ENCM 515: Digital Signal Processors

Assignment 2 - Block Processing

Group 7

Hanan Anam

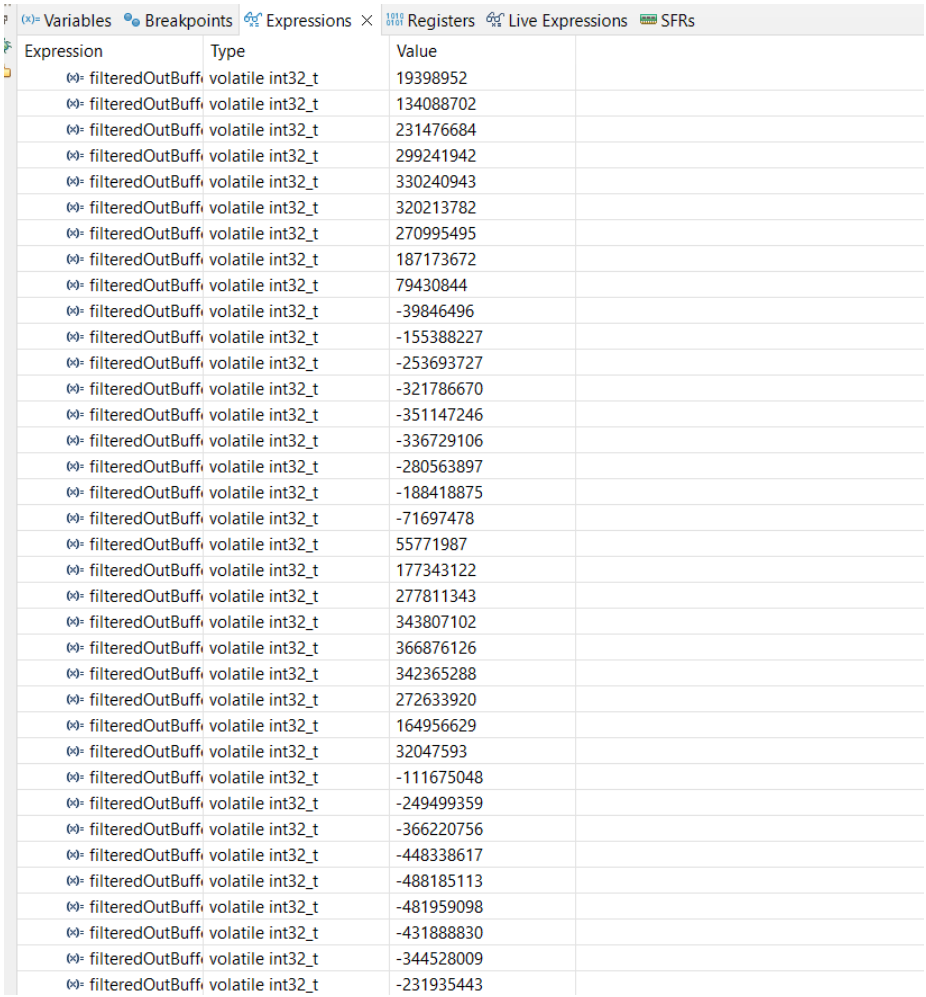
Nadia Duarte

April 5th, 2023

Introduction

In this assignment, we wrote four different functions to implement block processing with block sizes of 3 and 16, and then to include loop unrolling. We disabled timer interrupts while doing our analysis. This is because we would be required to perform buffering of new input samples in the timer callback, since in some cases the entire block would not finish processing before the next sample arrived. We felt we could more easily demonstrate the timing requirements by directly reading and buffering the input samples. We verified the output of our system by setting a breakpoint where filteredOutputBufferA filled for the first time. We compared each of the results to the result we got from running ProcessSample(). For this experiment, we chose to use 220 taps.

Original ProcessSample() Output



Expression	Type	Value
filteredOutBuffA	volatile int32_t	19398952
filteredOutBuffA	volatile int32_t	134088702
filteredOutBuffA	volatile int32_t	231476684
filteredOutBuffA	volatile int32_t	299241942
filteredOutBuffA	volatile int32_t	330240943
filteredOutBuffA	volatile int32_t	320213782
filteredOutBuffA	volatile int32_t	270995495
filteredOutBuffA	volatile int32_t	187173672
filteredOutBuffA	volatile int32_t	79430844
filteredOutBuffA	volatile int32_t	-39846496
filteredOutBuffA	volatile int32_t	-155388227
filteredOutBuffA	volatile int32_t	-253693727
filteredOutBuffA	volatile int32_t	-321786670
filteredOutBuffA	volatile int32_t	-351147246
filteredOutBuffA	volatile int32_t	-336729106
filteredOutBuffA	volatile int32_t	-280563897
filteredOutBuffA	volatile int32_t	-188418875
filteredOutBuffA	volatile int32_t	-71697478
filteredOutBuffA	volatile int32_t	55771987
filteredOutBuffA	volatile int32_t	177343122
filteredOutBuffA	volatile int32_t	277811343
filteredOutBuffA	volatile int32_t	343807102
filteredOutBuffA	volatile int32_t	366876126
filteredOutBuffA	volatile int32_t	342365288
filteredOutBuffA	volatile int32_t	272633920
filteredOutBuffA	volatile int32_t	164956629
filteredOutBuffA	volatile int32_t	32047593
filteredOutBuffA	volatile int32_t	-111675048
filteredOutBuffA	volatile int32_t	-249499359
filteredOutBuffA	volatile int32_t	-366220756
filteredOutBuffA	volatile int32_t	-448338617
filteredOutBuffA	volatile int32_t	-488185113
filteredOutBuffA	volatile int32_t	-481959098
filteredOutBuffA	volatile int32_t	-431888830
filteredOutBuffA	volatile int32_t	-344528009
filteredOutBuffA	volatile int32_t	-231935443

filteredOutBuffi	volatile int32_t	-107349606
filteredOutBuffi	volatile int32_t	14090455
filteredOutBuffi	volatile int32_t	119015192
filteredOutBuffi	volatile int32_t	194841501
filteredOutBuffi	volatile int32_t	233967090
filteredOutBuffi	volatile int32_t	232394202
filteredOutBuffi	volatile int32_t	191630188
filteredOutBuffi	volatile int32_t	116328175
filteredOutBuffi	volatile int32_t	16187639
filteredOutBuffi	volatile int32_t	-97256908
filteredOutBuffi	volatile int32_t	-210046085
filteredOutBuffi	volatile int32_t	-309727862

Part I. Processing Blocks Without Unrolling

ProcessBlock()

Implementation before unrolling using a block size of 3

To properly implement block processing with a block size of 3, we started off by changing our output buffer size to 48 to make it a multiple of both 3 and 16. This means that we don't switch to the second output buffer before the first one is full. After defining BLOCK_SIZE to 3, we declared the function ProcessBlock(). The processing of the input block of samples starts with storing the current block of samples into the history array with indices 0, 1, and 2, corresponding to the 3 samples used for each block. The next step is to apply the filter coefficients to the history array using a convolution operation. The result of this operation is stored in an accumulator array with the same length as the block size to process the 3 samples at the same time. We implemented this using nested for loops - one to iterate through all the filter taps, and one to iterate through each sample in the block. This means that each time a filter coefficient is loaded, we can use it to calculate three output values instead of just one. Lines 458-469 check for overflow and underflow, where if the value in the accumulator exceeds the maximum or minimum possible value that can be stored in the data type being used (int32_t), the statement sets the value to the maximum or minimum possible value, respectively. The function returns void, because it receives a pointer to the filter output array as an argument.

