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 In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others.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 band used in this lab is between 260 Hz and 450 Hz, and between 2250 Hz and 2500 Hz. The Matlab code used to generate the listing is given in section A.1.

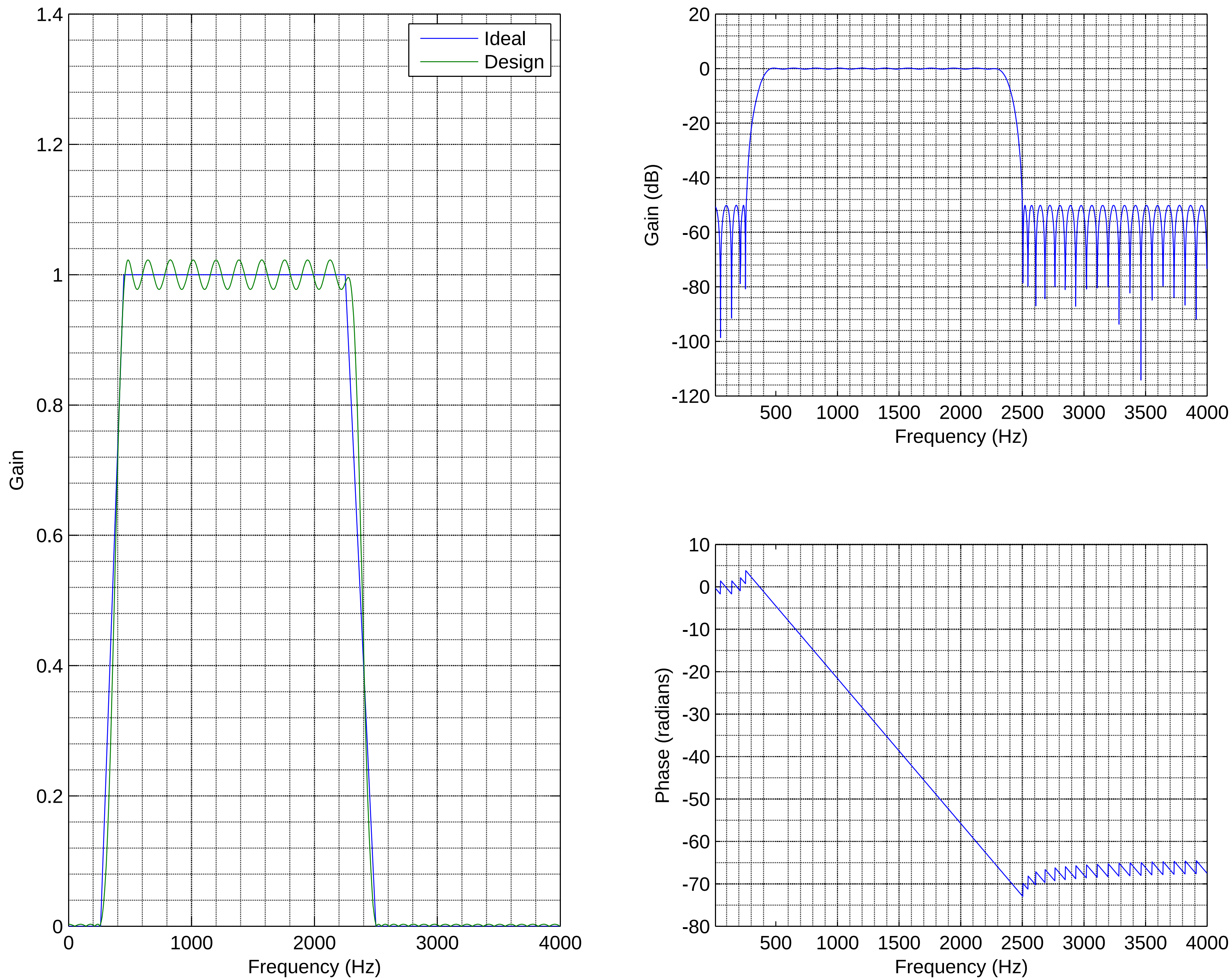
1.1 Coefficients

The coefficients generated by the Order 87 filter (with 88 coefficients) used is given by:

-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	1.1791976942964414e-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
1.1791976942964414e-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.


Ideal
Design
Gain
Frequency (Hz)
Gain (dB)
Frequency (Hz)
Phase (radians)
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 is 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. At the start of every ISR, the sample is first read from the input port.

```
Int16 sample = mono_read_16Bit(); // read
```

The buffer is then updated as though it's a shift register.

```
// Handle the buffer
for (i = N-1; i > 0; i--)
  buffer[i] = buffer[i-1];
buffer[0] = sample;
```

Finally, the convolution is done by a call to the `convoluteNonCircular` function and the output is written to. The convolution is done simply according to the following equation

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

and is implemented in code as below:

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

2.2 Oscilloscope Traces

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

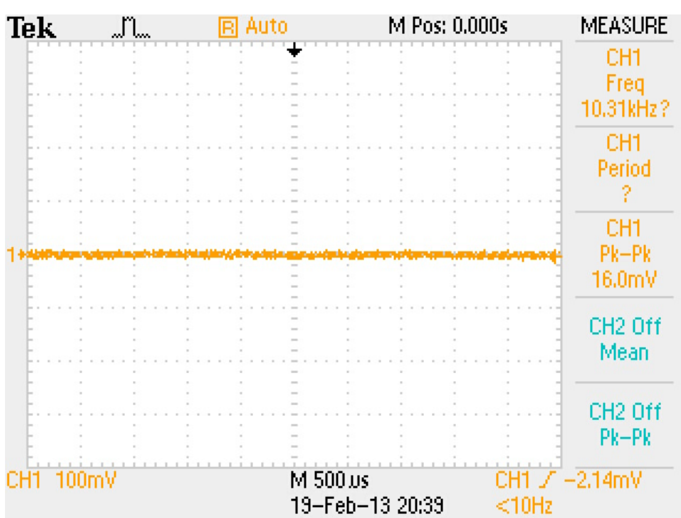


Figure 2.1: 200 Hz input, with almost zero output. This is in the stop-band.

