RTDSP Lab 4

Yong Wen Chua (ywc110) & Ryan Savitski (rs5010)

Declaration

Declaration: We confirm that this submission is our own work. In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others. We are aware that this course is bound by penalties as set out in the College examination offenses policy.

Signed: Yong Wen Chua & Ryan Savitski

Contents

1 Matlab Filter Design

The transition bands used in this lab are: 260 Hz -> 450 Hz and 2250 Hz -> 2500 Hz (choice was made before the spec was changed). The Matlab code used to generate the listing is given in section A.1.

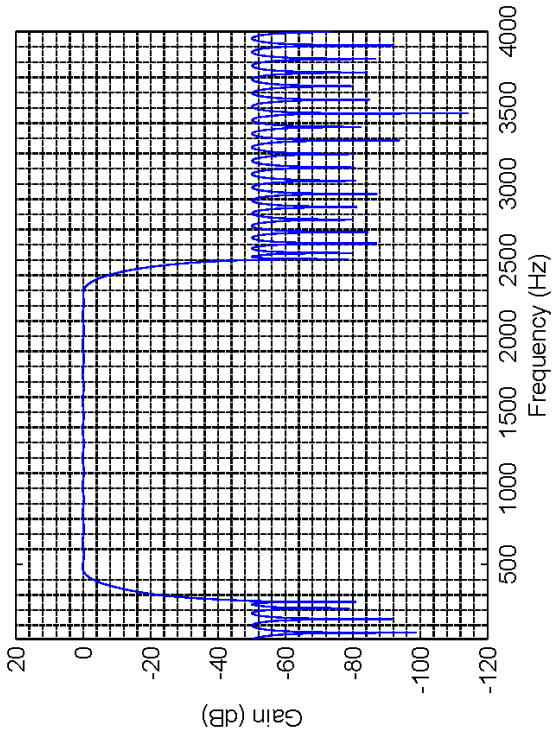
1.1 Coefficients

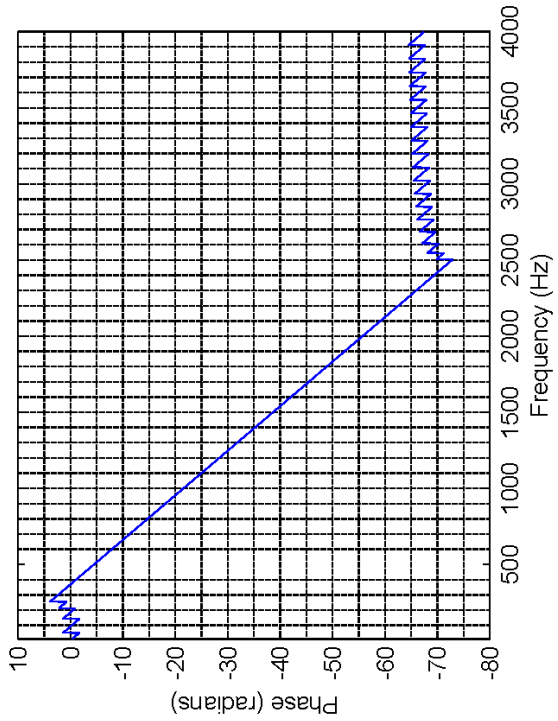
The coefficients generated by the Order 87 filter (with 88 coefficients) are given below. Note that the filter is linear phase and the coefficients are thus symmetric.

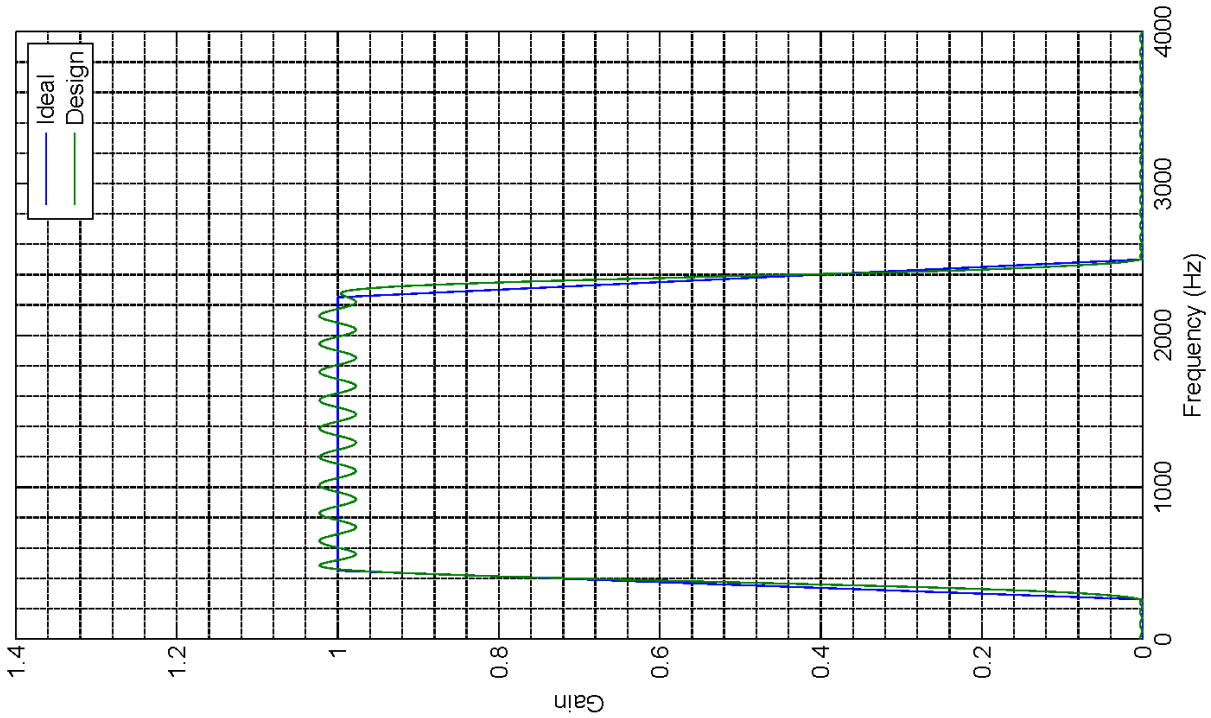
```
  -5.6238234861581632e-03   -4.8142851508671362e-03    3.2377476053097676e-03    5.2623077366623777e-03
3.8327678023773130e-04    2.5228524080704710e-03    6.6427594305550220e-03    2.0191540917237553e-03
6.0838154216067970e-04    6.0195074513261972e-03    2.2588854699456557e-03   -4.0581656174741142e-03
1.1053698037032480e-03    8.7570330682306904e-04   -9.4389095569342232e-03   -6.7133371831993478e-03
-4.4912094561377273e-04   -1.0781141008779919e-02   -1.2833025044814740e-02    7.4357338553088497e-04
-3.6475744956566657e-03   -1.2285472406529016e-02    4.9069216133462504e-03    1.7919769642964414e-02
-3.5124996299853682e-03    8.4328459566069963e-03    2.8242140033990469e-02    9.5414427887197482e-03
4.6527705138212187e-03    3.3181195014207174e-02    1.9161471979520985e-02   -1.1640938381470692e-02
1.4905816953372706e-02    1.8640626747436755e-02   -3.8390515525090867e-02   -3.0977635742666779e-02
6.9891233521773809e-03   -6.2265514731766294e-02   -1.0105444362367744e-01   -9.6437225383029998e-03
-5.1295032155504995e-02   -2.1091838197686907e-01   -2.1562212120427096e-02    4.2133153775698379e-01
4.2133153775698379e-01   -2.1562212120427096e-02   -2.1091838197686907e-01   -5.1295032155504995e-02
-9.6437225383029998e-03   -1.0105444362367744e-01   -6.2265514731766294e-02    6.9891233521773809e-03
-3.0977635742666779e-02   -3.8390515525090867e-02    1.8640626747436755e-02    1.4905816953372706e-02
-1.1640938381470692e-02    1.9161471979520985e-02    3.3181195014207174e-02    4.6527705138212187e-03
9.5414427887197482e-03    2.8242140033990469e-02    8.4328459566069963e-03   -3.5124996299853682e-03
1.7919769642964414e-02    4.9069216133462504e-03   -1.2285472406529016e-02   -3.6475744956566657e-03
7.4357338553088497e-04   -1.2833025044814740e-02   -1.0781141008779919e-02   -4.4912094561377273e-04
-6.7133371831993478e-03   -9.4389095569342232e-03    8.7570330682306904e-04    1.1053698037032480e-03
-4.0581656174741142e-03    2.2588854699456557e-03    6.0195074513261972e-03    6.0838154216067970e-04
2.0191540917237553e-03    6.6427594305550220e-03    2.5228524080704710e-03    3.8327678023773130e-04
5.2623077366623777e-03    3.2377476053097676e-03   -4.8142851508671362e-03   -5.6238234861581632e-03
```

1.2 Frequency Response

The frequency response of the generated filter is given on the following page.


Gain (dB)
20
0
-20
-40
-60
-80
-100
-120
500
1000
1500
2000
2500
3000
3500
4000
Frequency (Hz)


Phase (radians)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
500
1000
1500
2000
2500
3000
3500
4000
Frequency (Hz)


Ideal
Design
Gain
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1000
2000
3000
4000
Frequency (Hz)

2 Non-Circular Buffer FIR Filter

The code for the non-circular buffer FIR filter is given in section A.2.

2.1 Code Description

The coefficients for the filter are kept in a global `double` array with the name `b`. An array of size 88, `buffer`, is used as the storage for the previous inputs, required for the convolution. The code is inside the ISR and is presented below:At the start of every ISR, the buffer's contents are shifted:

```
int i;
double output = 0;

// shift buffer
for (i = N-1; i != 0; --i)
  buffer[i] = buffer[i-1];
buffer[0] = mono_read_16Bit(); // new sample

// mac loop
for (i = 0; i < N; ++i)
  output += b[i] * buffer[i];

mono_write_16Bit(output); // write
```

The code starts by doing a left shift of the buffer, then reading a new sample and writing it into the start of the buffer. Then a convolution is performed with a mac loop. The convolution is done according to the following equation:

$$output = \sum_{i=0}^{87} b[i] \times buffer[i]$$

Finally, the sample is written to the output codec.

2.2 Oscilloscope Traces

The oscilloscope trace of the filter implemented on the DSP behave as expected with the amplitude changing accordingly.

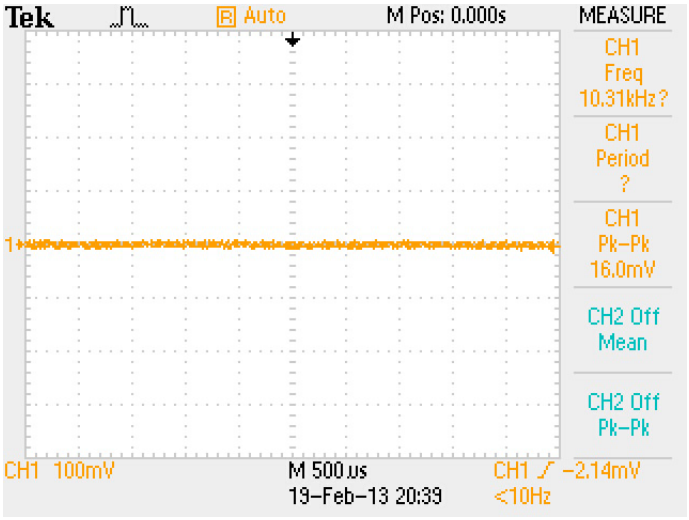


Figure 2.1: 200 Hz input, with almost zero output. This is in the stop-band.

