Timing Analysis of a Single-Threaded Application Implemented Using a FastMATH™ Processor

1. Overview

In any software system, when several kernel algorithms are put together to form the system, the total execution time (cycle count) for the entire system is usually greater than the sum of the individual pieces. This is due to additional overhead such as data movement between various blocks, control code overhead, and so on. This document presents example code for a system and illustrates how to combine various algorithms that use the FastMATH™ processor to create code for the entire system. The document also presents a timing profile of the code executed using the FastMATH cycle-accurate simulator, and shows the cycle count of the whole system versus the cycle count of its individual pieces.

The results presented here show that overhead arising in the single-threaded application implemented using the FastMATH processor is relatively small. Future work will show examples of a multi-threaded application implemented using the FastMATH processor, and corresponding overhead in implementing such applications.

2. Design Overview of the Single-Threaded Application

This document considers a simple application showing the typical overhead arising in a single-threaded FastMATH system. The example chosen is a smart baseband receiver system containing front end finite-length impulse response (FIR) filters and a smart antenna beamformer.

The example C-language file (Attachment A) describes the problem and provides a self-contained application that can be implemented with the FastMATH cycle-accurate simulator. For convenience, Sections 2.1. through 2.5. provide a summary of the comments embedded in this example code.

2.1. Brief Description

This file contains a stand-alone example that incorporates the front-end filtering (FIR filters) and a beamformer for a smart antenna receiver. System-level performance has not been verified for the parameters used in this code. This is simply an example to illustrate the usage of the FastMATH processor in a typical application.

2.2. Typical Applications

The example shown can be applicable in wireless communication systems using a smart antenna receiver. For example, this could be applicable in a global system for mobile communications (GSM) or a time division multiple access (TDMA) cellular base station system.

2.3. Receiver Structure

Figure 1 illustrates an example receiver structure.

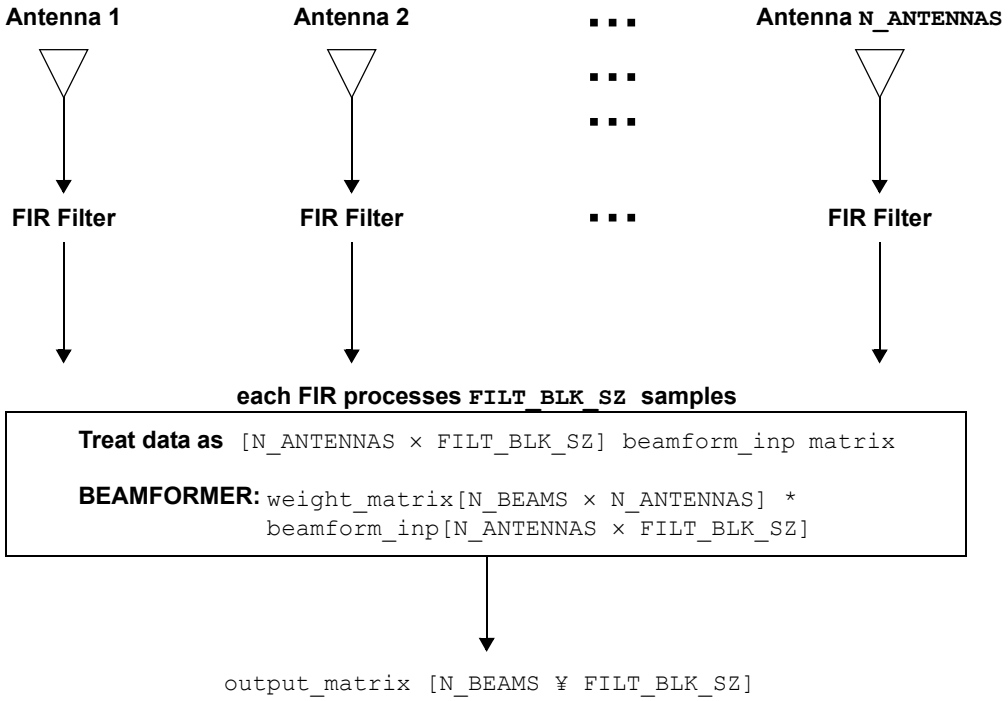

Figure 1: Receiver structure for independent frequency

2.4. High-Level Description of Design

Design parameters include the following:

N_FREQ	number of independent frequencies
N_ANTENNAS	number of antennas
N_BEAMS	number of beams created by the smart antenna receiver
N_TAPS	number of taps in the FIR filter used for front-end filtering
INP_BLK_SZ	number of samples in the input frame (processed in sub-frame “chunks”)
FILT_BLK_SZ	Size of subframe–block size for FIR and smart antenna processing

