ENCM 515: Digital Signal Processors – Lab 2

Group #	7	Date: March 16, 2023
Student name(s)	Hanan Anam	
	Nadia Duarte Amador	
Instructor/TA	Dr. Tan	

To submit:

- **Add your lab sheet (as PDF if possible) and any other added/modified .c or .h files to a zip archive and upload to D2L.**

PART ONE

Q1: How many seconds of audio is there in the input data?

number of samples * T = 64 000 samples * 1/8000Hz = 3 seconds.

Q2> Run the debugger until filteredOutBufferA is full. Take a screenshot of the contents of the buffer, as below. Do you get the expected output?

Expression	Type	Value
filteredOutBufferA	volatile int32_t [32]	0x200005e4 <filteredOut...
filteredOutBufferA[0]	volatile int32_t	19398952
filteredOutBufferA[1]	volatile int32_t	134088702
filteredOutBufferA[2]	volatile int32_t	231476684
filteredOutBufferA[3]	volatile int32_t	299241942
filteredOutBufferA[4]	volatile int32_t	330240943
filteredOutBufferA[5]	volatile int32_t	320213782
filteredOutBufferA[6]	volatile int32_t	270995495
filteredOutBufferA[7]	volatile int32_t	187173672
filteredOutBufferA[8]	volatile int32_t	79430844
filteredOutBufferA[9]	volatile int32_t	-39846496
filteredOutBufferA[10]	volatile int32_t	-155388227
filteredOutBufferA[11]	volatile int32_t	-253693727
filteredOutBufferA[12]	volatile int32_t	-321786670
filteredOutBufferA[13]	volatile int32_t	-351147246
filteredOutBufferA[14]	volatile int32_t	-336729106
filteredOutBufferA[15]	volatile int32_t	-280563897
filteredOutBufferA[16]	volatile int32_t	-188418875
filteredOutBufferA[17]	volatile int32_t	-71697478
filteredOutBufferA[18]	volatile int32_t	55771987
filteredOutBufferA[19]	volatile int32_t	177343122
filteredOutBufferA[20]	volatile int32_t	277811343
filteredOutBufferA[21]	volatile int32_t	343807102
filteredOutBufferA[22]	volatile int32_t	366876126
filteredOutBufferA[23]	volatile int32_t	342365288
filteredOutBufferA[24]	volatile int32_t	272633920
filteredOutBufferA[25]	volatile int32_t	164956629
filteredOutBufferA[26]	volatile int32_t	32047593
filteredOutBufferA[27]	volatile int32_t	-111675048
filteredOutBufferA[28]	volatile int32_t	-249499359
filteredOutBufferA[29]	volatile int32_t	-366220756
filteredOutBufferA[30]	volatile int32_t	-448338617
filteredOutBufferA[31]	volatile int32_t	-488185113

Yes, the expected output is an array of the length of BUFFER_SIZE, 32. This is the expected output because no samples are missed, and the first value matches that shown in the lab document. FUNCTIONAL_TEST skips the interrupt handler and instead processes the audio in a loop which is a useful way to test the results.

Q3> Run the debugger again until filteredOutBufferA is full. Take a screenshot of the contents of the buffer, as below. Do you get the expected output?

Expression	Type	Value
filteredOutBufferA	volatile int32_t [32]	0x200005e4 <filteredOut...
filteredOutBufferA[0]	volatile int32_t	-55771987
filteredOutBufferA[1]	volatile int32_t	-47645399
filteredOutBufferA[2]	volatile int32_t	-38994515
filteredOutBufferA[3]	volatile int32_t	-29491650
filteredOutBufferA[4]	volatile int32_t	-21889358
filteredOutBufferA[5]	volatile int32_t	-15859954
filteredOutBufferA[6]	volatile int32_t	-13435085
filteredOutBufferA[7]	volatile int32_t	-13238474
filteredOutBufferA[8]	volatile int32_t	-15794417
filteredOutBufferA[9]	volatile int32_t	-18350360
filteredOutBufferA[10]	volatile int32_t	-20250933
filteredOutBufferA[11]	volatile int32_t	-18350360
filteredOutBufferA[12]	volatile int32_t	-12714178
filteredOutBufferA[13]	volatile int32_t	-1703962
filteredOutBufferA[14]	volatile int32_t	12714178
filteredOutBufferA[15]	volatile int32_t	29622724
filteredOutBufferA[16]	volatile int32_t	44958382
filteredOutBufferA[17]	volatile int32_t	56296283
filteredOutBufferA[18]	volatile int32_t	59114374
filteredOutBufferA[19]	volatile int32_t	51708693
filteredOutBufferA[20]	volatile int32_t	31457760
filteredOutBufferA[21]	volatile int32_t	-458759
filteredOutBufferA[22]	volatile int32_t	-43188883
filteredOutBufferA[23]	volatile int32_t	-92472707
filteredOutBufferA[24]	volatile int32_t	-144902307
filteredOutBufferA[25]	volatile int32_t	-194448279
filteredOutBufferA[26]	volatile int32_t	-237702699
filteredOutBufferA[27]	volatile int32_t	-270012440
filteredOutBufferA[28]	volatile int32_t	-290459984
filteredOutBufferA[29]	volatile int32_t	-297669054
filteredOutBufferA[30]	volatile int32_t	-294326667
filteredOutBufferA[31]	volatile int32_t	-282202322
Add new expression		

