ENCM 515: Digital Signal Processors – Lab 3

Group #	7	Date: March 30th
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.**

TASKS

Q1: Write the code for a function to implement the Biquad filter with the following coefficients: `b0 = b2 = 10158, b1 = 20283, {-a1} = -2261, {-a2} = -10060`. Include an annotated version of your source code in your report. Take a screenshot of the assembly code for your function (after compiling), and explain which assembly instructions correspond to which parts of the filter's operation (e.g., what part is for managing the state variables (history of the input/output), what part corresponds to multiplication with the coefficients, etc.) How many cycles are required for processing a sample?

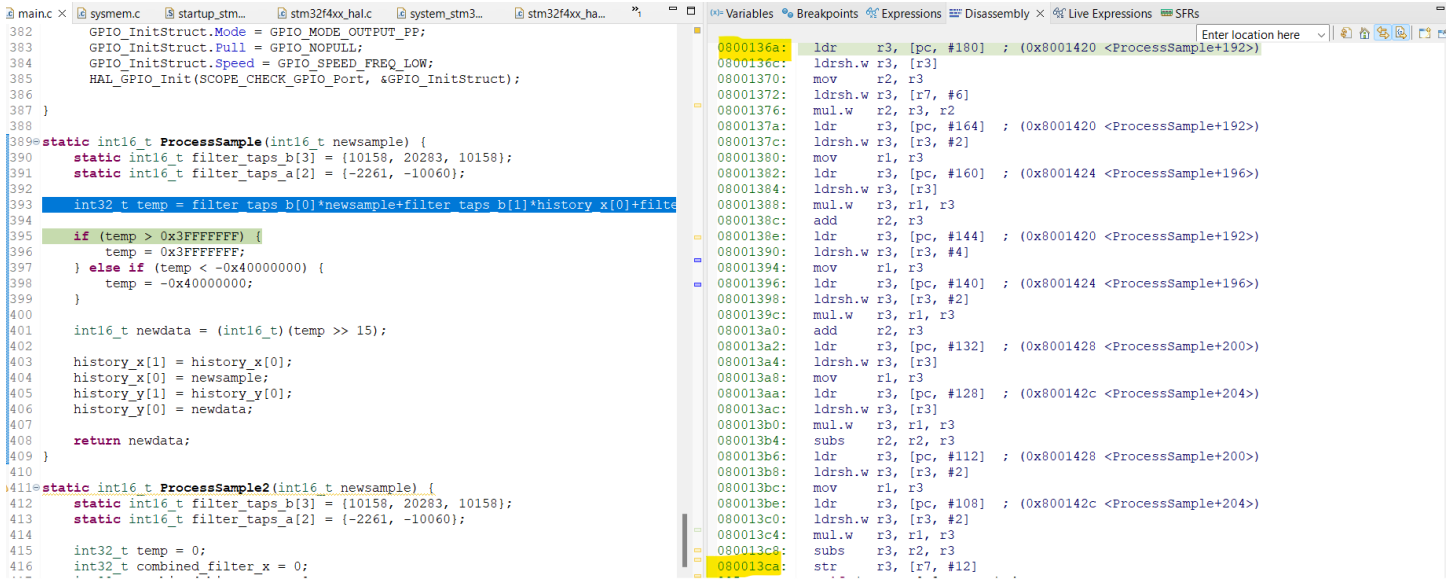
Created global variables

```
55 int16_t history_x[3] = {0,0,0}; //initialize history to 0
56 int16_t history_y[2] = {0,0};
```