For each frequency, the input frame (INP_BLK_SZ samples) is processed in “chunks” of sub-frames (FILT_BLK_SZ samples). The data is received from an external device, such as a RapidIO™ interface, or from a different task on the same FastMATH processor and stored directly into the on-chip L2 cache. The FILT_BLK_SZ size sub-frames are processed one at a time through the filtering and beamforming blocks. It is assumed that as sub-frame #n is written into on-chip cache, sub-frame #(n-1) is being processed by the receiver. An example of this would be a ping-pong buffer scheme.

2.5. Data Structures Used

Figure 2 illustrates the data structures used in the design for each frequency.

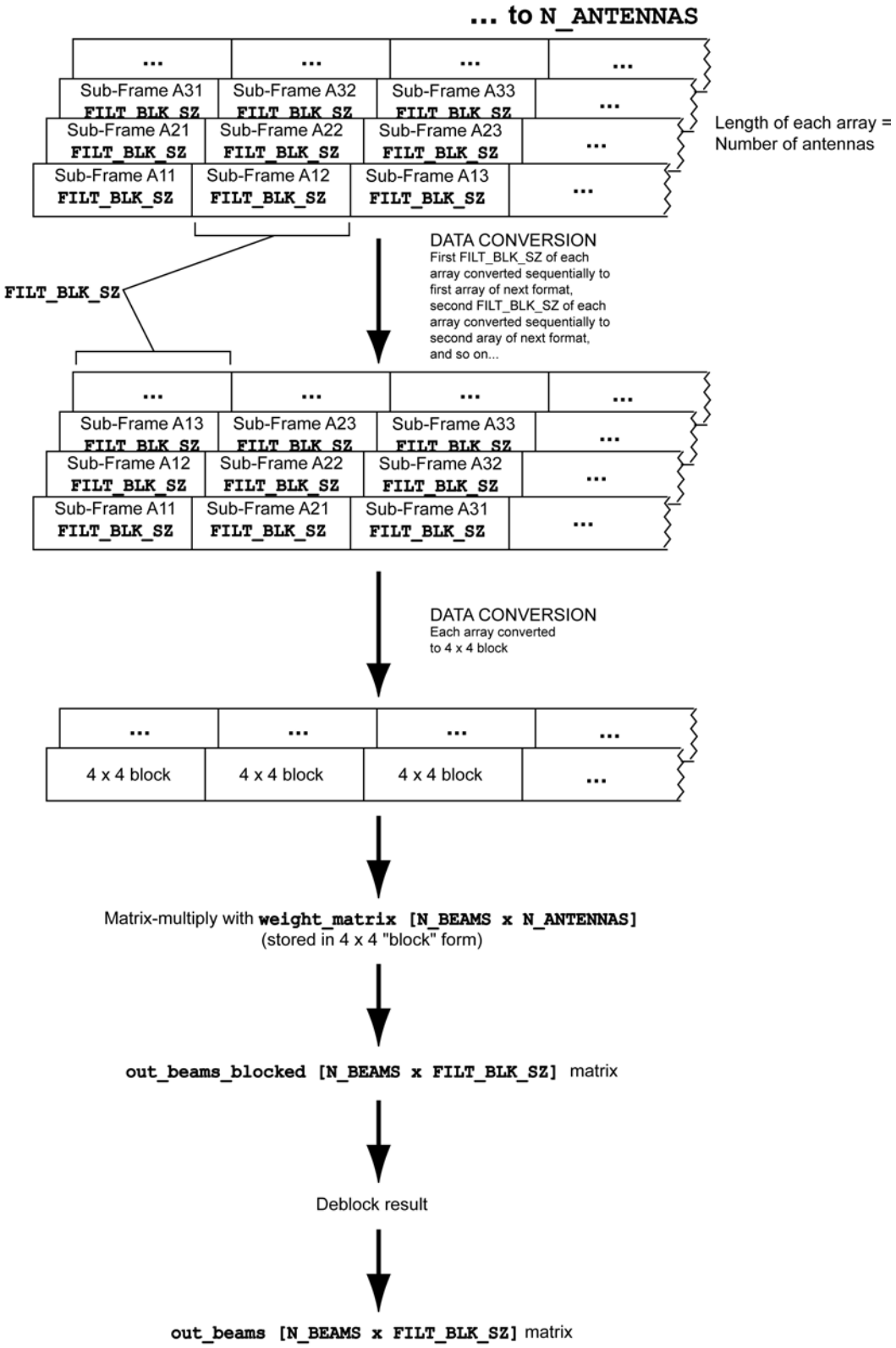

Figure 2: Data structures used for each frequency

3. Timing Analysis and Cycle Count Summary

The code is linked with the *preliminary* FastMATH libraries and executed with the FastMATH cycle accurate simulator. Since the FastMATH libraries used for this simulation were preliminary, we expect further improvements to the cycle counts described in Table 1. Table 1 shows the cycle counts measured for the various functional blocks, and the total cycle count measured for the entire system when two consecutive sub-frames of size `FILT_BLK_SZ` are processed from 16 antennas. The cycle counts are shown for the case when all the data is available in 1-Mbyte on-chip cache. Note that in typical applications the input data is expected to be available in the on-chip cache.

Table 1: Cycle count summary for the case when all data is available in on-chip cache[a]

	FastMATH Cycles	Estimated Time @ 2 GHz
FIR filter for *each* antenna (repeated for 16 antennas)	2,720	1.36 µs
Transform matrices from sequential format into block format for beamforming operation	1,369	0.68 µs
Beamforming	5,713	2.86 µs
Format final output	391	0.19 µs
Total cycles *calculated* from sum of individual blocks to process: 2 sub-frames 1 frequency 16 antennas 2* (16 × 2720 + 1369 + 5713 + 391) cycles	101,986	50.99 µs
Corresponding total cycles *measured* for the entire system	103,212	51.61 µs
Estimated overhead	1.2%	

a. The FastMATH cycle counts and estimated time do not represent the optimized performance achievable using the FastMATH processing libraries.

A comparison of the two *total* cycle counts in Table 1 shows that the additional overhead of integrating all the blocks is approximately 1.2% of the total. Moreover, if the data formatting functions (transforming matrices from sequential format into block format and transforming matrices from block format into sequential format) are considered as part of the overhead computation, the total overhead is approximately 4.6% of the total. It is clear that in this single-threaded application, the total overhead required to put together all the kernel algorithms into a complete system is small.

Figure 3 shows the timing diagram for the single-threaded smart baseband receiver system. The figure illustrates the sequence of processing for two consecutive sub-frames of size `FILT_BLK_SZ` for 16 antennas. The total time to complete the required processing is 51.61 µs.

