Digital System Design Report

Oskar Weigl - ow610
and
Ryan Savitski - rs510

March 21, 2013

Abstract

Write the abstract here

Contents

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 software 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. Entries are time in milliseconds per 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

1 Floating Point Hardware

the matrix algorithm is constant for each size: 49 KB

2 Determinant Calculation Hardware Accelerator

PUT INTRODUCTION HERE!

2.1 Theory

This hardware module uses a modified version of the Doolittle Algorithm to compute the determinant of a matrix. We implement a

2.2 Interface Overview

reffering to Algorithm 1, we can identify the expected growth of different parts of the algorithm.

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 LU decomposition of a matrix, as found in [1], but modified to compute the determinant, and include pivoting.

Table 3: Resource Utilisation of the Determinant module. Note that two fp_mul units are used.

	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 4: The result of the cubic regression shown in Figure 2.

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

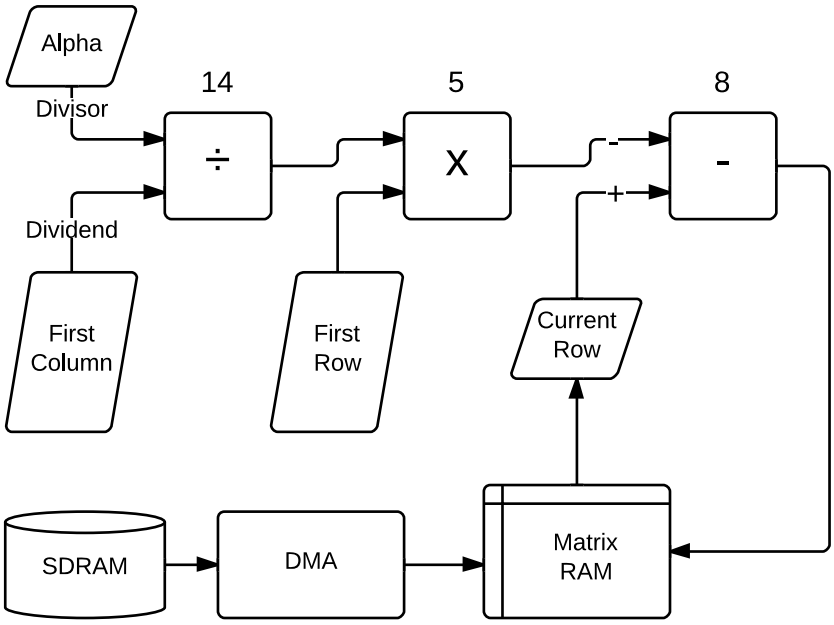

Figure 1: Block diagram of the hardware implementation of the Doolittle Algorithm. The number above the arithmetic blocks indicate latency.

