Optimizing Code Speed for the MaverickCrunch™ Coprocessor

Brett Davis

1. Introduction

This application note is intended to assist developers in optimizing their source code for use with the MaverickCrunch coprocessor. This document begins with a brief overview of the MaverickCrunch coprocessor, followed by optimization guidelines and concludes with an example applying the guidelines discussed.

Multiple facets of code optimization must be considered in order to realize the full benefit of the MaverickCrunch coprocessor. The guidelines in this document are categorized as algorithm, compiler, or hardware optimizations. The discussion on algorithm optimization centers on high level programming details such as compound expressions and loop unrolling. Next, the compiler optimization guidelines deal with the effects of compiler optimization on code performance - primarily code size and execution speed. Finally, the hardware optimization section enumerates optimization guidelines related to the MaverickCrunch coprocessor implementation such as IEEE-754 implementation and pipeline stalls.

Note: Algorithm selection will not be discussed in this applications note. It is assumed that the developer has selected and implemented the correct algorithm for their application.

2. MaverickCrunch

This section introduces and summarizes the features, instruction set and architecture of the MaverickCrunch coprocessor. For further in-depth information on these topics, please read Chapter 3 of the User's Guide.

2.1 Features

The MaverickCrunch coprocessor accelerates IEEE-754 floating point arithmetic, and 32-bit and 64-bit fixed point arithmetic. The MaverickCrunch coprocessor is an excellent candidate for encoding and decoding digital audio, digital signal processing (such as IIR, FIR, FFT) and numeric approximations. Key features of the MaverickCrunch include:

- IEEE-754 based single and double precision floating point support
- Full IEEE-754 rounding support
- Inexact, Overflow, Underflow, and Invalid Operator IEEE-754 exceptions
- 32/64-bit fixed point integer operations
- Add, multiply, and compare functions for all data types
- Fixed point integer MAC 32-bit input with 72-bit accumulate
- Fixed point integer shifts

- Conversion between floating point and integer data representations
- Sixteen (16) 64-bit general-purpose registers
- Four (4) 72-bit accumulators
- Status and control registers

2.2 Instruction Set

The MaverickCrunch coprocessor's instruction set is robust and includes memory, control, and arithmetic operations. MaverickCrunch mnemonics are translated by the compiler or assembler into ARM coprocessor instructions. For example, the MaverickCrunch mnemonic for double precision floating-point multiply is:

```
cfmuld c0, c1, c2
```

The equivalent ARM coprocessor instruction is:

```
cdp p4, 1, c0, c1, c2, 1
```

There are five categories of ARM coprocessor instructions: Data Path (CDP), Load (LDC), Store (STC), Coprocessor to ARM Moves (MCR), and ARM to coprocessor moves (MRC). CDP instructions include all arithmetic operations, and any other operation internal to the coprocessor. LDC and STC instructions include the set of operations responsible for moving data between memory and the coprocessor. MCR and MRC instructions are responsible for moving data between ARM and coprocessor registers.

Table 1, Table 2 and Table 3 summarize all of the MaverickCrunch's instruction mnemonics. For more information on the MaverickCrunch instruction set, please see the table: *MaverickCrunch Instruction Set* in the User's Guide.

Table 1. MaverickCrunch Load/Store Mnemonics

cfldrs Cd, [Rn]	cfldrd Cd, [Rn]	cfldr32 Cd, [Rn]
cfldr64 Cd, [Rn]	cfstrs Cd, [Rn]	cfstrd Cd, [Rn]
cflstr32 Cd, [Rn]	cfstr64 Cd, [Rn]	cfmvsr Cn, Rd
cfmvdlr Cn, Rd	cfmvdhr Cn, Rd	cfmv64lr Cn, Rd
cfmv64hr Cn, Rd	cfmvsr Rd, Cn	cfmvrdl Rd, Cn
cfmvrdh Rd, Cn	cfmvr64l Rd, Cn	cfmvr64h Rd, Cn
cfmval32 Cd, Cn	cfmvam32 Cd, Cn	cfmv32a Cd, Cn
cfmv64a Cd, Cn	cfmvsc32 Cd, Cn	cfmv32sc Cd, Cn
cfcpys Cd, Cn	cfcpyd Cd, Cn	