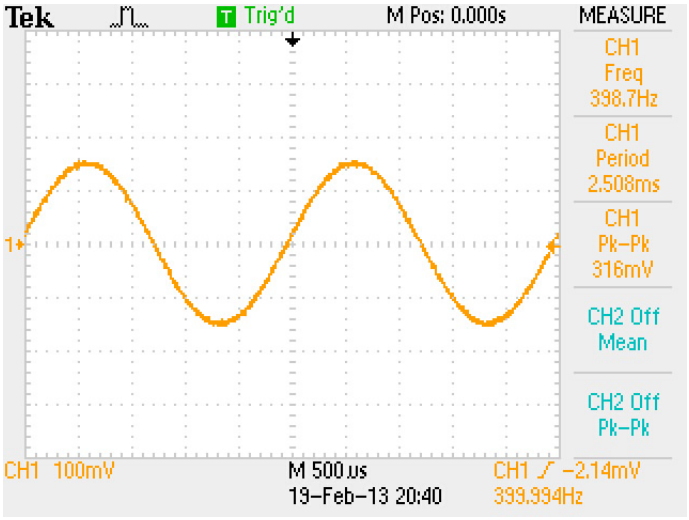


Figure 2.2: 400 Hz input, with increasing output amplitude. This is in the first transition band.

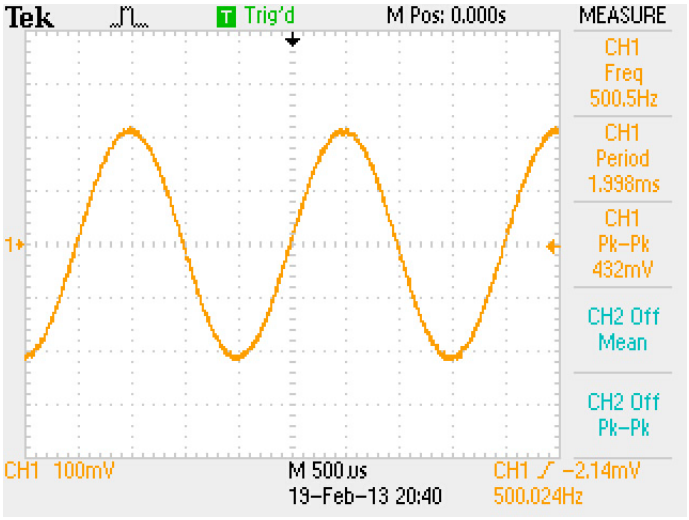


Figure 2.3: 500 Hz input, with maximum output amplitude. This is within the passband.

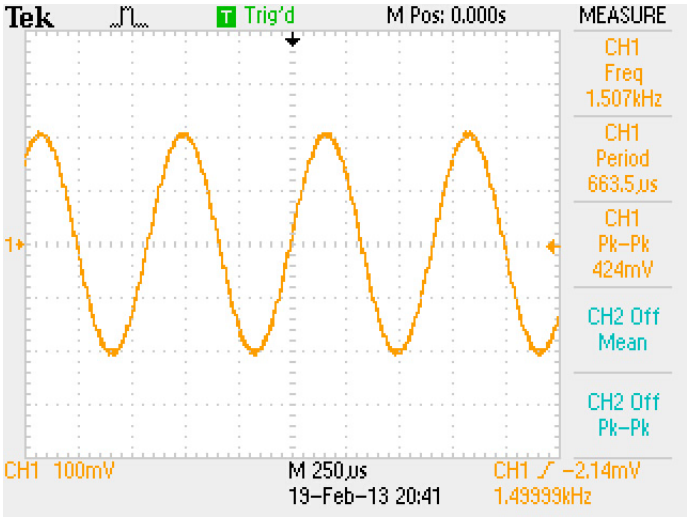


Figure 2.4: 1500 Hz input, with maximum amplitude. This is within the passband.

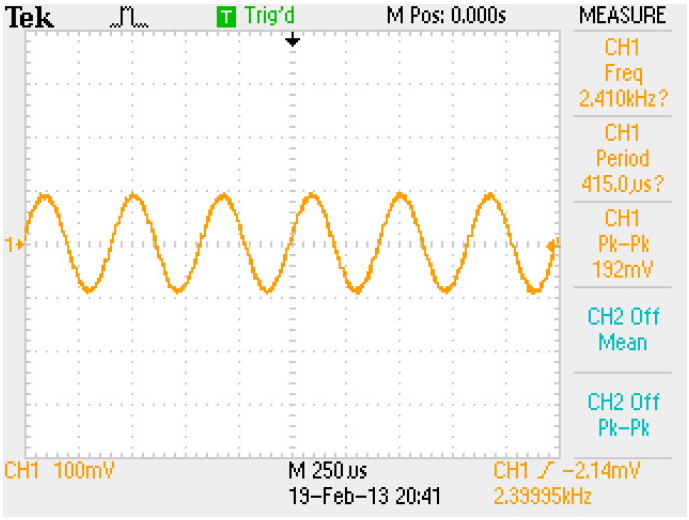


Figure 2.5: 2400 Hz, with decreasing amplitude. This is within the second transition band.

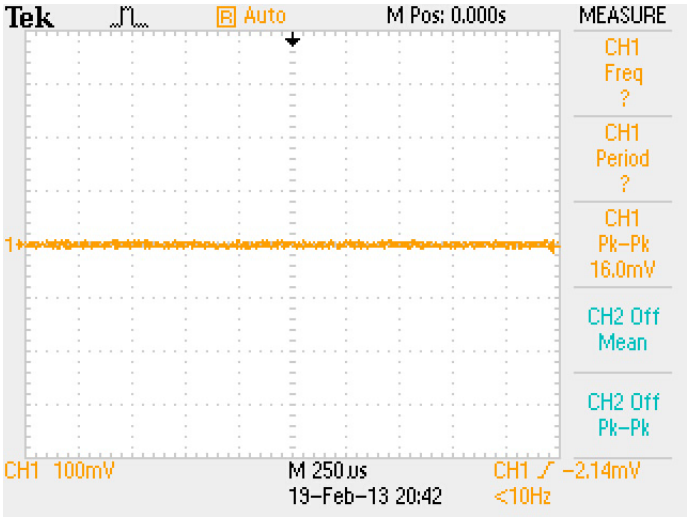


Figure 2.6: 3000 Hz input, with zero output. This is within the second stop-band.

2.3 Code Performance

The number of cycles for the simple non-circular FIR part with overhead of sample read/write cycles, are given below. Note that the actual overhead cycle counts depend on both the optimisation level and how/if the compiler inlines the calls. In addition, the breakpoint insertion offsets make it hard to time the overhead of just the sample input/output operations. Therefore it is not sensible to present the results without the overheads included. The estimated overhead at o2/o3 is 130 cycles.

Optimisation Level	Cycle count with read/write sample overhead
None	5766
o0	4666
o1	3855
o2/o3	1584

Note that seeing as this is a DSP board, the compiler toolchain is written to recognise mac-like loops and buffer operations, inserting templated assembly where needed. For example, for the above code the compiler did a triple in-flight branch shift loop that is much faster than the naive serial shifting. We are aware that simple mac loop code as above can also trigger a dual-unit pipelined template from the compiler at o2/o3 that would push the cycle counts towards 800, but we were unsuccessful in triggering that pattern. The compiler's pattern matcher is fairly unreliable from observations during lab work.

3 Circular Buffer FIR Filter

3.1 Naive Implementation

A simple version of the circular buffer was first implemented to ensure that it worked correctly. The code listing can be found in section A.3. Its operations are explained in the next section.

3.1.1 Code Description

A variable "`index`" is used to indicate the position in the array at which the "current" sample should reside. This `index` is incremented after every new sample is obtained, eventually wrapping around to the front of the array. Thus, if the current index is of value i, then the previous nth sample will be given by the index value of $[(i-n)+N]\%N$ where $N = 88$ is the total number of coefficients. The array, and `index` are defined by

```
Int16 buffer[N] = {0}; // initialise everything to zero
int index = 0;
```

The newly retrieved sample will first be written to the buffer.

```
*(buffer + index) = input; // equivalent to, and no faster than writing buffer[index] = input
```

A loop is then started to perform the Multiply and Accumulate (MAC) operation and the result is stored in `result`. Proper circular offset buffering is calculated using the method described earlier in this section.

```
for (i = 0; i < N; i++)
  result += b[i]* buffer[ ((index-i) + N) % N];
```

The index is then incremented. The mod operator ensures that proper wrapping around occurs.

```
index = (index + 1)%N;
```

3.1.2 Code Performance

The number of cycles taken between the start, and the end of the ISR routine is given in the table below. In general, this implementation of the buffer performed worse than the Non-Circular buffer version described in section §2. This is due to the compiler no longer seeing the trigger patterns for optimised template loops.

Optimisation Level	Cycle count with read/write overheads
None	7377
o0	5830
o2	3934

3.2 Optimised Implementation

The code listing for the optimised implementation can be found in section A.4.

3.2.1 Code Operation

Similar to the code operation described in section 3.1.1, a variable "`index`" is used to indicate the position in the array at which the "current" sample should reside. This `index` is decremented after every new sample is obtained, eventually wrapping around.

The optimised circular buffer code starts with the following pointer declarations for buffer indexing/walks:

```
double* coeffptr = b;
double* coeffEndptr = b + N; // points to the element AFTER the coefficient array
double* bufferEndptr = buffer + N; // one after last element
```

It is important to note that all of these values are compile time constants and are thus "free" in terms of performance.