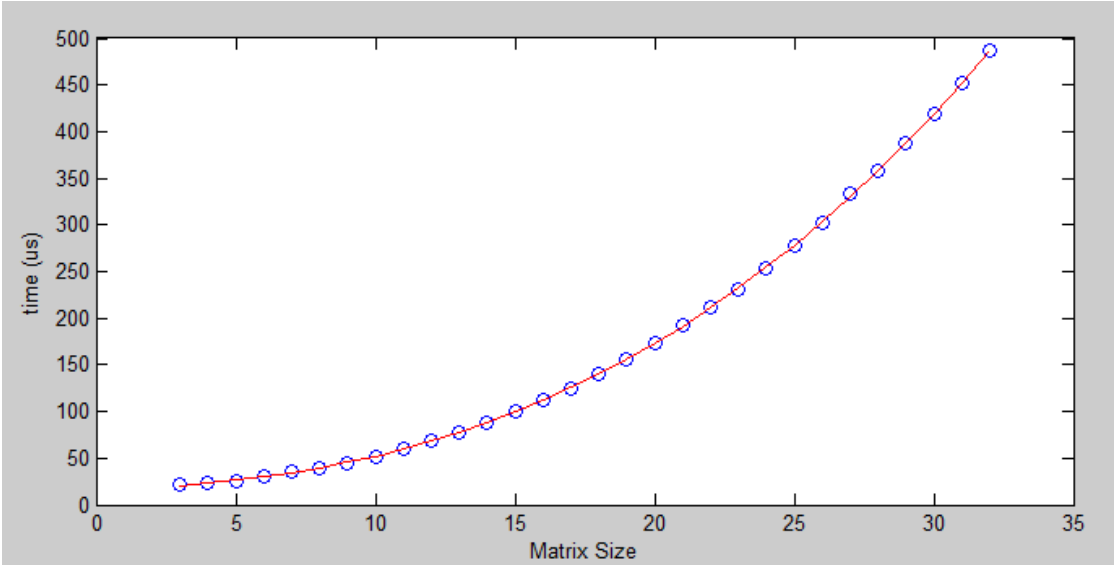

Figure 2: Benchmark results of the determinant hardware. Plotted in blue are the data points, and shown in red is a cubic regression, detailed in Table 4.

Table 5: Expected complexity of various operations in LU decomposition. Note that element multiplication and subtraction are pipelined and do not add.

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

3 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.

3.1 Filter Design

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 3.

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 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 of the resulting filter can been seen in Figure 4.

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 6, 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.

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

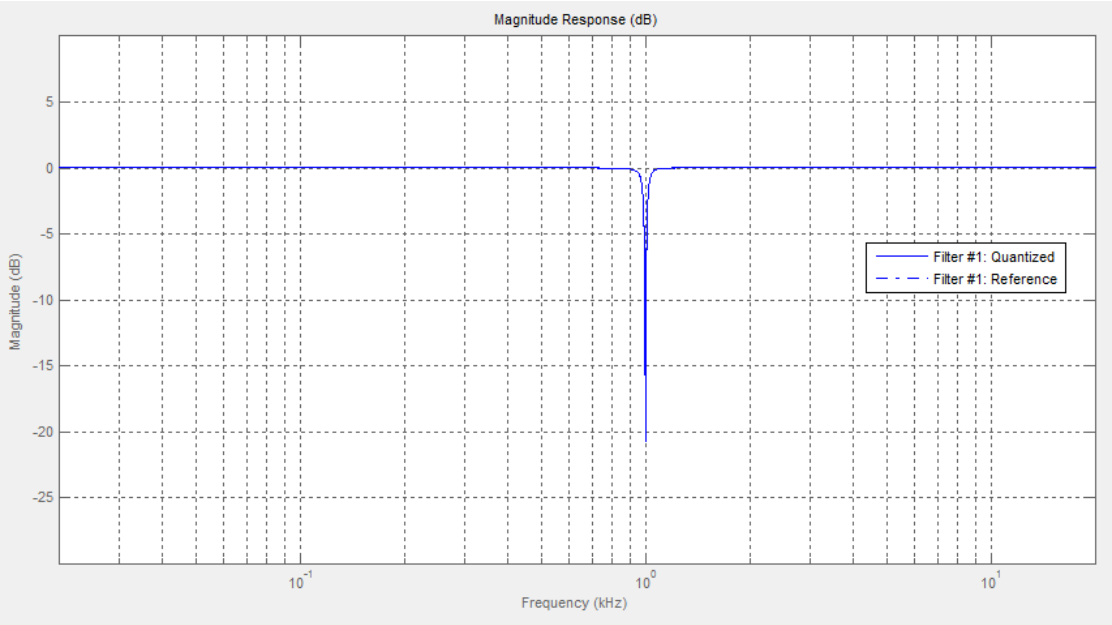

Figure 4: Magnitude response of both the full (double) precision and the quantised filter. The gain is negative infinity dB at 1 kHz.