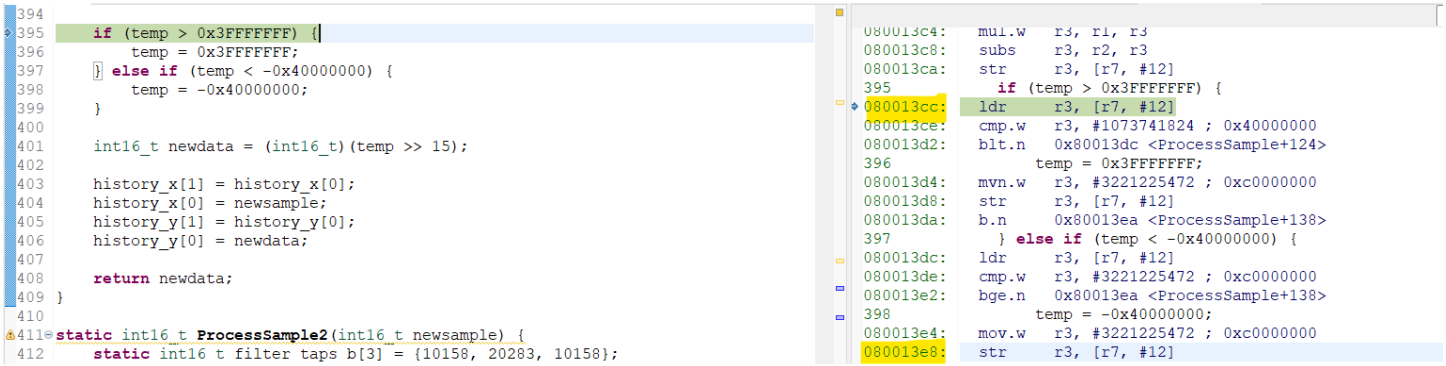
Function:

```
389 static int16_t ProcessSample(int16_t newsample) {
390     static int16_t filter_taps_b[3] = {10158, 20283, 10158};
391     static int16_t filter_taps_a[2] = {-2261, -10060};
392
393     history_x[0] = newsample;
394
395     int32_t temp = filter_taps_b[0]*history_x[0]+filter_taps_b[1]*history_x[1]+filter_taps_b[2]*history_x[2]-
396             filter_taps_a[0]*history_y[0]-filter_taps_a[1]*history_y[1];
397
398     if (temp > 0x3FFFFFFF) {
399         temp = 0x3FFFFFFF;
400     } else if (temp < -0x40000000) {
401         temp = -0x40000000;
402     }
403
404     int16_t newdata = (int16_t)(temp >> 15);
405
406     history_x[2] = history_x[1];
407     history_x[1] = history_x[0];
408     history_y[1] = history_y[0];
409     history_y[0] = newdata;
410
411     return newdata;
412 }
```

Instructions 0800136a-080013ca correspond to the convolution operation in line 393. In the assembly code, the instruction mul.w corresponds to the five filter coefficient multiplications with the new sample and history performed in line 393, similarly, the add and subs instruction perform the addition and subtraction of these five products. The ldrsh.w instruction sign extends the newsample and history half words to 32-bit integers. Mov and ldr correspond to the managing of the history input and output.

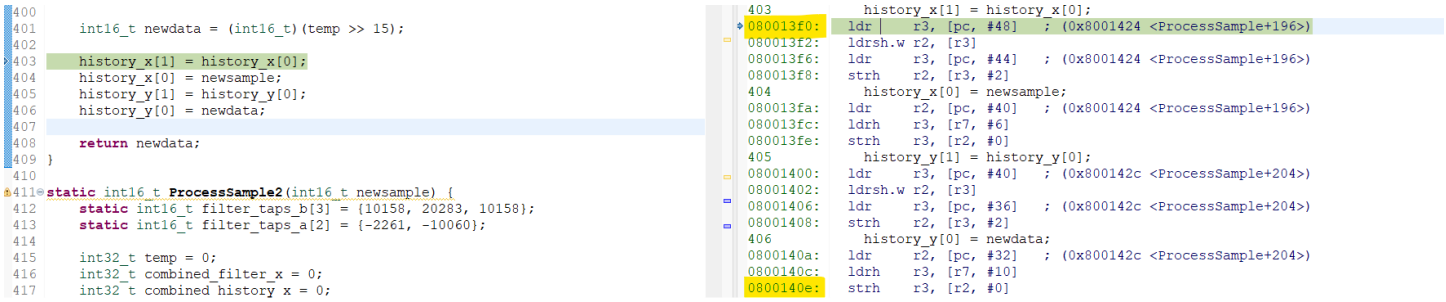


Instructions 080013cc-080013e8 are checking for overflow and underflow. The cmp.w instruction makes the comparison in the if statements, and the blt.n and bge.n instructions branch to the target address if the condition is met. Ldr is used to copy the value from the register that contains the data of the value being compared to.



