Digital System Design Report

Oskar Weigl - ow610
and
Ryan Savitski - rs5010

June 27, 2014

Introduction

This report describes the design considerations and implementation details of the Digital System Design assignment involving two sub-projects. The first part concentrates on various methods of adding hardware support to compute arithmetic operations, in particular in computing the determinant of a matrix. We implement a software algorithm based on LU Decomposition to efficiently compute the determinant of a matrix. We explore the use of different types of multipliers to achieve this, and compare the results to a custom floating point unit.

Further, a highly optimised hardware accelerator is developed as a hardware implementation of the Doolittle algorithm.

The second part of the project involves the implementation of a streaming filter hardware module. The module is capable of Direct Memory Access, includes optimisations for efficient access to the SDRAM, and uses a buffering and Clock Domain Crossing strategy, to maximise performance. The design process including topology consideration and a quantisation analysis is presented, and the detailed hardware implementation is described. Further, a considerable timing analysis and several optimisations to maximise performance are presented, and a performance and resource utilisation analysis is performed. Finally, we verify the operation of the module and conclude with some suggested improvements.

Contents

1 Floating Point Hardware

We use a modified version of the Doolittle Algorithm for LU decomposition to compute the determinant of a matrix. We implement a software version and perform several benchmarks with different hardware configurations, and finally implement and benchmark a custom instruction hardware module for the use with this software algorithm.

The modified Doolittle Algorithm is discussed in more detail in Section 2.3, but worth noting is that this algorithm has the time complexity of $O(n^3)$ as opposed to $O(n!)$ of Laplace expansion, and is therefore much faster for any matrix larger than 4x4. During our benchmarking tests, we saw virtually no change to the code size, which is 49 KB. Most of the code space is taken up by support functions such as `printf`.

Seen in Table 1 is the configuration of the system during testing. The performance results can be seen in Table 2 and Figure 1, and the resource usage can be seen in Table 3. We see an improvement in the runtime of the algorithm when using integer multiplication hardware over software multiplication. The resource penalty of implementing these integer multiplication units depends on the type implemented. When using embedded multiplier blocks, we consume four 18 by 18 multiplier blocks, and only around 200 logic elements. Conversely, to implement the multiplication using Logic Elements, we incur a penalty of 400 logic elements. Depending on the resource restrictions in the application, either may be better.

We see a considerable further improvement to the performance of the algorithm when using dedicated floating point hardware to perform the operations. When implementing the custom logic blocks for the floating-point arithmetic operations, we elected to use the floating point blocks provided by Altera's Megafunction tools. These tools provide a block for implementing floating point multiplication, or addition and subtraction, among many others [9].

There are several different versions of these blocks available, with different clock cycle latency and different Fmax. During this phase of the project, the target clock frequency of the system is 50 MHz. As such, all of the provided blocks have an Fmax that is higher than the system Fmax. Therefore, we chose the block with the shortest latency and minimum resource usage. Particularly when used as a custom instruction for the CPU, latency is much more important than in a pipelined application. This is because the CPU only has a single execution stage and cannot pipeline the execution [2].

We implement custom bypass logic that will compute the result of the requested operation in a single clock cycle if the request is "trivial". For example, if any of the operands are NaN [3], the result is always NaN, so we can bypass the arithmetic module, and we can finish the operation in a single cycle. Trivial or degenerate operations include multiplication by negative or positive unity, zero, negative or positive infinity, or NaN. Addition or subtraction by zero, infinity or NaN is also handled. The bypass module honours all combinations of degenerate operations correctly. For example, the combination of the signs of the operands, and doubly degenerate operations such as infinity multiplied by zero, is correctly handled.

Table 1: Hardware setup for the floating point performance measurements.

Processor Type	Nios II Standard
Instruction Cache	2 kB
Data Cache	none
Clock Speed	50 MHz

Table 2: Performance comparison of determinant calculation for various hardware configurations and matrix sizes. The algorithm used is based on LU decomposition and the hardware is set up as shown in Table 1.

(ms/matrix)	**3x3**	**6x6**	**8x8**	**10x10**	**20x20**
No hardware multipliers	0.940	6.6	15.3	30.1	231.75
Embedded multipliers	0.470	3.15	8.0	14.4	115.5
Custom floating-point instructions	0.122	0.468	0.984	1.450	7.8
Hardware Accelerator	0.02	0.03	0.04	0.052	0.17

Table 3: Resource usage of entire system with different types of multipliers used. Columns are logic elements, combinational functions, registers, memory bits and 18 bit multipliers.

	LE	**Comb.**	**Reg.**	**Mem.**	**Mul.**
Available resources	15,408	15,408	15,408	516,096	112
No hardware multipliers	2,728	2,519	1,655	29,248	0
Embedded multipliers	2,934	2,709	1,903	29,248	4
Logic Element multipliers	3,090	2,852	1,936	29,248	0

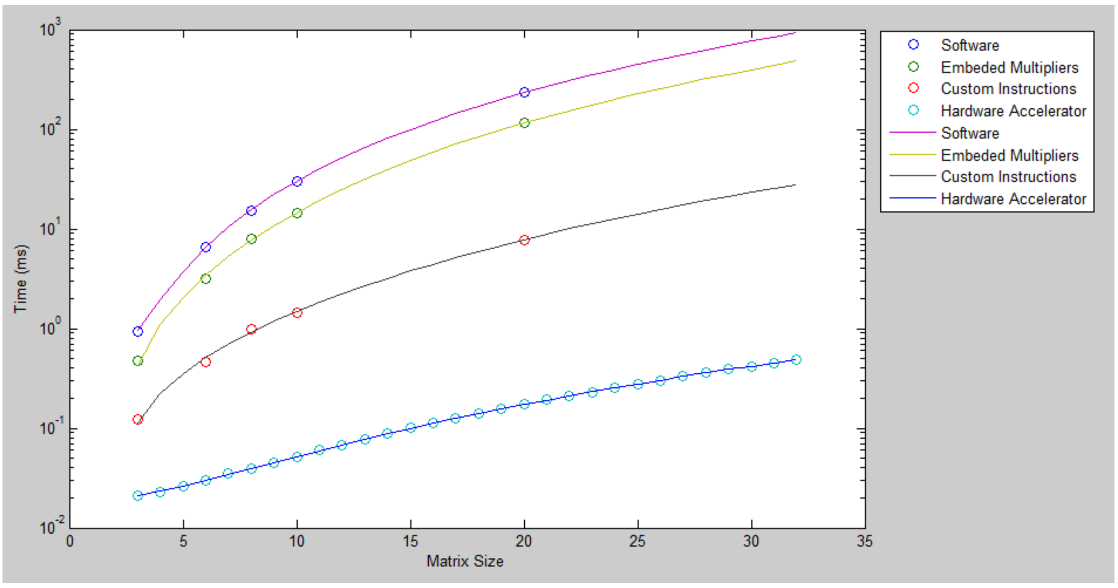


Figure 1: Performance comparison of determinant calculation for various hardware configurations and matrix sizes.

2 Determinant Calculation Hardware Accelerator

2.1 Introduction

This section describes a hardware accelerator solution for finding matrix determinants and its integration into the Nios system.

2.2 High level overview

The accelerator module is implemented as a memory mapped device in the Nios system. The completion of the operation is indicated to the processor via an Interrupt Request (IRQ). All software calls to the unit go through a custom software driver layer that handles all details of the interface, as well as providing an abstraction layer between hardware and software.

The entire computation is done on the hardware, thus the processor is left free to execute other instructions while the determinant is being computed. There is no need for explicit polling on the hardware done by the processor as completion is indicated via an interrupt. Although the driver layer does provide a software flag for the processor to check the status of the computation.