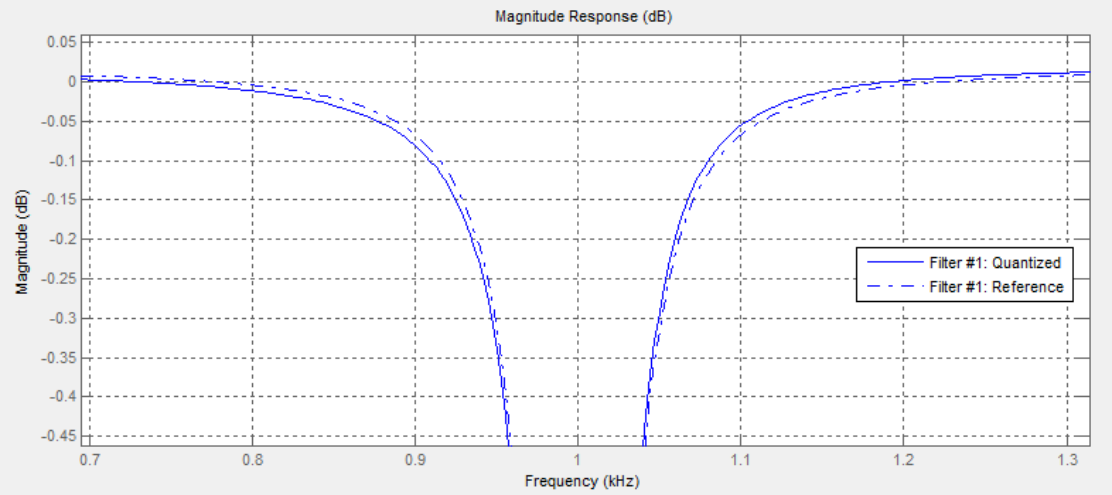

Figure 5: Magnitude response of both the full (double) precision and the quantised filter. The quantised and reference filters differ with less than 0.02dB in the passband.

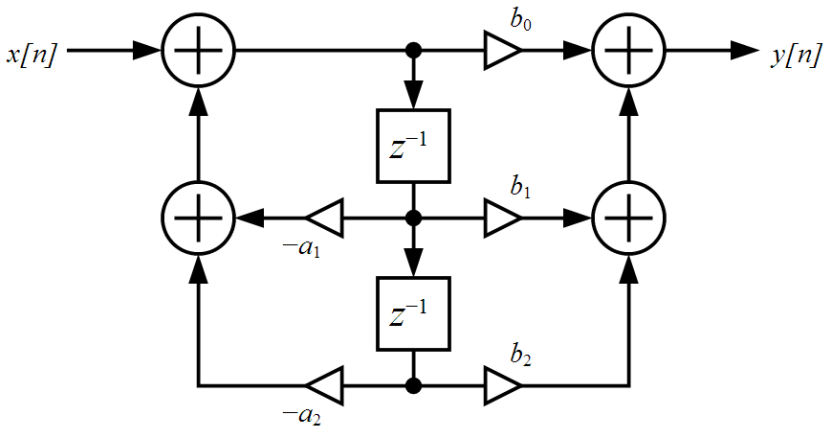

Figure 6: Second order Direct Form II filter topology.

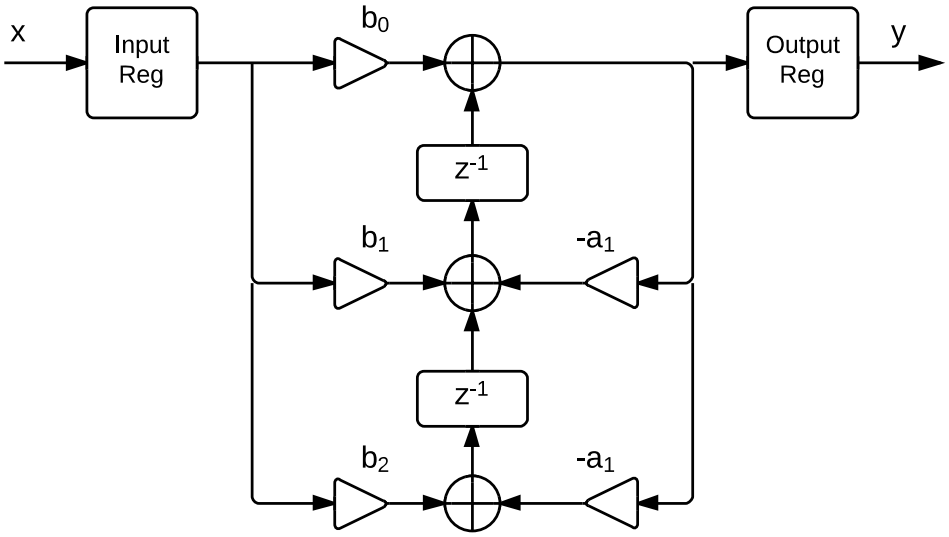

Figure 7: Second order Direct Form II Transposed filter topology.

```
   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 6: 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 7. Thus, for the critical path is constant for any order of the filter, and in all cases shorter than the non-transposed version. The timing of the 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 go 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.

During the initial design of the filter, it was established that a minimum of 12 bits was

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 7: Filtering Report generated when filtering the input sample

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 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 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. 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 7.

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 includes a feature for outputting the custom filter design as automatically generated Hardware Description Language (HDL). Using the `fdhdltool` command, we are able to generate Verilog HDL from the filter we have designed.

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.

This tool provides us with a Verilog source file with the interface as shown in Figure 8. The

```
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 8: The interface of the automatically generated filter.

filter input and outputs are signed 16 bit integer representation with no fractional parts, as required.