The code proceeds to declare the pointer to the oldest sample (to be overwritten by the new sample) and calculate the amount of entries there are between current sample and the end of the sample buffer (the MAC will be two part, first part walking the sample buffer forwards, so this is the loop counter for the forward walk).

```
double* sampleptr = buffer + index; // point to oldest sample initially

int loopcnt = bufferEndptr - sampleptr; // how many iterations are needed for a single-step loop
```

As the MAC loops will be manually unrolled 4 times in C (reasoning below), we also need to calculate how much misalignment there is for MAC loop iterations, to be handled separately after the first MAC loop. In addition, four accumulators are declared.

```
char modunroll = loopcnt % 4; // non-integral leftover of an unrolled loop

// accumulators
double result = 0;
double result2 = 0;
double result3 = 0;
double result4 = 0;
```

Then, the sample is read into the correct part of the circular buffer:

```
*sampleptr = mono_read_16Bit(); // read sample into buffer
```

Then the code drops into a 4-unrolled MAC loop that steps the buffers in the forwards direction until it hits the end of the sample buffer (can be a range of values since this is a circular buffer). Afterwards, the non-integral case is handled for one, two or three leftover values. Note that the loop count is known at the start of the `ISR` and thus the compiler will turn the condition check into a simple loop counter, which is the optimal solution.

```
// process samples until the end of the sample buffer is hit
while(sampleptr < bufferEndptr-3)
{
  result += (*coeffptr++) * (*sampleptr++);
  result2 += (*coeffptr++) * (*sampleptr++);
  result3 += (*coeffptr++) * (*sampleptr++);
  result4 += (*coeffptr++) * (*sampleptr++);
}
// take care of non-integral leftover iterations
if (modunroll>0) result += (*coeffptr++) * (*sampleptr++);
if (modunroll>1) result2 += (*coeffptr++) * (*sampleptr++);
if (modunroll>2) result3 += (*coeffptr++) * (*sampleptr++);
```

Then we reset the sample pointer to the start of the sample buffer, effectively wrapping it around and do another pass of the MAC for the remainder of the FIR operation. Note that the difference between the current coefficient pointer and the end of the coefficient buffer is the required amount of iterations. This is also turned into a simple loop counter by the compiler.

```
sampleptr = buffer; // wrap pointer to beginning of the buffer

// pass the remainder of the buffer (amount of iterations = how many coefficients there are left
    to process)
while (coeffptr < coeffEndptr-3)
{
  result += (*coeffptr++) * (*sampleptr++);
  result2 += (*coeffptr++) * (*sampleptr++);
  result3 += (*coeffptr++) * (*sampleptr++);
  result4 += (*coeffptr++) * (*sampleptr++);
}

// take care of non-integral leftover iterations
if (modunroll==1) result += (*coeffptr++) * (*sampleptr++);
if (modunroll==1 || modunroll==2) result2 += (*coeffptr++) * (*sampleptr++);
if (modunroll==1 || modunroll==2 || modunroll==3) result3 += (*coeffptr++) * (*sampleptr++);
```

The wrapup part of the code does the final accumulation of partial accumulators, stepping the index and writing the resulting sample to the codec.

```
// sum the accumulators
result = result + result2 + result3 + result4;

// advance index into circular buffer
index = (index == 0) ? N-1 : index-1;

mono_write_16Bit(result); // output sample
```

The reason for unrolling the loop four times is as follows (all with o2/o3 optimisation settings):

- with one accumulator, the compiler was not noticing the possible software pipelining of the results and was limited by serial writes to one accumulator.
- with two accumulators, the compiler notices that it can use both sides of the processor. But there are still several nop cycles.
- with four accumulators, the compiler is able to unroll its generated loop an extra time to remove amount of nop cycles.

3.2.2 Code Performance

With four manual loop unrolls to use as hints for the compiler and o2/o3 compiler setting, the C version can be pushed to 422 cycles for the entire ISR. From our measurements, we estimate the read/write overheads for the samples to be around 130 cycles at o2/o3, therefore we estimate that the actual optimised C version FIR pass is 192 cycles.

Optimisation Level	Est. Cycle count without sample overheads	Cycle count with read/write overheads
none	3042	3172
o1	1842	1972
o2/o3	192	422

Note that this code's performance at a high level of optimisation is very impressive, the C code is structured to take advantage of the compiler's pattern matcher to utilise the tight mac loops and automatic unrolling.

Potentially we could improve the code further to take advantage of the coefficient symmetry to reduce the amount of loads by a quarter and halve the amount of multiplications. However, at C level, the compiler could not pick up on the possible software pipelining in practice. Software pipelining will be discussed in section 4.2.1.

Another possible optimisation path would be to change the filter, while keeping it conformant to the spec, to have a significant amount of zero coefficients, which could be handled very quickly by the code.

3.2.3 Spectrum Analyser Output

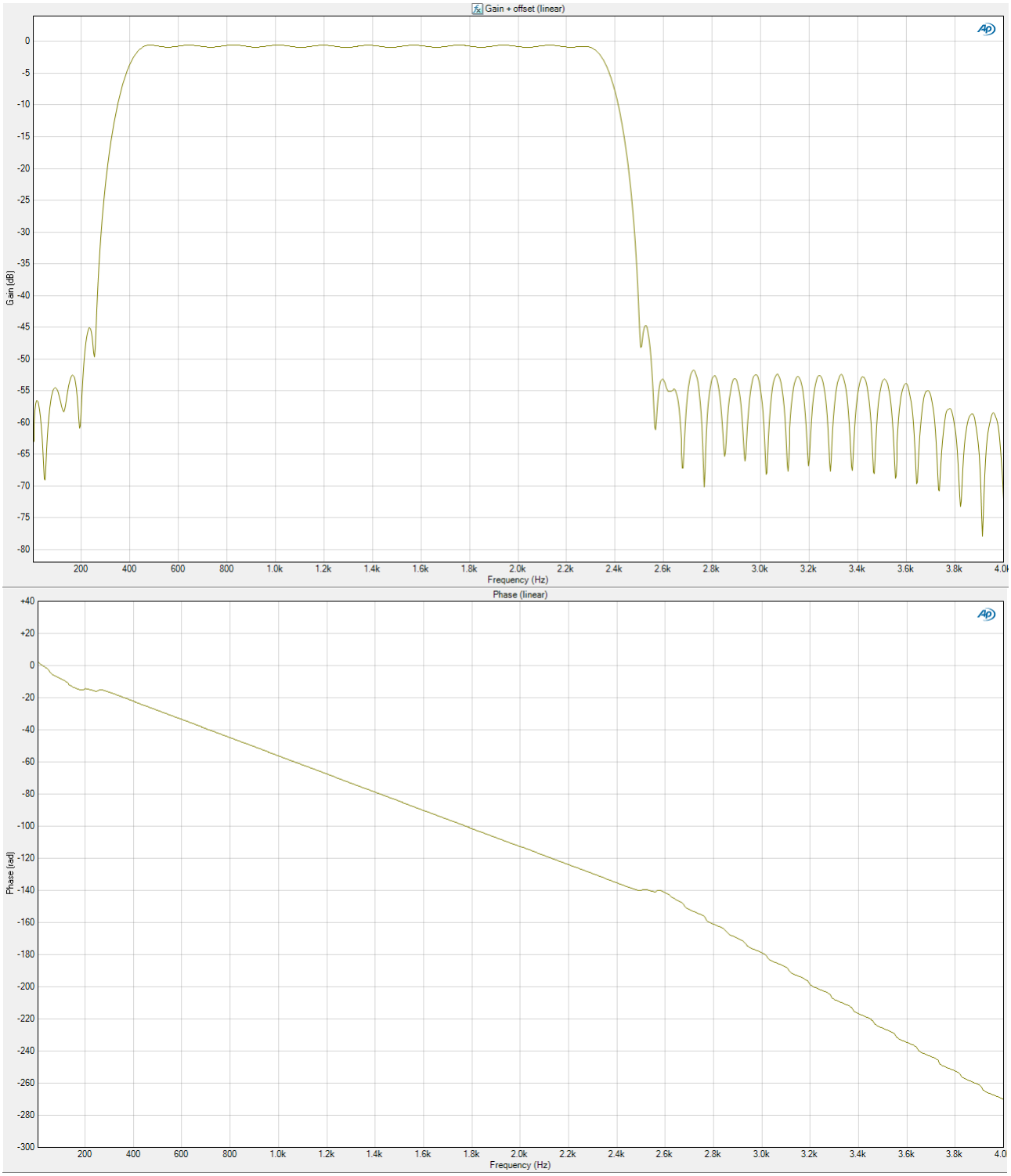
The output for the spectrum analyser is given in the figures below. Due to the input being fed to only one channel on the DSP, along with the potential divider in the circuitry, the value "seen" by the DSP will be one-fourth of what was provided by the analyser. This leads to an approximate -12 dB gain for the output in the frequency response. The figures given below have the necessary offset to reflect this.

The phase is observed to still be linear on hardware, but it is important to note that the overall change in the roughly 300 Hz to 2500 Hz range is 140 radians. This is more than the 70 radians change we observed in the Matlab plots, meaning that the slope of the phase response is steeper.

The relationship between the phase response$\phi(\omega)$, and the group delay $\tau_g(\omega)$ is given by

$$\tau_g(\omega) = -\frac{d\phi(\omega)}{d\omega}$$

Thus, a higher group delay will contribute to a steeper phase response, which was observed. This means that the hardware filter has a higher group delay, which can be attributed to the delay in the codec buffers.



4 Assembly Implementation

An implementation of the MAC operation was done in assembly. The code for the C file that calls the assembly function is given in section A.5.1. The ISR routine simply reads the sample from the output, calls the assembly function and writes the output. A buffer size of 1024 bytes was used. This is because there are 88 entries in the buffer, and 88 entries require $88 \times \frac{64}{8} = 704$ bytes of space. When rounded up to the nearest power of two, we get 1024.

Two versions of the assembly function were implemented, and will be detailed later in this section.

4.1 Linear Implementation

An assembly implementation of the MAC operation without any parallelism was implemented to test the output.

The code listing can be found in section A.5.2. In the comments to the code, the numbers in brackets after the code indicate the number of delay slots required after the instruction is sent to `E1` stage of the pipeline before its results can be used. For floating point instructions, a second number will indicate the number of latency cycles after the `E1` stage of the pipeline before the functional unit can execute another instruction.

4.1.1 Code Operation

The structure of the code before, and after the MAC loop is generally the same as the assembly code provided. The `AMR` register is set to have a value of `0x90004`, which sets the register `A5` to use circular buffering with a block size of 1024 bytes. The MAC loop then simply consists of code to load the sample data and the coefficients, multiply them together, and finally add them to an accumulator. The straightforward loop code is given below:

```
1     LDDW .D1     *A5++, A11:A10 ; (4) loads the (delayed) sample into A11:A10, and post increment
          pointer
2  ||  LDDW .D2     *B4++, B11:B10 ; (4) load the coefficient into B11:B10, and post increment pointer
3     NOP 4
4     MPYDP .M1X     A11:A10, B11:B10, A11:A10 ; (9, 4) DP multiply
5     NOP 9
6     ADDDP .L1    A15:A14, A11:A10, A15:A14 ; (6, 2) DP ADD
7     NOP 6
```

In the first execute packet of the loop, the coefficient and the sample are loaded into their respective registers (`A11:A10`, and `B11:B10`) in parallel using the `D` units on both sides. 4 delay slots are required before the results can be used. The values are then multiplied using the `MPYDP` instruction, which uses the `M1` unit, and utilises the cross path (thus the `.M1X`). 9 delay slots are required before the results are added using `ADDDP`. Then, a further six delay slots are required before the loop begins again.

4.1.2 Code Performance

The number of cycles taken between the start, and the end of the ISR routine is given in the table below. The number given is the lowest number of clock cycles observed. The C code in this case do not change much through the various optimisation level. This is because the compiler does not optimise the assembly code, and the assembly code has a constant number of clock cycles (including the five `NOP`s after the branch back to C). This linear and straightforward implementation of the MAC operation in assembly actually performs worse than the Non-Circular Buffer implemented in C at higher levels of optimisation. This is because at higher levels of optimisation, the compiler will attempt to optimise using techniques such as software pipelining. This will be further discussed in section §5.