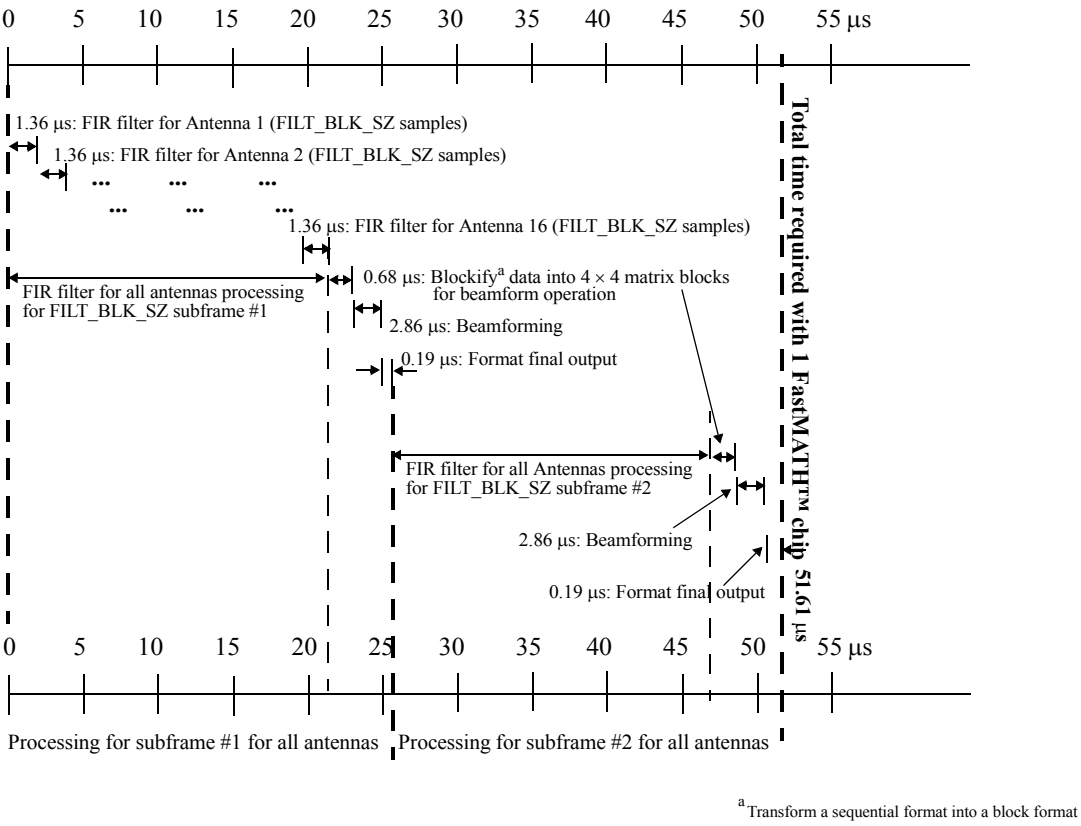

Figure 3: Timing diagram for the single-threaded smart baseband receiver system

4. Additional Assumptions and Notes

The information in this document is based on the following assumptions:

- The block size for filter processing and smart antenna processing is assumed to be the same in this implementation; for example, this could be `min(FILT_BLK_SZ, BEAMFORM_BLK_SZ)`.
- The total bytes of the selected block size should be equal to a multiple of 64. This is assumed in the code (Attachment A), as it allows for the most efficient implementation of the FastMATH processor.
- The random number generator is currently used to create both the input data from the antennas and the weight-matrix.
- The FastMATH cycle accurate simulator is used with the code (Attachment A) to obtain cycle counts for the various functional blocks. In order to simulate the effect of having the input data in on-chip cache, at least two iterations of the receiver should be used.

Note that the code is executed by linking with the optimized FastMATH libraries. Because, at the present time, only a preliminary version of these libraries is available, the cycle counts output from this code is *not* optimal.

5. Acronyms & Abbreviations

FIR	finite-length impulse response
GSM	global system for mobile communications
TDMA	time division multiple access

6. Revision History

Revision	Date	Description
1.0	9/17/02	Initial release
1.1	3/26/03	New format and logo with tag line
1.2	8/1/03	Added footnote to Table 1

