Lab 2

UNIVERSITY of WASHINGTON

EE 443 4/14/16

Jake Garrison (1126716)
Jisoo Jung (1236577)

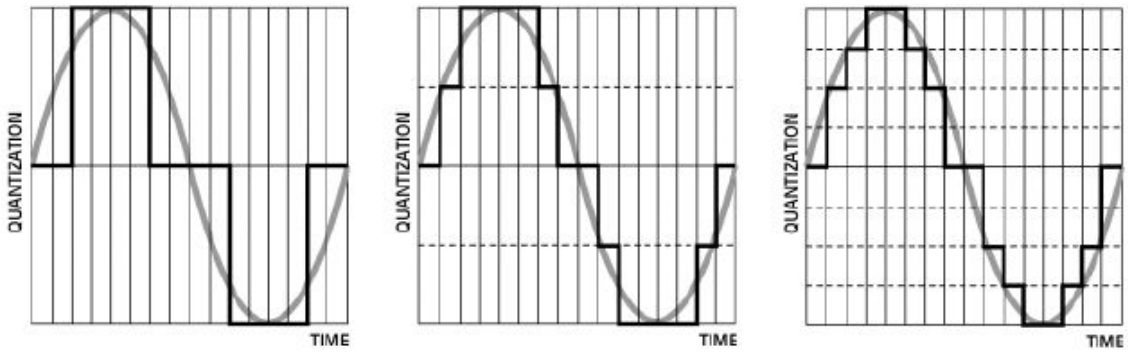
Problem 1

Description

In this task we saturated the mic input different quantization levels. Each setting was assigned to a key interrupt, where key0 had 3 levels, key1, 5 levels and key2, 7 levels. Each level was split into sections based off a constant min and max int that defines the ceiling and floor of the expected input. The lookup tables are shown below. The correct table is used based off the last key that was pressed. The level is selected based off the current sample's amplitude.

We had trouble selecting a good min and max value for human speech. If they are set too high (ie. > +/- 1000) background noise may get quantized as max and lead to a lot of background distortion. If it is set too low, human speech may not be understandable. We calibrated max based off the max amplitude we saw on the mic (due to blowing into it). It may have been better to calibrate based off regular human speech. We ended up using a min of -1000 and max of 9000, but in our demo it didn't work to its full potential. SInce it is difficult to judge the effectiveness based off the oscilloscope or FFT, we opted to just include code snippets to support our description.

In addition to the code shown below, we also had trivial logic in our key interrupts to select different quantization levels as well as code to pipe the output to the line out.



Code

yourISR.h

```
// ------ MAX and MIN -------
#define MAX 9000
#define MIN -1000

// ------ Quantization Table -------
int quantTable3[2] = {
  (2*MAX+6*MIN)/8,
  (6*MAX+2*MIN)/8,
};

int quantTable5[4] = {
  (MAX+7*MIN)/8,
  (3*MAX+5*MIN)/8,
  (5*MAX+3*MIN)/8,
  (7*MAX+MIN)/8,
};

int quantTable7[6] = {
  (MAX+7*MIN)/8,
  (2*MAX+6*MIN)/8,
  (3*MAX+5*MIN)/8,
  (5*MAX+3*MIN)/8,
  (6*MAX+2*MIN)/8,
  (7*MAX+MIN)/8,
};

// ------ Level-n Table -------
float levelTable3[3] = {-1, 0, 1};
float levelTable5[5] = {-1, -0.5, 0, 0.5, 1};
float levelTable7[7] = {-1, -0.75, -0.5, 0, 0.5, 0.75, 1};
```

```
// ---------------------------------
static double getQuantizedValue(int input) {
/*  if (input < 0) {
    input *= -1;
  }*/
  float* quantTable;
  float* levelTable;
  if (level == 7) {
    quantTable = quantTable7;
    levelTable = levelTable7;
  } else if (level == 5) {
    quantTable = quantTable5;
    levelTable = levelTable5;
  } else { // level == 3
    quantTable = quantTable3;
    levelTable = levelTable3;
  }

  int i = 0;
  while (input > quantTable[i]) {
    i++;
  }
  return levelTable[i];
}
```

Problem 2

Description

In this activity we created the delay effect using mic input, switch 0 to toggle the effect, and keys to increase or decrease the delay time. The delay is outputted as follows:

1. The switch0 interrupt checks if the switch is on
2. The last n samples are buffered, where n is 5000 in our code.
3. The delay is set from the `sampleDelay` var which gets updated in our `changeDelay()` function every time key1 or key2 are pressed.
4. A previous sample `index` is updated from the buffer and decayed by with a 0.6 multiplier, this is calculated with the `getEchoSample()` function.
5. The `getEchoSample()` returns the delayed sample which gets summed with the current sample and then outputted to lineout

The logic for the keys, switches and output interrupts are straightforward and rely on the helper functions described above

Code

yourISR.h: The code snippets below represent the steps above.

```
// either -1 or 1
#define MAX_DELAY 1000
#define MIN_DELAY 0
void changeDelay(short sign) {
  sampleDelay = sampleDelay + sign * delayInc;
  if (sampleDelay < MIN_DELAY) {
    sampleDelay = MIN_DELAY;
  } else if (sampleDelay > MAX_DELAY) {
    sampleDelay = MAX_DELAY;
  }
  printf("sampleDelay = %d\n", sampleDelay);
}
```

```
// calculate current sample
// output sum
// put it into buffer
int x_t = sw0State * getEchoSample(sampleDelay) + leftChanr
IOWR_ALTERA_AVALON_PIO_DATA(LEFTSENDDATA_BASE, x_t);
delayedBuffer[leftCount] = x_t;
// reset leftCount to zero if it reaches 512*/
leftCount = (leftCount + 1) % BUFFER_LENGTH;
```

```
104  static int getEchoSample(int index) {
105    if (index == 0) {
106      return 0;
107    }
108    return gain * delayedBuffer[(BUFFER_LENGTH + leftCount - index) % BUFFER_LENGTH];
109  }
```

Problem 3

Description

In this task, we wrote assembly code to reverse a buffered chunk of size 128 and used it in the C code to reverse the input in realtime. We also exported the original unprocessed samples followed by the reversed chunks of size 128 on UART with a `UART_BUFFER_SIZE` of 3*128 to capture three chunks. The resulting sound was not what we expected. Speech input sounded like a distorted pitched down and warped. It is possible that the 128 sized chunks are too small to notice the reversal.

The assembly code, `flip.s`, is shown below and essentially reverses the vector by first moving dst pointer to the last element, and then looping backwards through it. The pointer for src loops from the first element to the last element and each element is stored in an output vector in the location pointed by dst. Our method only requires a single loop and some temp storage.
If the assembly code were to be described in C, the following code achieves the same result as the assembly code:

```
1  void flip(int src[], int dst[], int N) {
2      int i;
3      for (i = 0; i < N; i++) {
4          dst[i] = src[N - 1 - i];
5      }
6  }
```

Most of our logic is in the `handle_leftready_interrupt_test()` which handles calling flip on the incoming data, and buffering the data using the code provided in the homework description. In addition, the uart data is packaged in this function so it can be sent with key0.

Code

flip.s

```
1   .global flip
2   # void flip(int src[], int dst[], int N)
3   #   - Reverses src's elements and put them in dst
4   #   - N is the number of elements in src and dst.
5   #   - Length of src and dst must be equal
6   # r4 = src,  r5 = dst, r6 = N
7
8   flip:
9     muli r6, r6, 4      # N = N * 4
10    add  r5, r5, r6     # dst += (initial N)
11    subi r5, r5, 4      # dst--; Now dst is at end
12
13    LOOP:
14      ldw  r9, 0(r4)    # temp = *src
15      stw  r9, 0(r5)    # *dst = temp
16      addi r4, r4, 4    # src++
17      subi r5, r5, 4    # dst--
18      subi r6, r6, 4    # N -= 4
19      bgt  r6, r0, LOOP # if (N > 0) { LOOP } else { DONE }
```

Explanation

The flip function takes three arguments: src, dst, and N, which are stored in argument register r4, r5, and r6, respectively. The first step is to have dst to point to the very its last element. Since each element in the array is 4-byte in size, the address of the last element is the address of dst + 4 * (N - 1) [NOTE: the description above the line 10 is C interpretation. It's written as dst += (initial N) because of pointer arithmetic]. Then in the loop the first element address at src gets loaded to the temporary variable which is stored in r9. Then this data is written to the address at dst. Then src pointer increments by one element and dst pointer decrements by one element (using addi and subi). In order to indicate when to stop the loop, N is decremented by one element size such that the loop stops when N becomes less than or equal to zero. N can be decremented by 4 because it was initially multiplied by 4 to locate the dst pointer to the last element.