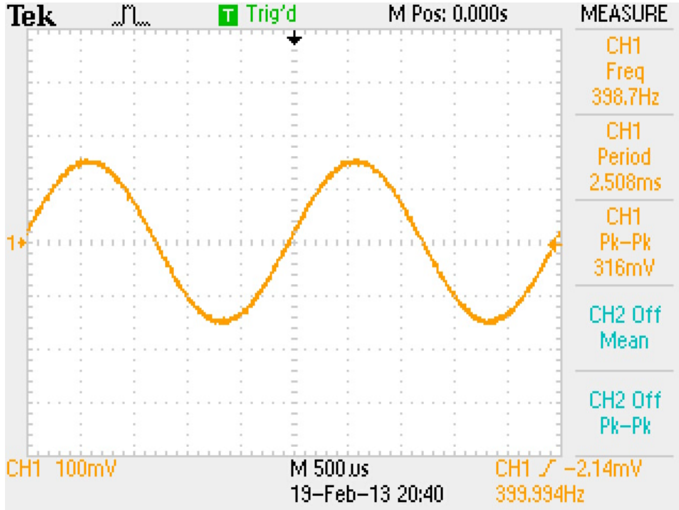


Figure 2.2: 400 Hz input, with increasing output amplitude. This is in the first transition band.

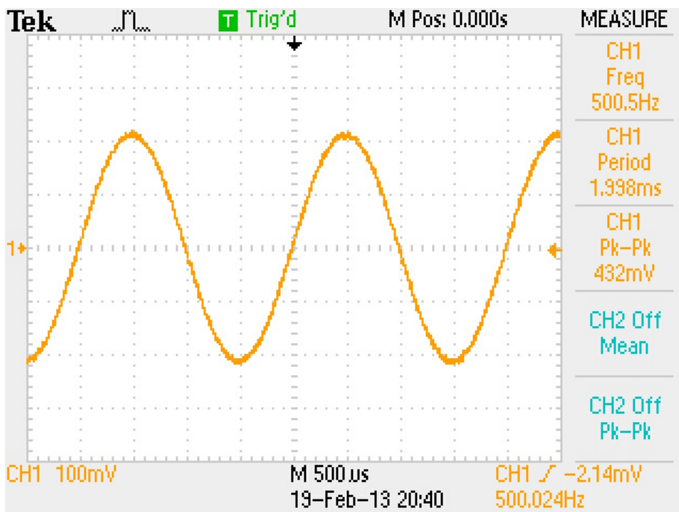


Figure 2.3: 500 Hz input, with maximum output amplitude. This is within the passband.

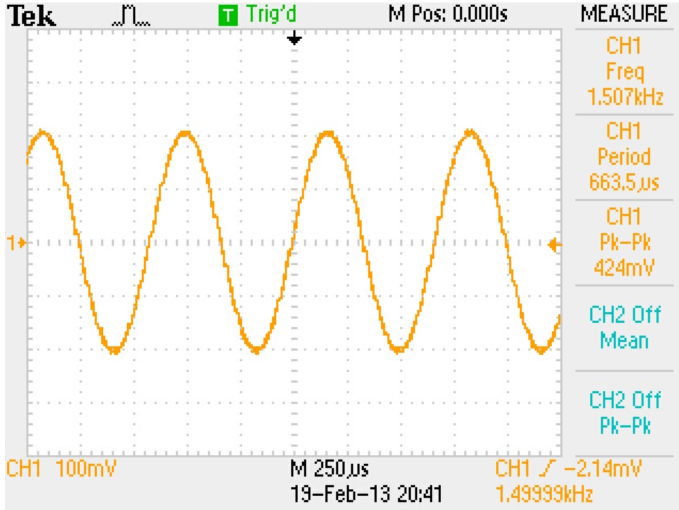


Figure 2.4: 1500 Hz input, with maximum amplitude. This is within the passband.

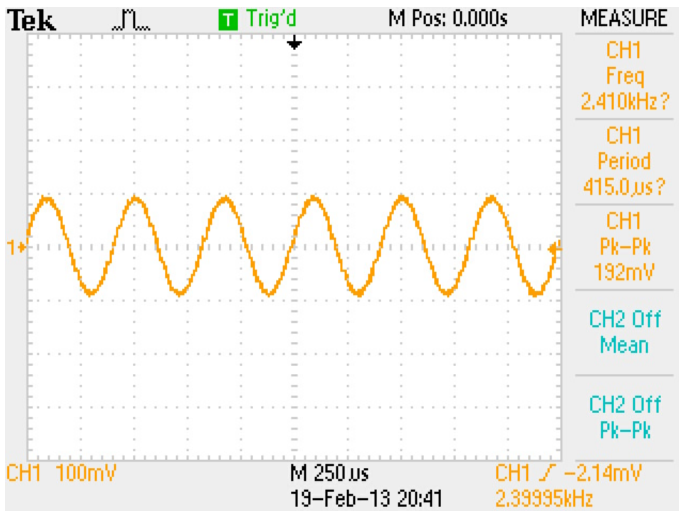


Figure 2.5: 2400 Hz, with decreasing amplitude. This is within the second transition band.