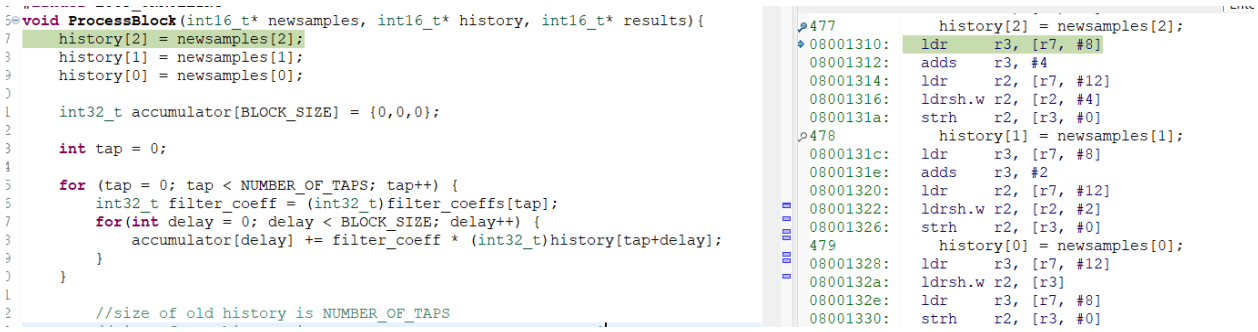
```
void ProcessBlock(int16_t* newsamples, int16_t* history, int16_t* results){
        history[2] = newsamples[2];
        history[1] = newsamples[1];
        history[0] = newsamples[0];

        int32_t accumulator[BLOCK_SIZE] = {0,0,0};

        int tap = 0;

        for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
                int32_t filter_coeff = (int32_t)filter_coeffs[tap];
                for(int delay = 0; delay < BLOCK_SIZE; delay++) {
                        accumulator[delay] += filter_coeff * (int32_t)history[tap+delay];
                }
        }

        for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
                        history[tap+BLOCK_SIZE] = history[tap];
        }

        for(int delay = 0; delay < BLOCK_SIZE; delay++) {
                if (accumulator[delay] > 0x3FFFFFFF) {
                                accumulator[delay] = 0x3FFFFFFF;
                                overflow_count++;
                } else if (accumulator[delay] < -0x40000000) {
                        accumulator[delay] = -0x40000000;
                        underflow_count++;
                }

                results[delay] = (int16_t)(accumulator[delay] >> 15); //results[0] is y[n]
        }
        return;
}
```

The output of ProcessBlock() is shown below, as we can see it produces the correct outputs as it matches the original ProcessSample output.

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

Assembly

Adding new samples to history buffer

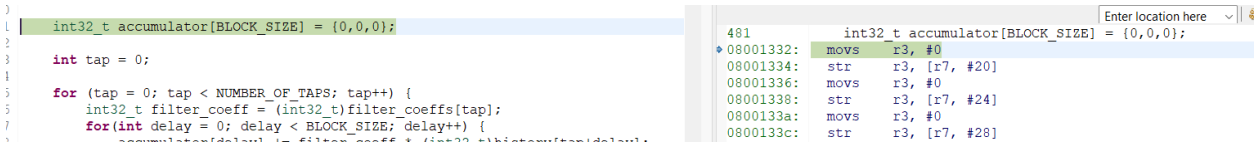


```
void ProcessBlock(int16_t* newsamples, int16_t* history, int16_t* results){
    history[2] = newsamples[2];
    history[1] = newsamples[1];
    history[0] = newsamples[0];

    int32_t accumulator[BLOCK_SIZE] = {0,0,0};

    int tap = 0;

    for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
        int32_t filter_coeff = (int32_t)filter_coeffs[tap];
        for(int delay = 0; delay < BLOCK_SIZE; delay++) {
            accumulator[delay] += filter_coeff * (int32_t)history[tap+delay];
        }
    }

        //size of old history is NUMBER_OF_TAPS
```

```
477           history[2] = newsamples[2];
08001310:   ldr     r3, [r7, #8]
08001312:   adds    r3, #4
08001314:   ldr     r2, [r7, #12]
08001316:   ldrsh.w r2, [r2, #4]
0800131a:   strh    r2, [r3, #0]
478           history[1] = newsamples[1];
0800131c:   ldr     r3, [r7, #8]
0800131e:   adds    r3, #2
08001320:   ldr     r2, [r7, #12]
08001322:   ldrsh.w r2, [r2, #2]
08001326:   strh    r2, [r3, #0]
479           history[0] = newsamples[0];
08001328:   ldr     r3, [r7, #12]
0800132a:   ldrsh.w r2, [r3]
0800132e:   ldr     r3, [r7, #8]
08001330:   strh    r2, [r3, #0]
```

To set an array element in history equal to an array element in newsamples, 5 RISC type instructions are needed. These instructions use general purpose registers to access the values in the pointer arrays.

Reset accumulator result to zero

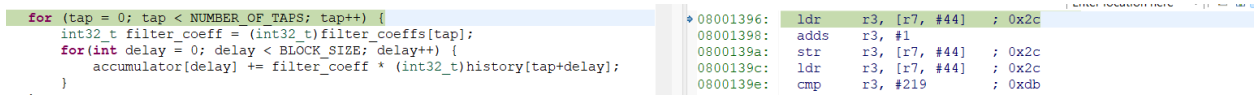


```
    int32_t accumulator[BLOCK_SIZE] = {0,0,0};

    int tap = 0;

    for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
        int32_t filter_coeff = (int32_t)filter_coeffs[tap];
        for(int delay = 0; delay < BLOCK_SIZE; delay++) {
```

```
Enter location here
481           int32_t accumulator[BLOCK_SIZE] = {0,0,0};
08001332:   movs    r3, #0
08001334:   str     r3, [r7, #20]
08001336:   movs    r3, #0
08001338:   str     r3, [r7, #24]
0800133a:   movs    r3, #0
0800133c:   str     r3, [r7, #28]
```

To reset the accumulator array to zero 6 instructions are required. As we can see in the assembly code, 3 accesses to memory were done, one for each array element. After the values were copied from memory, they were stored on their corresponding register.

Outer convolution loop

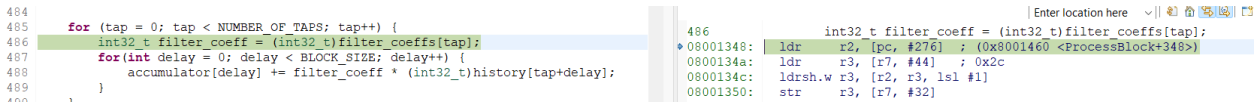


```
for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
    int32_t filter_coeff = (int32_t)filter_coeffs[tap];
    for(int delay = 0; delay < BLOCK_SIZE; delay++) {
        accumulator[delay] += filter_coeff * (int32_t)history[tap+delay];
    }
```

```
08001396:   ldr     r3, [r7, #44]   ; 0x2c
08001398:   adds    r3, #1
0800139a:   str     r3, [r7, #44]   ; 0x2c
0800139c:   ldr     r3, [r7, #44]   ; 0x2c
0800139e:   cmp     r3, #219        ; 0xdb
```

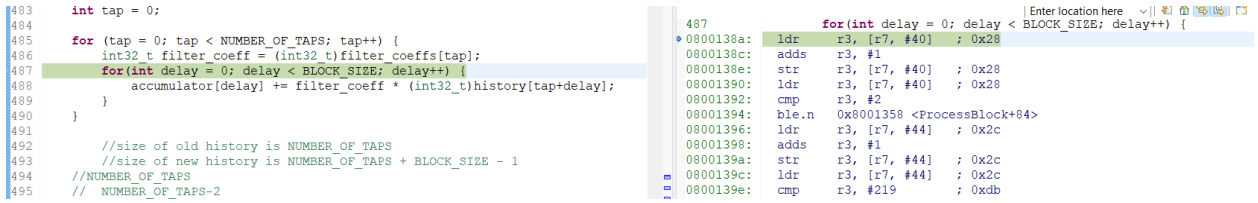
For the loop definition where we count the number of taps for the number of samples, we used 5 instructions. The instruction cmp compares tap and NUMBER_OF_TAPS every time the loop is run. The loads ldr copy the values from memory to use them in the comparison and the store str, updates the value of tap everytime it is increased by adds.

Loading new filter coefficient



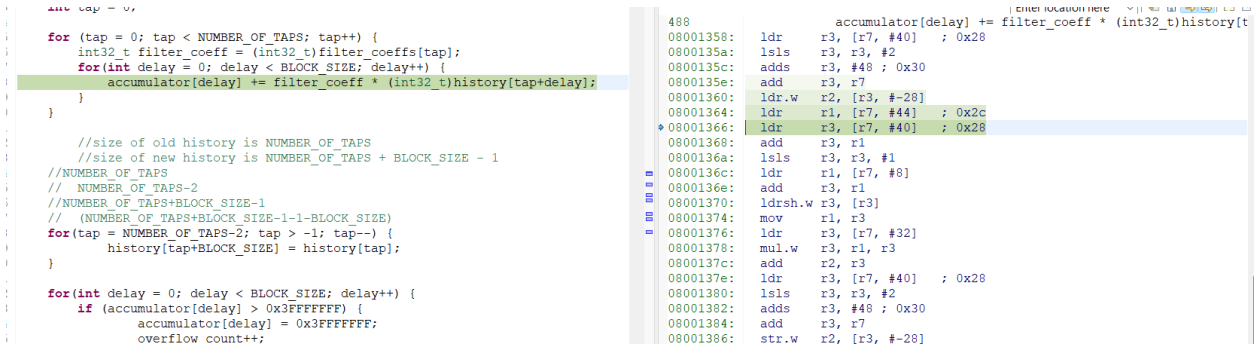
In line 486, we are loading a new filter coefficient to filter_coeff and casting it to an int32_t data type. Here we used 4 RISC type instructions, mainly loads and one store.