740	ITM Port 31	1	6417313	64.173130 ms
741	ITM Port 30	10	6429670	64.296700 ms
742	ITM Port 31	2	6430235	64.302350 ms
743	ITM Port 31	1	6442319	64.423190 ms
744	ITM Port 30	10	6454667	64.546670 ms
745	ITM Port 31	2	6455238	64.552380 ms
746	ITM Port 31	1	6467315	64.673150 ms
747	ITM Port 30	10	6479664	64.796640 ms
748	ITM Port 31	2	6480239	64.802390 ms

By disabling the FUNCTIONAL_TEST, we have enabled a periodic interrupt that assigns a value of 10 to ITM Port 30 whenever a sample is missed. Upon examining the SWV trace log, we discovered that approximately half of the samples were missed, resulting in incorrect output.

PART TWO

Q4> How much time/how many cycles are required for each sample, with the original ProcessSample function? Take a screenshot of the generated assembly code and explain where you might think the main bottlenecks are.

To measure the time required to process each sample, we added writes to ITM port 31 before and after the function call. Looking at the timings from the SWV Trace Log in question 3,

64.302350 ms - 64.173130 ms = 129.22 us

Since new samples are arriving every 125 us, each sample cannot finish being processed before the next one arrives.

After examining the assembly code, it is apparent that the convolution loop in the accumulator uses around nine assembly instructions per iteration. The function is taking longer than anticipated because the code used for convolution employs instructions that are executed in the ALU and are not specialized to DSP applications. This may require a greater number of instructions than other instructions that are executed in specialized functional units with DSP operations such as the MAC functional unit.

```
static int16_t ProcessSample(int16_t newsample, int16_t* history) {

    // set the new sample as the head
    history[0] = newsample;

    // set up and do our convolution
    int tap = 0;
    int32_t accumulator = 0;
    for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
        accumulator += (int32_t)filter_coeffs[tap] * (int32_t)history
    }

    // shuffle things along for the next one?
```

```
0800129c:   ldr     r2, [pc, #156]   ; (0x800133c <ProcessSample+192>)
0800129e:   ldr     r3, [r7, #20]
080012a0:   ldrsh.w r3, [r2, r3, lsl #1]
080012a4:   mov     r1, r3
080012a6:   ldr     r3, [r7, #20]
080012a8:   lsls    r3, r3, #1
080012aa:   ldr     r2, [r7, #0]
080012ac:   add     r3, r2
080012ae:   ldrsh.w r3, [r3]
080012b2:   mul.w   r3, r1, r3
080012b6:   ldr     r2, [r7, #16]
080012b8:   add     r3, r2
080012ba:   str     r3, [r7, #16]
371           for (tap = 0; tap < NUMBER OF TAPS; tap++) {
```

Similarly, the loop that shuffles the contents of the history buffer after processing each new sample requires about 7 assembly instructions per iteration.

```
    // set up and do our convolution
    int tap = 0;
    int32_t accumulator = 0;
    for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
        accumulator += (int32_t)filter_coeffs[tap] * (int32_t)history
    }

    // shuffle things along for the next one?
    for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
        history[tap+1] = history[tap];
    }

    if (accumulator > 0x3FFFFFFF) {
        accumulator = 0x3FFFFFFF;
        overflow_count++;
    } else if (accumulator < -0x40000000) {
        accumulator = -0x40000000;
        underflow_count++;
```

```
Use the "interrupt" command to stop the tar
and then try again.
080012d0:   lsls    r3, r3, #1
080012d2:   ldr     r2, [r7, #0]
080012d4:   add     r2, r3
080012d6:   ldr     r3, [r7, #20]
080012d8:   adds    r3, #1
080012da:   lsls    r3, r3, #1
080012dc:   ldr     r1, [r7, #0]
080012de:   add     r3, r1
080012e0:   ldrsh.w r2, [r2]
080012e4:   strh    r2, [r3, #0]
080012e6:   ldr     r3, [r7, #20]
080012e8:   subs    r3, #1
080012ea:   str     r3, [r7, #20]
080012ec:   ldr     r3, [r7, #20]
080012ee:   cmp     r3, #0
080012f0:   bge.n   0x80012ce <ProcessSample+82>
380           if (accumulator > 0x3FFFFFFF) {
```