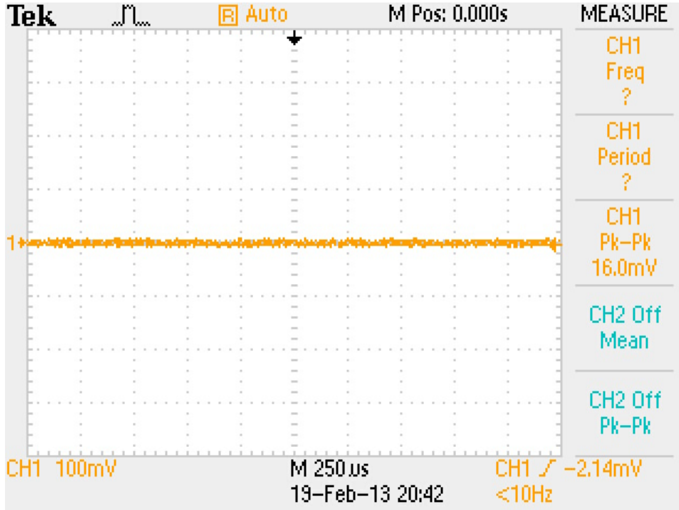


Figure 2.6: 3000 Hz input, with zero output. This is within the second stop-band.

2.3 Code Performance

The number of cycles 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.

Optimisation Level	Number of Clock Cycles
None	5825
Level 0	4829
Level 2	1719

The time taken for the functions `mono_read_16Bit()`, and `mono_write_16Bit()` were also recorded, and is shown in table 2.1. These number of cycles do not vary with the type of buffer used as the functions are independent of the buffer used.

Optimisation Level	`mono_read_16Bit()`	`mono_write_16Bit()`
None	110	67
Level 0	109	53
Level 2	77	48

Table 2.1: The number of cycles for the functions `mono_read_16Bit()`, and `mono_write_16Bit()`

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. The number given is the lowest number of clock cycles observed. The number might vary due to cache hits and/or misses. In general, this implementation of the buffer performed worse than the Non-Circular buffer version described in section §2. This is because the modulus operator is an expensive operation, and will be optimised in the next section.

Optimisation Level	Number of Clock Cycles
None	7377
Level 0	5830
Level 2	3934

3.2 Optimised Implementation

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

3.2.1 Code Operation

The code for the convolution was moved into the ISR routine, yielding negligible performance gains, but allowing for more optimisation to take place.

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 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 $(i+n)\%N$ where $N=88$ is the total number of coefficients. The variables are declared in the same way as in section 3.1.1.

Pointers are used to point to the appropriate values in the arrays for use during the MAC loop. This improves the performance of the code slightly, as offset addresses do not have to be calculated at the point of pointer dereference. Thus, at the start of the ISR routine, the following pointers are set up. `i` is a pointer to the first element in the coefficients and `bEnd` is the pointer to one element after the end of the coefficient array. `offset` is a pointer to the current entry in the `buffer` array to be written to, and `bufferEnd` is a pointer to one element after the end of the `buffer` array.

```
double *i = b;
double *bEnd = b + N; // one after last element
double *offset = buffer + index;
double *bufferEnd = buffer + N; // one after last element
```

The result is then read and written to the buffer.

```
*offset = mono_read_16Bit();  // read and write to current "zero" sample
```

The MAC loop then takes place.

Another way to implement circular buffering is to use `if/else` tests to see if the index of the array is below zero, or after the last element. However, these tests are expensive. It can be noted that the `for` loop already does its own tests for an index during an iteration. The check necessary for circular buffering can thus be integrated with the index check of a `for` loop. To achieve circular buffering without `if/else` tests, two separate `for` loops are implemented.

The first `for` loop will perform the MAC for all entries in the `buffer` between the current entry up to, and including, the last entry in the `buffer` (since the older entries have indices larger than the current entry).

```
for (; offset < bufferEnd; ++i, ++offset)
  result += (*i) * (*offset);
```

The `offset` variable is then reset to the beginning of the `buffer`, and is looped until the necessary number of coefficients have been multiplied (with the check performed against `bEnd`).

```
 for (offset = buffer; i < bEnd; ++i, ++offset)
        result += (*i) * (*offset);
```

The index is then decremented using an `if/else` check.

```
index = (index == 0) ? N-1 : index-1;
```

