Lab 6

Frame-based Filtering

ECE 406: Real-Time Digital Signal Processing

April 17, 2015

Christopher Daffron

cdaffron@utk.edu

There was only one objective for this lab. The objective was to practice frame-based filtering using CPU vs DMA. This involved two tasks. The first was to run some frame-based filtering code that uses the CPU for the memory operations and computes the sum of the inputs as the output of one of the channels and the difference of the outputs as the other output channel. The second was to implement a frame-based FIR filter that uses DMA for memory operations.

Step 1

The first step of the lab was to import and compile some code from chapter 6 of the welch book that uses frame-based filtering utilizing the CPU for memory operations. This code computes the sum of the input channels and outputs that to the left channel and it calculates the difference between the input channels and outputs that to the right channel. Once plugged into the signal generator and the oscilloscope, this code behaved as expected, outputting either the sum or the difference of the inputs depending on which output channel was being observed.

Step 2

The next step of the lab was to import some code from the welch book that implements frame-based filtering using the DMA buffers to handle the memory operations and to modify it to create an FIR filter. After compiling and running the code, I connected the DSK board to a signal generator and oscilloscope to take measurements comparing the actual frequency response of the filter to the theoretical frequency response. A graph of the results of these measurements is shown below.

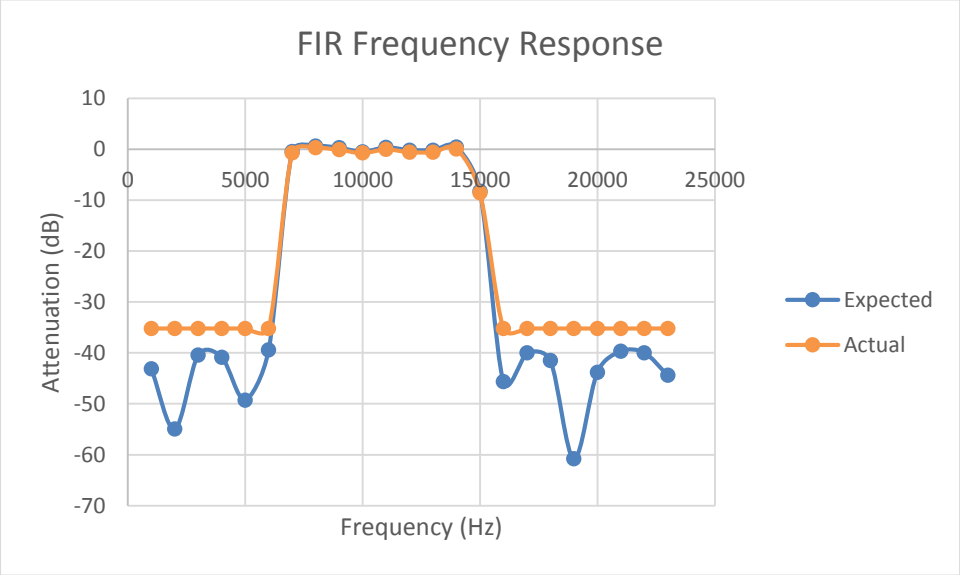

Figure 1: FIR DMA Frequency Response

As it has in previous labs, the actual frequency response follows the expected frequency response very well. The only difference is in the stopbands due to the expected attenuation being greater than can be measured by the oscilloscope.

Step 3

For the next step, several measurements were taken on the running time of the code to compare the frame-based FIR to the sample-based FIR code. First, I took measurements with all compiler optimization disabled. With this setup, the sample-based FIR code took 5,111 cycles to compute the value for a single sample and the frame-based FIR code took 3,500 cycles to compute the value for a single sample. This result is somewhat surprising since, while the frame-based code is faster, it is not as fast as I would expect due to all of the memory operations being handled by the DMA controller in the frame-based FIR code, but makes sense due to the extra code required to iterate through the frame. A key observation that I made about the structure of the code is that the compiler would probably do a good job of optimizing it since it is doing many operations in the same function call so the compiler can rearrange the operations so that multiple samples can be worked on at the same time. To test this, I again measured the cycle counts for both implementations, but with compiler optimization set to 2 this time. With this setup, the sample-based FIR code took 2,886 cycles and the frame-based FIR code took 262 cycles. These results show that, as expected, the frame-based implementation is able to be optimized much better and shows an impressive improvement.