yourISR.h

```
static void handle_leftready_interrupt_test(void* context, alt_u32 id) {
        volatile int* leftreadyptr = (volatile int *)context;
        *leftreadyptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(LEFTREADY_BASE);
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(LEFTREADY_BASE, 0);
        /*******Read, playback, store data*******/
        leftChannel = unsigned2signed(IORD_ALTERA_AVALON_PIO_DATA(LEFTDATA_BASE));
        if(leftCount < FRAME_SIZE){
                data1[leftCount] = leftChannel;
                if(leftCount == 0) {
                        flip(data2, flip_data, FRAME_SIZE);
                }
        } else {
                data2[leftCount % FRAME_SIZE] = leftChannel;
                if(leftCount == FRAME_SIZE) {
                        flip(data1, flip_data, FRAME_SIZE);
                }
        }
        // flipped  321 321 321
        // original 123 123 123
        int flipIndex = leftCount % FRAME_SIZE;
        IOWR_ALTERA_AVALON_PIO_DATA(LEFTSENDDATA_BASE, flip_data[flipIndex]);
        // original
        datatest[uartIndex] = flip_data[FRAME_SIZE - 1 - flipIndex];
        // flipped
        datatest[uartIndex + (3 * FRAME_SIZE)] = flip_data[flipIndex];
        uartIndex = (uartIndex+1) % (UART_BUFFER_SIZE / 2);  // [0, 3*frame size-1]
        leftCount = (leftCount+1) % BUFFER_SIZE;
}
```

Explanation

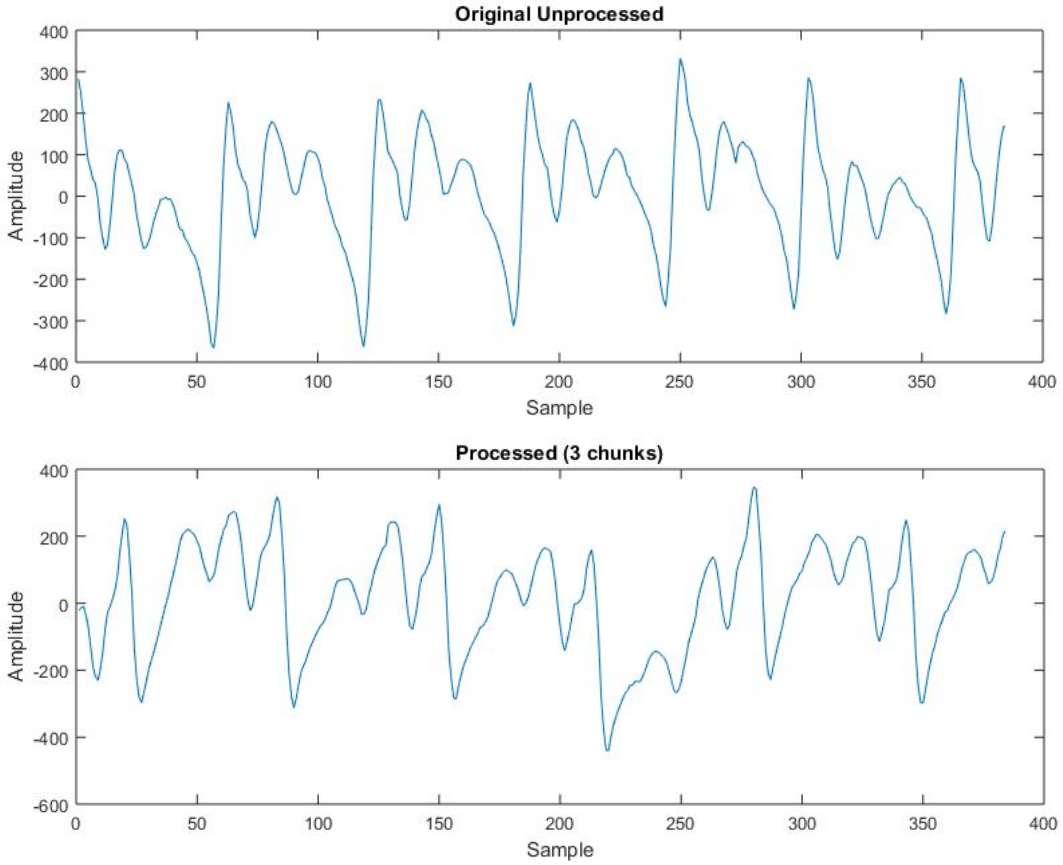
At every 0th and 128th sample flipped_data is repopulated with the reversed data from either data1 or data2. Because the all the samples in a single frame are contained in the flipped_data the original unflipped samples are located at the FRAME_SIZE - 1 flipIndex. The unflipped and flipped data are simply located symmetrical about the center of the buffer. In order to send the original and the flipped data, a buffer called datatest was initialized with 6 * FRAME_SIZE, holding the original buffer followed by the reversed buffer.

```
original = x(1:128*3);
flipped = x(128*3+1:end);
```