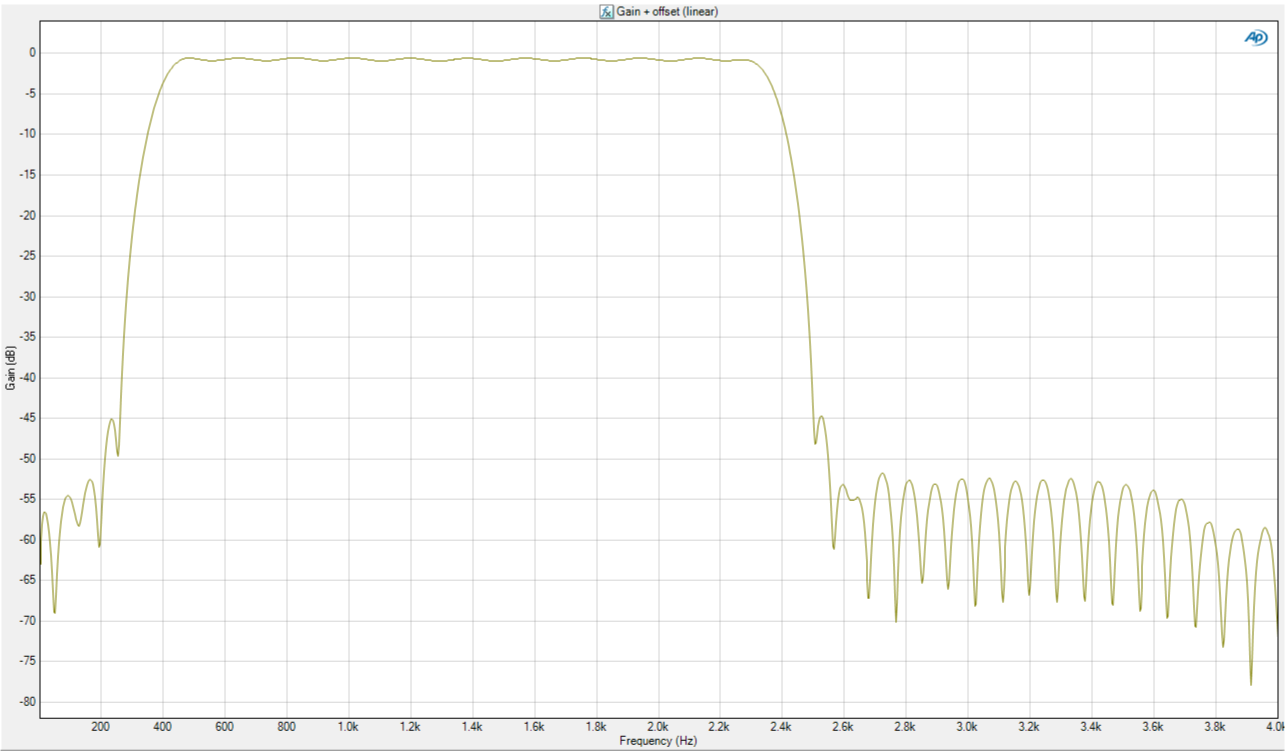
3.2.2 Code Performance

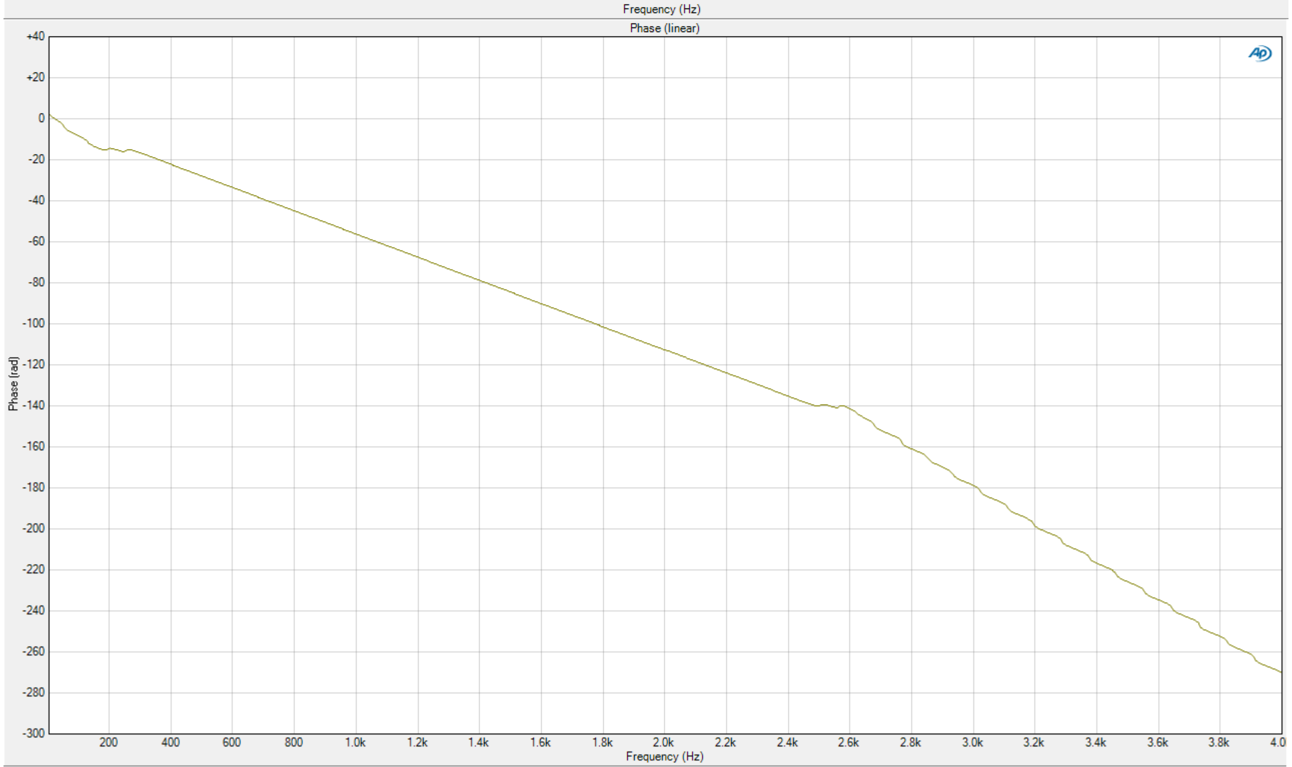
The number of cycles taken between the start, and the end of the ISR routine is given in the table below. The number given is the lowest number of clock cycles observed. The number might vary due to cache hits and/or misses. In general, this implementation gives massive improvement in terms of performance.