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, but we did not have time to implement it. As such, while it would be possible to have a fast system with variable coefficients, the system as it stands, has hard-wired coefficients.

3.5 Hardware Implementation

The filtering module, depicted in Figure 11, 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 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 9. 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 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.

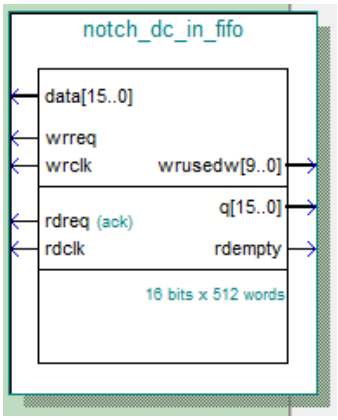

Figure 9: Block diagram of the First In First Out buffers used for both data buffering and Clock Domain Crossing.

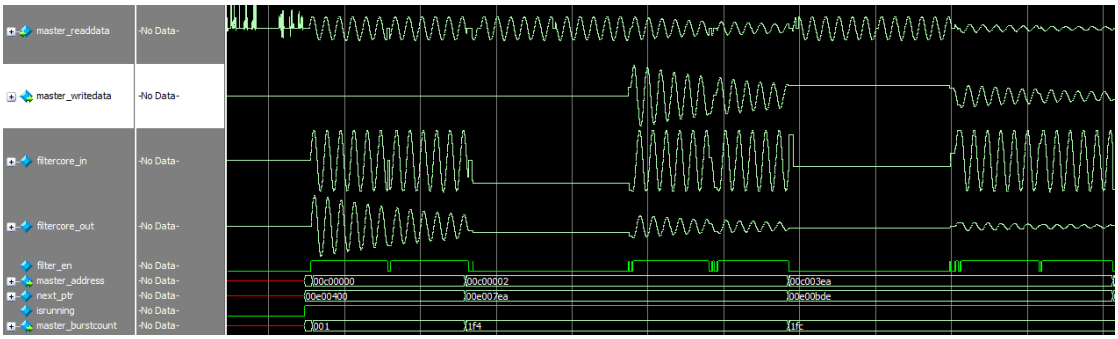

Figure 10: Simulated interleaved operation of the filter. Note that both Clock Domains are simulated at 50MHz for clarity.