Optimisation Level	Number of Clock Cycles	Assembly Code
None	2736	2594
o0	2736	
o2	2730	

4.2 Optimised Implementation

Various techniques can be employed to optimise the assembler code and shave the number of cycles required by five times. The techniques will be described in this section. The code listing can be found in section A.5.3.

4.2.1 Optimisation Techniques

There are various techniques that can be employed to take advantage of the VLIW architecture of the DSP hardware. This mostly include exploiting the ability to schedule multiple instructions that utilise different functional units to be executed in parallel, and also to understand how the pipeline works for the various instructions so as to interleave instructions. Some of the techniques used by the compiler (described in section §5) are also used.

Double precision (DP) instructions are the first area for optimisation. The delay slots between two consecutive DP instructions where the second instruction makes use of the result from the first instruction could be reduced by one (for example `MPYDP` followed by `ADDDP`). This is because the DP instructions write the lower half of the results to the register first, before writing the upper half of the results to the register in the final delay slot. DP instructions that read the lower half results first in `E1`, followed by the upper half in `E2` can be scheduled to start executing in the final delay slot of the previous DP instruction. Thus, the number of delay slots between `MPYDP` followed by `ADDDP` can be reduced from 9 to 8.

Utilising multiple functional units on both sides is the second area for optimisation. This works, so long as the operations do not write to the same registers in the same cycle. There is also a need to be careful to not read more than four registers in the same register file in the same execute packet. Thus, two `MPYDP` and `ADDDP` operations can take place in parallel utilising both of the functional units. This can roughly half the number of cycles required for the code to run, but does, however, require twice the number of registers required.

Software pipelining for loops is the third area for optimisation. Software pipelining is analogous to hardware pipelining where multiple instructions are interleaved so that the functional units can be maximally utilised during their delay slots, subject to their latencies, if any. Software pipelining, along with loop unrolling are techniques used by compilers to optimise code. In software pipelining, the pipeline is first primed using a pipeline prologue. The main loop kernel is then executed for the required number of times, with several loop cycles unrolled to execute interleaved. Then, the loop epilogue will finish up any outstanding tasks. This technique can roughly reduce the number of cycles by a factor roughly equivalent to the number of times the loop is unrolled, but requires proper planning and tracking.

Finally, taking advantage of the branch delay slots can also reduce the numbers of cycles in a non-trivial manner. The branch instruction requires five delay slots afterwards, whether the branch is taken or not. Those five execute packets are guaranteed to execute, and thus code can be executed during those execute packets.

These techniques are employed in the code implementation, to be explained later on in this section.

4.2.2 Code Operation

The register usage is described in the comments in the listing in section A.5.3. Where possible (restricted by the number of functional units available, and the avoidance of hazards due to dependencies), operations are run in parallel.

Code Setup

The assembly function first starts off by setting the AMR register is set to have a value of 0x90004, which sets the register `A5` to use circular buffering with a block size of 1024 bytes. At the same time, we save some of the values in registers we

are going to use later onto the stack using the Stack Pointer B15. The address pointer for the sample that was just read is dereferenced, along with the address of the circular buffer.

```
1        MVC.S2      AMR,B13        ;(0) Save contents of AMR reg to B13
2   ||   STW .D2     B3, *++B15     ; (0) save return to C to stack
3   ||   LDDW.D1     *A6,A11:A10    ;(4) Get the 32 bit data for read_samp put it in A11:A10
4
5        STW .D2     B6, *++B15     ; (0) save &filtered_samp to stack
6   ||   MVK .S2     4H,B2          ;(0)Set AMR to allow A5 to be used for circular addressing
          with BK0
7   ||   LDW .D1     *A4,A5         ;(4) Get the address of the circ_ptr, dereference then
         place in A5
8        MVKLH .S2   9H,B2          ;(0)Set BK0 to allow for 1024 bytes addressing
9        MVC.S2      B2,AMR         ;(0) set AMR reg
10
11       NOP 2                      ; A5 now holds address pointing into delay_circ    ; set
             circular mode using the AMR
12  MVC .S2     AMR,B13    ;(0) Save contents of AMR reg to B13
13  MVK .S2     4H,B2      ;(0) Lower half. set A5 to be circular buffering addressing mode using
         BK0
14  MVKLH .S2   9H,B2      ;(0) Upper half. Set BK0 to work for 1024 bytes
15  MVC .S2     B2,AMR     ;(0) set AMR reg
16       NOP 2                      ; A5 now holds address pointing into delay_circ
```

Next, the sample that was just read is written into the appropriate address in memory (the circular buffer), and the registers used for accumulations are ZEROed. The address for the next execution of the convolution function to write the new sample to is written back to memory.

```
1        STW .D1     A11,*--A5     ;(0) Store new input sample (MSB) to delay_circ array
2   ||   ZERO .S1    A1            ;(0) zero accumulator LSB
3   ||   ZERO .S2    B3
4
5        STW .D1     A10,*--A5     ;(0) Store new input sample (LSB) to delay_circ array
6   ||   ZERO.S1     A0            ;(0) zero accumulator MSB
7   ||   ZERO .S2    B2
8        STW.D1      A5,*A4    ;(0) write back the decremented pointer to circ_ptr
9                              ; this points to the end of the MSB of where the next sample
10                             ; will be stored on the next call to this function
```

Pipeline Description

The pipeline used in the implementation is complicated and warrants explanation. It can be summarised in figure 4.1.

Branch Considerations

The loop for the MAC operations takes four cycles. Thus, a branch operation for the end of the ith loop iteration must be scheculed on the $i-1$th loop iteration. In addition, the branch for the end of the first loop iteration must be scheduled 2 cycles before the start of the first loop iteration. This also results in a spurious loop being inserted at the end. Consider that there are N loop iterations expected. At loop iteration $N-1$, the branch back to the start of the loop for loop iteration N will be scheduled. In normal operations, the loop DOES NOT branch back to the beginning of the loop in the final loop iteration.

Instructions Properties

It should be noted that how the various instructions reads and writes registers, and their respective latency and delay slots play an important role in the operation of the pipeline. Without them, the pipeline would fail. Consider the following instructions being issued to the `E1` stage of the pipeline at cycle i:

- LDDW writes the results to the registers at $i+4$
- MPYDP reads the input registers from i to $i+3$, and outputs the lower half of the result at $i+8$ and the upper half at $i+9$
- ADDDP reads the lower half input at i and the upper half at $i+1$, and outputs the lower half of the result at $i+3$ and the upper half at $i+4$

Each iteration of a MAC operation basically involves all of the three instructions order in this order.

Pipeline Operation

Refer to figure 4.1. It can be seen from the lower diagram that the pipeline only fills up fully at the fifth loop iteration. Thus, we need a separate counter (register `B1`) to only allow accumulation to occur at the fifth loop iteration. Also, this means that we have to add 5 more iterations to the loop.

Both sides of architecture are used for the MAC operation. This results in side A and side B (see figure 4.1) being multiplied and accumulated separately. The accumulated results from both sides need to be added together at the end.s

Due to the fact that the `ADDDP` instruction is issued before the results from the previous invocation is complete, this will result in two separate accmulated sums for every other set of coefficients and samples being swapped in and out of the registers for each side of the functional units. Thus, there are virtually four data paths in operation at one go. At the end of the loop, the accumulated result from one of the virtual data pths must be prevented from being overwritten by the other virtual data path.

At the end of the loops, due to the added loop iterations so that the pipeline can be filled up, spurious operations will be performed, as described in the diagram. We must be careful not to use the registers that can be filled up with spurious data.

Loop Code

First, the extra counter `B1` required to kickstart the `ADDDP` instructions at the fifth iteration is initialised to 10. This coutner is added to the loop counter `B0`. `B1` is initialsed to 10 because the loop counter subtracts by two each time (two coefficients/samples are loaded to each side) and thus `B1` has to be multipled by 2. The branch necessary for the first loop iteration is also performed.

```
1     ||  MV .L2X          A8, B0      ;(0) move parameter (numCoefs) passed from C into b0
2     ||  MVK .S2          10, B1        ;(0) setup countdown to start addition
3
4
5         ADD              .L2 B0, B1, B0 ;(0) Branch needs 10/2=5 iterations to setup and running.
6     ||  B .S2            loop     ;(0) Loop is only 4 cycles,
7                                   ; so we need to kickstart the branch back for loop iteration 1
8
9     NOP                       ; NOP to allow the branch for Loop iteration 1 to happen right at
          the end
```

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
B1	8					6					4				2				0				0	
Loop	1			2			3				4			5			6							

A1				LA1								xA1 & LA3								+A1 & XA3 & LA3					
A2								LA2								xA2 & LA4									+A2 & XA4 & LA4
B1			LB1								xB1 & LB3								+B1 & XB3 & LB3						
B2							LB2								xB2 & LB4								B2 & XB4 & LB4		

Legend:

- Numbers: The ith MAC iteration involving the ith sample and coefficient
- A & B - The operations involving the M and L units on side A and B respectively. Note that loads use both sides.
- L - Load operation; x- Multiplication operation; + - Addition operation
- So +B2 refers to the second sample being accumulated (added) on the B side.

Iteration							
1	L1						
2	L2	L1					
3	L3	L2	x1				
4	L4	L3	x2	x1			
5	L5	L4	x3	x2	x1	+1	
6	L6	L5	x4	x3	x2	+2	
...							
45	L45	L44	x43	x42	x41	+41	
46	L46	L45	x44	x43	x42	+42	
47	L47	L46	x45	x44	x43	+43	
48	L48	L47	x46	x45	x44	+44	
49		L48		x46	x45	+45	
50					x46	+46	

Explanation:
Consider the entirety of each iteration of the MAC operation as a 6-stage pipeline (the columns).
We will ignore the the fact that sides A and B are both used at the same time and just consider the iterations.
Each iteration of the loop will move the pipeline stages forward by one
The loading operation (L) takes 2 pipeline stages.
The multiply operation (x) takes 3 pipleine stages, but the results can be used in the same iteration
The addition operation (+) takes 2 pipeline stages.
Red operation refers to spurious operations that cannot be prevent
The number indicates the ith iteration of the MAC operation

Figure 4.1: Pipeline description diagram.