Optimisation Level	Number of Clock Cycles
None	4526
Level 0	2898
Level 2	746

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 also roughly linear during in the passband.





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:

```
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
```

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 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 (including the five NOPs 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
Level 0	2736	
Level 2	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 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.

```
    ; 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
```

The sample that is just read is then loaded by dereferencing its address pointer, along with the address of the circular buffer by dereferencing its address pointer. At the same time, some registers are moved out of the way to prepare for the MAC loop. The register usage is described in the comments in the listing in section A.5.3.

```
        LDDW .D1        *A6,A9:A8     ;(4) Get the 64 bit data for read_samp put it in A9:A8
    ||  MV .S2X         A8, B0       ;(0) move parameter (numCoefs) passed from C into b0
        LDW .D1         *A4,A5         ;(4) Get the address of the circ_ptr, dereference then place in
    ||  MV .S2          B3, B1         ;(0) move return to C address
        MV .S2          B6, B5         ;(0) move &filtered_samp
        NOP 3                        ; A5 now holds address pointing into delay_circ
```

The sample is then stored into the buffer, and the registers used as the accumulators for the MAC loop is first zeroed.

```
        STW .D1         A9,*--A5     ;(0) Store new input sample (MSB) to delay_circ array
    ||  ZERO .S1        A0             ;(0) zero accumulator LSB
    ||  ZERO .S2        B2
        STW .D1         A8,*--A5      ;(0) Store new input sample (LSB) to delay_circ array
    ||  ZERO .S1        A1             ;(0) zero accumulator MSB
    ||  ZERO .S2        B3
```

Finally, the address for the next sample read from the input to is written back to the memory location pointed by `A4`.

```
        STW .D1         A5,*A4        ;(0) write back the decremented pointer to circ_ptr
```

The loop prologue, described in as part of the software pipeline is then primed to load the first two sets of sample values and coefficients into the respective registers so as to perform two MAC operations in parallel to separate accumulators. The `LDDW` instructions for each pair of values are loaded in the same execute packet, and the next pair of values are loaded in the subsequent execute packet. The instruction requires four delays slots before the registers are fully written.

```
        ;********************************** loop prologue **********************************
        ; prime the pipeline
       LDDW .D1          *A5++, A9:A8 ; (4) loads the (delayed) sample into A9:A8, and post increment pointer
    || LDDW .D2          *B4++, B9:B8 ; (4) load the coefficient into B9:B8, and post increment pointer

       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
```

The loop kernel is then executed next. The loop kernel will be executed N number of times, where N is the number of coefficients provided by the caller in C. In one execute packet, the kernel first attempts to multiply the two pairs of values loaded in the previous loop cycle (or in the loop prologue). It also performs a decrement of the loop counter by two. The `MPYDP` requires 8 delay slots (one less than 9, as described in section 4.2.1).

```
        ; ************************* loop kernel      ****************************
       MPYDP .M1X        A9:A8, B9:B8, A3:A2     ; (9, 4) DP multiply
```

```
    ||  MPYDP .M2X      B11:B10, A11:A10, B7:B6    ; (9, 4) DP multiply
    ||  SUB .S2         B0,2,B0                 ; (0) b0 - 2 -> b0
```

After three execute packets with NOPs, there are exactly five execute packets in the kernel, and thus the conditional branch instruction is executed.

```
        NOP 3
        [B0] B .S2          loop            ; (5) loop back if b0 is not zero
```

The next two pairs of values are then loaded. These LDDW instructions can actually be scheduled in the execute packet right after the MPYDP instruction as they write to the registers only on the fourth delay slot, but doing so would not change the number of cycles as the MPYDP still needs 8 delay slots. The delay slot requirements of these LDDW instructions will be met by the time the values are used again in the next loop's MPYDP instructions.

```
    ||  [B0] LDDW .D1     *A5++, A9:A8 ; (4) loads the (delayed) sample into A9:A8, and post
        increment pointer
    ||  [B0] LDDW .D2     *B4++, B9:B8 ; (4) load the coefficient into B9:B8, and post increment
        pointer

        [B0] LDDW .D1     *A5++, A11:A10 ; (4) loads the (delayed) sample into A11:A10, and post
            increment pointer
    ||  [B0] LDDW .D2     *B4++, B11:B10 ; (4) load the coefficient into B11:B10, and post
        increment pointer
```

Three more NOPs are then performed to meet MPYDP's delay slots requirement, and then the various accumulators are added.

```
        ADDDP .L1       A1:A0, A3:A2, A1:A0     ; (6, 2) DP ADD
    ||  ADDDP .L2       B3:B2, B7:B6, B3:B2     ; (6, 2) DP ADD
```

The epilogue of the loop is now performed. Five delay slots (one less than required, with similar reasons as MPYDP) are inserted for the final ADDDP instruction to complete execution before adding the two accumulators up. Five delay slots are needed for this ADDDP to complete. This is, again, one less than required because the lower half of the result will be used first later on.

```
        ;********************************** loop epilogue **********************************
        ; add both accumulators up
        NOP 5           ; for the final addition to be complete
        ADDDP .L1X      A1:A0, B3:B2, A1:A0     ; (6, 2) DP ADD
```