Instructions 080013ea-080013ee correspond to line 401 in the code, here ldr is used to load the temp value into register r3. Then asrs performs the arithmetic shift to get the upper 16 bits in temp. Strh stores the new value into newdata.

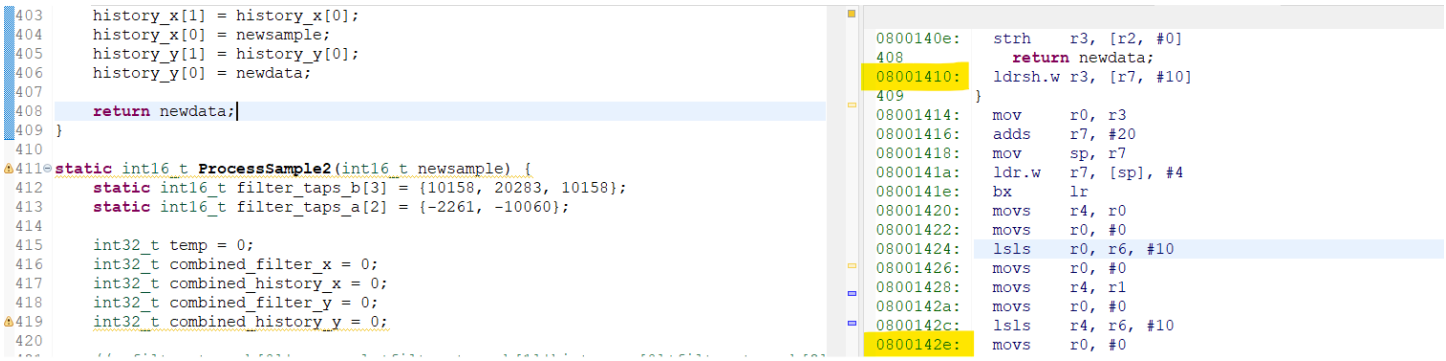
Instructions 080013f0-0800140e correspond to lines 403-406, where we update the history for both the input x and the output y. Ldrsh.w loads the 16-bit history halfword, similarly, strh stores the result into the halfword history.



```
400
401     int16_t newdata = (int16_t)(temp >> 15);
402
403     history_x[1] = history_x[0];
404     history_x[0] = newsample;
405     history_y[1] = history_y[0];
406     history_y[0] = newdata;
407
408     return newdata;
409 }
410
411 static int16_t ProcessSample2(int16_t newsample) {
412     static int16_t filter_taps_b[3] = {10158, 20283, 10158};
413     static int16_t filter_taps_a[2] = {-2261, -10060};
414
415     int32_t temp = 0;
416     int32_t combined_filter_x = 0;
417     int32_t combined_history_x = 0;
```

```
403         history_x[1] = history_x[0];
080013f0:   ldr     r3, [pc, #48]   ; (0x8001424 <ProcessSample+196>)
080013f2:   ldrsh.w r2, [r3]
080013f6:   ldr     r3, [pc, #44]   ; (0x8001424 <ProcessSample+196>)
080013f8:   strh    r2, [r3, #2]
404         history_x[0] = newsample;
080013fa:   ldr     r2, [pc, #40]   ; (0x8001424 <ProcessSample+196>)
080013fc:   ldrh    r3, [r7, #6]
080013fe:   strh    r3, [r2, #0]
405         history_y[1] = history_y[0];
08001400:   ldr     r3, [pc, #40]   ; (0x800142c <ProcessSample+204>)
08001402:   ldrsh.w r2, [r3]
08001406:   ldr     r3, [pc, #36]   ; (0x800142c <ProcessSample+204>)
08001408:   strh    r2, [r3, #2]
406         history_y[0] = newdata;
0800140a:   ldr     r2, [pc, #32]   ; (0x800142c <ProcessSample+204>)
0800140c:   ldrh    r3, [r7, #10]
0800140e:   strh    r3, [r2, #0]
```