The loop proper begins. Sides A and Sides B perform the MAC accordingly. This set of code results in the pipeline described in figure 4.1. The conditional execution is done so as to prevent some of the spurious operations from occurring at the end of the loop. As described above, to prevent one virtual data path on side B from overwriting the other virtual data path, we must move the registers that store the result for side B accumulation away. This happens at the end of the loop.

```
loop:
        [B0] SUB .S2      B0,2,B0  ;(0) Decrement loop counter by 2, because we are doing two
            calculations togerher


        [B1] SUB .D2      B1,2,B1  ; (0) countdown to allow start of addition. Countdown is done
            by two
                                   ; because the loop counter is decremented by two. And since we
                                       added B0
                                   ; and B1 together before the loop, we must also double B1's
                                       value and subtract
                                   ; by 2 each time

        [B0] B .S2        loop          ;(5) for current iteration i, kickstart the branch back
            for iteration i+1
    ||  [B0] LDDW .D1     *A5++, A11:A10 ; (4) B - Load delayed sample
    ||  [B0] LDDW .D2     *B4++, B11:B10 ; (4) B - Load coefficient
    ||  [B0] MPYDP .M2X   B11:B10, A11:A10, B3:B2 ; (9,4) B - Multiply
    || [!B1] ADDDP .L2    B7:B6, B3:B2, B7:B6 ; (6,2) B - Accumulate


        [B0] LDDW .D1     *A5++, A9:A8 ; (4) A - Load delayed sample
    ||  [B0] LDDW .D2     *B4++, B9:B8 ; (4) A - Load coefficient
    ||  [B0] MPYDP .M1X   A9:A8, B9:B8, A3:A2 ; (9,4) A - Multiply
    || [!B1] ADDDP .L1    A1:A0, A3:A2, A1:A0 ; (6,2) A - Accumulate
    || [!B0] MV .S2       B6, B12 ; (0) for the final iteration this cycle, the LH result for B-
        Addition-44 is
                                  ; written on this cycle. We move the LH result for B-Addition-43
                                      out of the
                                  ; way to prevent losing them
```

We then continue the moves for the upper half of the B-side and do the same for the A-side. A-side move is done one cycle later because its MAC operations are one cycle behind that of B-side. Both virtual data paths are then added.

```
        MV .D1            A0, A12 ; (0) the UH result for A-Addition-44 is
                                  ; written on this cycle. We move the UH result for A-Addition-43
                                      out of the
                                  ; way to prevent losing them
    ||  MV .S2            B7, B13 ; (0) the UH result for B-Addition-44 is
                                  ; written on this cycle. We move the UH result for B-Addition-43
                                      out of the
                                  ; way to prevent losing them

        MV .D1            A1, A13 ; (0) the LH result for A-Addition-44 is
                                  ; written on this cycle. We move the LH result for A-Addition-43
                                      out of the
                                  ; way to prevent losing them
    ||  ADDDP .L2         B7:B6, B13:B12, B13:B12  ; (6,2) the supurious B-Addition-45 will write
        the LH result in
                                                 ; 2 cycles after this. Better start adding
                                                     result of B-Addition-44

```

```
14          ADDDP .L1          A1:A0, A13:A12, A13:A12  ; (6,2) the supurious A-Addition-45 will write
                the LH result in
15                                                      ; 2 cycles after this. Better start adding
                                                            result of A-Addition-44
```

Code Cleanup

The registers saved to stack will be restored, and the accumulated values from A-side and B-side are added up. They are then saved to address and the code returns to C.

```
1           LDW .D2            *B15--, B6    ; (4) get &filtered_samp from stack
2           LDW .D2            *B15--, B0    ; (4) get return to C from stack
3
4           NOP 3
5           ADDDP .L1X         A13:A12, B13:B12, A13:A12 ; (6,2) Add the results of Side A and B
                together
6
7           NOP
8
9           ; return to C code
10
11  lend:   B.S2               B0            ; (5) branch to b3 (register b3 holds the return address)
12          NOP      3
13
14          ; send the result of MAC back to C
15          STW.D2             A12,*B6        ;(0) Write accumulator (LSB) into filtered_samp
16
17          STW.D2             A13,*+B6[1]    ;(0) Write accumulator (MSB) into filtered_samp
18      ||  MVC.S2             B1,AMR         ;(0) restore  AMR reg to previous contents
19
20          .end
```

It should be noted that with this optimisation, the number of coefficients, N, **MUST** be a multiple of two. If the number of coefficients is not a multiple of two, additional coefficients with values of zero should be added to make N a multiple of two. Otherwise, the code will compute the result wrongly.

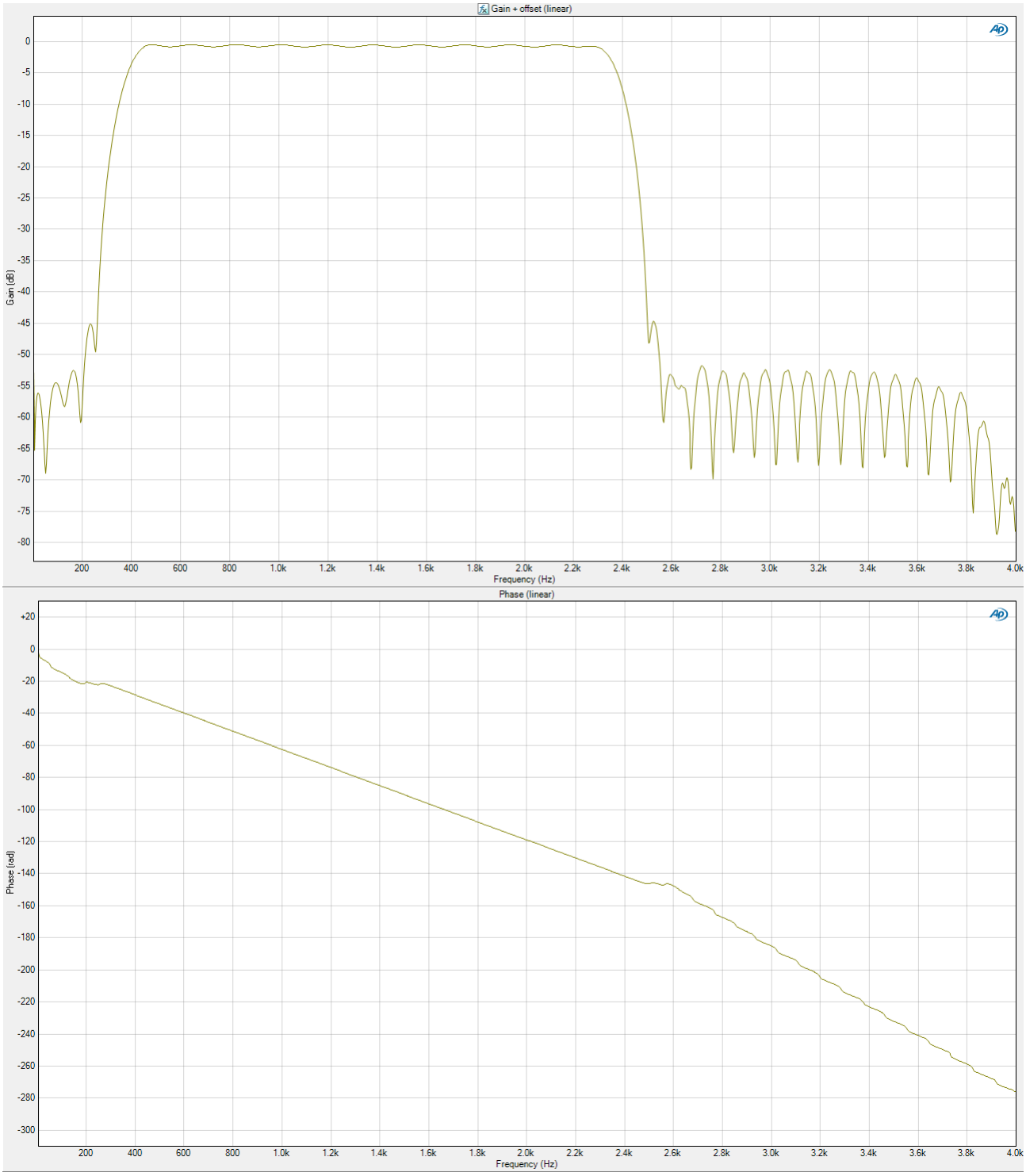
4.2.3 Code Performance

The number of cycles taken between the start, and the end of the ISR routine is given in the table below. The number given is the lowest number of clock cycles observed. The number might vary due to cache hits and/or misses. The C code in this case do not change much through the various optimisation level. This is because the compiler does not optimise the assembly code, and the assembly code has a constant number of clock cycles. The optimisation technique discussed in section 4.2.1 provide massive improvement to the code performance, by five fold.

Optimisation Level	Number of Clock Cycles	Assembly Code
None	378	239
o0	378	
o2	378	

4.2.4 Spectrum Analyser Traces

The output for the spectrum analyser is given below. As before, the -12 dB offset that occurs has been corrected in the trace below. The group delay that can be observed from the phase response is explained as before, from section 3.2.3.



5 Compiler Optimisation

The various optimisations performed by the compiler are described in the SPRU187O[1] document. Optimisation might result in a larger code size.

When no optimisation is done, the compiler generally generates assembly code "as-is" with no optimisation done to the code. This usually results in the fastest compilation time, and is easiest to debug (but with performance trade-off.). The section will examine the various optimisation performed by the compiler at various levels and examine how they could contribute to the increase in performance.

5.1 Level 0 Optimisation

The compiler will attempt to simplify the control-flow-graph (i.e. `if/else, for, switch` etc. statements). The code for both the different implementations do not use as much of these control statements, and thus not much improvement will arise from there. The compiler will also attempt to eliminate unused code, which is not present in both implementations. Next, the compiler will attempt to simplify statements and expressions. The implementations do not generally contain overly complicated expressions, and statements. However, the following statements contain expressions that always evaluate to the same constant values, and the compiler might attempt to "collapse" them into a constant value at compile-time, rather than ask them to be computed at run-time.

```
// from non-circular buffer
i = N-1;

// from circular buffer
double* coeffptr = b;
double* coeffEndptr = b + N; // points to the element AFTER the coefficient array
```

The compiler will also attempt to inline functions marked with the keyword `inline`. However, the keyword was not used in both implementations. The compiler will assign variables to registers, reducing the amount of memory access. This might have contributed a significant amount of code performance improvements to both implementation. It is likely that the circular buffer implementation benefited more from this optimisation, due to its use of pointers, which would have resulted in unnecessary amounts of dereferencing of pointers to pointers) .

Finally, the compiler will attempt to perform loop rotation (or loop inversion)[2]. Consider the following `for` loop in C code which will be transformed (essentially) by the compiler into an equivalent `while` loop in assembly. This results in two branches being run continually in a loop, and branches, whether taken or not, could lead to pipeline stalls (or in this architecture additional `NOP`s being inserted, which are wasteful if not optimised properly).

```
for (i = ; i < N; ++i) doSomething();

// transformed into
i = 0;
while (i < N) { // conditional branch to after end of loop if i >= N
  doSomething();
  ++i;
} // unconditional branch to start of loop
```

Loop rotation replaces the whole `while` block with an if block containing a `do..while` loop, which reduces the number of branches in the loop to one.

[1] http://www.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=spru187o&fileType=pdf
[2] See http://llvm.org/devmtg/2009-10/ScalarEvolutionAndLoopOptimization.pdf and http://en.wikipedia.org/wiki/Loop_inversion

```
i = 0;
if (i < N){ // conditional branch to after end of loop if i >= N OUTSIDE the loop
  do{
    doSomething();
    i++;
  } while (i < N); // conditional branch to beginning of loop if i < N
}
```

This technique contributes a significant improvement in both implementations. This technique also enable code that are loop-invariant to be moved out of the loop themselves[3].