Lab Report Questions

The primary difference between sample-based DSP and frame-based DSP is that the sample-based approach calculates a new output value every time a new sample comes in and that output usually either depends only on the new sample or on a relatively small number of previous samples. A frame-based DSP algorithm collects some number of samples and then does the processing on all of those samples at the same time. This allows more complex algorithms such as the Fast Fourier Transform to be implemented which allows for filtering and other modifications to be done in the frequency domain instead of the time domain. There are several considerations when determining the frame size for a particular DSP algorithm. The first is the amount of memory available on the device. To have an efficient frame-based algorithm, triple buffering is the most efficient method, so there must be enough memory to fit three frames in memory at any given time. Next, the DSP processor must be able to process all of the samples in a single frame before the input frame gets full. If it is unable to do this, a buffer overflow will occur. The primary difference between a non-DMA implementation of a DSP algorithm and an implementation that uses DMA is which component handles the memory transfers. In a non-DMA implementation, the CPU gets interrupted every time a new sample is ready, so it must switch tasks, handle the memory operation, and then continue whatever task it was doing. All of this switching takes time and is very inefficient, so it reduces the amount of time available to actually process the samples themselves. In a DMA implementation, the DMA controller handles all of the memory operations of the incoming and outgoing samples, so the processor only gets interrupted when the buffer gets full and it is time to actually process the samples. Because of this, DMA-based implementations allow the CPU more time to actually do the processing, allowing for more complex algorithms.

Conclusion

In this lab, I got hands-on experience with frame-based filtering and its comparison to sample-based filtering. I also got experience with both CPU-based and DMA-based implementations of different algorithms. I feel that my understanding of both has been improved by this lab.

Appendix

Task 1

ISRs.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

/////////////////////////////////////////////////////////////////////////
// Filename: ISRs.c
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
/////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "math.h"
#include "frames.h"

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
	Uint32 UINT;
	Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* add any global variables here */
// frame buffer declarations
#define BUFFER_LENGTH	96000  // buffer length in samples
#define NUM_CHANNELS	2		// supports stereo audio
#define NUM_BUFFERS		3		// don't change
#define INITIAL_FILL_INDEX  0		// start filling this buffer
#define INITIAL_DUMP_INDEX  1		// start dumping this buffer

#pragma DATA_SECTION (buffer, "CE0"); // allocate buffers in external SDRAM
volatile float buffer[NUM_BUFFERS][2][BUFFER_LENGTH];
// there are 3 buffers in use at all times, one being filled,
```

```
// one being operated on, and one being emptied
// fill_index   --> buffer being filled by the ADC
// dump_index --> buffer being written to the DAC
// ready_index --> buffer ready for processing
Uint8 buffer_ready = 0, over_run = 0, ready_index = 2;

void ZeroBuffers()
//////////////////////////////////////////////////////////////////////
// Purpose:   Sets all buffer locations to 0.0
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
//
// Notes:     None
//////////////////////////////////////////////////////////////////////
{
    Uint32 i = BUFFER_LENGTH * NUM_BUFFERS * NUM_CHANNELS;

    volatile float *p = buffer[0][0];

    while(i--)
        *p++ = 0.0;
}

void ProcessBuffer()
//////////////////////////////////////////////////////////////////////
// Purpose:   Processes the data in buffer[ready_index] and stores
//              the results back into the buffer
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
//
// Notes:     None
//////////////////////////////////////////////////////////////////////
{
       Uint32 i;
    volatile float *pL = buffer[ready_index][LEFT];
    volatile float *pR = buffer[ready_index][RIGHT];
    float temp;

/* zero out left channel
   for(i=0;i < BUFFER_LENGTH;i++){
              *pL = 0.0;
        pL++;
   }  */


/* zero out right channel
   for(i=0;i < BUFFER_LENGTH;i++){
              *pR = 0.0;
        pR++;
   }    */
```