Table 2. MaverickCrunch Data Manipulation Mnemonics

cfcvtsd Cd, Cn	cfcvtds Cd, Cn	cfcmp64 Rd, Cn, Cm
cfcvt32d Cd, Cn	cfcvt64s Cd, Cn	cfcvt32s Cd, Cn
cfcvts32 Cd, Cn	cfcvtd32 Cd, Cn	cfcvt64d Cd, Cn
cfrshl32 Cm, Cn, Rd	cftruncs32 Cd, Cn	cftruncd32 Cd, Cn
cfsh64 Cd, Cn, <imm>	cfrshl64 Cm, Cn, Rd	cfsh32 Cd, Cn, <imm>
cfcmp32 Rd, Cn, Cm	cfcmps Rd, Cn, Cm	cfcmpd Rd, Cn, Cm

Table 3. MaverickCrunch Arithmetic Mnemonics

cfabss Cd, Cn	cfnegs Cd, Cn	cfadds Cd, Cn, Cm
cfsubs Cd, Cn, Cm	cfnegd Cd, Cn	cfaddd Cd, Cn, Cm
cfsubd Cd, Cn, Cm	cfmuld Cd, Cn, Cm	cfabs32 Cd, Cn
cfadd64 Cd, Cn, Cm	cfneg32 Cd, Cn	cfadd32 Cd, Cn, Cm
cfsub32 Cd, Cn, Cm	cfmul32 Cd, Cn, Cm	cfmac32 Cd, Cn, Cm
cfmsc32 Cd, Cn, Cm	cfabs64 Cd, Cn	cfneg64 Cd, Cn
cfsub64 Cd, Cn, Cm	cfmul64 Cd, Cn, Cm	cfmadd32 Ca, Cd, Cn, Cm
cfmsub32 Ca, Cd, Cn, Cm	cfmadda32 Ca, Cd, Cn, Cm	cfmsuba32 Ca, Cd, Cn, Cm
cfmuls Cd, Cn, Cm	cfabsd Cd, Cn	

2.3 Architecture

The MaverickCrunch coprocessor uses the standard ARM coprocessor interface, sharing its memory interface and instruction stream. The MaverickCrunch coprocessor is pipelined, has data forwarding capabilities, and can run synchronously or asynchronously with respect to the ARM920T pipeline.

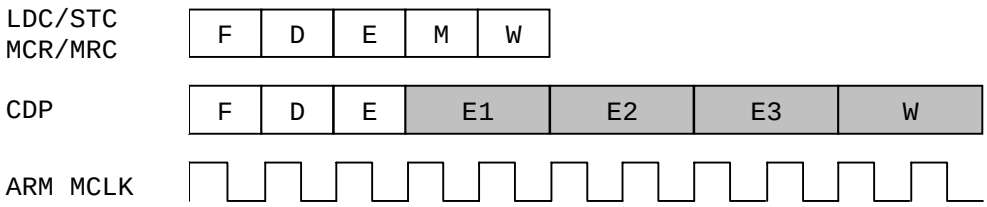
There are two separate pipelines in the MaverickCrunch coprocessor (see Figure 1). The first pipeline, five stages long, is used for LDC, STC, MCR, and MRC instructions. Its stages are Fetch (F), Decode (D), Execute (E), Memory Access (M), and Register Write-Back (W). The second pipeline, seven stages long, is used for the CDP instructions. Its stages are Fetch (F), Decode (D), Execute/Operand Fetch (E), Execute (E1), Execute (E2), Execute (E3), and Register Write-Back (W).

The MaverickCrunch LDC/STC/MCR/MRC pipeline is identical to, and 'follows' the ARM920T's pipeline. That is, the contents of the LDC/STC/MCR/MRC pipeline are identical to the contents of the ARM920T pipeline.

