          /* init output stream */

    FIR (H, X, h, x, t8, y1) ;       /* x * h + y => y1 */

    /* loop ((N/2)-1) times */
    n = 0;
    do
    {
        FIR (H, X, h, x, t8, y2) ;   /* x * h + y => y2 */
        WRPUTI(y1, 2) ;              /* put (y1) result */

        FIR (H, X, h, x, t8, y1) ;   /* x * h + y => y1 */
        WRPUTI(y2, 2) ;              /* put (y2) result */
    } while ( ++n < ((N>>1)-1) );

    FIR (H, X, h, x, t8, y2) ;       /* x * h + y => y2 */
    WRPUTI(y1, 2) ;                  /* put (y1) result */
    WRPUTI(y2, 2) ;                  /* put (y2) result */

    WRPUTFLUSH0() ;                  /* flush output stream */
    WRPUTFLUSH1() ;                  /* flush output stream */
}

```



## A.8 fir8c.c

```
/* *****\
 *
 * Copyright 2003 Stretch, Inc. All rights reserved.
 *
 * THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF
 * STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT
 * THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.
 *
 * THIS SOFTWARE IS CONSIDERED AS "REDISTRIBUTABLES" AND ITS USE IS
 * SUBJECT TO THE TERMS AND CONDITIONS OUTLINED IN STRETCH'S SOFTWARE
 * LICENSE AGREEMENT
 *
\*****/

/* Include the Stretch Instruction Specific Header */
#include "fir8.h"

#define ST_DECR1 /* Decrement Indicator */
#define ST_INCR0 /* Increment Indicator */

char isef_a_name[] = "fir8";
// char isef_b_name[] = "fir8_2"; /* Use this if there is a 2nd ISEF config.
*/

#define FIR(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, X) \
{
    WRGET0I( &(h1), 8 * sizeof(short) ); \
    WRGET1I( &(x1), 16 ) ; \
    X++ ; \
    WRGET0I( &(h2), 16 ); \
    WRGET1I( &(x2), 16 ) ; \
    FIR_MUL( (x1), (h1), &(y1) ); \
    \
    WRGET0I( &(h3), 16 ); \
    WRGET1I( &(x1), 16 ) ; \
    FIR_MAC( (x2), (h2), &(y1) ); \
    \
    WRGET0I( &(h4), 16 ); \
    \
}
```



```
    WRGET1I( &(x2), 16 ) ;          \
    FIR_MAC( (x1), (h3), &(y1) ); \
                                     \
    WRGET0I( &(h5), 16 );          \
    WRGET1I( &(x1), 16 ) ;          \
    FIR_MAC( (x2), (h4), &(y1) ); \
                                     \
    WRGET0I( &(h6), 16 );          \
    WRGET1I( &(x2), 16 ) ;          \
    FIR_MAC( (x1), (h5), &(y1) ); \
                                     \
    WRGET0I( &(h7), 16 );          \
    WRGET1I( &(x1), 16 ) ;          \
    FIR_MAC( (x2), (h6), &(y1) ); \
                                     \
    WRGET0I( &(h8), 16 );          \
    WRGET1I( &(x2), 16 ) ;          \
    FIR_MAC( (x1), (h7), &(y1) ); \
                                     \
    WRGET1INIT(ST_DECR, X) ;        \
    FIR_MAC( (x2), (h8), &(y1) ); \
}

#defineFIR1(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) \
{
    WRGET1I( &(x1), 16 ) ;          \
    FIR_MUL( (x1), (h1), &(y2) ); \
                                     \
    WRGET1I( &(x1), 16 ) ;          \
    FIR_MAC( (x1), (h2), &(y2) ); \
                                     \
    WRGET1I( &(x1), 16 ) ;          \
    FIR_MAC( (x1), (h3), &(y2) ); \
                                     \
    WRGET1I( &(x1), 16 ) ;          \
    FIR_MAC( (x1), (h4), &(y2) ); \
                                     \
    WRGET1I( &(x1), 16 ) ;          \
    X++ ;                             \
    FIR_MAC( (x1), (h5), &(y2) ); \
                                     \
    WRGET1I( &(x1), 16 ) ;          \
    WRGET1I( &(x2), 16 ) ;          \
}
```



```
FIR_MAC( (x1), (h6), &(y2) ); \
\
WRGET1I( &(x1), 16 ) ; \
WRGET1INIT0(ST_DECR, X) ; \
FIR_MAC( (x2), (h7), &(y2) ); \
\
WRGET1INIT1() ; \
WRPUTI(y1, 2) ; \
FIR_MAC( (x1), (h8), &(y2) ); \
}