2.3 Determinant algorithm

LU decomposition method is chosen as its complexity is O(n^3). Whereas a naive Laplace expansion approach is O(n!) and as such is computationally intractable for matrices of sizes of interest, as small ones can be handled in software without significant overheads.

LU decomposition can handle any size matrix without needing modifications or extra resources, it can be done fully in-place. Although for implementations of this algorithm there exists an upper bound at which the memory hierarchy of the target architecture becomes the limiting factor, this is discussed further in Section 2.5.1.

From examining the algorithm structure, it is noted that the hottest loop maps well to a dataflow pipeline, which the FPGA can readily take advantage of.

The particular implementation algorithm used for LU decomposition is the Doolittle algorithm. Its pseudocode is outlined in Figures 1.

2.4 Operation

At system initialisation, a custom Interrupt Service Routine (ISR) is registered by the driver to the determinant block's IRQ number. From the software's point of view, the determinant calculation is started by a call to the driver layer, the arguments passed are the memory pointer to the matrix and its size.

The driver layer first initialises a data cache flush on the processor side to ensure that the matrix is in a consistent state on the SDRAM. Then the driver proceeds to do a direct uncacheable IO write to the memory mapped registers of the accelerator, passing the same arguments to the hardware.

Upon detecting a write to its memory (via the avalon interface), the accelerator begins its operation by starting a DMA transfer of the matrix from the SDRAM to a local cache (which is private to the determinant). The alternative strategies, such as a shared data cache with the processor, are discussed in Section 2.7.

The DMA unit is a fully custom implementation and performs a pipelined avalon transfer of the matrix contents. A future iteration of the module will switch to a bursting interface due to the reasons outlined in Section 2.7.

```
**Data**: N x N matrix: A
**Result**: Determinant
for $k = 1$ to $N - 1$ do
    while $a_{k,k} = 0$ do
        rotate rows $k$ to $N$;
        if number of rotations $= k$ then
            determinant $\leftarrow 0$
            return;
        end
    end
    for $i = k + 1$ to $N$ do
        $a_{i,k} \leftarrow a_{i,k}/a_{k,k}$
    end
    for $j = k+1$ to $N$ do
        for $i = k+1$ to $N$ do
            $a_{i,j} \leftarrow a_{i,j} - a_{i,k} \times a_{k,j}$
        end
    end
end
determinant $\leftarrow \prod_{k=1}^{N} a_{k,k}$
```

Algorithm 1: The Doolittle Algorithm for computing the determinant of a matrix through LU decomposition, as found in [1], but modified to compute the determinant, and including pivoting.

The memory layout for the transferred matrix in the local ram is not the same as in the SDRAM (in other words, not the usual C array contiguous memory layout). The rows of the matrix are aligned with the ram row boundaries (32 four-byte word in the current configuration). An illustration is shown in figure 2. This layout simplifies the hardware for row pivots as required by the Doolittle algorithm.

To efficiently implement row permutations, instead of copying memory, a shift register with row pointers is used. All addressing into the local ram is done through this dereferencing register, hence rotations can be done by simply shifting the contents appropriately.

The hardware proceeds to drop into the decomposition algorithm sub-FSM pictured in figure 3.

The first step of an iteration of the decomposition is to ensure that the normalising factor

```
r\c  0:   1:   2:   3:   4:       31:
0:   [ 0][ 1][ 2][  ][  ]...[  ]
1:   [ 3][ 4][ 5][  ][  ]...[  ]
2:   [ 6][ 7][ 8][  ][  ]...[  ]
3:   [  ][  ][  ][  ][  ]...[  ]
4:   [  ][  ][  ][  ][  ]...[  ]
...
31:  [  ][  ][  ][  ][  ]...[  ]
```

Figure 2: Example layout of a 3x3 matrix within 32x32 cache with row alignedness

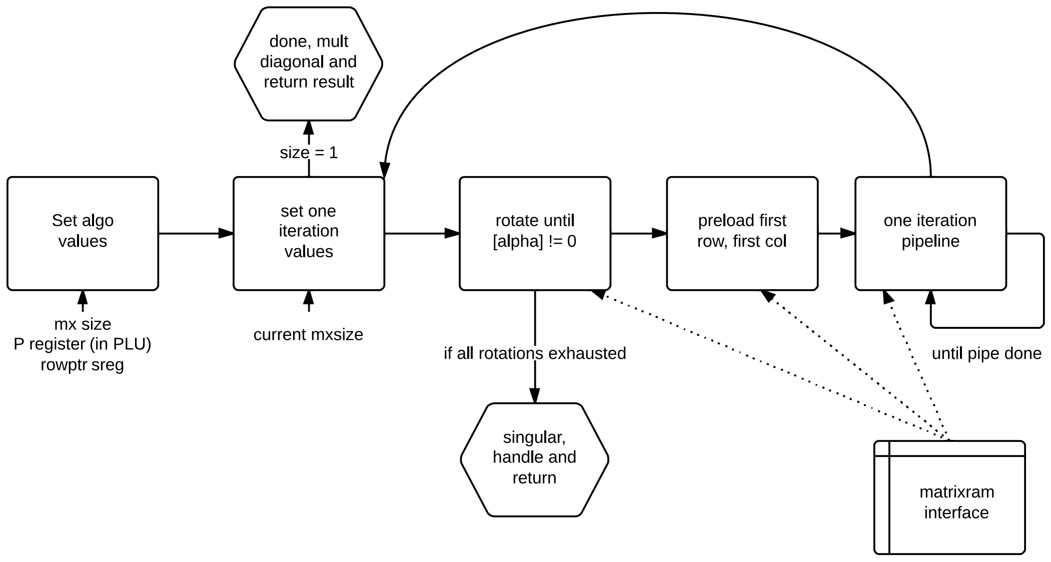


Figure 3: FSM part responsible for LU decomposition within doolittle algorithm

(the entry on the diagonal that will scale all the left row values) is non-zero. A stage in the FSM performs pivots until either a non-zero value is found or all rows have been exhausted. In the latter case, the matrix is singular and the algorithm terminates early, returning zero as the correct determinant.

After pivoting, the hardware reads the top row and left column of the current iteration matrix size (which is decremented after each full iteration) into extra registers. This is to ensure that the actual performance critical section of the pipeline can sustain itself with only one read and one write (of a matrix entry) per cycle.

The next state is the pipelined kernel of the hot loop. The pipeline is illustrated in the Figure 4.

The pipeline starts by feeding the first column divided by the normalisation factor (alpha) into the division block. Once the first result flows to the output of the division unit, the unit's enable is deasserted and handed over to the multiply stage to control. The multiply stage continuously does a wrapping sweep of the top row and multiplies them by the normalised values from the division stage. The div unit is enabled every time a row is finished. Next, the subtractor module reads submatrix values in row major order and subtracts the corresponding normalised factors. This performs the hot loop of the Doolittle algorithm in a fully pipelined way. It achieves 100% pipeline utilisation with 3 fp instructions per cycle and both read of a matrix entry and a write per cycle. Therefore a kernel pass for an iteration for the matrix size of N takes $(N-1)^2$ clock cycles (disregarding the pipeline latency).

Note: for determinant finding, the full LU decomposition is not required, only the correct diagonal entries in the L matrix. Therefore the hardware does not compute redundant zero values but rather treats them as zero implicitly.

Once an iteration is complete, the matrix size is decremented and the hardware transitions back to checking whether proper termination or singularity conditions are met, if not, the next iteration of the algorithm runs as above, ending when we reach a 1x1 matrix size.