Note: The ARM pipeline is not shown in Figure 1, but is identical to the LDC/STC/MCR/MRC pipeline.

The MaverickCrunch CDP pipeline is nearly twice as deep and runs at half the speed of the ARM920T's pipeline. The CDP pipeline may run asynchronously with respect to the ARM920T's pipeline after the initial execution stage. Specifically, the CDP pipeline may run asynchronously in the E1, E2, E3 and W stages. Running MaverickCrunch in synchronous mode forces the CDP pipeline to serialize the instruction stream resulting in an eight-cycle stall per data path instruction. The CDP pipeline's asynchronous capable stages are shaded in Figure 1.

Figure 1. MaverickCrunch Pipelines



3. Code Optimization for MaverickCrunch

This section describes guidelines for writing optimized code for the MaverickCrunch coprocessor. These guidelines are divided into algorithm, compiler and architecture sections. It is assumed that the correct algorithm has been chosen, and that all non-hardware specific optimizations have been completed. **However, optimization should not begin until all of the code has been written and tested for functionality.**

3.1 Algorithms

This section focuses on methods to reduce algorithm execution time. After the code's functionality is verified, profile and disassemble the objects. Look for and optimize the following:

- Sections of code that are executed most frequently
- Sections of code that take the most CPU cycles to execute
- Inefficiencies in assembly code from the compilation

When optimizing these sections keep in mind the following general concepts of code optimization:

- Avoid Redundancy - store computations rather than recomputing them
- Serialize Code - code should be designed with a minimum amount of branching. Code branching is expensive. The ARM920T does *not* support branch prediction
- Code Locality - code executed closely together in time should be placed closely together in memory, increasing spatial locality of reference and reducing expensive cache misses

Unless your goal is to create small code, code density is not always an indicator of code optimization. **Loop Unrolling** is an optimization technique that generally increases code size, but also increases code speed. This is because unrolled loops iterate fewer times than their unoptimized versions resulting in fewer index calculations, comparisons and branches taken.

Note: Taken branches are expensive operations, as they take three cycles to complete and cause the pipeline to be flushed. (There is no branch prediction in the ARM920T.)

Induction Variable Analysis is another speed optimization technique used in the case where a variable in a loop is a function of the loop index. This variable can be updated each time the index is updated, reducing the number of calculations in the loop.

Although not a loop optimization technique, **Common Sub-Expression Elimination** is a speed optimization technique that can be used in the case where a complex calculation has redundant sub-expressions. In this technique, these redundant sub-expressions are calculated once, stored and reused when needed.

Constant Folding is a speed optimization technique used in cases where constant expressions are replaced with their final value, rather than calculating these at run-time.

Various other algorithm techniques exist. Each optimization has trade-offs, and should be selected based on the algorithm's application. For further information, please consult one of the many available books on algorithm optimization.

3.2 Compilers

This section describes general optimization issues that arise when compiling for the ARM and MaverickCrunch coprocessor. Every compiler is different in how it implements and optimizes code. Most compilers consider the following types of optimizations:

- Peephole - combining several instructions into one simple instruction
- Local - analyzing, reordering and optimizing instructions in a basic block (serial code) for faster execution
- Loop - reduces time spent in loops by applying basic loop optimizations
- Intraprocedural - optimizes the way control and data are passed between procedures

The following guidelines will generally help the compiler produce optimal ARM and MaverickCrunch code.

Set the appropriate target processor for the compiler, linker and assembler. This would be the ARM920T. Furthermore, set the correct floating-point support for the compilation. This may either be hardware floating-point support through the MaverickCrunch coprocessor, soft-float support through a floating-point library, or no floating-point support.

Turn off all debug options. Compiling source code with debugging enabled generates un-optimized objects. These objects are typically larger, and slower than those compiled without the debugging options selected. The debugging option forces the compiler to not use all optimization techniques because some intermediate debug data is lost in the process. Furthermore, the objects are bloated with symbol data for the debugging tool.

