

# Title Here

Oskar Weigl - ow610  
and  
Ryan Savitski - rs510

March 17, 2013

## Abstract

Write the abstract here

## Contents

<b>1</b>	<b>Notch Filter</b>	<b>2</b>
1.1	Filter Design . . . . .	2
1.2	Filter Topology . . . . .	3
1.3	Quantization . . . . .	5
1.4	Filter Implementation . . . . .	6
1.5	Hardware Implementation . . . . .	6
1.6	Timing analysis . . . . .	8

# 1 Notch Filter

The Notch Filter module is a memory mapped hardware accelerator for the purpose of filtering a 16 bit signed input. It is a 2nd order notch filter tuned to to remove any spectral components at 1 kHz. The module is capable of Direct Memory Access, and includes optimisations for efficient access to the SDRAM, and uses a buffering, and Clock Domain Crossing strategy, to maximise performance.

## 1.1 Filter Design

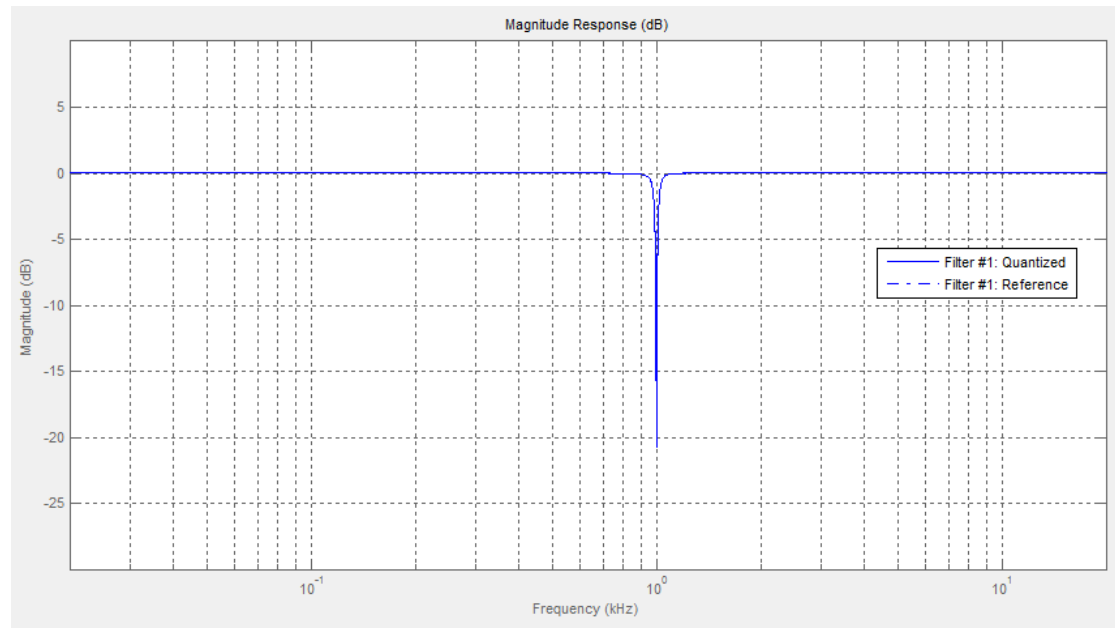


Figure 1: Magnitude response of both the full (double) precision and the quantised filter. The gain is negative infinity dB at 1 kHz.

Several different types of filter topologies are considered. Firstly, it is clear that as a 2nd order IIR filter meets the specification, it would be rather inappropriate to implement the same filter as an FIR filter. To get the same sharp transition band using an FIR filter, we would have to use over 10,000 taps. This metric is generated using MATLAB's Parks-McClellan optimal equiripple FIR order estimator; the MATLAB command and result can be seen in Figure 2.

```
beethmax = max(abs(fft(beeth5_noise)));  
firpmord([987.5 999 1001 1012.5], [1 0 1], [0.01 10/beethmax 0.01], 44100)  
ans = 13410
```

Figure 2: MATLAB command to estimate the order of FIR filter required for the notch filter specification.

The chosen filter type is a 2nd order IIR filter. This filter order is the minimum required to

generate the complex pole and zero pairs required to implement a notch filter. In fact, no higher order is needed, as the noise is a pure sinusoid, so the width of the stop-band can be very small, and as such, there are no additional zeroes required.

The filter is designed by simply placing a complex zero pair on the unit circle corresponding to the 1 KHz null we wish to create. To cancel the effect of this zero for other frequencies, we place a pole very close to it on the inside of the unit circle. The bandwidth of the filter is tuned by moving the complex pole radially.