Inner convolution loop



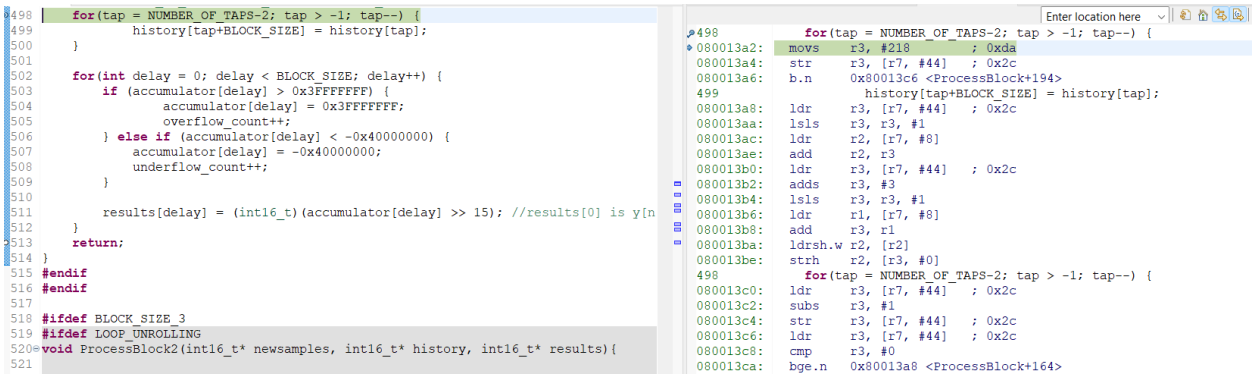
This inner loop requires 11 instructions, which include some loads, stores, additions, comparisons, and one conditional branch. This loop runs 3 times, because here the block size is 3.

Convolution sum



The convolution operation takes a lot more instructions from what we've been looking at before, with around 21 instructions. The first instructions ldr, lsls, add, and adds are reading the values of history[tap+delay], the following loads correspond to the casting of history. The multiplication and addition of the convolution are done by instructions mul.w, and add respectively. We can observe that one access to memory is made using mov just before doing the multiplication. The convolution is done 3 times, therefore we have a total of 21*3, 63 instructions to do the convolution operation.

Updating history buffer

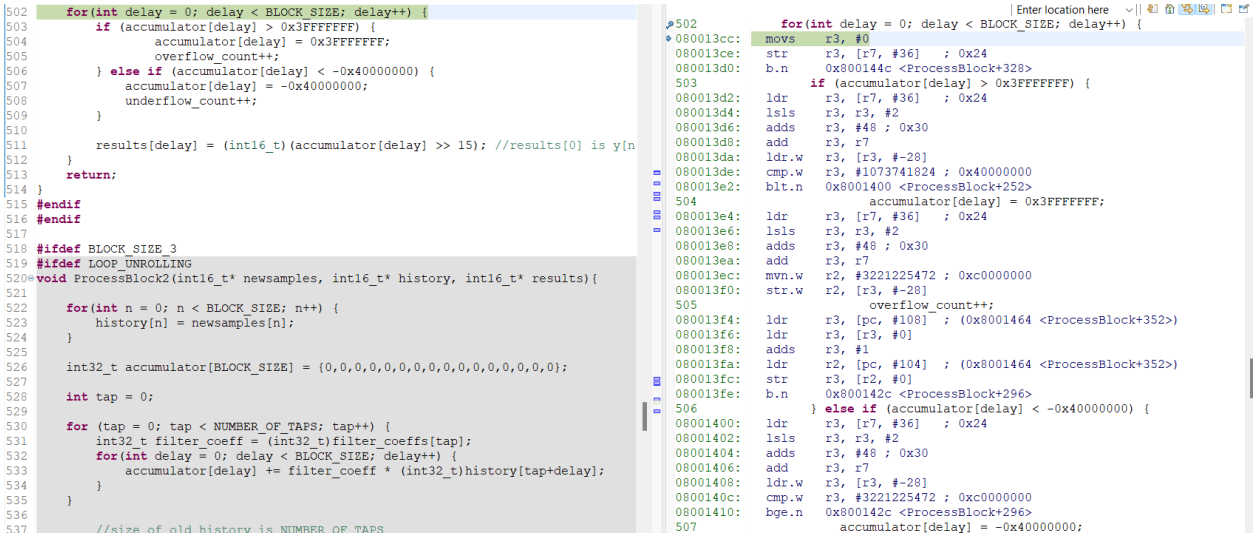


```
498     for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
499             history[tap+BLOCK_SIZE] = history[tap];
500     }
501
502     for(int delay = 0; delay < BLOCK_SIZE; delay++) {
503         if (accumulator[delay] > 0x3FFFFFFF) {
504                 accumulator[delay] = 0x3FFFFFFF;
505                 overflow_count++;
506         } else if (accumulator[delay] < -0x40000000) {
507             accumulator[delay] = -0x40000000;
508             underflow_count++;
509         }
510
511         results[delay] = (int16_t)(accumulator[delay] >> 15); //results[0] is y[n
512     }
513     return;
514 }
515 #endif
516 #endif
517
518 #ifdef BLOCK_SIZE_3
519 #ifdef LOOP_UNROLLING
520 void ProcessBlock2(int16_t* newsamples, int16_t* history, int16_t* results){
521
```

```
498         for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
080013a2:   movs    r3, #218        ; 0xda
080013a4:   str     r3, [r7, #44]   ; 0x2c
080013a6:   b.n     0x80013c6 <ProcessBlock+194>
499                   history[tap+BLOCK_SIZE] = history[tap];
080013a8:   ldr     r3, [r7, #44]   ; 0x2c
080013aa:   lsls    r3, r3, #1
080013ac:   ldr     r2, [r7, #8]
080013ae:   add     r2, r3
080013b0:   ldr     r3, [r7, #44]   ; 0x2c
080013b2:   adds    r3, #3
080013b4:   lsls    r3, r3, #1
080013b6:   ldr     r1, [r7, #8]
080013b8:   add     r3, r1
080013ba:   ldrsh.w r2, [r2]
080013be:   strh    r2, [r3, #0]
498           for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
080013c0:   ldr     r3, [r7, #44]   ; 0x2c
080013c2:   subs    r3, #1
080013c4:   str     r3, [r7, #44]   ; 0x2c
080013c6:   ldr     r3, [r7, #44]   ; 0x2c
080013c8:   cmp     r3, #0
080013ca:   bge.n   0x80013a8 <ProcessBlock+164>
```

In the loop to update history, the first set of instructions starting at address 080013a2 correspond to history[tap+BLOCK_SIZE] = history[tap]. The second set of instructions starting at address 080013c0 on the right correspond to the loop, where the instructions look similar to what we have seen in other loops, but here the subs instruction decreases tap on every iteration and bge.n is used to iterate through the loop.

Saturation



```
502     for(int delay = 0; delay < BLOCK_SIZE; delay++) {
503         if (accumulator[delay] > 0x3FFFFFFF) {
504                 accumulator[delay] = 0x3FFFFFFF;
505                 overflow_count++;
506         } else if (accumulator[delay] < -0x40000000) {
507             accumulator[delay] = -0x40000000;
508             underflow_count++;
509         }
510
511         results[delay] = (int16_t)(accumulator[delay] >> 15); //results[0] is y[n
512     }
513     return;
514 }
515 #endif
516 #endif
517
518 #ifdef BLOCK_SIZE_3
519 #ifdef LOOP_UNROLLING
520 void ProcessBlock2(int16_t* newsamples, int16_t* history, int16_t* results){
521
522     for(int n = 0; n < BLOCK_SIZE; n++) {
523         history[n] = newsamples[n];
524     }
525
526     int32_t accumulator[BLOCK_SIZE] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
527
528     int tap = 0;
529
530     for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
531         int32_t filter_coeff = (int32_t)filter_coeffs[tap];
532         for(int delay = 0; delay < BLOCK_SIZE; delay++) {
533             accumulator[delay] += filter_coeff * (int32_t)history[tap+delay];
534         }
535     }
536
537         //size of old history is NUMBER OF TAPS
```

```
502         for(int delay = 0; delay < BLOCK_SIZE; delay++) {
080013cc:   movs    r3, #0
080013ce:   str     r3, [r7, #36]   ; 0x24
080013d0:   b.n     0x800144c <ProcessBlock+328>
503             if (accumulator[delay] > 0x3FFFFFFF) {
080013d2:   ldr     r3, [r7, #36]   ; 0x24
080013d4:   lsls    r3, r3, #2
080013d6:   adds    r3, #48 ; 0x30
080013d8:   add     r3, r7
080013da:   ldr.w   r3, [r3, #-28]
080013de:   cmp.w   r3, #1073741824 ; 0x40000000
080013e2:   blt.n   0x8001400 <ProcessBlock+252>
504                   accumulator[delay] = 0x3FFFFFFF;
080013e4:   ldr     r3, [r7, #36]   ; 0x24
080013e6:   lsls    r3, r3, #2
080013e8:   adds    r3, #48 ; 0x30
080013ea:   add     r3, r7
080013ec:   mvn.w   r2, #3221225472 ; 0xc0000000
080013f0:   str.w   r2, [r3, #-28]
505                   overflow_count++;
080013f4:   ldr     r3, [pc, #108]  ; (0x8001464 <ProcessBlock+352>)
080013f6:   ldr     r3, [r3, #0]
080013f8:   adds    r3, #1
080013fa:   ldr     r2, [pc, #104]  ; (0x8001464 <ProcessBlock+352>)
080013fc:   str     r3, [r2, #0]
080013fe:   b.n     0x800142c <ProcessBlock+296>
506             } else if (accumulator[delay] < -0x40000000) {
08001400:   ldr     r3, [r7, #36]   ; 0x24
08001402:   lsls    r3, r3, #2
08001404:   adds    r3, #48 ; 0x30
08001406:   add     r3, r7
08001408:   ldr.w   r3, [r3, #-28]
0800140c:   cmp.w   r3, #3221225472 ; 0xc0000000
08001410:   bge.n   0x800142c <ProcessBlock+296>
507                   accumulator[delay] = -0x40000000;
```