Taking advantage of the five delay slots after a branch, the branch back to C instruction is executed while the results of the MAC is written back to C and the previous AMR register value is restored.

```
        NOP
        ; return to C code
lend:   B .S2           B1              ; (5) branch to b1 (moved C return address)
        NOP 3                   ; send the result of MAC back to C
        STW .D2         A0,*B5          ;(0) Write accumulator (LSB) into filtered_samp
        STW .D2         A1,*+B5[1]      ;(0) Write accumulator (MSB) into filtered_samp
        ; restore previous buffering mode
    ||  MVC .S2         B13,AMR         ;(0) restore AMR reg to previous contents
```

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

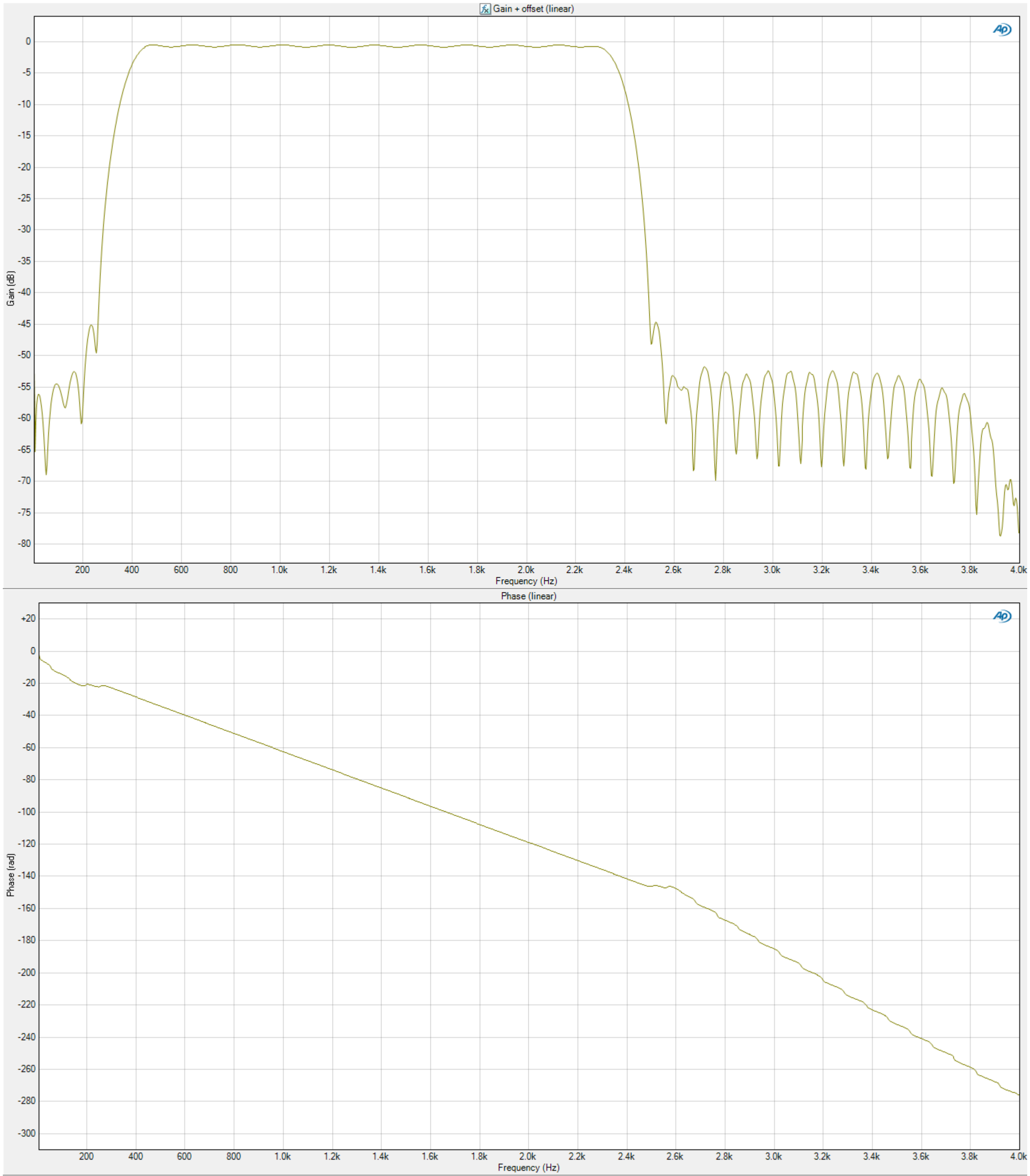
4.2.3 Code Performance

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	633	494
Level 0	633	
Level 2	648	

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.



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. The various clock cycles required for the different levels of optimisations from previous sections are copied in table 5.1, for comparison.

[1]http://www.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=spru187o&fileType=pdf

Optimisation Level	Non-circular Buffer	Optimised Circular Buffer
None	5825	4526
Level 0	4829	2898
Level 2	1719	746

Table 5.1: Comparison of the performance of the code at various compiler optimisation levels.

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 seen in table 5.1.

5.1 Level 0 Optimisation

As can be seen from table 5.1, level 0 results in some improvements in code performance (although not as drastic as level 2), with the non-circular and circular buffer achieving 17.1% and 36.0% improvement respectively.