In the Matlab program the buffer gets evenly divided into half. The first half of the buffer is the original and then another one is the flipped buffer.

Results

The results from Matlab are shown below. Essentially, both plots are of length 384. The unprocessed plot is the original signal, and the processed plot contains 3 chunks of size 128 reversed. To Validate the processed signal is indeed correct, each size 128 chunk can be reversed in place. The result is almost identical to the original signal, except shifted slightly due to hardware latency.



Problem 4

Description

In this question, we were instructed to utilize a sum of products (SoP) algorithm written in assembly to create a IIR low pass filter using the given coefficients in C. IIR filter is described by the following equations:

$$H(z) = \frac{\sum_{m=0}^{M-1} b_m z^{-m}}{1 + \sum_{l=1}^{N-1} a_l z^{-l}} = 4.29 \times 10^{-3} \frac{1 + z^{-1}}{1 - 0.801 z^{-1}} \frac{1 + 2z^{-1} + z^{-2}}{1 - 1.638 z^{-1} + 0.81 z^{-2}}$$

$$y(n) = -\sum_{l=1}^{N-1} a_l y(n-l) + \sum_{m=0}^{M-1} b_m x(n-m)$$

In summary, y(n) can be calculated using SoP and coefficients a_l and y(n - l), and coefficients b_m and x(n - m). The coefficients can be obtained from expanding the transfer function H(z) such that coefficients of the numerator becomes b_m and coefficients in the denominator becomes a_l. Because SoP requires both buffers to be in the same length, buffers a_l and y were in the same length (same idea applies for b_m and x). This implementation did not involve using circular buffers because the provided SoP function required elements in both buffers to be located in line with the corresponding coefficients. I.e. h(0)x(n)+h(1)x(n-1)+ ... + h(N-1)x(n-(N-1)). Thus input buffer shifts older data to put the new data in the first index.

Code

yourISR.h

```
static void handle_leftready_interrupt_test(void* context, alt_u32 id) {
        volatile int* leftreadyptr = (volatile int *)context;
        *leftreadyptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(LEFTREADY_BASE);
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(LEFTREADY_BASE, 0);

        /******* Apply Filter *******/
        input = unsigned2signed(IORD_ALTERA_AVALON_PIO_DATA(LEFTDATA_BASE));

        shiftInsert(input, x, M);

        // yn = - (sum from i = 1 to L - 1(a_i * y(n - i)))
        //      + (sum from i = 1 to M - 1(b_i * x(n - i)))
        yn = (sop(x, b, M) - sop(y, a, L)) / 10000;

        shiftInsert(yn, y, L);
        //scale output (shift right 15 bits)
        //IOWR_ALTERA_AVALON_PIO_DATA(LEFTSENDDATA_BASE, yn >> 15);
        IOWR_ALTERA_AVALON_PIO_DATA(LEFTSENDDATA_BASE, yn);

        // store data to be sent to Matlab
        UARTOutput[leftCount] = yn;
        UARTInput[leftCount] = input;
        leftCount = (leftCount + 1) % UART_BUFFER_SIZE;
}
```

Explanation

Algorithm for applying IIR filter for each new input value is done as the following:

1. Store input data in temporary variable
2. Insert the input data to the first index of buffer x
3. Compute y_n using SoP operation
4. Insert y_n to the first index of buffer y
5. Output y_n to LINE_OUT