The assembly code shown on the right, corresponds to the check of overflow and underflow. We can see that there are a big number of instructions but there is also a big number of lines of code written in the program. To compare the values of accumulate[delay] with the maximum and minimum values allowed in an int32_t, we are using the instruction cmp.w, which tells bge.n to branch under this condition.

Results

-O0

175	ITM Port 31	1	3395007	33.950070 ms
176	ITM Port 31	2	3433892	34.338920 ms
177	ITM Port 31	1	3434414	34.344140 ms
178	ITM Port 31	2	3473245	34.732450 ms
179	ITM Port 31	1	3473764	34.737640 ms
180	ITM Port 31	2	3512919	35.129190 ms

Total: 39155 cycles, 391.55us

total/BLOCK_SIZE = 13051.7 cycles, 130.52us

-OS

175	ITM Port 31	1	1253379	12.533790 ms
176	ITM Port 31	2	1264708	12.647080 ms
177	ITM Port 31	1	1264942	12.649420 ms
178	ITM Port 31	2	1276239	12.762390 ms
179	ITM Port 31	1	1276473	12.764730 ms
180	ITM Port 31	2	1287772	12.877720 ms

Total: 11299 cycles, 112.99us

total/BLOCK_SIZE = 3766.3 cycles, 37.66us

-Ofast

182	ITM Port 31	1	551643	5.516430 ms
183	ITM Port 31	2	557429	5.574290 ms
184	ITM Port 31	1	557821	5.578210 ms
185	ITM Port 31	2	563604	5.636040 ms
186	ITM Port 31	1	563830	5.638300 ms
187	ITM Port 31	2	569613	5.696130 ms

Total: 5783 cycles, 57.83us

total/BLOCK_SIZE = 1927.7 cycles, 19.28us

The function only satisfies the timing requirements when using size optimization -Os and speed optimization -Ofast, assuming 8kHz sampling 1/8000Hz = 125us.

ProcessBlock2()

Implementation before unrolling using a block size of 16

To process a block size of 16 samples, we first set BLOCK_SIZE to 16. In ProcessBlock2() the processing of the input block of samples starts with storing the current block of samples into the history array with indices 0 to 15, corresponding to the 16 samples used for each block. Same as before, the next step is to apply the filter coefficients to the history array using a convolution operation and the result of this operation is stored in an accumulator array with length 16. Like in ProcessBlock(), lines 458-469 check for overflow and underflow. The implementation of this function is identical to that in ProcessBlock(), except now BLOCK_SIZE has been changed to 16 by commenting out the define statement on line 39.

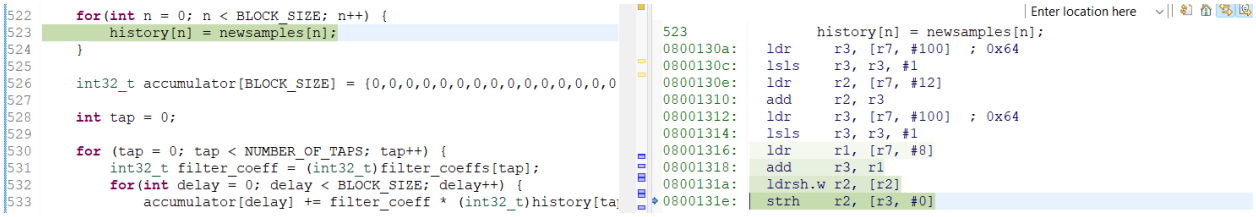
```
475
476  void ProcessBlock2(int16_t* newsamples, int16_t* history, int16_t* results){
477
478          for(int n = 0; n < BLOCK_SIZE; n++) {
479                  history[n] = newsamples[n];
480          }
481
482          int32_t accumulator[BLOCK_SIZE] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
483
484          int tap = 0;
485
486          for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
487                  int32_t filter_coeff = (int32_t)filter_coeffs[tap];
488                  for(int delay = 0; delay < BLOCK_SIZE; delay++) {
489                          accumulator[delay] += filter_coeff * (int32_t)history[tap+delay];
490                  }
491          }
492
493
494          for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
495                          history[tap+BLOCK_SIZE] = history[tap];
496          }
497
498          for(int delay = 0; delay < BLOCK_SIZE; delay++) {
499                  if (accumulator[delay] > 0x3FFFFFFF) {
500                                  accumulator[delay] = 0x3FFFFFFF;
501                                  overflow_count++;
502                  } else if (accumulator[delay] < -0x40000000) {
503                          accumulator[delay] = -0x40000000;
504                          underflow_count++;
505                  }
506
507                  results[delay] = (int16_t)(accumulator[delay] >> 15); //results[0] is y[n]
508          }
509          return;
510  }
```

Output

The output for ProcessBlock2() was the same as the output for ProcessBlock(). The values were correct again, as we were able to compare the values in the buffer to the output from the original function ProcessSample().

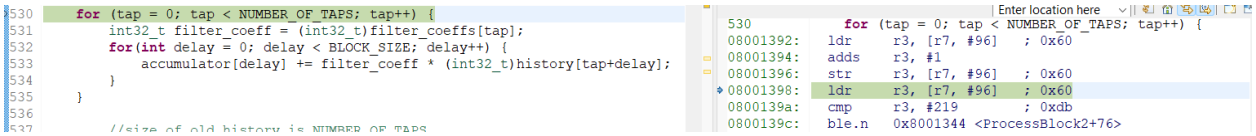
Assembly

Adding new samples to history buffer



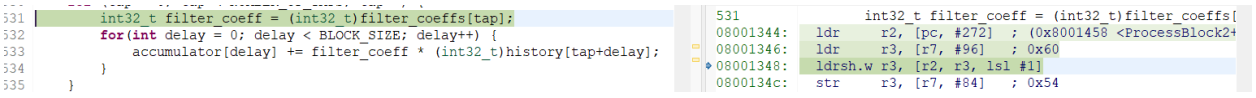
Here we are updating history using a loop, as we can there are 10 instructions required to do a single load and store to history[n]. This has more instructions than before as the value of [n] has to be copied in both history[n] and newsamples[n] for loading and storing this value. We chose not to unroll this instruction like we did in ProcessBlock() because there would be excessive lines of code.

Outer convolution loop



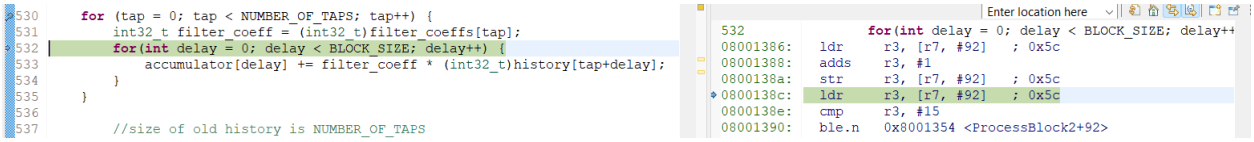
The instructions in the outer loop are the same as in ProcessBlock() as nothing was modified for this loop statement.