Q5> Create a new ProcessSample2 function, making use of the appropriate MAC instruction. You might remember from class that we used in-line assembly for this. How much time is required for each sample? Do we satisfy timing requirements now? Screenshot/copy your function and explain the changes that you made.

498	ITM Port 31	1	3380953	33.809530 ms
499	ITM Port 31	2	3392666	33.926660 ms
500	ITM Port 31	1	3393451	33.934510 ms
501	ITM Port 31	2	3405161	34.051610 ms
502	ITM Port 31	1	3405943	34.059430 ms
503	ITM Port 31	2	3417653	34.176530 ms

Now,

33.926660 ms - 33.809530 ms = 117.13 us

This satisfies the timing requirements because each sample is processed in less than 125 us, and no samples were missed. The modifications we implemented were in the convolution operation, where we replaced instructions that used the ALU with the MAC instruction by employing in-line assembly, which performs the convolution in the MAC functional unit. The syntax for the assembly instruction SMLABB is as follows:

SMLABB <Result>, <First multiply operand register source>, <Second multiply operand register source>, <Register that contains the accumulated value>. Where filter_coeffs[tap] is the first multiply operand and history[tap] is the second multiply operand, and accumulator is the register that contains the accumulate value.

```c
static int16_t ProcessSample2(int16_t newsample, int16_t* history) {
    // set the new sample as the head
        history[0] = newsample;

        // set up and do our convolution
        int tap = 0;
        int32_t accumulator = 0;
        for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
            __asm volatile ("SMLABB %[result], %[op1], %[op2], %[acc]"
                : [result] "=r" (accumulator)
                : [op1] "r" (filter_coeffs[tap]), [op2] "r" (history[tap]), [acc] "r" (accumulator)
            );
        }

        // shuffle things along for the next one?
        for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
            history[tap+1] = history[tap];
        }

        if (accumulator > 0x3FFFFFFF) {
            accumulator = 0x3FFFFFFF;
            overflow_count++;
        } else if (accumulator < -0x40000000) {
            accumulator = -0x40000000;
            underflow_count++;
        }

        int16_t temp = (int16_t)(accumulator >> 15);

        return temp;
}
```

Q6> Create a new ProcessSample3 function, making use of the appropriate SIMD instruction. You might remember from class that we might be able to use an intrinsic for this. How much time is required for each sample? Do we satisfy timing requirements now? Screenshot/copy your function and explain the changes that you made.

498	ITM Port 31	1	3362511	33.625110 ms
499	ITM Port 31	2	3373238	33.732380 ms
500	ITM Port 31	1	3375005	33.750050 ms
501	ITM Port 31	2	3385729	33.857290 ms
502	ITM Port 31	1	3387492	33.874920 ms
503	ITM Port 31	2	3398216	33.982160 ms

Now, 33.732380 ms - 33.625110 ms = 107.27 us

Yes, we satisfy the time requirements. The function takes 107.27 us to process each sample, giving the processor enough time between each 125 us interrupt. We modified the convolution step by employing the __SMLAD intrinsic to perform convolution. By employing SIMD (Single Instruction Multiple Data) intrinsics, we can enhance the processor's performance by executing the same operation multiple times with different data in a single cycle. So, we only have to iterate through the loop half as many times because we can operate on two array indices at a time. These types of instructions are commonly used for DSP optimization. The syntax of the __SMLAD is as follows:

__SMALD(uint32_t val1, uint32_t val2, uint32_t val3). Where val1 are the first 16-bit operands for each multiplication, val2 are the second 16-bit operands for each multiplication, and val3 is the accumulated value. In our code the loop does two additions per iteration where combined_filter is val1 and combined_history is val2.

```
static int16_t ProcessSample3(int16_t newsample, int16_t* history) {
    // set the new sample as the head
        history[0] = newsample;

        // set up and do our convolution
        int tap = 0;
        int32_t accumulator = 0;
        int32_t combined_filter = 0;
        int32_t combined_history = 0;

        //increment by 2 in the for loop since we do two additions per iteration
        for (tap = 0; tap < NUMBER_OF_TAPS; tap += 2) {
            combined_filter = *(int32_t*)(filter_coeffs + tap); //cast two 16 bit ints to a 32 bit int
            combined_history = *(int32_t*)(history + tap);      //cast two 16 bit ints to a 32 bit int
            accumulator = __SMLAD(combined_filter, combined_history, accumulator);
        }

        // shuffle things along for the next one?
        for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
            history[tap+1] = history[tap];
        }

        if (accumulator > 0x3FFFFFFF) {
            accumulator = 0x3FFFFFFF;
            overflow_count++;
        } else if (accumulator < -0x40000000) {
            accumulator = -0x40000000;
            underflow_count++;
        }

        int16_t temp = (int16_t)(accumulator >> 15);

        return temp;
}
```

Q7> Create a new ProcessSample4 function, this time, treating history_l as a circular buffer. How much time is required for each sample? Do we satisfy timing requirements now? Screenshot/copy your function and explain the changes that you made.

499	ITM Port 31	1	3368184	33.681840 ms
500	ITM Port 31	2	3379029	33.790290 ms
501	ITM Port 31	1	3380683	33.806830 ms
502	ITM Port 31	2	3391527	33.915270 ms
503	ITM Port 31	1	3393177	33.931770 ms
504	ITM Port 31	2	3404020	34.040200 ms

Now it takes

33.790290 ms - 33.681840 ms = 108.45 us to process each sample. This satisfies timing requirements, and when the results are compared to the “functional test” output, they match.

To add the circular buffer, we first created a global variable ‘start’ to keep track of the index of the newest sample.

```
static int16_t start = 0;
```

```
static int16_t ProcessSample4(int16_t newsample, int16_t* history) {

    // set the new sample as the head
    history[start] = newsample;

    // set up and do our convolution
    int tap = 0;
    int32_t accumulator = 0;
    for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
        if(tap > start) { //if we reach the start of the array, loop back to the last element
            accumulator += (int32_t)filter_coeffs[tap] * (int32_t)history[start-tap+NUMBER_OF_TAPS];
        }
        else{ //decrement index of history as i increases
            accumulator += (int32_t)filter_coeffs[tap] * (int32_t)history[start-tap];
        }
    }

    start ++; //increment start of buffer to overwrite the oldest sample
    // we don't need to 'shuffle' the buffer anymore because we've just changed the starting index
    if(start >= NUMBER_OF_TAPS) {
        start = 0;
    }


    if (accumulator > 0x3FFFFFFF) {
        accumulator = 0x3FFFFFFF;
        overflow_count++;
    } else if (accumulator < -0x40000000) {
        accumulator = -0x40000000;
        underflow_count++;
    }

    int16_t temp = (int16_t)(accumulator >> 15);

    return temp;
}
```

We made changes in the convolution loop, since the newest sample is no longer located at index 0. Instead, we start at index ‘start’ and move left through the buffer. Once we reach index 0, we jump to the end of the buffer (which is to the left of index 0 of the buffer in a circular buffer).

Our next change begins on line 486, where we no longer loop through the entire history array to update the samples. Since the buffer is circular, we can just overwrite the oldest sample and update the start index. We've set up our buffer so that the oldest sample is to the right of the start, which is slightly different from the linear interpretation. We chose to set our buffer up 'backwards' because it made the condition on line 478 simpler, and each sample could be processed about 3 us faster. If we had chosen to put the oldest sample to the left of the start, the condition would involve addition, followed by a comparison.

Q8> Write a reflection of what you have observed and learned in this lab, including an explanation of how you checked that your code optimizations preserved functional correctness.

In this lab, we attempted to use various instruction types to execute the convolution operation. We compared the performance of the processor while using the ALU versus the MAC unit by measuring the time taken to complete the ProcessSample function. The MAC unit executes convolution faster as it has direct access to the memory, whereas the ALU is connected to other registers. Additionally, we explored other optimization techniques, such as utilizing a SIMD intrinsic and implementing a circular buffer. After evaluating these various approaches, we can conclude that using SIMD instructions yielded the most significant optimization in this lab. However, the speedup resulting from the use of a circular buffer was comparable. We did not require any complex instructions to achieve significant speedup. This showed us that the best optimizations require an understanding of both hardware and software inefficiencies in the desired operation. To check that each of our optimized functions worked correctly, we checked the results against the original results from Q2 where we disabled the interrupt using FUNCTIONAL_TEST.