Attachment A

(Timing Analysis of a Single-Threaded Application Implemented Using a FastMATH™ Processor)

```
/*********************************************************************
 The Comments describing the design and the code have been deleted from
 this section, since it has been described in the above document.
*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <intrinsity/fastmath.h>
#include "fm.h"
#include "blas.h"
#include "../dsp/fm_fir.h"

#ifndef BLOCK_ALIGNMENT
#define BLOCK_ALIGNMENT 64 /* matrix memory blocks need to be on a 64 byte boundry */
#endif

/* random number generator - compatible with Green Hills compiler */
extern int rand(void);
extern void srand(unsigned int seed);
#define FM_RND(n)          ((int)(rand()%(n)))   /* random int from 0 to n-1 */
#define FM_RND_INIT()      srand(0) /* init RNG */

/* Data sets used for creating input data */
#define N_FREQ 1 /* 12 */
#define N_ANTENNAS 16
#define FILT_BLK_SZ 160 /* Approximately 1 GSM TS */
#define N_BEAMS 4
#define N_TAPS 32

#define INP_BLK_SZ 320
#define N_SUBFRAMES (INP_BLK_SZ/FILT_BLK_SZ) /* value of 2 => ping-pong buffer scheme
*/

#define WEIGHTS_RANGE 32768 /* 15 bit weights */
#define INPUT_RANGE 32768 /* 15 bit input data */

/* Main application function */

int main()
{
   /*************************************************************
   Initialization section
   *************************************************************/

   /* Input vector dimension: FILT_BLK_SZ samples, per sub-frame, per Antenna, per fre-
quency */
   fm_cs16a *input[N_SUBFRAMES][N_FREQ][N_ANTENNAS];

   /* Filt out vector ("blocked" form) dim: FILT_BLK_SZ samples for N_ANTENNAS
   antennas, per frequency */
   fm_cs16a *filt_out_blocked[N_FREQ];

   /* Beamforming weight_matrix dim: N_BEAMS x N_ANTENNAS, per frequency */
```

```
   fm_cs16a *weight_matrix[N_FREQ];

   /* Beamforming input matrix dim: N_ANTENNAS x FILT_BLK_SZ, per frequency */
   fm_cs16a *beamform_inp[N_FREQ];

   /* Output beams ("blocked" form) matrix dim: N_BEAMS x FILT_BLK_SZ, per frequenncy */
   fm_cs16a *out_beams_blocked[N_FREQ];

   /* Final output beams matrix dim: N_BEAMS x FILT_BLK_SZ, per frequenncy */
   fm_cs16a *out_beams[N_FREQ];

   /* FIR filter taps vector dim: N_TAPS taps per frequency */
   fm_cs16a *filt_taps[N_FREQ];

   int i, j, k, m, cycles_fir, cycles_blockify, cycles_matmatmul, cycles_deblock,
cycles_total;
   int cycles_t0, cycles_t1, cycles_t2, iterations;

   FM_RND_INIT(); /* Init random number generator */

   /* Allocate 64 byte alligned memory for data structs used with the FastMATH's matrix
      coprocessor. This ensures efficient implementation of low-level library funcs */

   for (i=0; i<N_SUBFRAMES; i++)
   {
     for (j=0; j<N_FREQ; j++)
     {
       for (k=0; k<N_ANTENNAS; k++)
       {
         /* Init input vector: FILT_BLK_SZ samples per sub-frame, per antenna, per freq
*/
         input[i][j][k] = (fm_cs16a *) memalign(BLOCK_ALIGNMENT,
FILT_BLK_SZ*sizeof(fm_cs16a));
         for (m=0; m<FILT_BLK_SZ; m++)
         {
           /* Init real & imag parts of cplx input to random numbers */