5.2 Level 2 Optimisation

Level 2 optimisation performs all the optimisation in Levels 0 and 1. The optimisation performed in Level 1 (Performs local copy/constant propagation, Removes unused assignments , and Eliminates local common expressions) are not applicable to the implementations

The compiler attempts to perform various loop optimisation such as software pipelining and loop unrolling as described in section 4.2.1. These optimisations contribute the most to the improvement in performance, seeing that most of the code is spent in loops.

The compiler also attempts to convert array references in loops to incremented pointer form, which was what was done already in the circular buffer implementation. In this case, it is the fact that the circular buffer only loops over the values once, rather than twice by the non-circular buffer, that gives it the performance advantage.

The various global optimisation done by the compiler is not relevant.

[3] See http://en.wikipedia.org/wiki/Loop-invariant_code_motion

A Code Listings

A.1 Matlab Code for Filter Generation

Based on the specification given, the following Matlab code was used to generate the filter:

```
clear;

rp = 0.4;                       % passband ripple
rs = 50;                        % stopband ripple
f = [0.065 0.1125 0.5625 0.625]; % Normalised frequencies
a = [0 1 0];                    % amplitude
fs = 8000;                      % sampling frequency

% calculate deviation
dev = [10^(-rs/20) (10^(rp/20)-1)/(10^(rp/20)+1) 10^(-rs/20)];

% determine the order
[n,fo,ao,w] = firpmord(f,a,dev);

b = firpm(n+3, fo, ao, w);

% time to plot
figure

% linear gain plot
subplot(2,2,[1 3]);
% [h,f] = freqz(b,a,n,fs)
[h, omega] = freqz(b, 1, 2048, fs);
plot( fo.*(fs/2), ao, omega, abs(h));
legend('Ideal', 'Design');
grid minor;
xlabel('Frequency (Hz)');
ylabel('Gain');

% magnitude bode plot
subplot(2,2,2)
%semilogx (omega, mag2db(abs(h)));
plot (omega, mag2db(abs(h)));
xlim([10 fs/2]);
grid minor;
xlabel('Frequency (Hz)');
ylabel('Gain (dB)');

% phase bode plot
subplot(2,2,4)
%semilogx (omega, unwrap(angle(h)));
plot (omega, unwrap(angle(h)));
xlim([10 fs/2]);
grid minor;
xlabel('Frequency (Hz)');
ylabel('Phase (radians)');

% write to file
format long e
save ('fir_coef.txt', 'b', '-ascii', '-double', '-tabs');
save ('fir_coef_float.txt', 'b', '-ascii','-tabs');
```

A.2 Non-Circular Buffer

```
/*********************************************************************************
             DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                    IMPERIAL COLLEGE LONDON

              EE 3.19: Real Time Digital Signal Processing
                 Dr Paul Mitcheson and Daniel Harvey

                                LAB 4 - Non-circular FIR
 *********************************************************************************/

/*************************** Pre-processor statements ****************************/

#include <stdlib.h>
#include <stdio.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

/******************************* Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
       /*****************************************************************/
       /*   REGISTER              FUNCTION            SETTINGS          */
       /*****************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                 */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                 */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                 */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                 */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off      */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on      */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit               */\
    0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ               */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                  */\
       /*****************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;


```

```
/****************************** Filter Stuff ******************************/
// The order of the FIR filter +1
#define N 88

// include the coefficients
#include "fir_coef.txt"

// define the buffer
Int16 buffer[N] = {0};

 /****************************** Function prototypes ******************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
Int16 convoluteNonCircular(void);
/****************************** Main routine ******************************/
void main(){


  // initialize board and the audio port
  init_hardware();

  /* initialize hardware interrupts */
  init_HWI();

  /* loop indefinitely, waiting for interrupts */
  while(1)
  {};

}

/****************************** init_hardware() ******************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

  /* Function below sets the number of bits in word used by MSBSP (serial port) for
  receives from AIC23 (audio port). We are using a 32 bit packet containing two
  16 bit numbers hence 32BIT is set for  receive */
  MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

  /* Configures interrupt to activate on each consecutive available 32 bits
  from Audio port hence an interrupt is generated for each L & R sample pair */
  MCBSP_FSETS(SPCR1, RINTM, FRM);

  /* These commands do the same thing as above but applied to data transfers to
  the audio port */
  MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
  MCBSP_FSETS(SPCR1, XINTM, FRM);


}

```

```
/********************************* init_HWI() ***********************************/
void init_HWI(void)
{
  IRQ_globalDisable();       // Globally disables interrupts
  IRQ_nmiEnable();         // Enables the NMI interrupt (used by the debugger)
  IRQ_map(IRQ_EVT_RINT1,4);   // Maps an event to a physical interrupt
  IRQ_enable(IRQ_EVT_RINT1);     // Enables the event
  IRQ_globalEnable();        // Globally enables interrupts

}

/******************* WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/

void ISR_AIC(void)
{
  int i;
  double output = 0;

  // shift buffer
  for (i = N-1; i != 0; --i)
    buffer[i] = buffer[i-1];

  // new sample
  buffer[0] = mono_read_16Bit();

  // mac loop
  for (i = 0; i < N; ++i)
    output += b[i] * buffer[i];

    mono_write_16Bit(output); // write
}
```

A.3 Naive Implementation for a Circular Buffer

```
/*********************************************************************************
             DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                    IMPERIAL COLLEGE LONDON

              EE 3.19: Real Time Digital Signal Processing
                 Dr Paul Mitcheson and Daniel Harvey

                                LAB 4 - Naive Circular FIR
 *********************************************************************************/

/*************************** Pre-processor statements ****************************/

#include <stdlib.h>
#include <stdio.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

```

```
// math library (trig functions)
#include <math.h>

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

/******************************* Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
        /*************************************************************************/
        /*   REGISTER              FUNCTION            SETTINGS         */
        /*************************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                 */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                 */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                 */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                 */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off      */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on      */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit               */\
    0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ               */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                  */\
        /*************************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;


/******************************* Filter Stuff ********************************/
// The order of the FIR filter +1
#define N 88

// include the coefficients
#include "fir_coef.txt"

// define the buffer
Int16 buffer[N] = {0};

// index of the current "current" (zero) sample
int index = 0;

 /******************************* Function prototypes ********************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
Int16 convolute(Int16 input);
/********************************** Main routine ************************************/
void main(){
  // initialize board and the audio port
  init_hardware();

  /* initialize hardware interrupts */
  init_HWI();
```

```
81
82    /* loop indefinitely, waiting for interrupts */
83    while(1)
84    {};
85
86  }
87
88  /******************************** init_hardware() ********************************/
89  void init_hardware()
90  {
91      // Initialize the board support library, must be called first
92      DSK6713_init();
93
94      // Start the AIC23 codec using the settings defined above in config
95      H_Codec = DSK6713_AIC23_openCodec(0, &Config);
96
97    /* Function below sets the number of bits in word used by MSBSP (serial port) for
98    receives from AIC23 (audio port). We are using a 32 bit packet containing two
99    16 bit numbers hence 32BIT is set for  receive */
100   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
101
102   /* Configures interrupt to activate on each consecutive available 32 bits
103   from Audio port hence an interrupt is generated for each L & R sample pair */
104   MCBSP_FSETS(SPCR1, RINTM, FRM);
105
106   /* These commands do the same thing as above but applied to data transfers to
107   the audio port */
108   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
109   MCBSP_FSETS(SPCR1, XINTM, FRM);
110
111
112 }
113
114 /******************************** init_HWI() ************************************/
115 void init_HWI(void)
116 {
117   IRQ_globalDisable();       // Globally disables interrupts
118   IRQ_nmiEnable();         // Enables the NMI interrupt (used by the debugger)
119   IRQ_map(IRQ_EVT_RINT1,4);    // Maps an event to a physical interrupt
120   IRQ_enable(IRQ_EVT_RINT1);     // Enables the event
121   IRQ_globalEnable();        // Globally enables interrupts
122
123 }
124
125 /******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
126
127 void ISR_AIC(void){
128     Int16 sample = mono_read_16Bit(); // read
129     sample =  convolute(sample); // convolute
130     mono_write_16Bit(sample); // write
131 }
132
133 // Perform convolution
134 Int16 convolute(Int16 input){
135   int i;
136   double result = 0;
137   // write to current "zero" sample
```

```
138   *(buffer + index) = input;
139
140   for (i = 0; i < N; i++)
141     result += b[i]* buffer[ ((index-i) + N) % N];
142
143   // advance index
144   index = (index + 1)%N;
145
146   return (Int16) round(result);
147 }
```