Set the compiler to optimize for code speed. Be sure to experiment with the optimization modes of the compiler, because some optimizations may have negative unintended effects. For instance, some intermediate results may be optimized out of the code whether these results are needed or not. Please note that some compilers will optimize code for speed or size better than other compilers.

3.3 Architecture

The following guidelines are specific to the MaverickCrunch hardware. **The MaverickCrunch should be running in asynchronous mode with data forwarding enabled to obtain the highest pipeline throughput.** Additionally, the ARM920T's master clock should be set to run at the highest allowed frequency. Please see the example of the initialization source code in the User’s Guide.

Avoid single or double floating-point division. These operations are not implemented in the MaverickCrunch and will be carried out by a soft-float library routine. When possible, calculate the reciprocal during compile-time and multiply instead. Be vigilant of compounding rounding errors from calculating the reciprocal, and repeated multiplication of the reciprocal.

Avoid creating data dependencies in algorithms when performing MaverickCrunch operations. A data dependency occurs when an instruction takes the output of a previous instruction as its operand. This dependency will stall the pipeline if the output of the previous instruction is not available for the current instruction. The following table contains the MaverickCrunch's instruction stall time in cycle counts for each type of coprocessor instruction.

Table 4. Instruction Stall Time

INSTRUCTION TYPE	CYCLE COUNT
CDP	5
CDP (Multiply Double & 64)	11
LDC/MCR	2
STR/MRC	0

Consider the following MaverickCrunch code:

```
fmuls c3, c1, c2 // CDP instruction
fadds c0, c0, c3 // CDP instruction - stalls on c3
```

In this example, the addition operation stalls for 5 cycles on the product (*c3*) of the multiplication. However, if the first instruction had been a double precision multiplication the addition operation would have stalled on the product (*c3*) for 11 cycles.

Considering the pipeline stall cycles, the above source code looks like:

```
fmuls c3, c1, c2 // CDP instruction
<stall cycle>
<stall cycle>
<stall cycle>
<stall cycle>
<stall cycle>
fadds c0, c0, c3 // CDP instruction - stalls on c3
```

Optimize data-dependent pipeline stalls by interleaving the dependent instructions with independent instructions. This will have a positive effect by increasing pipeline throughput. Again, this is especially important when executing a double precision multiply, and significantly important when executing adds, compares, or other data-path operations. These interleaved instructions may either be ARM or MaverickCrunch operations and should be placed just after the stalling instruction. However, be judicious about which interleaved instructions are used so that new data dependencies are not introduced into the source code.

Finally, the optimized example source code looks like:

```
fmuls c3, c1, c2
<Independent ARM/MaverickCrunch Instruction>
<Independent ARM/MaverickCrunch Instruction>
<Independent ARM/MaverickCrunch Instruction>
<Independent ARM/MaverickCrunch Instruction>
<Independent ARM/MaverickCrunch Instruction>
fadds c0, c0, c3
```

Note: Please be aware that some sequences of MaverickCrunch instructions are not supported. For an up-to-date list of these instruction sequences, please see the appropriate Errata Sheet. The Errata Sheets are available at www.cirrus.com. Furthermore, a parsing tool is available that can identify illegal sequences of MaverickCrunch instructions. This tool is also available from Cirrus Logic.

Attempt to maximize the throughput of the ARM920T, and MaverickCrunch CDP pipelines by interleaving independent ARM and CDP instructions. The maximum throughput of the CDP pipeline is one CDP instruction every other ARM CLK. This is because the E1, E2, E3, and W pipeline stages of the CDP pipeline take two ARM cycles to complete. Please note that MaverickCrunch must be operating in asynchronous mode to realize this optimization.

Consider the following code sequence:

```
<Independent MaverickCrunch Instruction A>
<Independent MaverickCrunch Instruction B>
<Independent MaverickCrunch Instruction C>
<Independent ARM Instruction A>
<Independent ARM Instruction B>
<Independent ARM Instruction C>
```

The above code is inefficient because MaverickCrunch Instructions A, B, and C will stall the ARM's pipeline during their execution of the second cycle in the E1, E2, E3, and W stages. The following code interleaves the MaverickCrunch and ARM instructions, which removes the stalls and maximizes the throughput of both pipelines.