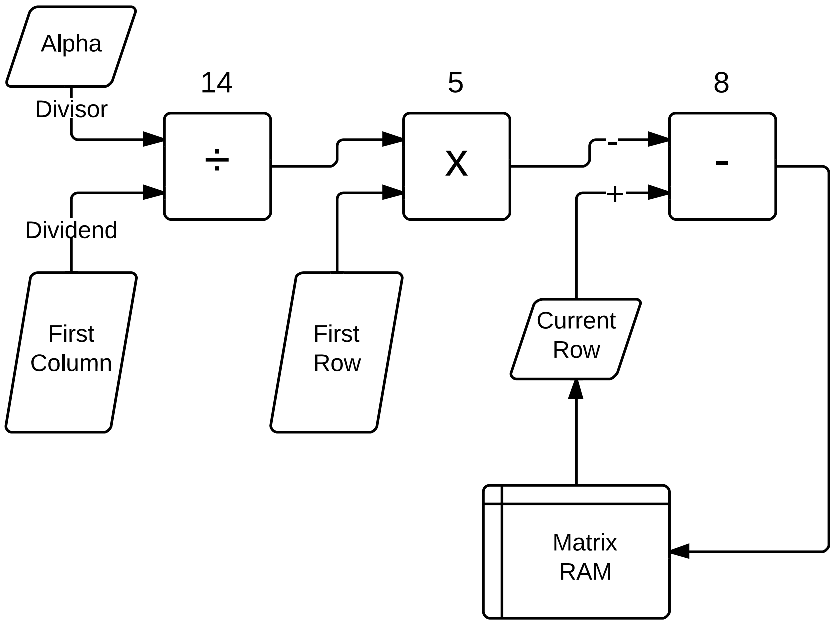


Figure 4: Block diagram of the hardware implementation of the pipeline implementing Doolittle's hot loop. The numbers above the arithmetic blocks indicate latency.

In the (usual) case of the algorithm not terminating early due to a singular matrix input, we need to multiply the diagonal entries together to obtain the determinant. This is done by a separate sub-fsm that does a naive serial multiplication with an accumulator.

Once the determinant is obtained, it is written to the result register and the interrupt is raised. When the ISR reads the result, hardware transitions to the idle state, ready to process the next request.

2.5 Resources

2.5.1 Resource usage

The resource usage for the determinant accelerator module is shown in table 4. Note that the big LE usage is due to row/col preloading memory synthesised to LEs instead of being packed into one m9k (which is just a synthesis setting).

2.6 Scaling

Due to the nature of the algorithm, the same hardware tackles all sizes up to the memory capacity. Current implementation does 32x32 upper bound, which is 2^{12} bytes as each entry is a four byte fp single precision. This maps to 4 m9k units for local ram. Row and column preloading registers requires 2^8 bytes.

There are different versions of the fp units available, with different clock cycle latency and maximum frequency max. The design choices are guided by smallest area possible block, while

Table 4: Resource utilisation of the determinant module. Note that two fp_mul units are used. Also note that the big LE usage is due to row/col preloading memory synthesised to LEs instead of being packed into one m9k (which is just a synthesis setting)

	Entire Module	**fp_div**	**fp_mul**	**fp_sub**	**matrixram**
Logic Cells	5978	404	264	827	0
Dedicated Registers	3663	291	208	426	0
M9K Memory Blocks	11	6	0	1	4
Multiplier Elements	31	16	7	0	0

Table 5: Timing benchmark results of various versions of the Altera floating point units. The chosen type is indicated by the word USE.

	Speed (MHz)	**Latency (cycles)**	
Division Unit	141	6	
	>160	14	USE
Multiplication Unit	>176	5	USE
Subtraction Unit	136	7	
	>176	8	USE
	>176	10	

maintaining an fmax that is at least as fast as the target system frequency, which is 160 MHz. The choices are shown in table 5.

Note that no additional fp units are needed as the pipeline stays the same for all possible matrix sizes.

If we were to scale the design from 32x32 to 64x64 max size. The m9ks for local ram scale from 4 to 16 and row/col registers scale from 2^8 (can be packed in 1 m9k) to 2^9 which can still be packed in one m9k. There is also a small footprint of extending several sweeping counters by one bit, but that does not a perceivable impact.

As the matrix size increases, there exists a limit at which it is impossible to cache to the entire matrix. At this point, the main memory (SDRAM) access times become the blocking factor with the usual access patterns that need to reload caches. Instead, the design would have to transition to maximising work done per cached piece of the matrix. Block LU decomposition [1]. decomposition does that, but is a different algorithm and would need different hardware as well.

2.7 Performance

As mentioned before, the algorithm's hot loop of N^2 complexity is implemented in a pipeline that achieves one matrix entry per cycle (3 fp instructions, one read and one write per cycle). A cubic regression comparison to the previous solutions is shown in 5 and the cubic regression results are shown in figure 5. The complexities of the respective operations are shown in table 7.

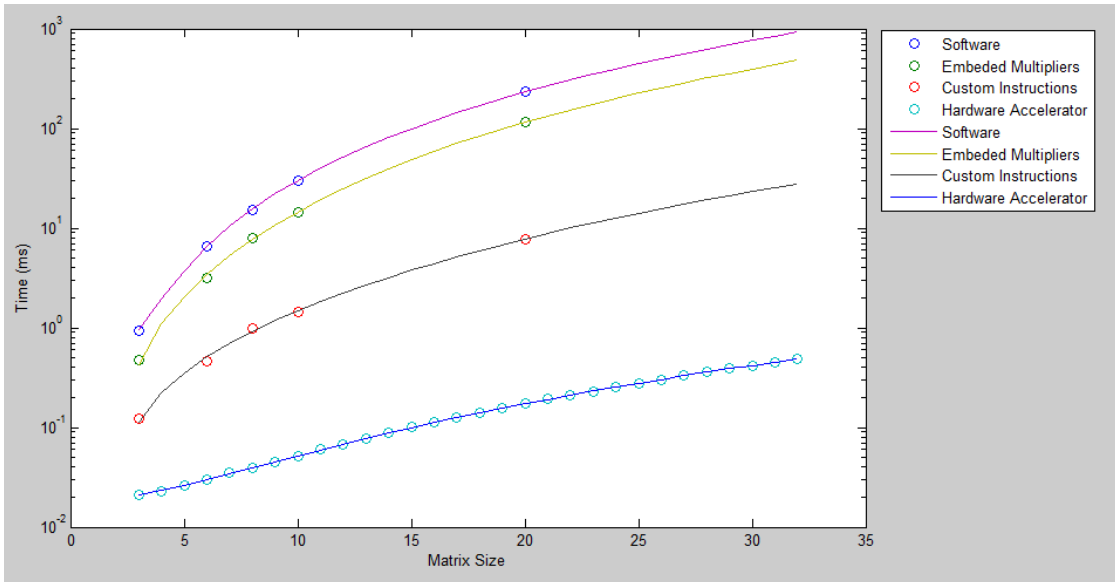


Figure 5: Performance comparison of determinant accelerator to the previous implementations

Table 6: The result of the cubic regression shown in Figure 5. Note that the actual cycle performance matches the expected complexity of N^3.

	N^3	N^2	N	1
Time (ns)	6.49	236	490	17100
Time (cycles)	0.324	11.8	24.5	855

Table 7: Expected complexity of various operations in LU decomposition. Note that element multiplication and subtraction are pipelined and do not add.