Note that the filter coefficients are multiplied by 10,000 and y_n was divided by the same amount. When the variables stored the real numbers (of type float), the generated sound was just a loud noise. The type of input value returned from IORD_ALTERA_AVALON_PIO_DATA(LEFTDATA_BASE) is int and when the other variables were all converted back to integers the filter functioned properly. Because the filter coefficients were in decimal numbers with some of them being less than 1, the coefficients were multiplied by 10,000 to maintain all the decimal values. Then the output was divided by 10,000 to compensate the initial gain.

sop.s

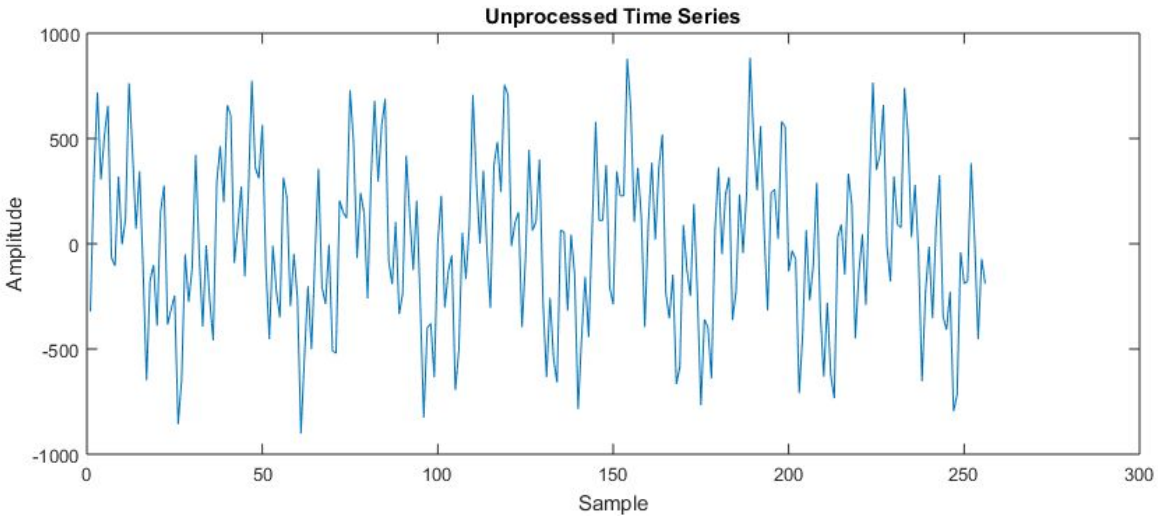
```
.global sop
/*
r4 - data (x)
r5 - coefficients (h)
r6 - N
*/
sop:
    add r2, r0, r0
    LOOP:
        ldw r9, 0(r4)
        ldw r10, 0(r5)
        mul r8, r9, r10
        add r2, r2, r8
        addi r4, r4, 4
        addi r5, r5, 4
        subi r6, r6, 1
        bgt r6, r0, LOOP
    ret
```

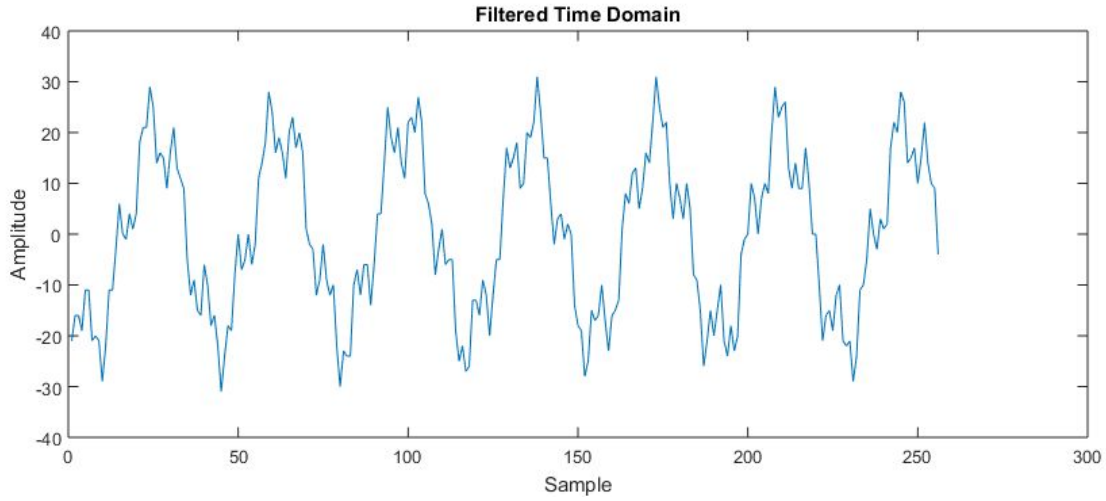
This is the provided SoP function written in assembly code. Basically, it takes in three arguments: array x, h, and an int N. Both arrays have the same N number of elements. The returned value is the sum of two buffers' element-wise multiplication.