#defineFIR2(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) \
{
\
WRGET1I( &(x1), 16 ) ; \
FIR_MUL( (x1), (h1), &(y1) ); \
\
WRGET1I( &(x1), 16 ) ; \
FIR_MAC( (x1), (h2), &(y1) ); \
\
WRGET1I( &(x1), 16 ) ; \
FIR_MAC( (x1), (h3), &(y1) ); \
\
WRGET1I( &(x1), 16 ) ; \
FIR_MAC( (x1), (h4), &(y1) ); \
\
WRGET1I( &(x1), 16 ) ; \
X++ ; \
FIR_MAC( (x1), (h5), &(y1) ); \
\
WRGET1I( &(x1), 16 ) ; \
WRGET1I( &(x2), 16 ) ; \
FIR_MAC( (x1), (h6), &(y1) ); \
\
WRGET1I( &(x1), 16 ) ; \
WRGET1INIT0(ST_DECR, X) ; \
FIR_MAC( (x2), (h7), &(y1) ); \
\
WRGET1INIT1() ; \
WRPUTI(y2, 2) ; \
FIR_MAC( (x1), (h8), &(y1) ); \
}

/*
```



```
* - FIR using 8 multipliers in ISEF
* - Loop optimized / Hand unrolled
*/
void fir(short *X, short *H, short *Y, short N, short T)
{
    int n, t, t8 ;
    WR h1, h2, h3, h4, h5, h6, h7, h8 ;
    WR x1, x2;
    WR y1;
    WR y2;

    WRPUTINIT(ST_INCR, Y) ;           /* init output stream */

    WRGET0INIT(ST_INCR, H) ;         /* init coefficient stream */
    X++ ;
    WRGET1INIT(ST_DECR, X) ;        /* init input stream */

    /* compute Y[0] in y1 */
    FIR(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, X) ;

    /* loop ((N/2)-1) times */
    for (n = 0; n < ((N>>1)-1); n++)
    {
        /* FIR1 writes previous output (y1) and computes current output (y2) */
        FIR1(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) ;

        /* FIR1 writes previous output (y2) and computes current output (y1) */
        FIR2(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) ;
    }
    /* compute Y[N-1] in y2 and write Y[N-2] from y1 */
    FIR1(h1, h2, h3, h4, h5, h6, h7, h8, x1, x2, y1, y2, X) ;
    WRPUTI(y2, 2) ;                 /* write U[N-1] */

    WRPUTFLUSH0() ;                 /* flush output stream */
    WRPUTFLUSH1() ;                 /* flush output stream */
}
```



## A.9 fir16.xc

```
/* *****\
 *
 * Copyright 2003 Stretch, Inc. All rights reserved.
 *
 * THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF
 * STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT
 * THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.
 *
 * THIS SOFTWARE IS CONSIDERED AS "REDISTRIBUTABLES" AND ITS USE IS
 * SUBJECT TO THE TERMS AND CONDITIONS OUTLINED IN STRETCH'S SOFTWARE
 * LICENSE AGREEMENT
 *
 * *****/

#include <stretch.h>

/* 4 partial sums for 4 outputs per call */
static se_sint<32> acc[4] ;

/* Performs 16 parallel MAC */
SE_FUNC void
firFunc(SE_INST FIR_MUL, SE_INST FIR_MAC, WRB X, WRA H, WRB *Y)
{
    se_sint<16> x, h, res[4];
    se_sint<32> sum ;
    int i, j ;

    /* Requires 4 coefficients / 7 data points to produce 4 outputs */
    for(i = 0; i < 4; i++) {
        /* loop per # of outputs */
        sum = 0;
        for(j = 0; j < 4; j++) {
            /* 4 MACs per output */
            h = H(63-(j*16), 48-(j*16)) ; /* coeff[] */
            x = X >> ( (j*16) + (i*16) ); /* data[] */
            sum += x * h ;
        }
        acc[i] = FIR_MUL ? sum : se_sint<32> (sum + acc[i]) ;
        res[i] = acc[i] >> 14 ;
    }
    *Y = (res[3], res[2], res[1], res[0]); /* concatenate outputs */
}
```



}

---

## A.10 fir32.xc