Element multiplication	$N^3/3$
Element subtraction	$N^3/3$
Matrix memory copy	N^2
Pipeline flushes	N
Division latency	N
Diagonal multiplication	N
Interrupt latency	1

2.7.1 Improving performance

Memory transfer time by the DMA is significant up to a point where N^2 times the SDRAM access time gets dominated by the N^3 Doolittle algorithm complexity. For current matrix sizes up to 32x32, the transfer time is relevant at all sizes. Assuming the processor does operations on the matrix which determinant it wants to compute (otherwise it would be a compile time constant). We could share the cache with the processor with cache coherency algorithms.

The accelerator with processor versions that don't have either instruction or data cache means that instruction/data fetches create contention for the SDRAM and do not let the pipelined transfer copy the entire matrix in one go. A bursting interface would not suffer from this and always do an entire transfer in a single transaction.

Several parts of the FSMs can be "improved" by pipelining certain states, such as doing a multiply tree for the diagonal multiplication at the end or starting the kernel pipeline as the hardware is preloading the first row and column. However, all of these optimisations are small constant factors of several cycles and are completely dominated by the n^3 complexity of the algorithm. Furthermore, these alterations increase the complexity of design and make it less modular.

One way of speeding up the algorithm's kernel pipeline by a factor of two is to instead of storing the matrix entries in the m9k ram blocks serially, we can stripe the bytes of one entry across several blocks. Then we can read two continuous fp single instruction entries in a cycle, therefore the pipeline can be duplicated and the relevant N^2 loop is sped by a factor of two.

3 Notch Filter

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

3.1 Filter Design

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, no additional zeroes are required.

The filter is designed by simply placing a complex zero 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 complex 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 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 3.3. 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 response of the resulting filter can been seen in Figure 7.

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 6.

3.2 Filter Topology

As a second order IIR filter is settled upon, we need to chose 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 9, the Direct Form II filter topology requires an adder chain on the feedback path. This adder chain grows with one adder in the chain for every increased order of the filter.

```
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 6: MATLAB command to estimate the order of FIR filter required for the notch filter specification.

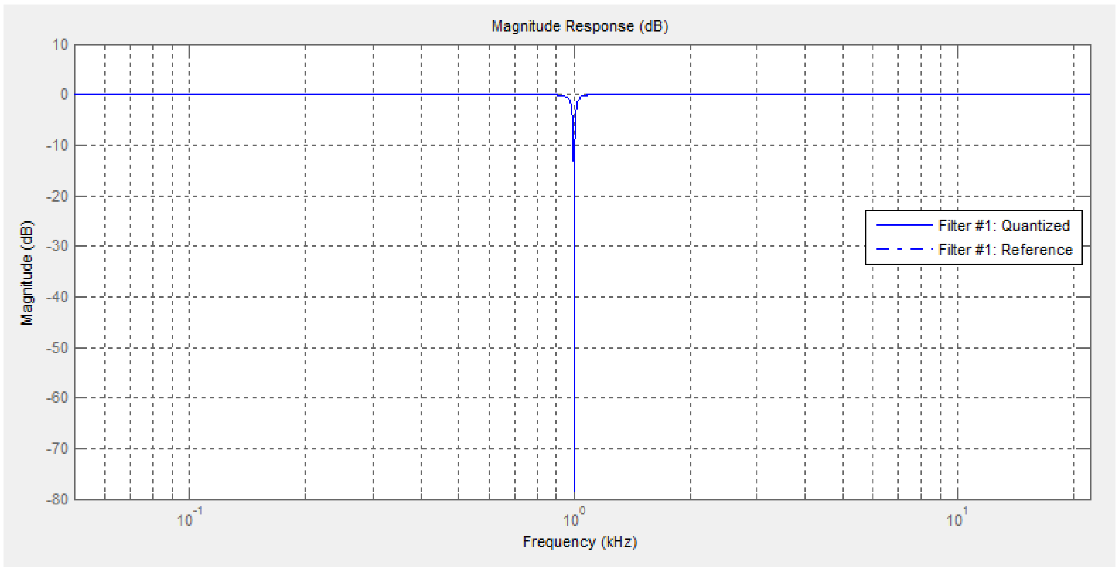


Figure 7: Magnitude response of both the full (double) precision and the quantised filter. The gain is negative infinity dB at 1 kHz.

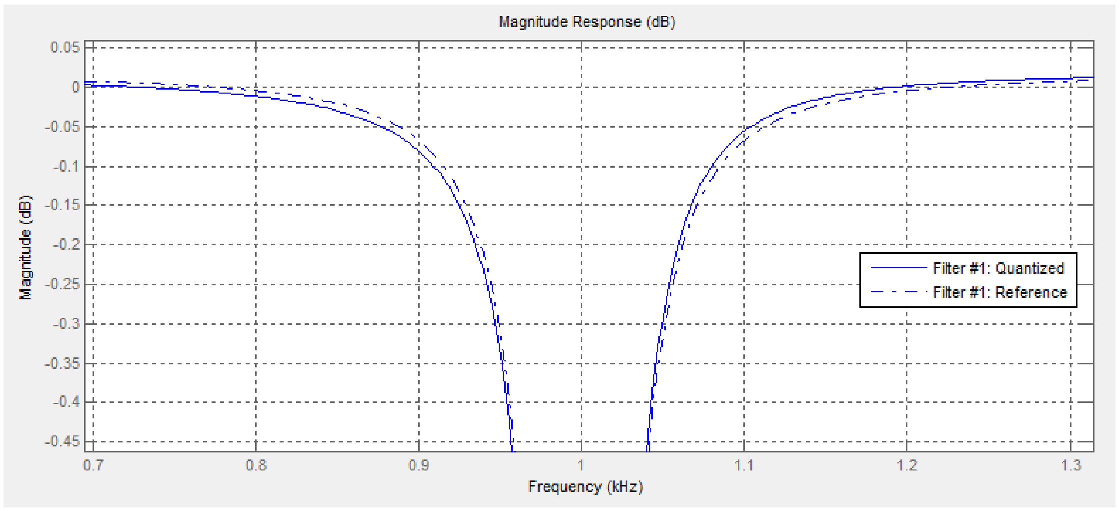


Figure 8: Magnitude response of both the full (double) precision and the quantised filter. The quantised and reference filters differ with less than 0.02dB in the passband.

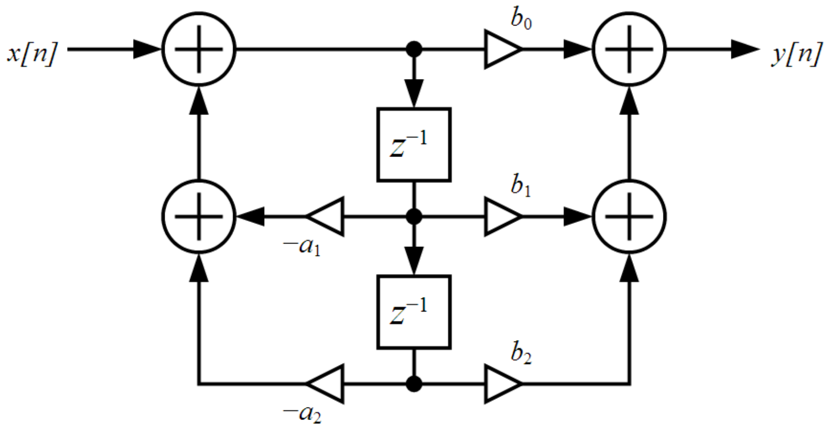


Figure 9: Second order Direct Form II filter topology.

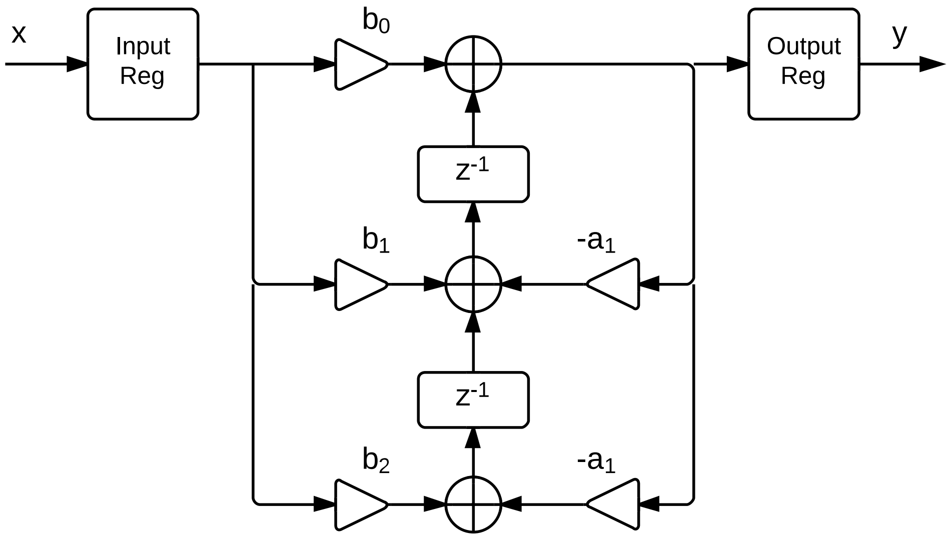


Figure 10: Second order Direct Form II Transposed filter topology.