Loading filter coefficient



The loading of the filter coefficients has also the same instructions as before, nothing was modified.

Inner convolution loop

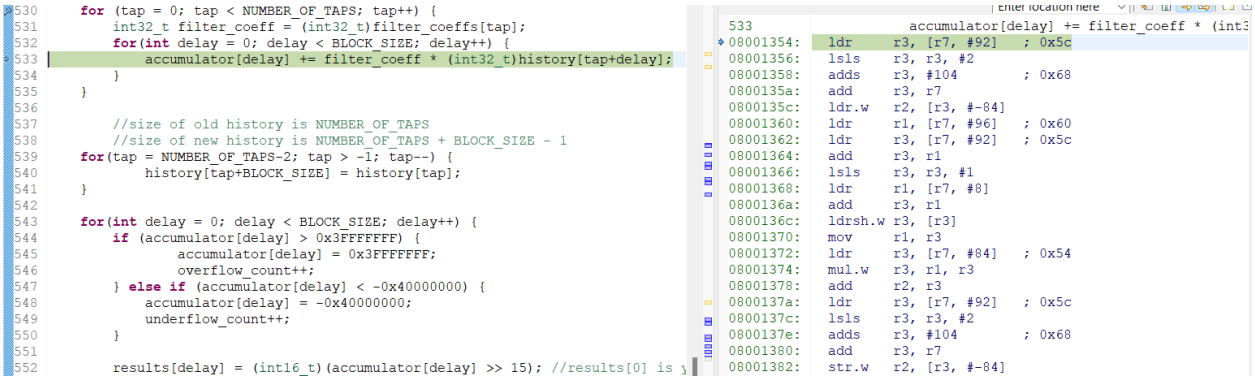


```
530     for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
531         int32_t filter_coeff = (int32_t)filter_coeffs[tap];
532         for(int delay = 0; delay < BLOCK_SIZE; delay++) {
533             accumulator[delay] += filter_coeff * (int32_t)history[tap+delay];
534         }
535     }
536
537         //size of old history is NUMBER_OF_TAPS
```

```
532             for(int delay = 0; delay < BLOCK_SIZE; delay++
08001386:   ldr     r3, [r7, #92]   ; 0x5c
08001388:   adds    r3, #1
0800138a:   str     r3, [r7, #92]   ; 0x5c
0800138c:   ldr     r3, [r7, #92]   ; 0x5c
0800138e:   cmp     r3, #15
08001390:   ble.n   0x8001354 <ProcessBlock2+92>
```

The instructions are the same as before, however because we have changed the block size we know that this loop is going to be running 16 times, branching with ble.n.

Convolution sum



```
530     for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
531         int32_t filter_coeff = (int32_t)filter_coeffs[tap];
532         for(int delay = 0; delay < BLOCK_SIZE; delay++) {
533             accumulator[delay] += filter_coeff * (int32_t)history[tap+delay];
534         }
535     }
536
537         //size of old history is NUMBER_OF_TAPS
538         //size of new history is NUMBER_OF_TAPS + BLOCK_SIZE - 1
539     for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
540             history[tap+BLOCK_SIZE] = history[tap];
541     }
542
543     for(int delay = 0; delay < BLOCK_SIZE; delay++) {
544         if (accumulator[delay] > 0x3FFFFFFF) {
545                 accumulator[delay] = 0x3FFFFFFF;
546                 overflow_count++;
547         } else if (accumulator[delay] < -0x40000000) {
548             accumulator[delay] = -0x40000000;
549             underflow_count++;
550         }
551
552         results[delay] = (int16_t)(accumulator[delay] >> 15); //results[0] is y
```

```
533                 accumulator[delay] += filter_coeff * (int3
08001354:   ldr     r3, [r7, #92]   ; 0x5c
08001356:   lsls    r3, r3, #2
08001358:   adds    r3, #104        ; 0x68
0800135a:   add     r3, r7
0800135c:   ldr.w   r2, [r3, #-84]
08001360:   ldr     r1, [r7, #96]   ; 0x60
08001362:   ldr     r3, [r7, #92]   ; 0x5c
08001364:   add     r3, r1
08001366:   lsls    r3, r3, #1
08001368:   ldr     r1, [r7, #8]
0800136a:   add     r3, r1
0800136c:   ldrsh.w r3, [r3]
08001370:   mov     r1, r3
08001372:   ldr     r3, [r7, #84]   ; 0x54
08001374:   mul.w   r3, r1, r3
08001378:   add     r2, r3
0800137a:   ldr     r3, [r7, #92]   ; 0x5c
0800137c:   lsls    r3, r3, #2
0800137e:   adds    r3, #104        ; 0x68
08001380:   add     r3, r7
08001382:   str.w   r2, [r3, #-84]
```

The instructions are the same as before for the convolution sum in ProcessBlock(), nothing was modified. However, we know that the convolution operation will now happen 16 times, therefore we have a total of 21*16, 336 instructions to do convolution for the 16 samples.

Results:

-O0

31	ITM Port 31	1	2551299	25.512990 ms
32	ITM Port 31	2	2711621	27.116210 ms
33	ITM Port 31	1	2714165	27.141650 ms
34	ITM Port 31	2	2874432	28.744320 ms
35	ITM Port 31	1	2877018	28.770180 ms
36	ITM Port 31	2	3037286	30.372860 ms

Total: 160268 cycles, 1602.68us

total/BLOCK_SIZE = 10016.8 cycles, 100.17us

-Os

30	ITM Port 31	1	845455	8.454550 ms
31	ITM Port 31	2	880827	8.808270 ms
32	ITM Port 31	1	882020	8.820200 ms
33	ITM Port 31	2	917391	9.173910 ms
34	ITM Port 31	1	918584	9.185840 ms
35	ITM Port 31	2	953986	9.539860 ms

Total: 35402 cycles, 354.02us

total/BLOCK_SIZE = 2212.6 cycles, 22.13us

-Ofast

30	ITM Port 31	1	644952	6.449520 ms
31	ITM Port 31	2	669542	6.695420 ms
32	ITM Port 31	1	670694	6.706940 ms
33	ITM Port 31	2	695325	6.953250 ms
34	ITM Port 31	1	696480	6.964800 ms
35	ITM Port 31	2	721064	7.210640 ms

Total: 24584 cycles, 245.84us

total/BLOCK_SIZE = 1536.5 cycles, 15.37us

The function satisfies the timing requirements in all of the compilations assuming 8kHz sampling 1/8000Hz = 125us. The average time to process each cycle has decreased because, as we can see based on the assembly code, the only change is the number of iterations of the inner convolution loop. So, the same total overhead of the function call is now being divided by 16 instead of 3, resulting in a smaller processing time per sample.

Part ll. Processing Blocks With Unrolling

Unrolling is a technique used in digital signal processing (DSP) optimization to improve performance by reducing the amount of time it takes to execute loops. This is achieved by performing multiple loop iterations simultaneously, which reduces the overhead associated with loop iteration and results in faster execution times. Essentially, the compiler takes the code that would normally be executed in a loop and "unrolls" it, creating a sequence of instructions that can be executed more efficiently.

ProcessBlock3()

Implementation after unrolling using a block size of 3

To improve the performance of the ProcessBlock() function, we made some modifications to allow for loop unrolling of the inner loop. With a block size of 3 in ProcessBlock(), we noticed that there were only 3 samples to compute per inner loop iteration. Therefore, we decided to remove the inner loop entirely and perform the accumulation directly to the accumulator with indices 0, 1, and 2. In the unrolled version, we do not need the inner loop as 1 is the only multiple of 3 that would work as a factor of 3 for loop unrolling. This allowed us to simplify the code and eliminate the overhead of the inner loop. The history and accumulator sizes remained the same, and the only change we made was to manually perform the convolution for the 3 samples processed. By removing the inner loop and performing the convolution manually, we were able to reduce the number of iterations and instructions required to process each block.