Results

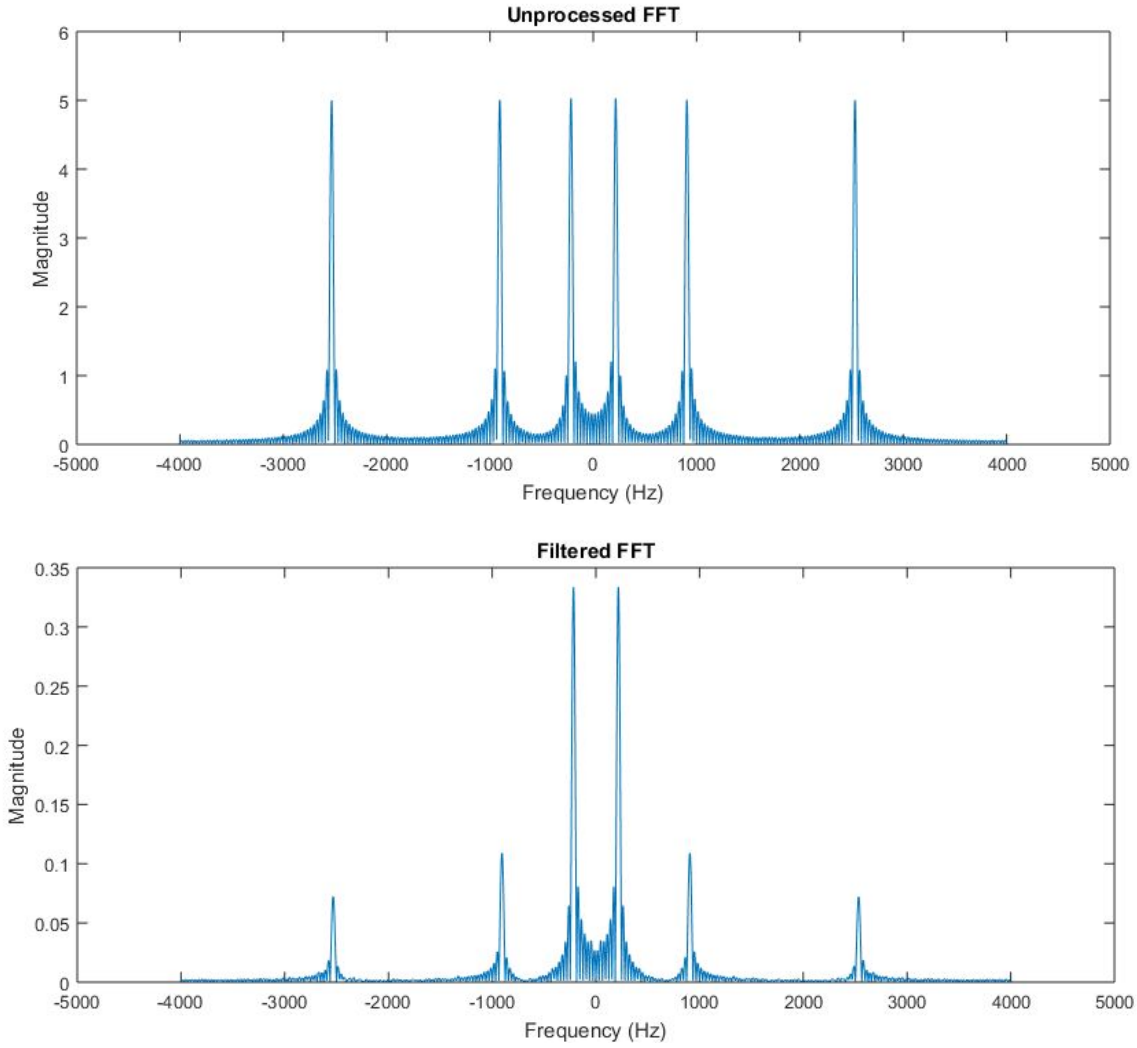
The resulting effect on the input sound is precisely what would be expected from a low pass filter. See the figures below to visualize the effect. Compared to the original input, in which all frequencies have equal magnitude, in filtered version of the input, it is clear that as the frequencies increase, they also decay more.