In our case, we elect to place the pole such that the bandwidth of rejection is about 25Hz. This is chosen as a good trade-off for several reasons. Firstly, placing the pole at this location makes the filter robust to coefficient quantisation, as discussed in Section 1.3. More importantly, this bandwidth is narrow enough to not affect the music, but wide enough to quickly suppress the overlaid sinusoid at the beginning of the sample. The magnitude of the resulting filter can be seen in Figure 1.

## 1.2 Filter Topology

As a second order IIR filter is settled upon, we need to choose what filter topology we will use. The candidates are Direct Forms I and II, and their transposed versions. It is immediately clear that the Direct Form II versions are better than the Direct Form I, as the number of state registers can be halved. Thus, the choice remains between regular and transposed version of Direct Form II.

In a software implementation, the difference between the two would be quite small, but in a hardware implementation they differ significantly. As can be seen in Figure 3, the Direct Form II filter topology requires an adder chain on the feedback path. This adder chain adds one adder in a chain for every added order of the filter.

Conversely, the Direct Form II Transposed filter topology only has a single 3-input adder for each stage, which is invariant to the filter order, as illustrated by Figure 4. Thus, for the critical path is constant for any order of the filter, and in all cases shorter than the. The timing of the filter is explored further in Section 1.6.



### 1.3 Quantization

The effects of quantization depends entirely on the chosen filter topology. An FIR filter, which has no poles (except at the origin), can never go unstable, and as such, it is less sensitive to coefficient quantization. However, as seen in Section 1.1, an FIR filter is simply not feasible for this application. Thus, as an IIR filter is required, careful consideration of the effects of quantization is required.

As this is a hardware implementation, it is possible to independently designate the bit widths for each of the different stages of the filter. That is, it is not only the bit width of the coefficients themselves that need to be tuned, but also the width of the data-paths, the input, the output, the state registers, the multipliers, and the adders. Furthermore, the resources of the target hardware architecture has to be considered. For instance, the target FPGA has hardware multipliers that either multiply two 9 bit numbers, or alternatively, multiply two 18 bit numbers. That is, any bit-width between these two values would be underutilising the resources available on the FPGA.

During the initial design of the filter, it was established that a minimum of 12 bits was required for the coefficients of the filter in order to not create any resonance in the filter which would artificially amplify a portion of the passband. Furthermore, it was empirically found that playback of the audio at 8 bits was simply not high enough fidelity to do the music justice. At 16 bits, the audio was deemed to be of good enough quality, and as such, the input and output widths of the filter were chosen to be 16 bits.

As we require at least 12 bits wide coefficients, and as the multipliers will have to be 18 bits wide, it is no penalty in terms of hardware resources to simply elect to have 18 bits wide coefficients. That is, as the coefficients do not have to change during runtime, no registers are required to hold the coefficients. Furthermore, as the coefficients are 18 bits, and the input is 16 bits, the rest of the system is required to be at least 16 bits wide in order to preserve the precision. However, as the adder and state stages are in series, it is in fact beneficial to have 2-3 more bits. Thus, it was elected to set the adder and state register bit-widths to 18 bits.

Finally, we must consider where to place the binary point in the fixed point representation. As this is a hardware implementation, shifts of the binary point are free. In order to scale the binary point, we used the command `Hdsdsos = autoscale(Hdsdsos, x)` from the digital filter toolbox in MATLAB. This command will automatically scale the binary point of the different stages of the filter. It uses the provided input (in this case a 16 bit signed integer sample of the noisy music) to scale the binary point to include as much precision as possible while still avoiding overflow. The result of the operation can be seen in Figures 6 and 5.

Fixed-Point Report						
	Min	Max		Range		Number of Overflows
Input:	-32768	32767		-32768 32767		0/963144 (0%)
Section In:	-32768	32767		-32768 32767		0/1926288 (0%)
Section Out:	-23436	23293		-131072 131071		0/1926288 (0%)
Output:	-23436	23293		-32768 32767		0/963144 (0%)
States:	-16423	16408.5		-32768 32767.75		0/1926288 (0%)
Num Prod:	-64870.02	64872		-524288 524288		0/2889432 (0%)
Den Prod:	-46021.588	46304.123		-524288 524288		0/1926288 (0%)
Num Acc:	-64870	64872		-65536 65535.5		0/1926288 (0%)
Den Acc:	-54472.5	54347		-65536 65535.5		0/1926288 (0%)