           input[i][j][k][m].r = FM_RND(INPUT_RANGE);
           input[i][j][k][m].i = FM_RND(INPUT_RANGE);
         }
       }
     }
   } /* End init input[][][] */

   for (i=0; i<N_FREQ; i++)
   {
     /* Init filter output buff: FILT_BLK_SZ samples for N_ANTENNAS antennas, per fre-
quency */
     filt_out_blocked[i] = (fm_cs16a *) memalign(BLOCK_ALIGNMENT,
                           FILT_BLK_SZ*N_ANTENNAS*sizeof(fm_cs16a));
     /* Init weight_matrix buff: N_BEAMS x N_ANTENNAS matrix, per frequency */
     weight_matrix[i] = (fm_cs16a *)memalign( BLOCK_ALIGNMENT,
                                          N_BEAMS*N_ANTENNAS*sizeof(fm_cs16a) );
     /* Init beamforming input matrix: N_ANTENNAS x FILT_BLK_SZ, per frequency */
     beamform_inp[i] = (fm_cs16a *)memalign( BLOCK_ALIGNMENT,
                                         N_ANTENNAS*FILT_BLK_SZ*sizeof(fm_cs16a) );
    /* Init output beams ("blocked" form) matrix: N_BEAMS x FILT_BLK_SZ, per frequenncy
*/
     out_beams_blocked[i] = (fm_cs16a *)memalign( BLOCK_ALIGNMENT,
                                           N_BEAMS*FILT_BLK_SZ*sizeof(fm_cs16a) );
```

```
  /* Init output beams: N_BEAMS x FILT_BLK_SZ, per frequenncy */
  out_beams[i] = (fm_cs16a *)memalign( BLOCK_ALIGNMENT,
                                  N_BEAMS*FILT_BLK_SZ*sizeof(fm_cs16a) );
  /* FIR filter taps vector init: N_TAPS taps per frequency */
  filt_taps[i] = (fm_cs16a *)memalign( BLOCK_ALIGNMENT, N_TAPS*sizeof(fm_cs16a) );

  for (j=0; j<N_BEAMS*N_ANTENNAS; j++)
  {
    /* Use random weights for now */
    weight_matrix[i][j].r = FM_RND(WEIGHTS_RANGE);
    weight_matrix[i][j].i = FM_RND(WEIGHTS_RANGE);
  }
  for (j=0; j<N_ANTENNAS*FILT_BLK_SZ; j++)
  {
    filt_out_blocked[i][j].r = 0;
    filt_out_blocked[i][j].i = 0;
    beamform_inp[i][j].r = 0;
    beamform_inp[i][j].i = 0;
  }
  for (j=0; j<N_BEAMS*FILT_BLK_SZ; j++)
  {
    out_beams_blocked[i][j].r = 0;
    out_beams_blocked[i][j].i = 0;
    out_beams[i][j].r = 0;
    out_beams[i][j].i = 0;
  }
  for (j=0; j<N_TAPS; j++)
  {
    /* Just use constant taps for now (example: averaging filter) */
    filt_taps[i][j].r = 1;
    filt_taps[i][j].i = 1;
  }

}
/*****************************************************************
For each independent frequency, implement the receiver.
*****************************************************************/

/* In order to simulate the effect of having the input data in on-chip
   cache (which will happen in the real application), iterate at least
   twice to obtain the cycle-counts
*/
for (iterations=0; iterations<3; iterations++)
{
cycles_t0 = MatrixGetCycleCount();

/* Process sub-frames of size FILT_BLK_SZ one at a time */
for (i=0; i<N_SUBFRAMES; i++)
{
  /* Process the data for all the frequencies */
  for (j=0; j<N_FREQ; j++)
  {
    for (k=0; k<N_ANTENNAS; k++)
    {
      /* Perform the complex-number FIR filter function for each
         antenna  using the efficient FastMATH low-level library call.
         Write the output data to the appropriate output block
      */