A.4 Optimised Circular Buffer Implementation

```
1  /*************************************************************************
2               DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3                       IMPERIAL COLLEGE LONDON
4
5                EE 3.19: Real Time Digital Signal Processing
6                   Dr Paul Mitcheson and Daniel Harvey
7
8                                  LAB 4 - Circular FIR
9   *************************************************************************/
10
11 /*************************** Pre-processor statements ***************************/
12
13 #include <stdlib.h>
14 #include <stdio.h>
15 //  Included so program can make use of DSP/BIOS configuration tool.
16 #include "dsp_bios_cfg.h"
17
18 /* The file dsk6713.h must be included in every program that uses the BSL.  This
19    example also includes dsk6713_aic23.h because it uses the
20    AIC23 codec module (audio interface). */
21 #include "dsk6713.h"
22 #include "dsk6713_aic23.h"
23
24 // math library (trig functions)
25 #include <math.h>
26
27 // Some functions to help with writing/reading the audio ports when using interrupts.
28 #include <helper_functions_ISR.h>
29
30 /******************************* Global declarations ********************************/
31
32 /* Audio port configuration settings: these values set registers in the AIC23 audio
33    interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
34 DSK6713_AIC23_Config Config = { \
35       /*****************************************************************/
36       /*   REGISTER              FUNCTION            SETTINGS         */
37       /*****************************************************************/\
38     0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                */\
39     0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                */\
40     0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                */\
41     0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                */\
42     0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
43     0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off      */\
44     0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on      */\
```

```
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit                */\
    0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ                */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                   */\
       /************************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;


/******************************* Filter Stuff *******************************/
// The order of the FIR filter +1
#define N 88

// include the coefficients
#include "fir_coef.txt"

// define the buffer
double buffer[N] = {0};

// index of the current "current" (zero) sample
int index = 0;

// macro that based on the index of the current zero sample,
// calculate the index of the array to read
// including handling wrap arounds
//#define GET_INDEX(index, offset) (index + offset)%N

 /***************************** Function prototypes ****************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
/********************************** Main routine ******************************/
void main(){
  // initialize board and the audio port
  init_hardware();

  /* initialize hardware interrupts */
  init_HWI();

  /* loop indefinitely, waiting for interrupts */
  while(1)
  {};

}

/********************************* init_hardware() ****************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

  /* Function below sets the number of bits in word used by MSBSP (serial port) for
```

```
102   receives from AIC23 (audio port). We are using a 32 bit packet containing two
103   16 bit numbers hence 32BIT is set for  receive */
104   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
105
106   /* Configures interrupt to activate on each consecutive available 32 bits
107   from Audio port hence an interrupt is generated for each L & R sample pair */
108   MCBSP_FSETS(SPCR1, RINTM, FRM);
109
110   /* These commands do the same thing as above but applied to data transfers to
111   the audio port */
112   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
113   MCBSP_FSETS(SPCR1, XINTM, FRM);
114
115
116 }
117
118 /********************************* init_HWI() ************************************/
119 void init_HWI(void)
120 {
121   IRQ_globalDisable();       // Globally disables interrupts
122   IRQ_nmiEnable();         // Enables the NMI interrupt (used by the debugger)
123   IRQ_map(IRQ_EVT_RINT1,4);   // Maps an event to a physical interrupt
124   IRQ_enable(IRQ_EVT_RINT1);    // Enables the event
125   IRQ_globalEnable();        // Globally enables interrupts
126
127 }
128
129 /******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
130
131 void ISR_AIC(void)
132 {
133   // FIR filter
134   // operation principle: do a forward pass of the sample buffer until the end is hit,
135   // then wrap the sample pointer to the start of the buffer and do the remainder of
136   // iterations that can be computed from the amount of coefficients left to process (with pointer
        arithmetic).
137
138   double* coeffptr = b;
139   double* coeffEndptr = b + N; // points to the element AFTER the coefficient array
140   double* sampleptr = buffer + index; // point to oldest sample initially
141   double* bufferEndptr = buffer + N; // one after last element
142
143   int loopcnt = bufferEndptr - sampleptr; // how many iterations are needed for a single-step loop
144   char modunroll = loopcnt % 4; // non-integral leftover of an unrolled loop
145
146   // accumulators
147   double result = 0;
148   double result2 = 0;
149   double result3 = 0;
150   double result4 = 0;
151
152   *sampleptr = mono_read_16Bit(); // read sample into buffer
153
154   // process samples until the end of the sample buffer is hit
155   while(sampleptr < bufferEndptr-3)
156   {
157     result += (*coeffptr++) * (*sampleptr++);
```

```
158     result2 += (*coeffptr++) * (*sampleptr++);
159     result3 += (*coeffptr++) * (*sampleptr++);
160     result4 += (*coeffptr++) * (*sampleptr++);
161   }
162
163   // take care of non-integral leftover iterations
164   if (modunroll>0) result += (*coeffptr++) * (*sampleptr++);
165   if (modunroll>1) result2 += (*coeffptr++) * (*sampleptr++);
166   if (modunroll>2) result3 += (*coeffptr++) * (*sampleptr++);
167
168     sampleptr = buffer; // wrap pointer to beginning of the buffer
169
170     // pass the remainder of the buffer (amount of iterations = how many coefficients there are
          left to process)
171     while (coeffptr < coeffEndptr-3)
172     {
173       result += (*coeffptr++) * (*sampleptr++);
174     result2 += (*coeffptr++) * (*sampleptr++);
175     result3 += (*coeffptr++) * (*sampleptr++);
176     result4 += (*coeffptr++) * (*sampleptr++);
177     }
178
179     // take care of non-integral leftover iterations
180   if (modunroll==1) result += (*coeffptr++) * (*sampleptr++);
181   if (modunroll==1 || modunroll==2) result2 += (*coeffptr++) * (*sampleptr++);
182   if (modunroll==1 || modunroll==2 || modunroll==3) result3 += (*coeffptr++) * (*sampleptr++);
183
184     // sum the accumulators
185     result = result + result2 + result3 + result4;
186
187   // advance index into circular buffer
188   index = (index == 0) ? N-1 : index-1;
189
190   mono_write_16Bit(result); // output sample
191 }
```

A.5 Assembly Implementation

A.5.1 C File

```
1  /*************************************************************************
2               DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3                       IMPERIAL COLLEGE LONDON
4
5                EE 3.19: Real Time Digital Signal Processing
6                   Dr Paul Mitcheson and Daniel Harvey
7
8                                  LAB 4 - ASM FIR
9   *************************************************************************/
10
11 /************************** Pre-processor statements **************************/
12
13 #include <stdlib.h>
14 #include <stdio.h>
15 //  Included so program can make use of DSP/BIOS configuration tool.
16 #include "dsp_bios_cfg.h"
17
```

```
18 /* The file dsk6713.h must be included in every program that uses the BSL.  This
19    example also includes dsk6713_aic23.h because it uses the
20    AIC23 codec module (audio interface). */
21 #include "dsk6713.h"
22 #include "dsk6713_aic23.h"
23
24 // math library (trig functions)
25 #include <math.h>
26
27 // Some functions to help with writing/reading the audio ports when using interrupts.
28 #include <helper_functions_ISR.h>
29
30 /****************************** Global declarations ********************************/
31
32 /* Audio port configuration settings: these values set registers in the AIC23 audio
33    interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
34 DSK6713_AIC23_Config Config = { \
35        /*****************************************************************/
36        /*   REGISTER              FUNCTION            SETTINGS         */
37        /*****************************************************************/\
38     0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                */\
39     0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                */\
40     0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                */\
41     0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                */\
42     0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
43     0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off      */\
44     0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on      */\
45     0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit               */\
46     0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ               */\
47     0x0001   /* 9 DIGACT     Digital interface activation    On                  */\
48        /*****************************************************************/
49 };
50
51
52 // Codec handle:- a variable used to identify audio interface
53 DSK6713_AIC23_CodecHandle H_Codec;
54
55
56 /****************************** Filter Stuff ********************************/
57 // The order of the FIR filter + 1
58 #define N 88
59
60 // The size, in bytes, of the buffer
61 #define BUFFER_BYTE_SIZE 1024
62
63 // the buffer
64 double x_buffer[BUFFER_BYTE_SIZE/8] = {0};
65
66 // Byte align
67 #pragma DATA_ALIGN(x_buffer, BUFFER_BYTE_SIZE)
68
69 // pointer to first element
70 double *X_PTR = x_buffer;
71
72 // include the coefficients
73 #include "fir_coef.txt"
74
```

```
75 // index of the current "current" (zero) sample
76 int index = 0;
77
78 // Assembly circular FIR
79 extern void circ_FIR_DP(double **ptr, double *coef, double *input_samp, double *filtered_samp ,
       unsigned int numCoefs);
80
81  /****************************** Function prototypes ********************************/
82 void init_hardware(void);
83 void init_HWI(void);
84 void ISR_AIC(void);
85 /********************************* Main routine ***********************************/
86 void main(){
87   // initialize board and the audio port
88   init_hardware();
89
90   /* initialize hardware interrupts */
91   init_HWI();
92
93   /* loop indefinitely, waiting for interrupts */
94   while(1)
95   {};
96
97 }
98
99 /********************************* init_hardware() ********************************/
100 void init_hardware()
101 {
102     // Initialize the board support library, must be called first
103     DSK6713_init();
104
105     // Start the AIC23 codec using the settings defined above in config
106     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
107
108   /* Function below sets the number of bits in word used by MSBSP (serial port) for
109   receives from AIC23 (audio port). We are using a 32 bit packet containing two
110   16 bit numbers hence 32BIT is set for  receive */
111   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
112
113   /* Configures interrupt to activate on each consecutive available 32 bits
114   from Audio port hence an interrupt is generated for each L & R sample pair */
115   MCBSP_FSETS(SPCR1, RINTM, FRM);
116
117   /* These commands do the same thing as above but applied to data transfers to
118   the audio port */
119   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
120   MCBSP_FSETS(SPCR1, XINTM, FRM);
121
122
123 }
124
125 /********************************* init_HWI() *************************************/
126 void init_HWI(void)
127 {
128   IRQ_globalDisable();       // Globally disables interrupts
129   IRQ_nmiEnable();          // Enables the NMI interrupt (used by the debugger)
130   IRQ_map(IRQ_EVT_RINT1,4);    // Maps an event to a physical interrupt
```

```
131   IRQ_enable(IRQ_EVT_RINT1);     // Enables the event
132   IRQ_globalEnable();        // Globally enables interrupts
133
134 }
135
136 /******************* WRITE YOUR INTERRUPT SERVICE ROUTINE HERE**********************/
137
138 void ISR_AIC(void){
139   double sample = 0, output = 0;
140   sample = mono_read_16Bit(); // read
141   circ_FIR_DP(&X_PTR, b, &sample, &output, N);
142   mono_write_16Bit((Int16) output);
143 }
```

A.5.2 Linear Assembly Implementation

```
1  ; ************************************************************************************
2  ;             DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3  ;                        IMPERIAL COLLEGE LONDON
4  ;
5  ;                EE 3.19: Real Time Digital Signal Processing
6  ;                  Course  by: Dr Paul Mitcheson
7  ;
8  ;         LAB 4: Double precision FIR using Circular Buffer Hardware
9  ;
10 ;                     ********** circ_FIR_DP.ASM **********
11 ;
12 ; ************************************************************************************
13 ;                     Written by D. Harvey: 18 Jan 2010
14 ;
15 ; ************************************************************************************
16 ;
17         .global _circ_FIR_DP
18
19         .text
20 ;
21 ; ************************** _circ_FIR_DP description ******************************
22 ;
23 ;   The input delay buffer has a data length of (size in bytes)/(data type length).
24 ;       The buffer you create must have a power of 2 size in bytes
25 ; i.e its length in bytes must equal 2^X bytes (where X is integer between 1 and 32).
26 ;
27 ;       Also ensure that its data length (size in bytes/8) is longer than the
28 ;     coefficient array data length. The buffer will need to be data aligned
29 ;     using  #pragma DATA_ALIGN(delay_buff_name, B) before it is defined
30 ;          where B is your chosen delay buffer size in bytes.
31 ;
32 ; circ_FIR_DP function call in C;
33 ;
34 ; circ_FIR_DP( &circ_ptr, &coef[0], &read_samp, &filtered_samp, N );
35 ;
36 ; ************************** Register Assignments ***********************************
37
38 ; A0 LSB Multiplication  result     B0 Loop Counter
39 ; A1 MSB "        "       B1
40 ; A2                  B2 Used to set AMR to circular mode
41 ; A3                  B3 Return to C Address
```

```
; A4 &circ_ptr              B4 &coef[k]
; A5 circ_ptr             B5
; A6 &read_samp             B6 &filtered_samp
; A7                  B7
; A8 Number of Coefs (N)        B8
; A9                  B9
; A10 LSB delay_circ[j]         B10 LSB coef[k]
; A11 MSB "             B11 MSB  "
; A12                   B12
; A13                   B13 Temp Store for previous AMR register value
; A14 MSB Accumulator           B14
; A15 LSB   "             B15
;  See Real Time Digital Signal Processing by Nasser Kehtarnavaz (page 146) for more
;  info on mixing C and Assembly.
; ******************************************************************************

_circ_FIR_DP:
    ; set circular mode using the AMR

    MVC .S2     AMR,B13    ;(0) Save contents of AMR reg to B13
    MVK .S2     4H,B2      ;(0) Lower half. set A5 to be circular buffering addressing mode using
        BK0
    MVKLH .S2   9H,B2      ;(0) Upper half. Set BK0 to work for 1024 bytes
    MVC .S2     B2,AMR     ;(0) set AMR reg

    ; get the data passed from C

    LDDW .D1    *A6,A11:A10 ;(4) Get the 64 bit data for read_samp put it in A11:A10
    LDW .D1     *A4,A5     ;(4) Get the address of the circ_ptr, dereference then place in A5
    NOP 4           ; A5 now holds address pointing into delay_circ

    STW .D1     A11,*--A5 ;(0) Store new input sample (MSB) to delay_circ array
  || ZERO .S1     A14      ;(0) zero accumulator LSB
    STW .D1     A10,*--A5   ;(0) Store new input sample (LSB) to delay_circ array
  || ZERO .S1     A15      ;(0) zero accumulator MSB


    STW .D1     A5,*A4     ;(0) write back the decremented pointer to circ_ptr
                  ; this points to the end of the MSB of where the next sample
                  ; will be stored on the next call to this function

    || MV .S2X      A8, B0       ;(0) move parameter (numCoefs) passed from C into b0

    ;******************************** loop begin *********************************

loop:

    ; ************************ INSERT YOUR MAC CODE HERE ***************************
    LDDW .D1    *A5++, A11:A10 ; (4) loads the (delayed) sample into A11:A10, and post increment
        pointer
  || LDDW .D2     *B4++, B11:B10 ; (4) load the coefficient into B11:B10, and post increment
      pointer
    NOP 4
    MPYDP .M1X     A11:A10, B11:B10, A11:A10 ; (9, 4) DP multiply
    NOP 9
    ADDDP .L1   A15:A14, A11:A10, A15:A14 ; (6, 2) DP ADD
    NOP 6
```

```
96
97
98
99      ; MAC must use 64 bit IEEE double floating point data obtained from arrays defined in C
100
101
102
103     ; ************************************************************************
104
105     ; manage loop
106
107         SUB .D2        B0,1,B0      ; (0) b0 - 1 -> b0
108     [B0] B.S2       loop       ; (5) loop back if b0 is not zero
109         NOP        5
110
111     ;******************************** loop end ********************************
112
113     ; send the result of MAC back to C
114
115     STW .D2      A14,*B6    ;(0) Write accumulator (LSB) into filtered_samp
116     STW .D2      A15,*+B6[1] ;(0) Write accumulator (MSB) into filtered_samp
117
118     ; restore previous buffering mode
119
120   || MVC .S2      B13,AMR    ;(0) restore  AMR reg to previous contents
121
122     ; return to C code
123
124 lend:   B .S2        B3       ; (5) branch to b3 (register b3 holds the return address)
125         NOP        5
126
127         .end
```