```
   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'
```

Table 8: Fixed point filter designed in MATLAB.

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 10. Thus, the critical path is constant for any order of the filter, and in all cases shorter than the non-transposed version. For these reasons, a Direct Form II Transposed filter topology is chosen. The timing of this filter is explored further in Section 3.6.

3.3 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 become unstable, and as such, it is less sensitive to coefficient quantization. However, as seen in Section 3.1, 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.

```
                           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%)
```

Table 9: Filtering Report generated when filtering the input sample

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 are 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 of penalty in terms of hardware resources to simply elect to have 18 bits wide coefficients. That is, as the multiplier blocks have dedicated input registers [7], no extra 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 adders and state stages are in series, it is in fact beneficial to have 2-3 more bits. Thus, it is 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. That is, we can independently assign the size of the fractional part of all the stages of the filter. 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 8 and 9.

It should be noted that the aforementioned automatic scaling will optimise the filter for a representative input. Therefore, it will by design not guarantee the absence of overflow for any input. That is, if the filter were exposed to very extreme inputs, such as rail-to-rail amplitude signal, especially containing high amplitude 1kHz components, it would overflow. To make a more robust filter, one could either over-estimate the range required during the binary-point placement, or implement saturating arithmetic. The trade-off is that the former will introduce more noise, while the latter will increase the critical path and use more hardware resources.

3.4 Filter Implementation

The filter described in Section 3.3 is engineered to minimise quantisation noise while avoiding any overflow. This design is carried out using the digital filter toolbox in MATLAB. This toolbox

```
module Hdsdsos_copy_hardwired
               (
                clk,
                clk_enable,
                reset,
                filter_in,
                filter_out
                );