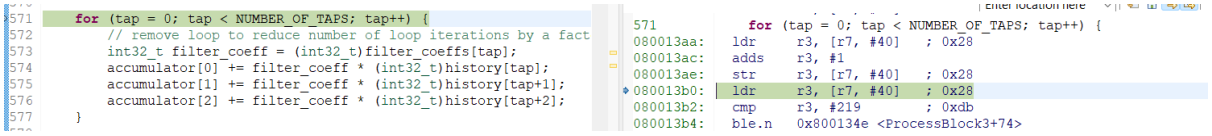
```
519  void ProcessBlock3(int16_t* newsamples, int16_t* history, int16_t* results){
520
521          for(int n = 0; n < BLOCK_SIZE; n++) {
522                  history[n] = newsamples[n];
523          }
524
525          int32_t accumulator[BLOCK_SIZE] = {0,0,0};
526
527          int tap = 0;
528
529          for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
530                  // remove loop to reduce number of loop iterations by a factor of 3
531                  int32_t filter_coeff = (int32_t)filter_coeffs[tap];
532                  accumulator[0] += filter_coeff * (int32_t)history[tap];
533                  accumulator[1] += filter_coeff * (int32_t)history[tap+1];
534                  accumulator[2] += filter_coeff * (int32_t)history[tap+2];
535          }
536
537
538          for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
539                          history[tap+BLOCK_SIZE] = history[tap];
540          }
541
542          for(int delay = 0; delay < BLOCK_SIZE; delay++) {
543                  if (accumulator[delay] > 0x3FFFFFFF) {
544                                  accumulator[delay] = 0x3FFFFFFF;
545                                  overflow_count++;
546                  } else if (accumulator[delay] < -0x40000000) {
547                          accumulator[delay] = -0x40000000;
548                          underflow_count++;
549                  }
550
551                  results[delay] = (int16_t)(accumulator[delay] >> 15); //results[0] is y[n]
552          }
553          return;
554  }
```

Output

The values were correct again, as we were able to compare the values in the buffer with the results from the original function ProcessSample().

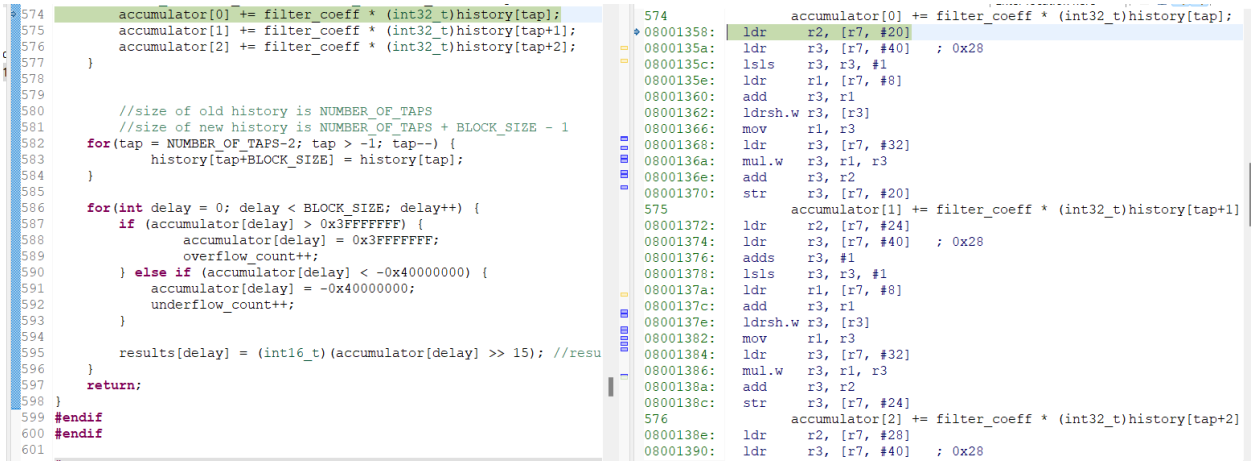
Assembly

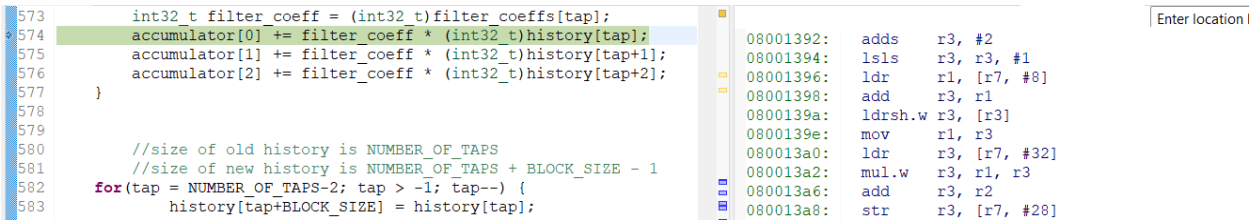
Convolution loop



The outer loop for the convolution stayed the same, no instructions were modified. However, the inner loop was removed as part of implementing the unrolling method.

Convolution Sum





This is the part of the code that we modified after implementing unrolling. We can see that each sum to an array element in the accumulator takes around 11 instructions, making it a total of 35 instructions to compute the convolution of 3 samples using the unrolling method. By using the unrolling method we were able to use only 35 instructions compared to using 63 before unrolling it and using a loop.

The rest of the instructions including the update of history, outer convolution loop, and loading filter coefficient stayed the same as in ProcessBlock(), nothing else was modified in the code.

Results

-O0

174	ITM Port 31	1	2173890	21.738900 ms
175	ITM Port 31	2	2195646	21.956460 ms
176	ITM Port 31	1	2196168	21.961680 ms
177	ITM Port 31	2	2217856	22.178560 ms
178	ITM Port 31	1	2218375	22.183750 ms
179	ITM Port 31	2	2240070	22.400700 ms

Total: 21695 cycles, 216.95us

total/BLOCK_SIZE = 7231.7 cycles, 72.32us

-Os

3	ITM Port 31	1	275941	2.759410 ms
4	ITM Port 31	2	282700	2.827000 ms
5	ITM Port 31	1	282901	2.829010 ms
6	ITM Port 31	2	289810	2.898100 ms

Total: 6909 cycles, 69.09us

total/BLOCK_SIZE = 2303 cycles, 23.03us

-Ofast

2	ITM Port 31	1	291050	2.910500 ms
3	ITM Port 31	2	297005	2.970050 ms
4	ITM Port 31	1	297172	2.971720 ms
5	ITM Port 31	2	303083	3.030830 ms

Total: 5911 cycles, 59.11us

total/BLOCK_SIZE = 1970.3 cycles, 19.70us

The function satisfies the timing requirements in all of the compilations assuming 8kHz sampling 1/8000Hz = 125us. The significant speedup can be attributed to us being able to perform the convolution sum using only 35 assembly instructions instead of 63.

ProcessBlock4()

Implementation after unrolling using a block size of 16

When we are processing a block of 16 samples using the ProcessBlock4() function, we can optimize the code by reducing the number of loop iterations needed to compute the filtered output values. In this case, since 4 is a multiple of 16, we can compute 4 filtered output values per iteration of the loop. So instead of running the inner loop 16 times to compute 16 filtered output values, we can reduce the number of loop iterations by a factor of 4, meaning we only need to run the inner loop 4 times to compute all 16 filtered output values. This can significantly improve the performance of the code and reduce the execution time of the ProcessBlock4() function. To achieve this optimization, we can keep the inner loop where the index delay starts at 0 and is increased by 4 on each iteration. In each iteration of the inner loop, we compute 4 filtered output values and store them in the filteredSamplesL array. At the end of the inner loop, we have computed all 16 filtered output values for the current block.

```
556  void ProcessBlock4(int16_t* newsamples, int16_t* history, int16_t* results){
557
558          for(int n = 0; n < BLOCK_SIZE; n++) {
559                  history[n] = newsamples[n];
560          }
561
562          int32_t accumulator[BLOCK_SIZE] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
563
564          int tap = 0;
565
566          for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
567                  int32_t filter_coeff = (int32_t)filter_coeffs[tap];
568                  for(int delay = 0; delay < BLOCK_SIZE; delay+=4) { // reducing number of loop iterations by a factor of 4
569                          accumulator[delay] += filter_coeff * (int32_t)history[tap+delay];
570                          accumulator[delay+1] += filter_coeff * (int32_t)history[tap+delay+1];
571                          accumulator[delay+2] += filter_coeff * (int32_t)history[tap+delay+2];
572                          accumulator[delay+3] += filter_coeff * (int32_t)history[tap+delay+3];
573                  }
574          }
575
576
577          for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
578                          history[tap+BLOCK_SIZE] = history[tap];
579          }
580
581          for(int delay = 0; delay < BLOCK_SIZE; delay++) {
582                  if (accumulator[delay] > 0x3FFFFFFF) {
583                                  accumulator[delay] = 0x3FFFFFFF;
584                                  overflow_count++;
585                  } else if (accumulator[delay] < -0x40000000) {
586                          accumulator[delay] = -0x40000000;
587                          underflow_count++;
588                  }
589
590                  results[delay] = (int16_t)(accumulator[delay] >> 15); //results[0] is y[n]
591          }
592          return;
593  }
```

Output

The values were correct again, as we were able to check the values in the buffer with the original function ProcessSample().