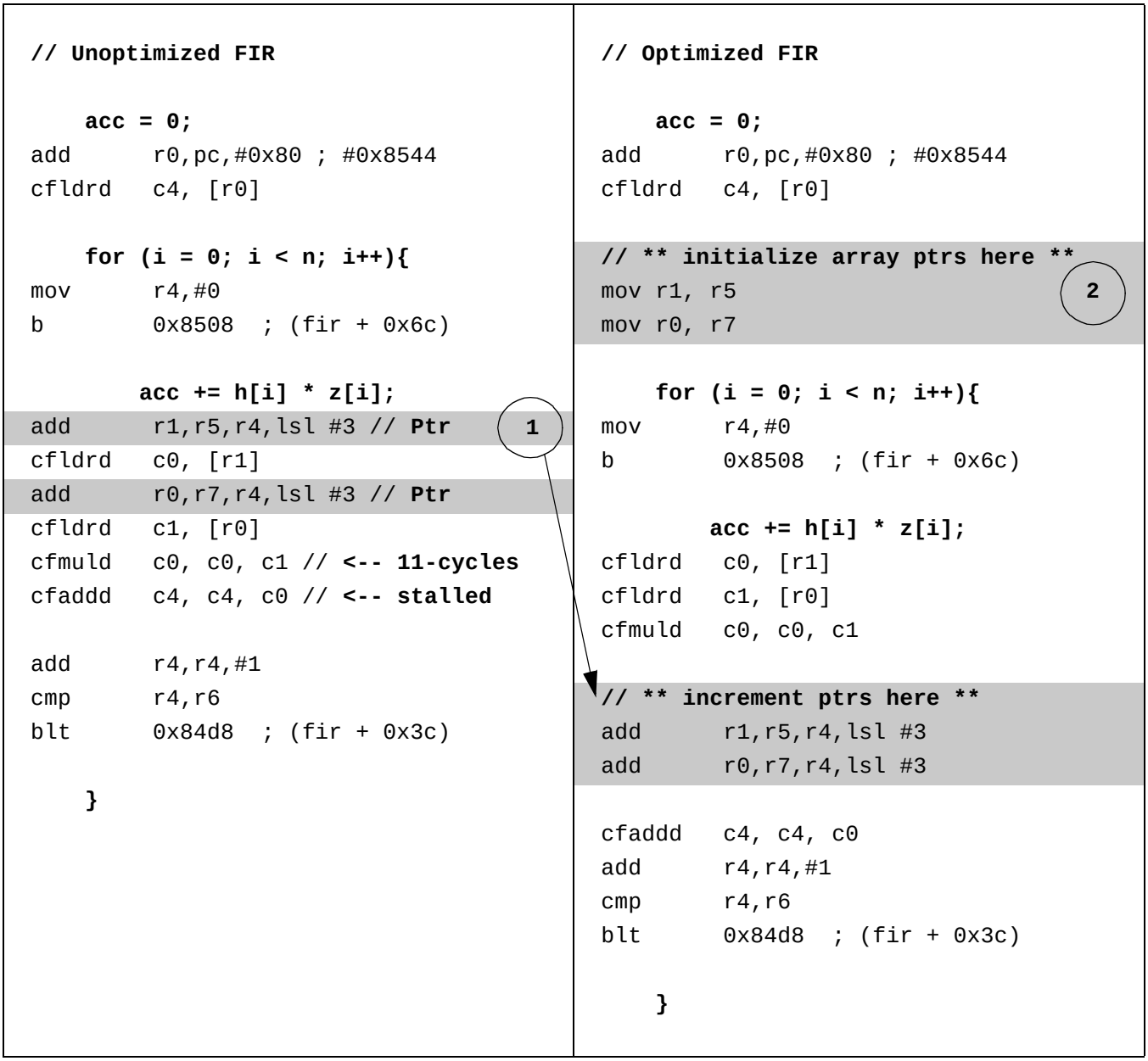
```
<Independent MaverickCrunch Instruction A>
<Independent ARM Instruction A>
<Independent MaverickCrunch Instruction B>
<Independent ARM Instruction B>
<Independent MaverickCrunch Instruction C>
<Independent ARM Instruction C>
```