Figure 5: Filtering Report generated when filtering the input sample

```

FilterStructure: 'Direct-Form II Transposed, Second-Order Sections'
  Arithmetic: 'fixed'
    Numerator: [1 -1.979736328125 1]
    Denominator: [1 -1.97576904296875 0.996002197265625]
  PersistentMemory: false

  CoeffWordLength: 18
    NumFracLength: 16
    DenFracLength: 16
    Signed: true

  InputWordLength: 16
  InputFracLength: 0

  OutputWordLength: 16
  OutputFracLength: 0

  StateWordLength: 18
  StateFracLength: 2

  ProductWordLength: 36
  NumProdFracLength: 16
  DenProdFracLength: 16

  AccumWordLength: 18
  NumAccumFracLength: 1
  DenAccumFracLength: 1

  RoundMode: 'convergent'
  OverflowMode: 'wrap'

```

Figure 6: Fixed point filter designed in MATLAB.

## 1.4 Filter Implementation

Using the `generatehdl` command..

## 1.5 Hardware Implementation

The filtering module, depicted in Figure 7, is implemented as a memory mapped device. The CPU communicates with the hardware block by writing to a Avalon slave interface. The hardware module also has a Avalon Memory-Mapped Master interface so that the data required for the filtering operation can be fetched independently of the CPU, an operation known as Direct Memory Access (DMA).

The target architecture, the DE0 board, only has a single off chip memory with high memory bandwidth, the SDRAM memory. The Flash memory that is present on this board has such low memory bandwidth that if it were to be used, it would slow the system down several times over. Thus, to complete the filtering operation, the data must be fetched from SDRAM, filtered, and then stored back in the same memory (but at a different location). The solution to this contention is to use a custom arbitrator that will interleave the reads and writes to SDRAM.

As non-sequential access to SDRAM is slow and sequential access is fast, we use a feature of the Avalon interface known as Bursting. This feature locks the slave device, in this case the SDRAM controller, and forces it to fetch or store a sequential chunk of data as one uninterrupted operation. That is, we are able to lock the arbitration of the Avalon interconnect to finish serving

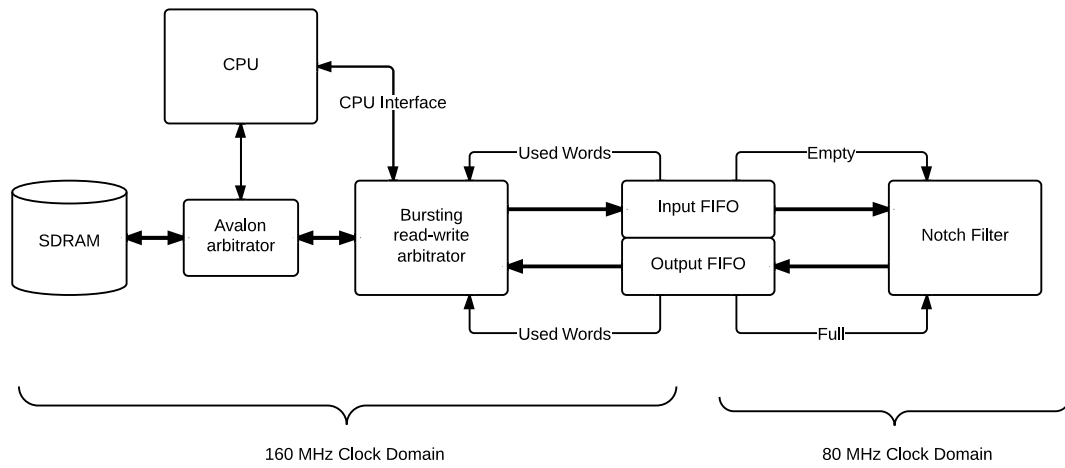


Figure 7: Block diagram of the Notch Filter hardware. FIFO buffers are used to allow efficient interleaved reads and writes to SDRAM. The buffers also serve as Clock Domain Crossing interfaces.

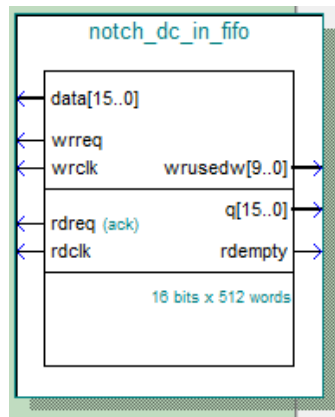


Figure 8: Block diagram of the First In First Out buffers used for both data buffering and Clock Domain Crossing

our sequential burst before it interleaves any requests from other masters, such as the CPU.