Assembly

Inner convolution loop

```
614     for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
615         int32_t filter_coeff = (int32_t)filter_coeffs[tap];
616         for(int delay = 0; delay < BLOCK_SIZE; delay+=4) { // reduci
617             accumulator[delay] += filter_coeff * (int32_t)history[ta
618             accumulator[delay+1] += filter_coeff * (int32_t)history[
619             accumulator[delay+2] += filter_coeff * (int32_t)history[
620             accumulator[delay+3] += filter_coeff * (int32_t)history[
621         }
```

```
616             for(int delay = 0; delay < BLOCK_SIZE; delay+=4) { // re
0800142e:   ldr     r3, [r7, #92]   ; 0x5c
08001430:   adds    r3, #4
08001432:   str     r3, [r7, #92]   ; 0x5c
08001434:   ldr     r3, [r7, #92]   ; 0x5c
08001436:   cmp     r3, #15
08001438:   ble.n   0x8001354 <ProcessBlock4+92>
```

The inner convolution loop was modified, here we are increasing delay by a factor of 4. Even though the assembly instructions look the same as before in ProcessBlock2() this loop will only run 4 times, compared to 16 times before unrolling.

Convolution sum

```
614     for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
615         int32_t filter_coeff = (int32_t)filter_coeffs[tap];
616         for(int delay = 0; delay < BLOCK_SIZE; delay+=4) { // reduci
617             accumulator[delay] += filter_coeff * (int32_t)history[ta
618             accumulator[delay+1] += filter_coeff * (int32_t)history[
619             accumulator[delay+2] += filter_coeff * (int32_t)history[
620             accumulator[delay+3] += filter_coeff * (int32_t)history[
621         }
622     }
623
624         //size of old history is NUMBER_OF_TAPS
625         //size of new history is NUMBER_OF_TAPS + BLOCK_SIZE - 1
626     for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
627             history[tap+BLOCK_SIZE] = history[tap];
628     }
629
630     for(int delay = 0; delay < BLOCK_SIZE; delay++) {
631         if (accumulator[delay] > 0x3FFFFFF) {
632                 accumulator[delay] = 0x3FFFFFF;
633                 overflow_count++;
634         } else if (accumulator[delay] < -0x40000000) {
635             accumulator[delay] = -0x40000000;
636             underflow_count++;
637         }
638
639         results[delay] = (int16_t)(accumulator[delay] >> 15); //resu
640     }
641     return;
```

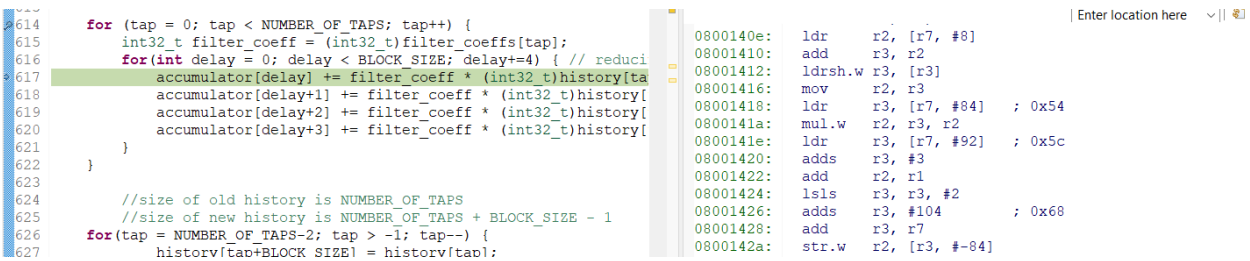
```
617                 accumulator[delay] += filter_coeff * (int32_t)histor
08001354:   ldr     r3, [r7, #92]   ; 0x5c
08001356:   lsls    r3, r3, #2
08001358:   adds    r3, #104        ; 0x68
0800135a:   add     r3, r7
0800135c:   ldr.w   r2, [r3, #-84]
08001360:   ldr     r1, [r7, #96]   ; 0x60
08001362:   ldr     r3, [r7, #92]   ; 0x5c
08001364:   add     r3, r1
08001366:   lsls    r3, r3, #1
08001368:   ldr     r1, [r7, #8]
0800136a:   add     r3, r1
0800136c:   ldrsh.w r3, [r3]
08001370:   mov     r1, r3
08001372:   ldr     r3, [r7, #84]   ; 0x54
08001374:   mul.w   r3, r1, r3
08001378:   add     r2, r3
0800137a:   ldr     r3, [r7, #92]   ; 0x5c
0800137c:   lsls    r3, r3, #2
0800137e:   adds    r3, #104        ; 0x68
08001380:   add     r3, r7
08001382:   str.w   r2, [r3, #-84]
618                 accumulator[delay+1] += filter_coeff * (int32_t)hist
08001386:   ldr     r3, [r7, #92]   ; 0x5c
08001388:   adds    r3, #1
0800138a:   lsls    r3, r3, #2
0800138c:   adds    r3, #104        ; 0x68
0800138e:   add     r3, r7
```

```
614     for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
615         int32_t filter_coeff = (int32_t)filter_coeffs[tap];
616         for(int delay = 0; delay < BLOCK_SIZE; delay+=4) { // reduci
617             accumulator[delay] += filter_coeff * (int32_t)history[ta
618             accumulator[delay+1] += filter_coeff * (int32_t)history[
619             accumulator[delay+2] += filter_coeff * (int32_t)history[
620             accumulator[delay+3] += filter_coeff * (int32_t)history[
621         }
622     }
623
624         //size of old history is NUMBER_OF_TAPS
625         //size of new history is NUMBER_OF_TAPS + BLOCK_SIZE - 1
626     for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
627             history[tap+BLOCK_SIZE] = history[tap];
628     }
629
630     for(int delay = 0; delay < BLOCK_SIZE; delay++) {
631         if (accumulator[delay] > 0x3FFFFFF) {
632                 accumulator[delay] = 0x3FFFFFF;
633                 overflow_count++;
634         } else if (accumulator[delay] < -0x40000000) {
635             accumulator[delay] = -0x40000000;
636             underflow_count++;
637         }
638
639         results[delay] = (int16_t)(accumulator[delay] >> 15); //resu
640     }
641     return;
```

```
08001390:   ldr.w   r1, [r3, #-84]
08001394:   ldr     r2, [r7, #96]   ; 0x60
08001396:   ldr     r3, [r7, #92]   ; 0x5c
08001398:   add     r3, r2
0800139a:   adds    r3, #1
0800139c:   lsls    r3, r3, #1
0800139e:   ldr     r2, [r7, #8]
080013a0:   add     r3, r2
080013a2:   ldrsh.w r3, [r3]
080013a6:   mov     r2, r3
080013a8:   ldr     r3, [r7, #84]   ; 0x54
080013aa:   mul.w   r2, r3, r2
080013ae:   ldr     r3, [r7, #92]   ; 0x5c
080013b0:   adds    r3, #1
080013b2:   add     r2, r1
080013b4:   lsls    r3, r3, #2
080013b6:   adds    r3, #104        ; 0x68
080013b8:   add     r3, r7
080013ba:   str.w   r2, [r3, #-84]
619                 accumulator[delay+2] += filter_coeff * (int32_t)hist
080013be:   ldr     r3, [r7, #92]   ; 0x5c
080013c0:   adds    r3, #2
080013c2:   lsls    r3, r3, #2
080013c4:   adds    r3, #104        ; 0x68
080013c6:   add     r3, r7
080013c8:   ldr.w   r1, [r3, #-84]
080013cc:   ldr     r2, [r7, #96]   ; 0x60
080013ce:   ldr     r3, [r7, #92]   ; 0x5c
```

```
614     for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
615         int32_t filter_coeff = (int32_t)filter_coeffs[tap];
616         for(int delay = 0; delay < BLOCK_SIZE; delay+=4) { // reduci
617             accumulator[delay] += filter_coeff * (int32_t)history[ta
618             accumulator[delay+1] += filter_coeff * (int32_t)history[
619             accumulator[delay+2] += filter_coeff * (int32_t)history[
620             accumulator[delay+3] += filter_coeff * (int32_t)history[
621         }
622     }
623
624         //size of old history is NUMBER_OF_TAPS
625         //size of new history is NUMBER_OF_TAPS + BLOCK_SIZE - 1
626     for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
627             history[tap+BLOCK_SIZE] = history[tap];
628     }
629
630     for(int delay = 0; delay < BLOCK_SIZE; delay++) {
631         if (accumulator[delay] > 0x3FFFFFF) {
632                 accumulator[delay] = 0x3FFFFFF;
633                 overflow_count++;
634         } else if (accumulator[delay] < -0x40000000) {
635             accumulator[delay] = -0x40000000;
636             underflow_count++;
637         }
638
639         results[delay] = (int16_t)(accumulator[delay] >> 15); //resu
640     }
641     return;
```

```
080013d0:   add     r3, r2
080013d2:   adds    r3, #2
080013d4:   lsls    r3, r3, #1
080013d6:   ldr     r2, [r7, #8]
080013d8:   add     r3, r2
080013da:   ldrsh.w r3, [r3]
080013de:   mov     r2, r3
080013e0:   ldr     r3, [r7, #84]   ; 0x54
080013e2:   mul.w   r2, r3, r2
080013e6:   ldr     r3, [r7, #92]   ; 0x5c
080013e8:   adds    r3, #2
080013ea:   add     r2, r1
080013ec:   lsls    r3, r3, #2
080013ee:   adds    r3, #104        ; 0x68
080013f0:   add     r3, r7
080013f2:   str.w   r2, [r3, #-84]
620                 accumulator[delay+3] += filter_coeff * (int32_t)hist
080013f6:   ldr     r3, [r7, #92]   ; 0x5c
080013f8:   adds    r3, #3
080013fa:   lsls    r3, r3, #2
080013fc:   adds    r3, #104        ; 0x68
080013fe:   add     r3, r7
08001400:   ldr.w   r1, [r3, #-84]
08001404:   ldr     r2, [r7, #96]   ; 0x60
08001406:   ldr     r3, [r7, #92]   ; 0x5c
08001408:   add     r3, r2
0800140a:   adds    r3, #3
0800140c:   lsls    r3, r3, #1
```



```
614     for (tap = 0; tap < NUMBER_OF_TAPS; tap++) {
615         int32_t filter_coeff = (int32_t)filter_coeffs[tap];
616         for(int delay = 0; delay < BLOCK_SIZE; delay+=4) { // reduci
617             accumulator[delay] += filter_coeff * (int32_t)history[ta
618             accumulator[delay+1] += filter_coeff * (int32_t)history[
619             accumulator[delay+2] += filter_coeff * (int32_t)history[
620             accumulator[delay+3] += filter_coeff * (int32_t)history[
621         }
622     }
623
624         //size of old history is NUMBER_OF_TAPS
625         //size of new history is NUMBER_OF_TAPS + BLOCK_SIZE - 1
626     for(tap = NUMBER_OF_TAPS-2; tap > -1; tap--) {
627             history[tap+BLOCK_SIZE] = history[tap];
```

```
Enter location here
0800140e:   ldr     r2, [r7, #8]
08001410:   add     r3, r2
08001412:   ldrsh.w r3, [r3]
08001416:   mov     r2, r3
08001418:   ldr     r3, [r7, #84]   ; 0x54
0800141a:   mul.w   r2, r3, r2
0800141e:   ldr     r3, [r7, #92]   ; 0x5c
08001420:   adds    r3, #3
08001422:   add     r2, r1
08001424:   lsls    r3, r3, #2
08001426:   adds    r3, #104        ; 0x68
08001428:   add     r3, r7
0800142a:   str.w   r2, [r3, #-84]
```