      cycles_t1 = MatrixGetCycleCount();
```

```
          fm_cfir (filt_out_blocked[j] + k*FILT_BLK_SZ,
                   input[i][j][k],
                   filt_taps[i], FILT_BLK_SZ, N_TAPS);
          cycles_t2 = MatrixGetCycleCount();
          cycles_fir = cycles_t2 - cycles_t1;
//          printf("%d ", cycles_fir);
        }

        /* Format the data for the subsequent beamforming operation (into 4x4 matrix
blocks) */
        cycles_t1 = MatrixGetCycleCount();
        fm_cblockify (beamform_inp[j], /* Formated output - ready for beamforming pro-
cessing */
                      filt_out_blocked[j], /* Input into this function (output of filter-
ing) */
                      N_ANTENNAS, FILT_BLK_SZ); /* N_ANTENNAS x FILT_BLK_SZ inp matrix */
        cycles_t2 = MatrixGetCycleCount();
        cycles_blockify = cycles_t2 - cycles_t1;

        /*
         Perform the beamforming operation for the data output from the filter for all
         antennas. This involves a matrix-matrix multiply of the weight matrix with the
         appropriately formated (into 4x4 blocks) filtered inp_data from all the
antennnas.
         Use the FastMATH low-level function library to implement the matrix-matrix mul-
tiply.
        */
        cycles_t1 = MatrixGetCycleCount();
        fm_cgemmblock (out_beams_blocked[j], /* Output matrix dimensions: N_BEAMS x
FILT_BLK_SZ] */
                       weight_matrix[j], /* Input matrix 1: N_BEAMS x N_ANTENNAS */
                       beamform_inp[j],  /* Input matrix 2: N_ANTENNAS x FILT_BLK_SZ */
                       N_BEAMS, N_ANTENNAS, FILT_BLK_SZ);
        cycles_t2 = MatrixGetCycleCount();
        cycles_matmatmul = cycles_t2 - cycles_t1;

        /* Format the data for final output ("de-blockify") */
        cycles_t1 = MatrixGetCycleCount();
        fm_cdeblockify (out_beams[j], out_beams_blocked[j], N_BEAMS, FILT_BLK_SZ);
        cycles_t2 = MatrixGetCycleCount();
        cycles_deblock = cycles_t2 - cycles_t1;

//        printf("\nblockify_cyc: %d, cycles_matmul: %d, cycles_deblock: %d\n\n",
//          cycles_blockify, cycles_matmatmul, cycles_deblock);

      } /* for loop: N_FREQ */
    } /* for loop: N_SUBFRAMES */

    /*****************************************************************
    Output cycle-counts and results
    *****************************************************************/
    cycles_t2 = MatrixGetCycleCount();
    cycles_total = cycles_t2 - cycles_t0;
    printf("Total cycles: %d\n\n", cycles_total);
    printf("\n cycles_fir: %d, blockify_cyc: %d, cycles_matmul: %d, cycles_deblock:
%d\n\n",
         cycles_fir, cycles_blockify, cycles_matmatmul, cycles_deblock);

    } /* for loop: iterations */
```

```
   printf("\n\n--- Results for sub-frame #%d, freq #%d ---\n", N_SUBFRAMES-1, N_FREQ-
1);
   printf("\nWeight_matrix:\n");
   for (i=0; i<N_BEAMS*N_ANTENNAS; i++)
     printf(" %08X", weight_matrix[N_FREQ-1][i] );
   printf ("\n");

   printf("\nOutput matrix:\n");
   for (i=0; i<N_BEAMS*FILT_BLK_SZ; i++)
     printf(" %08X", out_beams_blocked[N_FREQ-1][i] ) ;

   printf ("\n");
}
```