The input we used for the filter came from a sound generating phone app. It is capable of outputting three sine waves simultaneously at specified frequencies. For this demonstration, we set the three sine oscillators to 210 hz, 880 hz and 2.5 khz respectively.





One effect that LPF has in time domain is that the filter gets rid of sudden fluctuations in the data. Therefore, the resulting plot shows the general trend of the original plot. It smoothes out the original plot.



Notice that in the filtered plot, the magnitude of high frequency is significantly smaller than that of low frequency. It shows that the IIR LPF is working properly.

Problem 5

Description

In this part we toggled a stereo mix effect using switch0. When the switch is off, the original mix is played. When the switch is on, the left and right channels are somewhat merged creating an alternative more overlapped stereo effect. We tested this using a music track with defined left and right channels such that drums were on the left channel and vocals and guitar on the right. After turning the effect on, the isolated left and right channels cleared bleed into the opposite channel creating a more centered, stereo mix.

Code

yourISR.h

```
81  //Stereo SWITCH
82  int stereo_on = 1;
83  static void handle_switch0_interrupt(void* context, alt_u32 id) {
84     volatile int* switch0ptr = (volatile int *)context;
85     *switch0ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(SWITCH0_BASE);
86
87     /* Write to the edge capture register to reset it. */
88     IOWR_ALTERA_AVALON_PIO_EDGE_CAP(SWITCH0_BASE, 0);
89
90     // Set mute state
91     stereo_on = IORD_ALTERA_AVALON_PIO_DATA(SWITCH0_BASE);
92     printf("Stereo ON: %d\n", stereo_on);
93  }
```

Explanation

The code above defines the logic for switch0, which toggles the effect. The variable `stereo_on` is either 0 or 1 and gets multiplied by the additional channel that is combined with the original channel, thus enabling or disabling the effect. The code below demonstrates this. Notice the left output is defined by : `leftChannel + stereo_on * rightChannel`, and the right chennel : `rightChannel - stereo_on * leftChannel`. This indicates if `stero_on` is 0, the original unprocessed sound is outputted. Since it is difficult to see this effect in via oscilloscope reading, no figures are included.

yourISR.h

```
168  /*
169   * Stereo Mix Logic
170   * Left when stereo_on = 0
171   * Right + Left when stereo_on = 1
172   */
173  static void handle_leftready_interrupt_test(void* context, alt_u32 id) {
174     volatile int* leftreadyptr = (volatile int *)context;
175     *leftreadyptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(LEFTREADY_BASE);
176     IOWR_ALTERA_AVALON_PIO_EDGE_CAP(LEFTREADY_BASE, 0);
177     /*******Read, playback, store data*******/
178     leftChannel = IORD_ALTERA_AVALON_PIO_DATA(LEFTDATA_BASE);
179     IOWR_ALTERA_AVALON_PIO_DATA(LEFTSENDDATA_BASE, (leftChannel + stereo_on * rightChannel));
180     leftCount = (leftCount+1)%256;
181  //   /************************************/
182
183  }
184
185  /*
186   * Stereo Mix Logic
187   * Right when stereo_on = 0
188   * Right - Left when stereo_on = 1
189   */
190  static void handle_rightready_interrupt_test(void* context, alt_u32 id) {
191     volatile int* rightreadyptr = (volatile int *)context;
192     *rightreadyptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(RIGHTREADY_BASE);
193     IOWR_ALTERA_AVALON_PIO_EDGE_CAP(RIGHTREADY_BASE, 0);
194     /*******Read, playback, store data*******/
195     rightChannel = IORD_ALTERA_AVALON_PIO_DATA(RIGHTDATA_BASE);
196     IOWR_ALTERA_AVALON_PIO_DATA(RIGHTSENDDATA_BASE, (rightChannel - stereo_on * leftChannel));
197     rightCount = (rightCount+1) % BUFFERSIZE;
198     /************************************/
199  }
```