In order to maximise efficiency, we use a pair of First In First Out (FIFO) buffers to absorb the data that is obtained during a read burst, or to ensure sustained write throughput during write burst. These buffers are constructed from the architecture specific double ported memory blocks (M9K memory blocks), as shown in Figure 8. These FIFO blocks support different clocks on the read and write ports. Thus they can be, and are, used as Clock Domain Crossing (CDC) logic. This means that we can operate the filter core at a different clock speed than the rest of the system.

The core of the filter is limited to operating at 80MHz, while the rest of the system operates at 160MHz (see Section 1.6 for a detailed timing analysis). This means that even though the memory access is interleaved and cannot fetch one new word every clock cycle on average, the filter can still operate at almost full saturation.

Describe diagram 1, (SDRAM slow, but good for sequential, interleaving) As described in Section 1.6.

## 1.6 Timing analysis

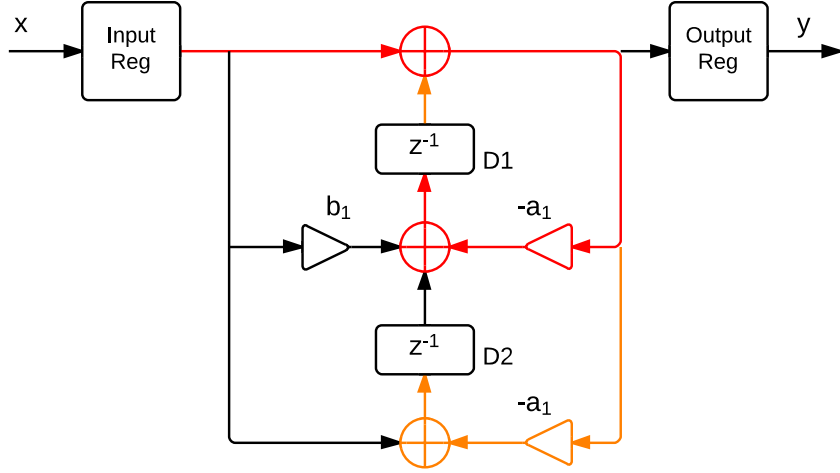


Figure 9: Annotated version of Figure 4, showing the main critical paths in the filter core. Shown in red is the critical path; from the input register to the delay register D1. Shown in orange are the alternative paths that have almost as little slack as the red path.

As mentioned in Section 1.2, Direct Form II Transposed is the chosen filter topology because of its superior timing characteristics. Nevertheless, the performance of the filter core is limited to processing 80 Million samples per second. This is because the critical path, depicted in red in Figure 9, is forced to pass through two adder stages and one multiplier. Because the filter is recursive in nature, it is not possible to pipeline this path and as such it is not possible to increase the clock speed in a simple way. There is a method called Look-Ahead pipe-lining, which involves creating pole and zero pairs that cancel each other out, in order to be able to pipeline the feedback path. However, this approach was deemed too involved for the purposes of this project.

One optimisation is however possible. As seen in Figure 9, the  $b_0$  coefficient multiplier is not present. That is because, in our case, this coefficient, representing the overall gain of the system,



is unity. However, because of the linearity of IIR filters, if this gain is required to be non-unity, it can still be present, but moved to the other side of the input register. Thus, the critical path is still maintained to be two adders and one multiplier for any Direct Form II Transposed filter implementation. In our case, however, to optimise the resources used for the filter, we elected to optimise away both multiplier  $b_0$  and  $b_2$ , as they are both unity.

When performing the timing analysis we found that there were several paths that had a timing slack that was almost as tight as the critical path. These are depicted in orange in Figure 9. That is, the paths from the input register to  $D1$ , from  $D1$  to  $D1$ , from the input register to  $D2$  and from  $D1$  to  $D2$ , all have similar timing characteristics, and are the limiting paths in the design. It is interesting to compare the fact that they all have such similar timing characteristics (within 100ps slack of each other), to the timing expected from inspecting the data-paths in a Direct Form II Transposed filter topology. That is, as seen in Figure 9, all of these paths pass through two adders and one multiplier. It is interesting to see that the expected timing from a high level topology perspective translates so well to the final hardware implementation.

85 degC 80.55 MHz 162.0 MHz Metastability MTBF:  $\geq 1$  Billion Years  
 0 degC 90.59 MHz 181.82 MHz Metastability MTBF:  $\geq 1$  Billion Years  
 2stage synch.