```
/* reverb on right channel
   for(i=0;i < BUFFER_LENGTH-4;i++){
              *pR = *pR + (0.9 * pR[2]) + (0.45 * pR[4]);
        pR++;
   }              */


// addition and subtraction
   for(i=0;i < BUFFER_LENGTH;i++){
      temp = *pL;
      *pL = temp + *pR; // left = L+R
      *pR = temp - *pR; // right = L-R
      pL++;
      pR++;
    }


/* add a sinusoid
   for(i=0;i < BUFFER_LENGTH;i++){
      *pL = *pL + 1024*sinf(0.5*i);
      pL++;
    }
*/

/* AM modulation
    for(i=0;i < BUFFER_LENGTH;i++){
      *pR = *pL * *pR; // right = L*R
      *pL = *pL + *pR; // left = L*(1+R)
      pL++;
      pR++;
    }
*/
    buffer_ready = 0;
}

////////////////////////////////////////////////////////////////////////
// Purpose:   Access function for buffer ready flag
//
// Input:     None
//
// Returns:   Non-zero when a buffer is ready for processing
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
int IsBufferReady()
{
    return buffer_ready;
}

////////////////////////////////////////////////////////////////////////
// Purpose:   Access function for buffer overrun flag
//
// Input:     None
//
```

```
// Returns:   Non-zero if a buffer overrun has occurred
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
int IsOverRun()
{
    return over_run;
}


interrupt void Codec_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       /* add any local variables here */
    static Uint8 fill_index = INITIAL_FILL_INDEX; // index of buffer to fill
    static Uint8 dump_index = INITIAL_DUMP_INDEX; // index of buffer to dump
    static Uint32 sample_count = 0; // current sample count in buffer

       if(CheckForOverrun())                                    // overrun error occurred
(i.e. halted DSP)
              return;                                                       // so serial
port is reset to recover

       CodecDataIn.UINT = ReadCodecData();             // get input data samples

       /* add your code starting here */

    // store input in buffer
    buffer[fill_index][ LEFT][sample_count] = CodecDataIn.Channel[ LEFT];
    buffer[fill_index][RIGHT][sample_count] = CodecDataIn.Channel[RIGHT];

       // bound output data before packing
       // use saturation of SPINT to limit to 16-bits
       CodecDataOut.Channel[ LEFT] = _spint(buffer[dump_index][ LEFT][sample_count] *
65536) >> 16;
       CodecDataOut.Channel[RIGHT] = _spint(buffer[dump_index][RIGHT][sample_count] *
65536) >> 16;
    // pack output data without bounding
//  CodecDataOut.channel[ LEFT] = buffer[dump_index][LEFT][sample_count];
//  CodecDataOut.channel[RIGHT] = buffer[dump_index][RIGHT][sample_count];

   // update sample count and swap buffers when filled
    if(++sample_count >= BUFFER_LENGTH) {
        sample_count = 0;
        ready_index = fill_index;
        if(++fill_index >= NUM_BUFFERS)
```

```
            fill_index = 0;
        if(++dump_index >= NUM_BUFFERS)
            dump_index = 0;
        if(buffer_ready == 1) // set a flag if buffer isn't processed in time
            over_run = 1;
        buffer_ready = 1;
    }