Utilize both the ARM and co-processor registers, and the data caching capabilities of the ARM core to reduce the latency of fetching data. For example, FIR filters typically have many filter coefficients - more coefficients than there are available registers. Many costly memory accesses will be executed to load the filter coefficients. One solution to this problem is to load and lock-in the filter coefficients in a data cache at start-up. This will result in an increase in performance because of the reduction in pipeline stalling while waiting for data to transfer from (slower) external memory to the coprocessor.

4. Examples

This example illustrates the process of optimizing code for the MaverickCrunch coprocessor at the architecture level. The following source code is part of a simple real-time implementation of an FIR filter (Figure 2). There is a data dependency in the accumulate portion of the basic source code (on the left). This data dependency stalls the MaverickCrunch CDP pipeline for 11 cycles. To reduce this penalty, it is possible to re-arrange the source code to use the ARM cycles in the shadow of the stall.

Figure 2: FIR Optimization Example



To accomplish this optimization (see Figure 2): 1.) Move the index initialization/update code behind the cfmuld operation; 2.) Add index initialization code just above the loop construct. This optimization results

in a 2-cycle increase in performance per iteration of the loop construct. This increase in performance can be significant for FIR filters with large numbers of coefficients.

In this next algorithm and architectural optimization example, a Biquad IIR filter (Figure 3) is modified to reduce the number of loads/stores in the inner loop. The Biquad IIR filter's difference equation is:

```
y[n] = b[0]*x[n] + b[1]*x[n-1] + b[2]*x[n-2] - a[1]*y[n-1] - a[2]*y[n-2]
```

One C implementation of this IIR Biquad difference equation, and the one used for this example is:

```
double inp, accumulator;
long i;
for (i=0; i<(long)nn; i++)
{
           inp = data[i];
           accumulator = 0;
           accumulator -= filtcoefs[0]*bqd1k_dCstates[0];
           accumulator -= filtcoefs[0]*bqd1k_dCstates[1];
           accumulator += filtcoefs[2]*inp;
           accumulator += filtcoefs[3]*bqd1k_dCstates[2];
           accumulator += filtcoefs[4]*bqd1k_dCstates[3];

           // Update y[n] states (output history for feedback lines)
           bqd1k_dCstates[1] = bqd1k_dCstates[0];
           bqd1k_dCstates[0] = accumulator;

           // Store the output
           data[i] = accumulator;

           // Update x[n] states (input history for delay lines)
           bqd1k_dCstates[3] = bdq1k_dCstates[2];
           bqd1k_dCstates[2] = inp;
}
```

In the assembly version of the original algorithm (on the left), the filter coefficients are loaded when needed. Consequently, they are loaded in each iteration of the loop. Additionally, the state variables are loaded when needed and then stored when updated. They are also loaded and stored in each iteration of the loop.