Instructions 08001410-0800142e correspond to lines 411-419 in the code. Here the instructions mov, ldr.w, lsls are used to save the values of the static and temporary variables into memory. Lsls shifts left to load the integer arrays into the static variables defined in line 412-413.



```
403     history_x[1] = history_x[0];
404     history_x[0] = newsample;
405     history_y[1] = history_y[0];
406     history_y[0] = newdata;
407
408     return newdata;
409 }
410
411 static int16_t ProcessSample2(int16_t newsample) {
412     static int16_t filter_taps_b[3] = {10158, 20283, 10158};
413     static int16_t filter_taps_a[2] = {-2261, -10060};
414
415     int32_t temp = 0;
416     int32_t combined_filter_x = 0;
417     int32_t combined_history_x = 0;
418     int32_t combined_filter_y = 0;
419     int32_t combined_history_y = 0;
420
```

```
0800140e:   strh    r3, [r2, #0]
408           return newdata;
08001410:   ldrsh.w r3, [r7, #10]
409       }
08001414:   mov     r0, r3
08001416:   adds    r7, #20
08001418:   mov     sp, r7
0800141a:   ldr.w   r7, [sp], #4
0800141e:   bx      lr
08001420:   movs    r4, r0
08001422:   movs    r0, #0
08001424:   lsls    r0, r6, #10
08001426:   movs    r0, #0
08001428:   movs    r4, r1
0800142a:   movs    r0, #0
0800142c:   lsls    r4, r6, #10
0800142e:   movs    r0, #0
```

The function takes 134 cycles to process a sample.

62498	ITM Port 31	1	393339106	3.933391 s
62499	ITM Port 31	2	393339240	3.933392 s
62500	ITM Port 31	1	393351595	3.933516 s
62501	ITM Port 31	2	393351729	3.933517 s
62502	ITM Port 31	1	393364126	3.933641 s
62503	ITM Port 31	2	393364260	3.933643 s

Q2> Analyze the code that you have written and use one or more techniques to optimize different parts of the code. Explain which technique(s) you've used and why you've used them. How many cycles per sample do you achieve after using your optimizations?

150 cycles.

We tried using different approaches, one of these is shown below. We used the MAC instruction 5 times, one for each tap. We chose to use this instruction which is written in assembly code and directly accesses the Multiply Accumulate unit to perform convolution. After running the project, we found that this new function actually took more cycles to complete than the original function before optimizing it. The convolution we are doing for this IIR filter is simple and requires only very few taps, therefore using these optimization techniques may not improve its performance as it is already using very few instructions.

48396	ITM Port 31	1	305212420	3.052124 s
48397	ITM Port 31	2	305212570	3.052126 s
48398	ITM Port 31	1	305224918	3.052249 s
48399	ITM Port 31	2	305225068	3.052251 s
48400	ITM Port 31	1	305237416	3.052374 s
48401	ITM Port 31	2	305237566	3.052376 s

```
413 static int16_t ProcessSample2(int16_t newsample) {
414     static int16_t filter_taps_b[3] = {10158, 20283, 10158};
415     static int16_t filter_taps_a[2] = {-2261, -10060};
417     int32_t temp = 0;
445
446         int32_t temp = 0;
447
448         history_x[0] = newsample;
449
450         __asm volatile ("SMLABB %[result], %[op1], %[op2], %[acc]"
451                         : [result] "=r" (temp)
452                         : [op1] "r" (filter_taps_b[0]), [op2] "r" (newsample), [acc] "r" (temp)
453                         );
454
455           __asm volatile ("SMLABB %[result], %[op1], %[op2], %[acc]"
456                         : [result] "=r" (temp)
457                         : [op1] "r" (filter_taps_b[1]), [op2] "r" (history_x[0]), [acc] "r" (temp)
458                         );
459
460          __asm volatile ("SMLABB %[result], %[op1], %[op2], %[acc]"
461                         : [result] "=r" (temp)
462                         : [op1] "r" (filter_taps_b[2]), [op2] "r" (history_x[1]), [acc] "r" (temp)
463                         );
464
465         __asm volatile ("SMLABB %[result], %[op1], %[op2], %[acc]"
466                         : [result] "=r" (temp)
467                         : [op1] "r" (filter_taps_a[0]), [op2] "r" (history_y[0]), [acc] "r" (temp)
468                         );
469
470        __asm volatile ("SMLABB %[result], %[op1], %[op2], %[acc]"
471                         : [result] "=r" (temp)
472                         : [op1] "r" (filter_taps_a[1]), [op2] "r" (history_y[1]), [acc] "r" (temp)
473                         );
474
475         if (temp > 0x3FFFFFFF) {
476             temp = 0x3FFFFFFF;
477         } else if (temp < -0x40000000) {
478             temp = -0x40000000;
479         }
480
481         int16_t newdata = (int16_t)(temp >> 15);
482
```