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 *bEnd = b + N;
double *bufferEnd = buffer + N
```

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 `NOPs` being inserted, which are wasteful if not optimised properly).

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

// transformed into
i = 0;
```

[2]See http://llvm.org/devmtg/2009-10/ScalarEvolutionAndLoopOptimization.pdf and http://en.wikipedia.org/wiki/Loop_inversion

```
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.

```
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. As seen from table 5.1, the non-circular and circular buffer implementation saw a 64.4% and 74.3% improvement in performance when compared to Level 0 optimisation.

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

```c
/*************************************************************************************
              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;
    Int16 output;
    Int16 sample = mono_read_16Bit(); // read

    // Handle the buffer
    for (i = N-1; i > 0; i--)
      buffer[i] = buffer[i-1];

    buffer[0] = sample;
    output = convoluteNonCircular();
    mono_write_16Bit(output); // write
}

// Perform convolution
Int16 convoluteNonCircular(void){
  double output = 0;
  int i;

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

  return (Int16) round(output);
}
```

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();

  /* 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){
    Int16 sample = mono_read_16Bit(); // read
    sample =  convolute(sample); // convolute
    mono_write_16Bit(sample); // write
}
```

```

// Perform convolution
Int16 convolute(Int16 input){
  int i;
  double result = 0;
  // write to current "zero" sample
  *(buffer + index) = input;

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

  // advance index
  index = (index + 1)%N;

  return (Int16) round(result);
}
```

A.4 Optimised Circular Buffer Implementation

```
/*************************************************************************************
             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 - 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
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
  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){
  double *i = b;
  double *bEnd = b + N; // one after last element
  double *offset = buffer + index;
  double *bufferEnd = buffer + N; // one after last element

  double result = 0;
  *offset = mono_read_16Bit();  // read and write to current "zero" sample

  for (; offset < bufferEnd; ++i, ++offset)
    result += (*i) * (*offset);


    for (offset = buffer; i < bEnd; ++i, ++offset)
        result += (*i) * (*offset);

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

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

A.5 Assembly Implementation

A.5.1 C File

```
/*************************************************************************************
             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 - ASM 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

// The size, in bytes, of the buffer
#define BUFFER_BYTE_SIZE 1024

// the buffer
double x_buffer[BUFFER_BYTE_SIZE/8] = {0};

// Byte align
#pragma DATA_ALIGN(x_buffer, BUFFER_BYTE_SIZE)

// pointer to first element
double *X_PTR = x_buffer;

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

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

// Assembly circular FIR
extern void circ_FIR_DP(double **ptr, double *coef, double *input_samp, double *filtered_samp ,
    unsigned int numCoefs);

 /******************************* 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
  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){
  double sample = 0, output = 0;
  sample = mono_read_16Bit(); // read
  circ_FIR_DP(&X_PTR, b, &sample, &output, N);
  mono_write_16Bit((Int16) output);
}
```

A.5.2 Linear Assembly Implementation

```
; ***********************************************************************************
;              DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
;                           IMPERIAL COLLEGE LONDON
;
;                  EE 3.19: Real Time Digital Signal Processing
;                     Course  by: Dr Paul Mitcheson
;
;         LAB 4: Double precision FIR using Circular Buffer Hardware
;
;                        *********** circ_FIR_DP.ASM ***********
;
; ***********************************************************************************
;                        Written by D. Harvey: 18 Jan 2010
;
; ***********************************************************************************
;
        .global _circ_FIR_DP

        .text
;
```

```
; ***************************** _circ_FIR_DP description ********************************
;
;   The input delay buffer has a data length of (size in bytes)/(data type length).
;       The buffer you create must have a power of 2 size in bytes
; i.e its length in bytes must equal 2^X bytes (where X is integer between 1 and 32).
;
;       Also ensure that its data length (size in bytes/8) is longer than the
;     coefficient array data length. The buffer will need to be data aligned
;     using  #pragma DATA_ALIGN(delay_buff_name, B) before it is defined
;          where B is your chosen delay buffer size in bytes.
;
; circ_FIR_DP function call in C;
;
; circ_FIR_DP( &circ_ptr, &coef[0], &read_samp, &filtered_samp, N );
;
; ***************************** Register Assignments *************************************

; A0 LSB Multiplication  result     B0 Loop Counter
; A1 MSB "         "        B1
; A2                    B2 Used to set AMR to circular mode
; 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



    ; MAC must use 64 bit IEEE double floating point data obtained from arrays defined in C



    ; ********************************************************************************

    ; manage loop

        SUB .D2        B0,1,B0      ; (0) b0 - 1 -> b0
   [B0] B.S2       loop       ; (5) loop back if b0 is not zero
        NOP        5

    ;********************************** loop end **********************************

    ; send the result of MAC back to C

    STW .D2      A14,*B6    ;(0) Write accumulator (LSB) into filtered_samp
    STW .D2      A15,*+B6[1] ;(0) Write accumulator (MSB) into filtered_samp

    ; restore previous buffering mode

  ||  MVC .S2      B13,AMR    ;(0) restore  AMR reg to previous contents

    ; return to C code

lend:   B .S2        B3       ; (5) branch to b3 (register b3 holds the return address)
        NOP        5

        .end
```