A.5.3 Optimised Assembly Implementation

```
1   ; ************************************************************************************
2   ;                    DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3   ;                                 IMPERIAL COLLEGE LONDON
4   ;
5   ;                        EE 3.19: Real Time Digital Signal Processing
6   ;                              Course  by: Dr Paul Mitcheson
7   ;
8   ;                  LAB 4: Double precision FIR using Circular Buffer Hardware
9   ;
10  ;                             ********** circ_FIR_DP.ASM **********
11  ;
12  ; ************************************************************************************
13  ;                               Written by D. Harvey: 18 Jan 2010
14  ;
15  ; ************************************************************************************
16  ;
17          .global _circ_FIR_DP
18
19          .text
20  ;
21  ; ************************** _circ_FIR_DP description ******************************
22  ;
```

```
23 ;        The input delay buffer has a data length of (size in bytes)/(data type length).
24 ;               The buffer you create must have a power of 2 size in bytes
25 ;    i.e its length in bytes must equal 2^X bytes (where X is integer between 1 and 32).
26 ;
27 ;             Also ensure that its data length (size in bytes/8) is longer than the
28 ;            coefficient array data length. The buffer will need to be data aligned
29 ;            using  #pragma DATA_ALIGN(delay_buff_name, B) before it is defined
30 ;                     where B is your chosen delay buffer size in bytes.
31 ;
32 ; circ_FIR_DP function call in C;
33 ;
34 ; circ_FIR_DP( &circ_ptr, &coef[0], &read_samp, &filtered_samp, N );
35 ;
36 ; ***************************** Register Assignments ***********************************
37
38 ; A0 LSB Accumulator A                    B0 Loop Counter
39 ; A1 MSB     "                           B1
40 ; A2 LSB Multiplied result    A           B2 Used to set AMR to circular mode - then reused LSB
     Multiplied Result B
41 ; A3 MSB    "        "                   B3 Return to C Address (original)   - then reused MSB "
42 ; A4 &circ_ptr    - possible reuse        B4 &coef[k] - don't use for calc
43 ; A5 circ_ptr    - don't use for calc     B5
44 ; A6 &read_samp    - possible reuse       B6 &filtered_samp - (original) - then reused LSB
     Accumulator B
45 ; A7                                     B7                                          MSB "
46 ; A8 N, then LSB delay_circ[j] A          B8 LSB coef[k] A
47 ; A9         MSB "                       B9 MSB    "
48 ; A10 LSB delay_circ[j]    B             B10 LSB coef[k] B
49 ; A11 MSB    "                           B11 MSB  "
50 ; A12                                    B12
51 ; A13                                    B13
52 ; A14                                    B14 Data pointer (DO NOT USE)
53 ; A15                                    B15 Stack Pointer (DO NOT USE)
54
55 ; ***************************************************************************************
56
57 _circ_FIR_DP:
58         MVC.S2          AMR,B13         ;(0) Save contents of AMR reg to B13
59     ||  STW .D2         B3, *++B15      ; (0) save return to C to stack
60     ||  LDDW.D1         *A6,A11:A10     ;(4) Get the 32 bit data for read_samp put it in A11:A10
61
62         STW .D2         B6, *++B15      ; (0) save &filtered_samp to stack
63     ||  MVK .S2         4H,B2 ;(0)Set AMR to allow A5 to be used for circular addressing with
         BK0
64     ||  LDW .D1         *A4,A5          ;(4) Get the address of the circ_ptr, dereference then
         place in A5
65         MVKLH .S2       9H,B2 ;(0)Set BK0 to allow for 1024 bytes addressing
66         MVC.S2          B2,AMR          ;(0) set AMR reg
67
68         NOP 2                           ; A5 now holds address pointing into delay_circ
69
70         STW .D1         A11,*--A5       ;(0) Store new input sample (MSB) to delay_circ array
71     ||  ZERO .S1        A1              ;(0) zero accumulator LSB
72     ||  ZERO .S2        B3
73
74         STW .D1         A10,*--A5       ;(0) Store new input sample (LSB) to delay_circ array
75     ||  ZERO.S1         A0              ;(0) zero accumulator MSB
```

```
76      ||  ZERO .S2          B2
77
78          STW.D1            A5,*A4    ;(0) write back the decremented pointer to circ_ptr
79                                      ; this points to the end of the MSB of where the next sample
80                                      ; will be stored on the next call to this function
81
82      ||  MV .L2X           A8, B0      ;(0) move parameter (numCoefs) passed from C into b0
83      ||  MVK .S2           10, B1        ;(0) setup countdown to start addition
84
85
86          ADD               .L2 B0, B1, B0 ;(0) Branch needs 10/2=5 iterations to setup and running.
87      ||  B .S2             loop    ;(0) Loop is only 4 cycles,
88                                      ; so we need to kickstart the branch back for loop iteration 1
89
90      NOP                         ; NOP to allow the branch for Loop iteration 1 to happen right at
            the end
91
92          ;********************************* loop begin *********************************
93
94  loop:
95          [B0] SUB .S2      B0,2,B0  ;(0) Decrement loop counter by 2, because we are doing two
                calculations togerher
96
97
98          [B1] SUB .D2      B1,2,B1  ; (0) countdown to allow start of addition. Countdown is done
                by two
99                                      ; because the loop counter is decremented by two. And since we
                                            added B0
100                                     ; and B1 together before the loop, we must also double B1's
                                            value and subtract
101                                     ; by 2 each time
102
103         [B0] B .S2        loop          ;(5) for current iteration i, kickstart the branch back
                for iteration i+1
104     ||  [B0] LDDW .D1     *A5++, A11:A10 ; (4) B - Load delayed sample
105     ||  [B0] LDDW .D2     *B4++, B11:B10 ; (4) B - Load coefficient
106     ||  [B0] MPYDP .M2X   B11:B10, A11:A10, B3:B2 ; (9,4) B - Multiply
107     || [!B1] ADDDP .L2    B7:B6, B3:B2, B7:B6 ; (6,2) B - Accumulate
108
109
110         [B0] LDDW .D1     *A5++, A9:A8 ; (4) A - Load delayed sample
111     ||  [B0] LDDW .D2     *B4++, B9:B8 ; (4) A - Load coefficient
112     ||  [B0] MPYDP .M1X   A9:A8, B9:B8, A3:A2 ; (9,4) A - Multiply
113     || [!B1] ADDDP .L1    A1:A0, A3:A2, A1:A0 ; (6,2) A - Accumulate
114     || [!B0] MV .S2       B6, B12 ; (0) for the final iteration this cycle, the LH result for B-
            Addition-44 is
115                                     ; written on this cycle. We move the LH result for B-Addition-43
                                            out of the
116                                     ; way to prevent losing them
117
118
119         ;********************************* loop end *********************************
120         MV .D1            A0, A12 ; (0) the UH result for A-Addition-44 is
121                                     ; written on this cycle. We move the UH result for A-Addition-43
                                            out of the
122                                     ; way to prevent losing them
123     ||  MV .S2            B7, B13 ; (0) the UH result for B-Addition-44 is
```

```
124                                     ; written on this cycle. We move the UH result for B-Addition-43
                                            out of the
125                                     ; way to prevent losing them
126
127         MV .D1          A1, A13 ; (0) the LH result for A-Addition-44 is
128                                     ; written on this cycle. We move the LH result for A-Addition-43
                                            out of the
129                                     ; way to prevent losing them
130     ||  ADDDP .L2       B7:B6, B13:B12, B13:B12  ; (6,2) the supurious B-Addition-45 will write
            the LH result in
131                                                 ; 2 cycles after this. Better start adding
                                                        result of B-Addition-44
132
133         ADDDP .L1       A1:A0, A13:A12, A13:A12  ; (6,2) the supurious A-Addition-45 will write
                the LH result in
134                                                 ; 2 cycles after this. Better start adding
                                                        result of A-Addition-44
135
136         LDW .D2         *B15--, B6    ; (4) get &filtered_samp from stack
137         LDW .D2         *B15--, B0    ; (4) get return to C from stack
138
139         NOP 3
140         ADDDP .L1X      A13:A12, B13:B12, A13:A12 ; (6,2) Add the results of Side A and B
                together
141
142         NOP
143
144         ; return to C code
145
146 lend:   B.S2            B0            ; (5) branch to b3 (register b3 holds the return address)
147         NOP     3
148
149         ; send the result of MAC back to C
150         STW.D2          A12,*B6        ;(0) Write accumulator (LSB) into filtered_samp
151
152         STW.D2          A13,*+B6[1]    ;(0) Write accumulator (MSB) into filtered_samp
153     ||  MVC.S2          B1,AMR         ;(0) restore  AMR reg to previous contents
154
155         .end
```