Q3> Implement a new function that produces an output with echo. Aim for a delay of around 125 ms. How large does your history (state variable array) have to be for this delay? Is this an FIR or IIR filter? Add to your report an analysis of the compiled code. Discuss any "optimizations" that you needed to use to make this function work under the time constraints.

```
61 int16_t delayBuffer[DELAY_NUMBER];
62 int16_t end = DELAY_NUMBER - 1;
```

Here we used delayBuffer as our state variable with a size of 1000 (we set DELAY_NUMBER to 1000). The sample time is 1/8000 Hz = 125us. So, we need to go back 1000 samples to get a 125ms delay. This is an FIR filter, the response depends on the current and previous input only. To optimize the echo function to make it work under the time constraints, we used a circular buffer. We did this by having a variable called end that works as the index of the state variable array to update the values of the array from right to left. When the last element has been reached (end of the buffer is at index 0), the state variable array starts again where it began, in this case at end = 999.

```
478 static int16_t echoEffect(int16_t newsample) {
479
480     int16_t newdata = newsample + 0.7*delayBuffer[end];
481
482     delayBuffer[end] = newsample;
483
484     end--;
485     if(end < 0) {
486         end = 999;
487     }
488
489     return newdata;
490 }
491
```

Q4> Implement a new function that produces an output with reverb. Is this an FIR or IIR filter? Add to your report an analysis of the compiled code. Add to your report an analysis of the compiled code. Discuss any "optimizations" that you needed to use to make this function work under the time constraints.

This is a IIR filter, as we used the previous output to compute the response and there is feedback present. As in question 3, the only optimization required to make the function work under time restraints was to implement a circular buffer for the delayBuffer, which contains the last 1000 output values. The circular buffer updates the state variable array from right to left, starting at end = 999, and ending at 0. When the last value is reached, the buffer starts again at the array element with index 999.

```
493 static int16_t reverbEffect(int16_t newsample) {
494
495     int16_t newdata = newsample + 0.7*delayBuffer[end];
496
497     delayBuffer[end] = newdata;
498
499     end--;
500     if(end < 0) {
501         end = 999;
502     }
503
504     return newdata;
505 }
```

Q5> Write a reflection of what you have observed and learned in this lab, drawing connections between what we've discussed in class and what you've done in this lab.

This lab allowed us to design a second order IIR filter using only five coefficients, which proved to be a highly efficient way to filter out noise in audio signals. Compared to the FIR filter designed in Lab1, we were able to achieve similar results while using less memory space, since fewer taps were used in the IIR filter. Additionally, the IIR filter was found to be easier to design and implement, making it a preferred choice in certain applications. However, it's important to note that using IIR filters also comes with a tradeoff. Due to the use of feedback in the filter response, there is a risk of instability, which must be taken into account when designing the filter. Nonetheless, the IIR filter remains a valuable tool in audio signal processing, especially in situations where memory efficiency is a critical factor. We tried to optimize our IIR filter using the MAC unit and intrinsics, however we concluded that in some cases where we have very few taps it may not be ideal to use these techniques as the cycle count may increase due to the increased number of instructions. Overall, the lab allowed us to explore the advantages and disadvantages of IIR filters, and we gained valuable insights into designing and implementing them.