Figure 3: IIR Optimization Example

```
// Floating-point Biquad IIR
// (Basic: Non-optimized)

main_loop

  cfldrd  temp2, [fcoef]
  cfldrd  bqd1k_s0, [bdq1k]
  cfmuld  acc, temp2, bqd1k_s0
  cfnegd  acc, acc
  cfldrd  temp2, [fcoef, 8]
  cfldrd  bqd1k_s1, [bdq1k, 8]
  cfmuld  temp, temp2, bqd1k_s1
  cfsubd  acc, acc, temp
  cfldrd  temp2, [fcoef, 16]
  cfldrd  temp4, [data]
  cfmuld  temp, temp2, temp4
  cfaddd  acc, acc, temp
  cfldrd  temp2, [fcoef, 24]
  cfldrd  bqd1k_s2, [bdq1k, 16]
  cfmuld  temp, temp2, bqd1k_s2
  cfaddd  acc, acc, temp
  cfldrd  temp2, [fcoef, 32]
  cfldrd  bqd1k_s3, [bdq1k, 24]
  cfmuld  temp, temp2, bqd1k_s3
  cfaddd  acc, acc, temp
  cfstrd  acc, [data], 8
  cfstrd  acc, [bdq1k]
  cfstrd  bqd1k_s0, [bdq1k, 8]
  cfstrd  temp4, [bdq1k, 16]
  cfstrd  bqd1k_s2, [bdq1k, 24]

  subs    nn, nn,  1
  bgt     main_loop
```

```
// Floating-point Biquad IIR
// (Optimized)

  cfldrd  bqd1k_s0, [bdq1k]
  cfldrd  bqd1k_s1, [bdq1k, 8]
  cfldrd  bqd1k_s2, [bdq1k, 16]
  cfldrd  bqd1k_s3, [bdq1k, 24]
  cfldrd  outp, [fcoef]
  cfldrd  temp1, [fcoef, 8]
  cfldrd  temp2, [fcoef, 16]
  cfldrd  temp3, [fcoef, 24]
  cfldrd  temp4, [fcoef, 32]
  cfnegd  outp, outp
  cfnegd  temp1, temp1

main_loop

  cfldrd  inp, [data]
  cfmuld  acc, outp, bqd1k_s0
  cfmuld  temp, temp1, bqd1k_s1
  cfcpyd  bqd1k_s1, bqd1k_s0
  cfaddd  acc, acc, temp
  cfmuld  temp, temp2, inp
  cfaddd  acc, acc, temp
  cfmuld  temp, temp3, bqd1k_s2
  cfaddd  acc, acc, temp
  cfmuld  temp, temp4, bqd1k_s3
  cfcpyd  bqd1k_s3, bqd1k_s2
  cfcpyd  bqd1k_s2, inp
  cfaddd  acc, acc, temp
  cfcpyd  bqd1k_s0, acc

  subs    nn, nn, 1
  bgt     main_loop

  ldr     temp1, =bqd1k_dCrstates
  cfstrd  bqd1k_s0, [temp1]
  cfstrd  bqd1k_s1, [temp1, 8]
  cfstrd  bqd1k_s2, [temp1, 16]
  cfstrd  bqd1k_s3, [temp1, 24]
```

To accomplish this optimization (see Figure 3): 1.) Use the additional registers in the MaverickCrunch co-processor to load the filter and state variables once before the inner loop; 2.) Shuffle the state variables around in registers during the inner loop; 3.) Store the state variables after the inner loop. This removes nine loads and four stores from the inner loop. Also note that the copy instructions used to shuffle the state

variables have been interleaved with data dependant instructions in order to reduce their respective stall penalties.

5. Summary

The optimization guidelines suggested in this document are:

- Write and test all source code before attempting any optimizations
- Focus optimizations on sections of code that are executed most frequently, and take the most CPU cycles to execute
- Identify and resolve any implementation inefficiencies generated by the compiler
- Set the appropriate target processor for the compiler, linker and assembler
- Turn off all of the compilers debug options
- Set the compiler to optimize the code for speed
- Set the MaverickCrunch coprocessor to run in asynchronous mode with data-forwarding enabled
- Avoid single or double floating-point division
- Avoid creating data dependencies in algorithms when performing MaverickCrunch operations
- Optimize data-dependent pipeline stalls by interleaving the dependent instructions with independent instructions
- Utilize both the ARM and MaverickCrunch's registers, and the data caching capabilities of the ARM core to reduce the latency of fetching data from external memory

In summary, this document has detailed many optimization techniques that can improve the performance of applications written for the MaverickCrunch coprocessor.

Revision	Date	Changes
1	23 January 2004	Initial Release

Contacting Cirrus Logic Support

For all product questions and inquiries contact a Cirrus Logic Sales Representative.
To find one nearest you go to **http://www.cirrus.com/corporate/contacts/sales.cfm**