       /* end your code here */

       WriteCodecData(CodecDataOut.UINT);       // send output data to  port
}
```

Step 2

ISRs.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

///////////////////////////////////////////////////////////////////////
// Filename: ISRs.c
//
// Synopsis: Interrupt service routines for EDMA
//
///////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "math.h"
#include "frames.h"
#include "coeff.h"      // load the filter coefficients, B[n] ... extern

// frame buffer declarations
#define BUFFER_COUNT        1024   // buffer length in McBSP samples (L+R)
#define BUFFER_LENGTH       BUFFER_COUNT*2 // two shorts read from McBSP each time
#define NUM_BUFFERS         3     // don't change this!

#pragma DATA_SECTION (buffer, "CE0"); // allocate buffers in SDRAM
Int16 buffer[NUM_BUFFERS][BUFFER_LENGTH];
// there are 3 buffers in use at all times, one being filled from the McBSP,
// one being operated on, and one being emptied to the McBSP
// ready_index --> buffer ready for processing
volatile Int16 buffer_ready = 0, over_run = 0, ready_index = 0;

void EDMA_Init()
///////////////////////////////////////////////////////////////////////
// Purpose:   Configure EDMA controller to perform all McBSP servicing.
//            EDMA is setup so buffer[2] is outbound to McBSP, buffer[0] is
//            available for processing, and buffer[1] is being loaded.
//            Conditional statement ensure that the correct EDMA events are
//            used based on the McBSP that is being used.
//            Both the EDMA transmit and receive events are set to automatically
//            reload upon completion, cycling through the 3 buffers.
//            The EDMA completion interrupt occurs when a buffer has been filled
//            by the EDMA from the McBSP.
```

```
//            The EDMA interrupt service routine updates the ready buffer index,
//            and sets the buffer ready flag which is being polled by the main
//            program loop
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       EDMA_params* param;

       // McBSP tx event params
       param = (EDMA_params*)(EVENTE_PARAMS);
       param->options = 0x211E0002;
       param->source = (unsigned int)(&buffer[2][0]);
       param->count = (0 << 16) + (BUFFER_COUNT);
       param->dest = 0x34000000;
       param->reload_link = (BUFFER_COUNT << 16) + (EVENTN_PARAMS & 0xFFFF);

       // set up first tx link param
       param = (EDMA_params*)EVENTN_PARAMS;
       param->options = 0x211E0002;
       param->source = (unsigned int)(&buffer[0][0]);
       param->count = (0 << 16) + (BUFFER_COUNT);
       param->dest = 0x34000000;
       param->reload_link = (BUFFER_COUNT << 16) + (EVENTO_PARAMS & 0xFFFF);

       // set up second tx link param
       param = (EDMA_params*)EVENTO_PARAMS;
       param->options = 0x211E0002;
       param->source = (unsigned int)(&buffer[1][0]);
       param->count = (0 << 16) + (BUFFER_COUNT);
       param->dest = 0x34000000;
       param->reload_link = (BUFFER_COUNT << 16) + (EVENTP_PARAMS & 0xFFFF);

       // set up third tx link param
       param = (EDMA_params*)EVENTP_PARAMS;
       param->options = 0x211E0002;
       param->source = (unsigned int)(&buffer[2][0]);
       param->count = (0 << 16) + (BUFFER_COUNT);
       param->dest = 0x34000000;
       param->reload_link = (BUFFER_COUNT << 16) + (EVENTN_PARAMS & 0xFFFF);


       // McBSP rx event params
       param = (EDMA_params*)(EVENTF_PARAMS);
       param->options = 0x203F0002;
       param->source = 0x34000000;
       param->count = (0 << 16) + (BUFFER_COUNT);
       param->dest = (unsigned int)(&buffer[1][0]);
       param->reload_link = (BUFFER_COUNT << 16) + (EVENTQ_PARAMS & 0xFFFF);

       // set up first rx link param
       param = (EDMA_params*)EVENTQ_PARAMS;
```

```
        param->options = 0x203F0002;
        param->source = 0x34000000;
        param->count = (0 << 16) + (BUFFER_COUNT);
        param->dest = (unsigned int)(&buffer[2][0]);
        param->reload_link = (BUFFER_COUNT << 16) + (EVENTR_PARAMS & 0xFFFF);

        // set up second rx link param
        param = (EDMA_params*)EVENTR_PARAMS;
        param->options = 0x203F0002;
        param->source = 0x34000000;
        param->count = (0 << 16) + (BUFFER_COUNT);
        param->dest = (unsigned int)(&buffer[0][0]);
        param->reload_link = (BUFFER_COUNT << 16) + (EVENTS_PARAMS & 0xFFFF);