A.5.3 Optimised Assembly Implementation

```
; ********************************************************************************
```

```
;                       DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
;                                       IMPERIAL COLLEGE LONDON
;
;                           EE 3.19: Real Time Digital Signal Processing
;                                   Course by: Dr Paul Mitcheson
;
;                       LAB 4: Double precision FIR using Circular Buffer Hardware
;
;                               *********** circ_FIR_DP.ASM ***********
;
; ******************************************************************************
;                               Written by D. Harvey: 18 Jan 2010
;
; ******************************************************************************
;
        .global _circ_FIR_DP

        .text
;
; ***************************** _circ_FIR_DP description ******************************
;
;       The input delay buffer has a data length of (size in bytes)/(data type length).
;               The buffer you create must have a power of 2 size in bytes
;   i.e its length in bytes must equal 2^X bytes (where X is integer between 1 and 32).
;
;           Also ensure that its data length (size in bytes/8) is longer than the
;           coefficient array data length. The buffer will need to be data aligned
;           using  #pragma DATA_ALIGN(delay_buff_name, B) before it is defined
;                   where B is your chosen delay buffer size in bytes.
;
; circ_FIR_DP function call in C;
;
; circ_FIR_DP( &circ_ptr, &coef[0], &read_samp, &filtered_samp, N );
;
; ***************************** Register Assignments ************************************

; A0 LSB Accumulator 1                  B0 Loop Counter
; A1 MSB     "                          B1 Moved return to C Address
; A2 LSB Multiplied result    1         B2 Used to set AMR to circular mode - then reused LSB
    Accumulator 2
; A3 MSB    "        "                  B3 Return to C Address (original)  - then reused MSB "
; A4 &circ_ptr    - possible reuse      B4 &coef[k] - don't use for calc
; A5 circ_ptr    - don't use for calc   B5 Moved &filtered_samp
; A6 &read_samp    - possible reuse     B6 &filtered_samp - (original) - then reused LSB
    Multiplied result 2
; A7                                    B7                                    MSB "
; A8 N, then LSB delay_circ[j] 1        B8 LSB coef[k] 1
; A9         MSB "                      B9 MSB    "
; A10 LSB delay_circ[j]    2            B10 LSB coef[k] 2
; A11 MSB    "                          B11 MSB  "
; A12                                   B12
; A13                                   B13 Temp Store for previous AMR register value
; A14                                   B14 Data pointer (DO NOT USE)
; A15 Frame Pointer                     B15 Stack Pointer (DO NOT USE)
;  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,A9:A8       ;(4) Get the 64 bit data for read_samp put it in A9:A8
    ||  MV .S2X         A8, B0          ;(0) move parameter (numCoefs) passed from C into b0
        LDW .D1         *A4,A5          ;(4) Get the address of the circ_ptr, dereference then place in A5
    ||  MV .S2          B3, B1          ;(0) move return to C address
        MV .S2          B6, B5          ;(0) move &filtered_samp
        NOP 3                           ; A5 now holds address pointing into delay_circ

        STW .D1         A9,*--A5        ;(0) Store new input sample (MSB) to delay_circ array
    ||  ZERO .S1        A0              ;(0) zero accumulator LSB
    ||  ZERO .S2        B2

        STW .D1         A8,*--A5        ;(0) Store new input sample (LSB) to delay_circ array
    ||  ZERO .S1        A1              ;(0) zero accumulator MSB
    ||  ZERO .S2        B3


        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

        ;********************************* loop prologue *********************************
        ; prime the pipeline
        LDDW .D1        *A5++, A9:A8 ; (4) loads the (delayed) sample into A9:A8, and post increment pointer
    ||  LDDW .D2        *B4++, B9:B8 ; (4) load the coefficient into B9:B8, and post increment pointer

        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
loop:

        ; ************************ loop kernel     **************************
        MPYDP .M1X      A9:A8, B9:B8, A3:A2     ; (9, 4) DP multiply
    ||  MPYDP .M2X      B11:B10, A11:A10, B7:B6     ; (9, 4) DP multiply
    ||  SUB .S2         B0,2,B0         ; (0) b0 - 2 -> b0

        NOP 3

        [B0] B .S2      loop            ; (5) loop back if b0 is not zero
    ||  [B0] LDDW .D1   *A5++, A9:A8 ; (4) loads the (delayed) sample into A9:A8, and post increment pointer
```

```
    || [B0] LDDW .D2    *B4++, B9:B8 ; (4) load the coefficient into B9:B8, and post increment pointer


       [B0] LDDW .D1    *A5++, A11:A10 ; (4) loads the (delayed) sample into A11:A10, and post increment pointer
    || [B0] LDDW .D2    *B4++, B11:B10 ; (4) load the coefficient into B11:B10, and post increment pointer
       NOP 3

       ADDDP .L1        A1:A0, A3:A2, A1:A0    ; (6, 2) DP ADD
    || ADDDP .L2        B3:B2, B7:B6, B3:B2    ; (6, 2) DP ADD



       ;********************************* loop epilogue *********************************
       ; add both accumulators up
       NOP 5          ; for the final addition to be complete
       ADDDP .L1X       A1:A0, B3:B2, A1:A0    ; (6, 2) DP ADD
       NOP
       ; return to C code
lend:  B .S2            B1            ; (5) branch to b1 (moved C return address)
       NOP 3

       ; send the result of MAC back to C

       STW .D2           A0,*B5        ;(0) Write accumulator (LSB) into filtered_samp
       STW .D2           A1,*+B5[1]    ;(0) Write accumulator (MSB) into filtered_samp

       ; restore previous buffering mode

    || MVC .S2           B13,AMR        ;(0) restore AMR reg to previous contents

       .end
```