In the convolution sum, we have 93 assembly instructions per loop, this means that we have a total of 93*4, 372 instructions every time we perform convolution for a block of samples which is a slightly larger number than before applying unrolling.

The rest of the instructions including the update of history, outer convolution loop, and loading filter coefficient stayed the same as in ProcessBlock2(), nothing else was modified in the code.

Results

-O0

30	ITM Port 31	1	2313561	23.135610 ms
31	ITM Port 31	2	2454831	24.548310 ms
32	ITM Port 31	1	2457641	24.576410 ms
33	ITM Port 31	2	2598920	25.989200 ms
34	ITM Port 31	1	2601467	26.014670 ms
35	ITM Port 31	2	2743059	27.430590 ms

Total: 141592 cycles, 1041.59us

total/BLOCK_SIZE = 8849.5 cycles, 88.495us

-Os

30	ITM Port 31	1	812115	8.121150 ms
31	ITM Port 31	2	841632	8.416320 ms
32	ITM Port 31	1	842837	8.428370 ms
33	ITM Port 31	2	872533	8.725330 ms
34	ITM Port 31	1	873735	8.737350 ms
35	ITM Port 31	2	903283	9.032830 ms

Total: 29548 cycles, 295.58us

total/BLOCK_SIZE = 1846.75 cycles, 18.47us

-Ofast

38	ITM Port 31	1	736538	7.365380 ms
39	ITM Port 31	2	761178	7.611780 ms
40	ITM Port 31	1	762294	7.622940 ms
41	ITM Port 31	2	786936	7.869360 ms
42	ITM Port 31	1	788051	7.880510 ms
43	ITM Port 31	2	812724	8.127240 ms

Total: 24673 cycles, 246.73us

total/BLOCK_SIZE = 1542.06 cycles, 15.42us

The function satisfies the timing requirements in all of the compilations assuming 8kHz sampling 1/8000Hz = 125us. Although we required more instructions to perform the convolution sum, the unrolled version of this function still ran faster. This is likely because we are checking the condition of the for loop 4 times instead of 16 times. This involves multiple load operations. The speedup resulting in the reduction of load operations is greater than the slowdown resulting from the increased convolution sum instructions.

Experimental Results

	Cycles per sample / size of the program		
Function Name	**-O0**	**-Os**	**-Ofast**
ProcessBlock()	13051.7 / 24.89 KB	3766.3 / 21.03 KB	1927.7 / 21.71 KB
ProcessBlock2()	10016.8 / 24.89 KB	2212.6 / 21.04 KB	1536.5 / 23.16 KB
ProcessBlock3()	7231.7 / 24.92 KB	2303 / 21.02 KB	1970.3 / 21.71 KB
ProcessBlock 4()	8849.5 / 25.05 KB	1846.75 / 21.08 KB	1542.06 / 23.18 KB

The table shows the performance of four different functions that process blocks of audio samples, each with a different block size and implementation method, under three different optimization settings: -O0, -Os, and -Ofast.

The results indicate that the implementation method and block size have a big impact on the functions' performance, as well as the optimization level. ProcessBlock() and ProcessBlock2() seem to be less optimized and efficient compared to the other functions, based on their higher

cycle per sample values. ProcessBlock3() and ProcessBlock4() seem to perform better because of their implementation method, especially when optimization is enabled. The unrolling method used in ProcessBlock3() and ProcessBlock4() seems to improve performance significantly as these functions have lower cycle per sample values than the other two functions with the same block size.

When optimization is enabled, all four functions perform significantly better, and among the three optimization levels, -Ofast provides the best performance in terms of cycle per sample values for all four functions. From looking at the table, the program size isn't affected much by the block size and implementation method, but enabling optimization does reduce the size of the program, with -Os providing the smallest program size for all four functions. Overall, it seems like choosing the right block size and implementation method, as well as enabling optimization, can make a big difference in the performance of audio processing functions.

RUBRIC FOR SELF-ASSESSMENT

Poor (0-5%)	Marginal (6-9%)	Good (10-12%)	Very good (12-15%)
No attempt or non-functional implementation of most of the functions. Missing or unconvincing tests of the functional correctness of the implementation in the source file(s). Documentation (report) is poorly presented or incomplete.	Mostly correct attempted implementation of block processing. Attempts some sort of test for the functional correctness of the implementations. Documentation (report) is minimal or incomplete but enough to run the code.	Implemented functions correctly implement block processing. Adequately tests the functional correctness of the implementations. Documentation (report) explains the implemented functions (with annotated screenshots/source code) and an attempt to explain the observed results.	Implemented functions correctly implement block processing. Adequately tests the functional correctness of the implementations. Documentation (report) explains the implemented functions (with annotated screenshots/source code) and compares the different observed results with discussions as to reasons why. Explains all relevant parts of the code (e.g., the role of pointers, variables, etc.) Instructions to run the submitted code are clear and allow easy reproduction of the presented results.

A minimal checklist

- ☑ ~~Implemented with frame size 3~~
- ☑ ~~Wrote test(s) to show correct implementation with frame size 3~~
- ☑ ~~Implemented with frame size 16~~
- ☑ ~~Wrote test(s) to show correct implementation with frame size 3~~
- ☑ ~~Wrote detailed explanations/investigations of how/why this works~~
- ☑ ~~Implemented an unrolled implementation~~
- ☑ ~~Wrote an explanation of how unrolling works, referred to assembly code~~
- ☑ ~~Explained the changes with different compiler flags~~
- ☑ ~~Table is complete and explained, talked about timing requirements~~
- ☑ ~~Provided a detailed README/walkthrough~~

We met all the assignment requirements and we have clear instructions on how to interpret and run our code.

Team Work

Hanan 50%: Helped with the code, debugging, and some of the report.

Nadia 50%: Helped with the unrolled functions in the code, the report, and some debugging.

Signature _____Hanan_____

Signature ____Nadia_____