        // set up third rx link param
        param = (EDMA_params*)EVENTS_PARAMS;
        param->options = 0x203F0002;
        param->source = 0x34000000;
        param->count = (0 << 16) + (BUFFER_COUNT);
        param->dest = (unsigned int)(&buffer[1][0]);
        param->reload_link = (BUFFER_COUNT << 16) + (EVENTQ_PARAMS & 0xFFFF);

        *(unsigned volatile int *)ECR = 0xf000;   // clear all McBSP events
        *(unsigned volatile int *)EER = 0xC000;
        *(unsigned volatile int *)CIER = 0x8000; // interrupt on rx reload only
}

void ZeroBuffers()
//////////////////////////////////////////////////////////////////////////
// Purpose:   Sets all buffer locations to 0
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
//
// Notes:     None
//////////////////////////////////////////////////////////////////////////
{
    Int32 i = BUFFER_COUNT * NUM_BUFFERS;
    Int32 *p = (Int32 *)buffer;

    while(i--)
        *p++ = 0;
}

void ProcessBuffer()
//////////////////////////////////////////////////////////////////////////
// Purpose:   Processes the data in buffer[ready_index] and stores
//            the results back into the buffer
//            Data is packed into the buffer, alternating right/left
//              Be careful of the order of packing
//
// Input:     None
//
// Returns:   Nothing
//
```

```
// Calls:     Nothing
//
// Notes:     None
/////////////////////////////////////////////////////////////////////////
{
    Int16 *pBuf = buffer[ready_index];
    // extra buffer room for convolution "edge effects"
    // N is filter order from coeff.h
    static float Left[BUFFER_COUNT+N]={0}, Right[BUFFER_COUNT+N]={0};
    float *pL = Left, *pR = Right;
    float yLeft, yRight;
    Int32 i, j, k;

    // offset pointers to start filling after N elements
    pR += N;
    pL += N;

    for(i = 0;i < BUFFER_COUNT;i++) { // extract data to float buffers
    // order is important here: must go right first then left
       *pR++ = *pBuf++;
       *pL++ = *pBuf++;
    }

    // reinitialize pointer before FOR loop
    pBuf = buffer[ready_index];

////////////////////////////////////////
// Implement FIR filter
// Ensure COEFF.C is part of project
////////////////////////////////////////
   for(i=0;i < BUFFER_COUNT;i++){
      yLeft  = 0;                     // initialize the LEFT output value
      yRight = 0;                     // initialize the RIGHT output value

      for(j=0,k=i;j <= N;j++,k++){
        yLeft  += Left[k] * B[j];        // perform the LEFT dot-product
        yRight += Right[k] * B[j];        // perform the RIGHT dot-product
      }

      // pack into buffer after bounding (must be right then left)
      *pBuf++ = _spint(yRight * 65536) >> 16;
      *pBuf++ = _spint(yLeft * 65536) >> 16;
   }

   // save end values at end of buffer array for next pass
   //  by placing at beginning of buffer array
   for(i=BUFFER_COUNT,j=0;i < BUFFER_COUNT+N;i++,j++){
      Left[j]=Left[i];
      Right[j]=Right[i];
   }

//////// end of FIR routine ///////////

    // reinitialize pointer
    pBuf = buffer[ready_index];

    buffer_ready = 0; // signal we are done
}
```

```
////////////////////////////////////////////////////////////////////////
// Purpose:   Access function for buffer ready flag
//
// Input:     None
//
// Returns:   Non-zero when a buffer is ready for processing
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
int IsBufferReady()
{
    return buffer_ready;
}

////////////////////////////////////////////////////////////////////////
// Purpose:   Access function for buffer overrun flag
//
// Input:     None
//
// Returns:   Non-zero if a buffer overrun has occurred
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
int IsOverRun()
{
    return over_run;
}

interrupt void EDMA_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   EDMA interrupt service routine.  Invoked on every buffer
//            completion
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       *(volatile Uint32 *)CIPR = 0xf000; // clear all McBSP events
    if(++ready_index >= NUM_BUFFERS) // update buffer index
        ready_index = 0;
    if(buffer_ready == 1) // set a flag if buffer isn't processed in time
        over_run = 1;
    buffer_ready = 1; // mark buffer as ready for processing
}
```

NOTE: The same coeff.c and coeff.h were used as in previous labs