input   clk;
input   clk_enable;
input   reset;
input   signed [15:0] filter_in; //sfix16
output  signed [15:0] filter_out; //sfix16
```

Figure 11: The interface of the automatically generated filter.

includes a feature for producing automatically generated Hardware Description Language (HDL). Using the `fdhdltool` command, we are able to generate Verilog HDL for the filter we have designed. This tool provides us with a Verilog source file with the interface as shown in Figure 11. The filter input and outputs are signed 16 bit integer representation with no fractional parts, as required.

As this method is guaranteed to generate the same digital filter that is already extensively tested in MATLAB, we are able to save a great deal of time in design and especially debugging of the filter core itself. The time saved is put to use in performing the integration of the filter with the rest of the system, and especially in developing the custom memory controller described in Section 3.5 and 3.7.

The HDL generation tool can also generate a second type of interface, one specifically designed to be capable of interfacing to a CPU for loading custom coefficients. However, due to the optimisations described in Section 3.6, we elected to use hard-wired coefficients. That is, the filter design tool cannot assume that the overall gain of the filter is unity when electing to supply coefficients from the processor interface.

However, it is possible to modify four (and with some modifications, five) out of the six coefficients while still maintaining the speed advantage. This design alteration is described in Section 3.6. Unfortunately we did we did not have time to implement it. As such, while the design is focused around having a fast system with variable coefficients, the system currently has hard-wired coefficients.

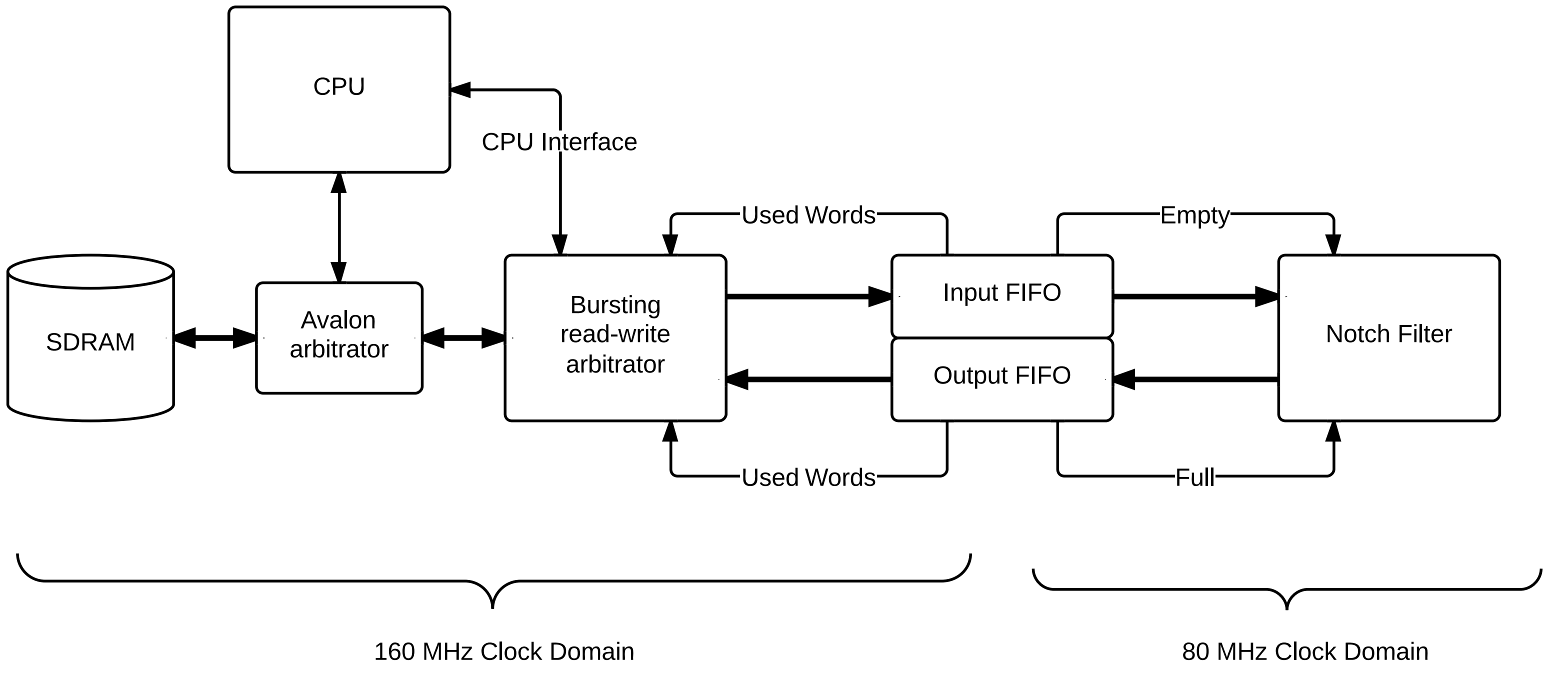


Figure 12: Block diagram of the Notch Filter hardware. FIFO buffers are used to allow efficient interleaved reads and writes to SDRAM. The buffers also serve as Clock Domain Crossing interfaces.

3.5 Hardware Implementation

The filtering module, depicted in Figure 12, is implemented as a memory mapped device. The CPU communicates with the hardware block by writing to an Avalon Memory-Mapped 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 [4], we use a feature of the Avalon interface known as Bursting [5]. 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 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 a write burst. These buffers are constructed from the architecture specific double ported memory blocks (M9K memory blocks) [6], as shown in Figure 13. 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.

Another critical consideration is that once a burst has been initiated, it cannot be aborted [5]. Thus, the custom memory access controller will request at most the same number of words that remain unused in the input FIFO, as seen in Figure 14. The same thing applies to the write requests, the controller only requests to write as much data as is available in the output FIFO.

The core of the filter is limited to operating at 80MHz, while the rest of the system operates at 160MHz (see Section 3.6 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.

While the Avalon Bus is a multi-mastered matrix [5], the heavily contended SDRAM is the bottle-neck for both the CPU and the filter hardware. For this reason we give the CPU both data and instruction caches, so that in the best case, it will not be completely starved during filtering. However, in the worst case, as the Avalon arbitration is by default Round-Robin [5], the CPU will only be able to interleave with the filter hardware for a single word of data every 512 clock cycles. This is deemed to not be a problem in this case, as the CPU performs no real-time critical tasks. If the CPU did require high priority access to the SDRAM in a different application, a priory aware arbitrator can be used.

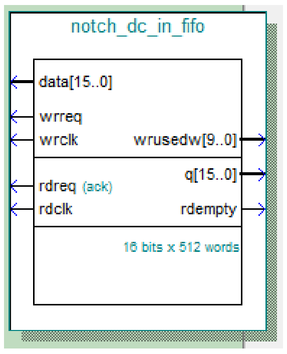


Figure 13: Block diagram of the First In First Out buffers used for both data buffering and Clock Domain Crossing.

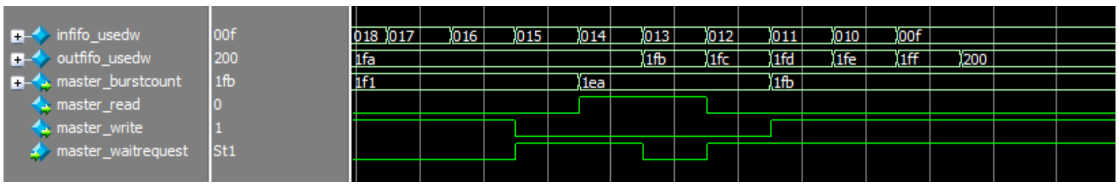


Figure 14: Simulation of the system, showing how the number of requested words depends on how much free space/used words is present in the input/output FIFO. Note that there is a 2 cycle latency from the capture of the state of the FIFOs to the generation of the request. Also note that there is a long delay (30 cycles) after the read-request before any data is returned as the CPU is serviced in this period.

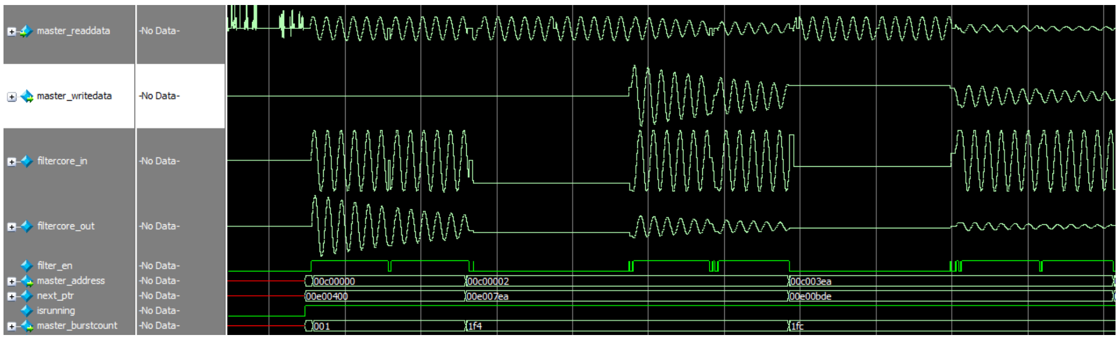


Figure 15: Simulated interleaved operation of the filter. Note that both Clock Domains are simulated at 50MHz for clarity.

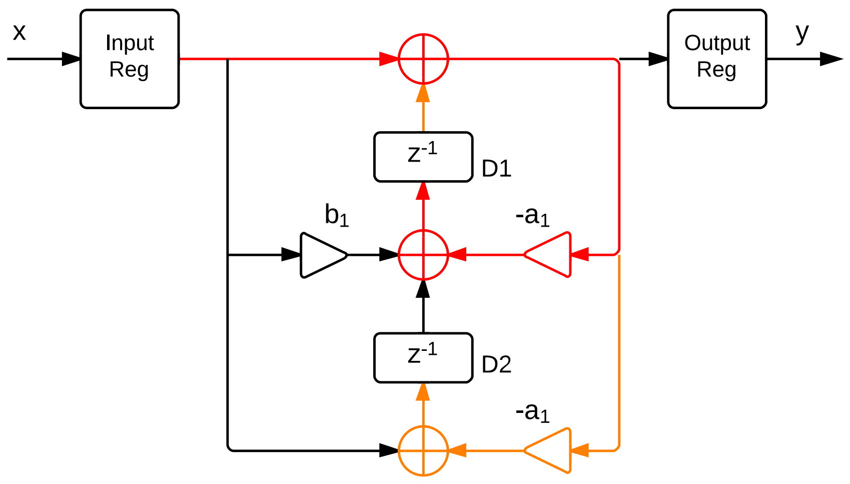


Figure 16: Annotated version of Figure 10, showing the main critical paths in the filter core. Shown in red is the critical path; from the input register to the delay register D1. Shown in orange are the alternative paths that have almost as little slack as the red path.

3.6 Timing analysis

As mentioned in Section 3.2, 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 16, is forced to pass through two adder stages and one multiplier block. 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 pipelining, which involves creating pole an zero pairs that cancel each other out, in order to be able to pipeline the feedback path. However, this approach is deemed too involved for the purposes of this project.

One optimisation is however possible. As seen in Figure 16, 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, the multiplier can be 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 for any notch filter.

Furthermore, when performing the timing analysis, we found that there are several paths that had a timing slack that is almost as tight as the critical path. These are depicted in orange in Figure 16. 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.

In fact, it is interesting to note that the high degree of similarity of the timing of these paths is expected. When inspecting the data-paths in a Direct Form II Transposed filter, as seen in Figure 16, all of these paths pass through two adders and one multiplier. It is fascinating to see that the expected timing from a high level topology perspective translates so well to the final hardware implementation.

Table 10: Timing performance of the system at various operating conditions.

	0°C	85°C
Maximum System Frequency	181.82 MHz	162.0 MHz
Maximum Filter Core Frequency	90.59 MHz	80.55 MHz
Metastability MTBF	>1 Billion Years	>1 Billion Years

Moreover, as the FIFO buffers are used as Clock Domain Crossing interfaces, there is a risk of metastability. However as the generated clocks are phase locked, as they are generated using the same PLL module, this risk is minimal. That is, as the clock frequency ratio is an exact integer, in this case two, the clock edges should always line up, and no metastability should be possible. Nevertheless, the FIFO buffers are set-up to use two stage synchroniser stages to synchronise operation. While this may be overkill for the purposes of eliminating metastability, it does allow register re-timing to have more freedom to optimise the circuit for higher speed.

3.7 Tuning for Speed

As the target speed of the filtering module is quite high: 160MHz for the interface side, and 80MHz for the filtering side, some special considerations are required. In particular, it was found that the address calculation for the fetch stage of the custom memory controller could not initially operate at 160MHz. This was because, it was designed to update the current fetch pointer, compare it with the final pointer, and conditionally request new data. As these pointers are 32 bits wide, the critical path for this operation was simply too slow.

To remedy this, we redesigned the addressing to use a base pointer and an offset. This reduced the arithmetic to 20 bits. Furthermore, we pipelined the address calculation, so that a fetch operation has a latency of 2 cycles. Thus, are were able meet the timing requirements for operation at the target clock frequency.

3.8 Performance Results

The performance of the filter is benchmarked by executing a complete filtering pass of the filter, and observing a dedicated cycle-counting register. This register is simply reset at the start of the filtering operation and will increment every clock cycle until the filtering operation is complete.

Due to the efficiency of the custom bursting interleaved memory access controller, operating at twice the clock frequency of the filter core, as described in Section 3.5, we are able to exceed 90% saturation of the filter core. Thus, as the sampling rate of the data is 44.1 KHz, we are able to exceed 1600 times real-time performance.

Table 11: Performance results of the Notch Filter module.

Time to process beeth5_noise	13.3	ms
Length of beeth5_noise	963,144	Samples
Average Throughput	72.24	MSamples/s
Filter Capacity	80.0	MSamples/s
Filter Saturation	90.3	%

Table 12: Resource Utilisation of the Notch Filter module.

	Entire Module	**Filter Core**	**Per FIFO**
Logic Cells	1278	322	136
Dedicated Registers	689	82	117
M9K Memory Blocks	2	0	2
Multiplier Elements	4	4	0

3.9 Resource Utilisation

We chose to favour speed for this design as it makes for the most interesting optimisation trade-offs. That is, it is only when pressing for speed that one is required to think very hard about the critical path and the effects of pipe-lining. Furthermore, as the device has more than 22 thousand logic elements, we found it was only natural to favour speed over area as the target device has a lot of spare resources. That is, this filter module only consumes slightly over 5% of the resources available in the device.

Outlined in Table 12 is the resource utilisation per entity of the notch filter module. The first thing to note is that we use a large number of logic cells and registers. This is primarily due to the fact that the synthesis optimisations are weighted towards speed. Furthermore, register duplication, a physical synthesis optimisation, is enabled. That is, to enable register re-timing for high fanout signals, we allow for automatic generation of extra registers post-fanout.

An interesting note is that, as seen in Figure 16, we only require three multipliers. However, as the multipliers are arranged in pairs on this FPGA architecture, we end up using the next even number of multipliers, which in this case is four.

Also, we note that one way to cut down on the use of scarce resources, such as multiplier blocks, would be to use constant coefficient multipliers that are constructed out of adder trees. However, while there was not enough time to implement this interface, we intended the notch filter to accept new coefficients from the CPU. Thus, to facilitate this feature, we are forced to use real multiplier blocks.

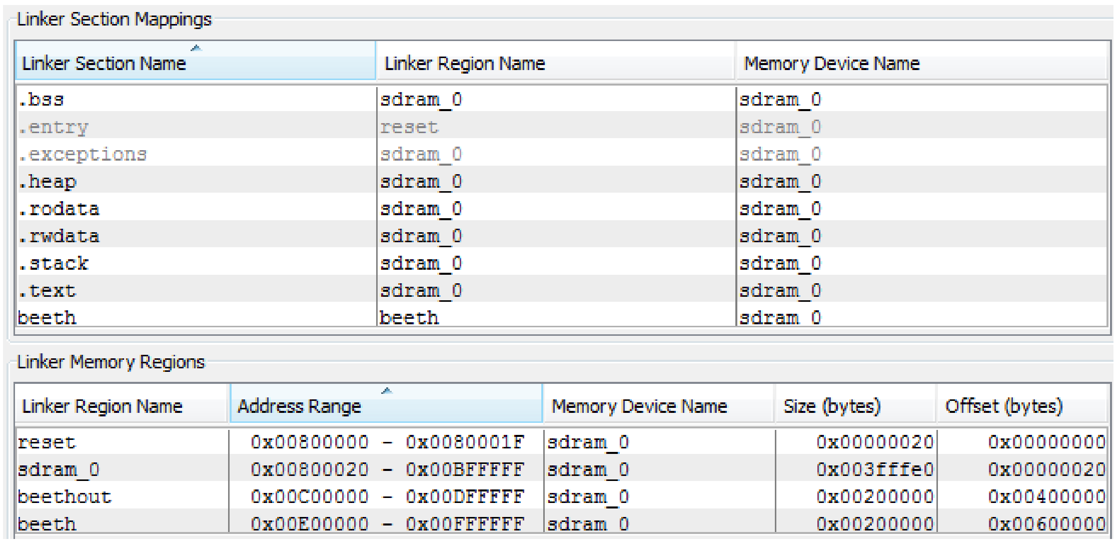
From a DSP perspective it is possible to use the Overlap-Save [8] algorithm to combine the outputs of two IIR filter cores that execute on different segments of the input data in parallel. However there would be no speed advantage in our case, as our implementation is already memory bandwidth limited with only one filter core, as seen in Table 11.

3.10 Operation

The audio data that will be filtered needs to be copied onto the SDRAM. We found that the easiest way to achieve this is to edit the linker script to reserve a memory region for the data. This is easily done using the Board Support Package Editor, as shown in Figure 17. Two megabytes is reserved for the input data, and another two megabytes for output data.

In order to get the 2MB data file onto the SDRAM, we define a linker section called `beeth`. We then generate a relocatable object file using the command line linker tool, and finally rename the automatically named data section to `beeth`. The commands are as follows:

```
nios2-elf-ld -r -b binary -o beeth.o beeth5_noise.bin
nios2-elf-objcopy --rename-section .data=beeth beeth.o beethsect.o
```


Linker Section Mappings

Linker Section Name	Linker Region Name	Memory Device Name
.bss	sdram_0	sdram_0
.entry	reset	sdram_0
.exceptions	sdram_0	sdram_0
.heap	sdram_0	sdram_0
.rodata	sdram_0	sdram_0
.rwdata	sdram_0	sdram_0
.stack	sdram_0	sdram_0
.text	sdram_0	sdram_0
beeth	beeth	sdram_0

Linker Memory Regions

Linker Region Name	Address Range	Memory Device Name	Size (bytes)	Offset (bytes)
reset	0x00800000 - 0x0080001F	sdram_0	0x00000020	0x00000000
sdram_0	0x00800020 - 0x00BFFFFF	sdram_0	0x003fffe0	0x00000020
beethout	0x00C00000 - 0x00DFFFFF	sdram_0	0x00200000	0x00400000
beeth	0x00E00000 - 0x00FFFFFF	sdram_0	0x00200000	0x00600000

Figure 17: The linker sections and memory regions defined in the linker script.

To link this object file to the rest of the system, we append the following to the end of the user-modifiable section of the Makefile:

```
# User object files
OBJS := beethsect.o
```

As this object file is linked with the rest of the system, the following symbols are automatically defined:

```
_binary_beeth5_noise_bin_start
_binary_beeth5_noise_bin_size
_binary_beeth5_noise_bin_end
```

This makes for seamless system integration, as these symbols can readily be used in the C code. Furthermore, the data itself will automatically be included in the .elf, and will as such automatically be downloaded to the target without requiring any extra user action.

We found that the easiest way to retrieve data from the system was to use the memory manipulation commands of GDB. That is, to download the memory contents to a file on the host machine, the following command is used in the GDB console:

```
dump binary memory <file> <start_addr> <end_addr>
```

3.11 Operational Results

We verify the operation of the hardware in both simulation and in hardware. Shown in Figure 18 are the simulation results of filtering during the first couple of microseconds of the operation of the filter. This filter response matches the expected response, and also matches the response generated in hardware, as shown in Figure 19. Shown in Figure 20 is the entire music sample after filtering. By a simple listening test, we also verify that no trace of the 1 kHz sine wave can be perceived, and no perceivable noise from the filtering operation is introduced.

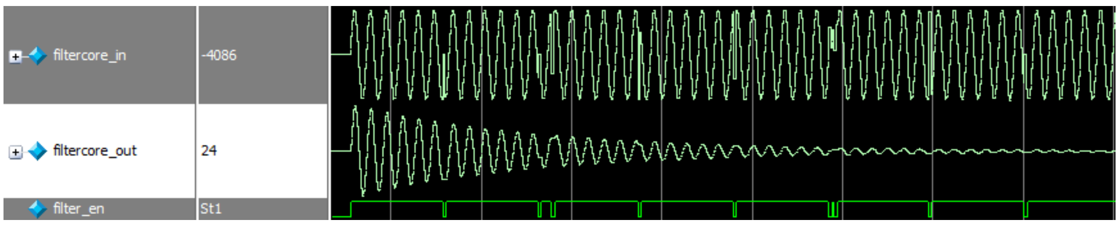


Figure 18: Simulation of filter in operation.

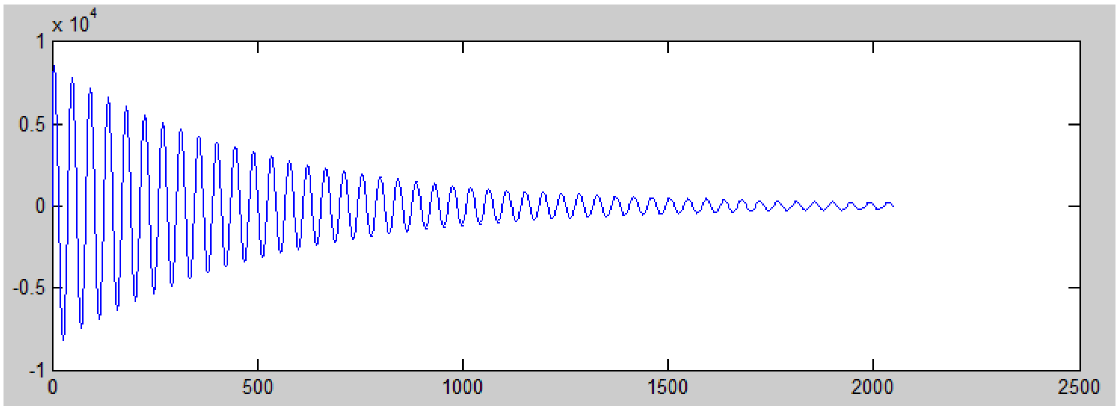


Figure 19: First 2048 output samples generated on hardware.

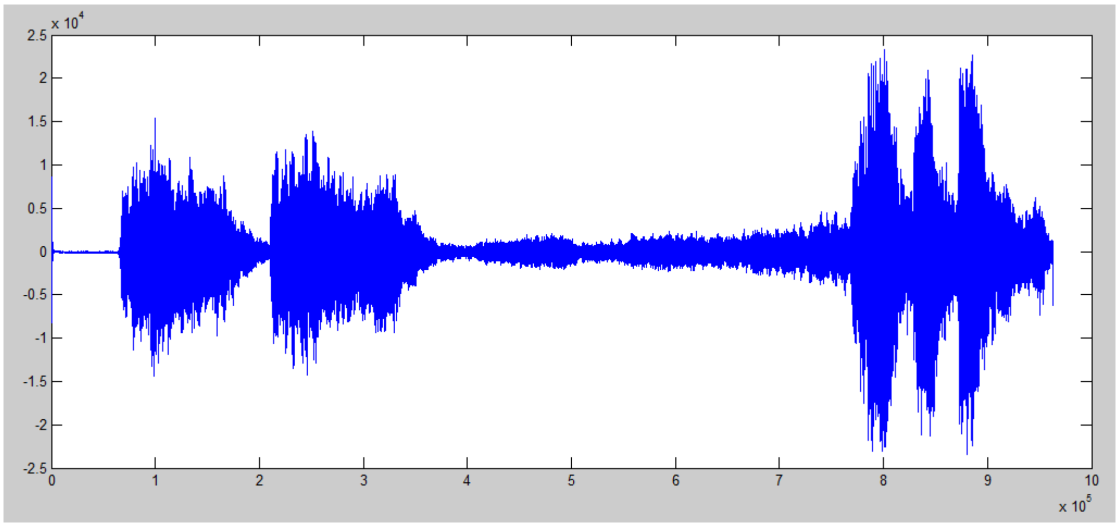


Figure 20: Output after filtering on hardware.

3.12 Future Improvements

The hardware was designed with the intention to be integrated with an interface to the CPU for updating filter coefficients. This was never implemented, so an obvious improvement would be to add this functionality. As the multiplier blocks have input registers with enable lines on them [7], it should be straightforward to memory map these registers to be accessible from the CPU interface.

We have designed a Pulse Width Modulation (PWM) audio output module. If implemented, this module would be able to accept 16 bit signed audio data through an Avalon Memory-Mapped Slave interface with only a single destination address. As such, only a slight modification of the main filtering block would be required to be able to stream the output of the filter to this audio output module instead of writing the result back to the SDRAM. The interface from the software's point of view would be the same, with the supplied destination address substituted for the PWM module's address.

References

[1] Wei Zhang, *Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver.* `http://www.eecg.toronto.edu/~weizer/LUgen/WeiZhang.pdf`

[2] Altera Corp, *Nios II Core Implementation Details.* `http://www.altera.co.uk/literature/hb/nios2/n2cpu_nii51015.pdf`

[3] IEEE, *IEEE Standard for Floating-Point Arithmetic.* `http://ieeexplore.ieee.org/stampPDF/getPDF.jsp?tp=&arnumber=4610935`

[4] Zentel Electronics Corp, *64Mb Synchronous DRAM Specification.* `http://tsl-ds.googlecode.com/svn-history/r23/SDRAM/A3V64S40ETP_(Zentel).pdf`

[5] Altera Corp, *Avalon Interface Specifications.* `http://www.altera.co.uk/literature/manual/mnl_avalon_spec.pdf`

[6] Altera Corp, *Memory Blocks in the Cyclone III Device Family.* `http://www.altera.co.uk/literature/hb/cyc3/cyc3_ciii51004.pdf`

[7] Altera Corp, *Embedded Multipliers in the Cyclone III Device Family.* `http://www.altera.co.uk/literature/hb/cyc3/cyc3_ciii51005.pdf`

[8] Dr Patrick A Naylor, *Digital Signal Processing Lecture Notes.* `https://bb.imperial.ac.uk/bbcswebdav/pid-84364-dt-content-rid-271372_1/courses/DSS-EE3_07-12_13/DSP3_Convolution.pdf`

[9] Altera Corp, *Floating-Point Megafunctions User Guide.* `www.altera.co.uk/literature/ug/ug_altfp_mfug.pdf`