```
/*\n * Copyright 2003 Stretch, Inc. All rights reserved. *\n * THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF *\n * STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT *\n * THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC. *\n *\n * THIS SOFTWARE IS CONSIDERED AS "REDISTRIBUTABLES" AND ITS USE IS *\n * SUBJECT TO THE TERMS AND CONDITIONS OUTLINED IN STRETCH'S SOFTWARE *\n * LICENSE AGREEMENT *\n */\n\n#include <stretch.h>\n\n#include "firx.h"\n\ntypedef se_uint<8> u8;\ntypedef se_uint<16> u16;\n\n/*\n * FIR ISEF Instruction doing 32 multiplies.\n * Computes 8 partial sums Y[j] of products for 8 outputs per call:\n * Inputs are one set of 4 filter values H[], 11 data samples X[i]\n * thru X[i+10].\n * multiplies the 4 H[i] by X[j+i], advancing j for each output.\n */\n\n// 8 partial sums carried across for next block of X ..\nstatic se_sint<32> sum[8]; // old Y[3] thru [10]\n\nstatic se_sint<16> oldx[8]; // old X[4] thru [11] is new [0] thru [7].\n\n/*\n * Performs 32 parallel MAC:\n * FIR_INIT_MUL: Start new sums of multiplies and also provide initial 8 X.\n * FIR_MAC: successive multiply-and-accumulate.
```



```
*
* Forms 8 partial sums Y of products H[i]*X[j], for 4 values of filter H,
* by 8 data samples X; the set of X is shifted 1 for each H (see below).
*
* In: WR newx(63,0) is new X[11] thru [8],
*     WR H(63,0) is H[3] thru [0],
*     WR Y (FIR_INIT_MUL only): X[7] thru [0].
* State:
*     sum: sums Y[3] thru [10],
*     oldx: old X[0] thru [7].
*
* Out: WR Y has Y[10] thru [3], 16-bit, each shifted right 14,
*     State:
*     sum: sums Y[3] thru [10] (32 bit),
*     oldx: old X[4] thru [11] in state.
*
* Adds to accumulated Y[3] thru [10], by going across 8 X for each
* H from 3 down thru 0, picking the X block 1 to the right for each
* next lower H, so that:
*     for j=3 thru 0
*         for i=3-j thru 10-j
*             Y[i+j] += X[i] * H[j]
*/
SE_FUNC void fir_macs( SE_INST FIR_MAC, SE_INST FIR_INIT_MUL,
                      WRB newx, WRA H, WRB *Y )
{
    se_sint<16> x[12], h, res[8];
    int ii, i, j = 0;

    // For _INIT: Initialize FIR: Zero the 8 Y totals, take in the first 8 X:
    if ( FIR_INIT_MUL )
    {
        for ( i = 0; i < 8; i++ )
            oldx[i] = *Y >>(16*i);
    }

    for ( i = 0; i < 8; i++ )
        x[i] = oldx[i];
    for ( i = 0; i < 4; i++ )
        x[i+8] = newx >>(i*16);

#ifdef DBNATIVE
    printf( "\n      ISEF H: " );
#endif
#endif
```



```
    for( ii = 0; ii < 4; ii++ )
    {
        i = 3 -ii;                // for H from 3 thru 0,
        h = H >>(i*16);          // next H[i], i going down,
#ifdef (DBNATIVE >= 2)
        printf( " %d", integer(h) );
#endif

        // X block is 1 farther right for each H[i] ..
        for(j = 0; j < 8; j++)
        {
            if ( FIR_INIT_MUL )
                sum[j] = 0;      // Init: clear sums.

            sum[j] += x[3 -i +j] * h; // sum of products, in state.
        }

        for ( i = 0; i < 8; i++ )
        {
            oldx[i] = x[i+4];    // keep old X.
            res[i] = sum[i] >>14; // output sum so far.
        }
#ifdef (DBNATIVE >= 2)
        printf( ",\n      ISEF Y: %8x %8x %8x %8x %8x %8x %8x %8x.\n",
            integer(sum[0]), integer(sum[1]), integer(sum[2]), integer(sum[3]),
            integer(sum[4]), integer(sum[5]), integer(sum[6]), integer(sum[7]) );
#endif
    }

    /* Output 8 sums [3] .. [10]: */
    *Y = ( res[7], res[6], res[5], res[4],
          res[3], res[2], res[1], res[0] );
}
```



## A.11 firx.h

```
/*
 * Copyright 2003 Stretch, Inc. All rights reserved.
 *
 * THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF
 * STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT
 * THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.
 */
\*****\

#ifndef _FIRX_H_
#define _FIRX_H_ 1

#ifdef DEBUG
#ifdef STRETCH_NATIVE

#define DBNATIVE DEBUG /* Def == Native and Debug; == DEBUG value. */

#endif
#endif

/*
 * countof(array) is number of elements in the array ..
 */
#define countof(ray) (sizeof(ray)/sizeof(ray[0]))

#define align16 __attribute__ ( (aligned (16)) )

// Name of the ISEF configs (actual is in the firNNV.c files) ..
extern char isef_a_name[];
// extern char isef_b_name[]; /* Use this if there is a 2nd ISEF config. */

#endif /* _FIRX_H_ */
```