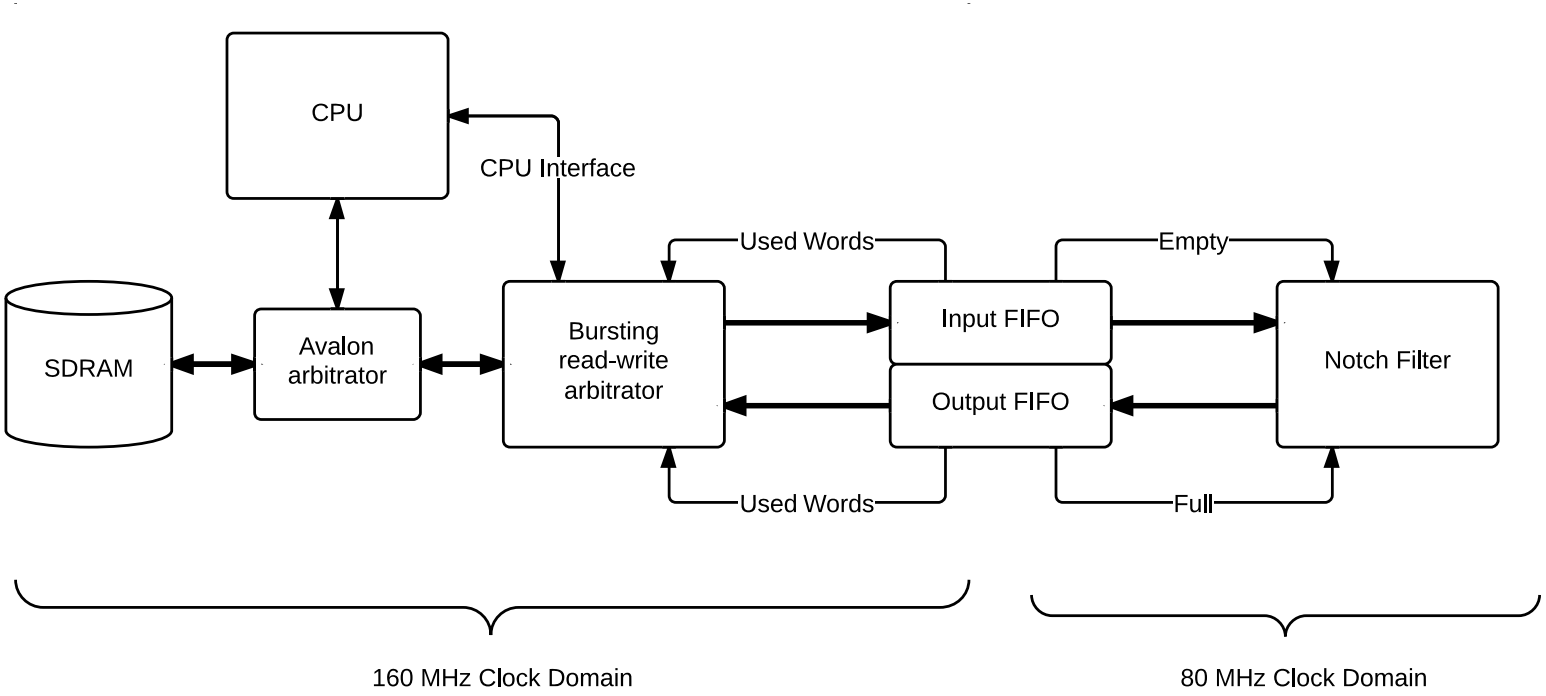

Figure 11: Block diagram of the Notch Filter hardware. FIFO buffers are used to allow efficient interleaved reads and writes to SDRAM. The buffers also serve as Clock Domain Crossing interfaces.

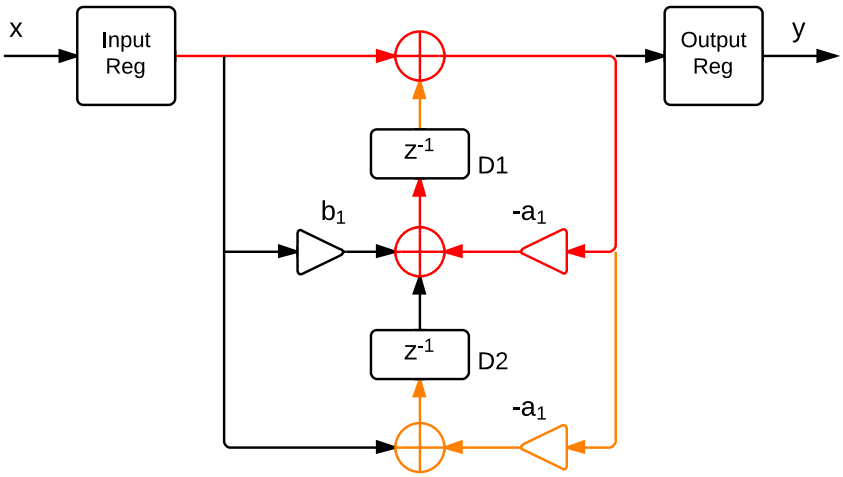

Figure 12: Annotated version of Figure 7, showing the main critical paths in the filter core. Shown in red is the critical path; from the input register to the delay register D1. Shown in orange are the alternative paths that have almost as little slack as the red path.

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 12, 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 pipe-lining, 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 12, 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 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.

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 12. 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 similarity of the timing of these paths is expected expected when inspecting the data-paths in a Direct Form II Transposed filter topology. That is, as seen in Figure 12, 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 8: 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 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.

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, initially, 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 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 9: 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 10: 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 10 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 12, 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.

3.10 Operation

The audio data that will be filtered needs to be copied onto the SDRAM. We found that the smoothest way to achive 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 13. 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
```

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
```

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

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 14 are the simulation results of filtering during the first couple of microseconds of the operation of the filter. This represents several thousand data samples. This filter response matches the expected response, and also matches the response generated in hardware, as shown in Figure 15. Shown in Figure 16 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.

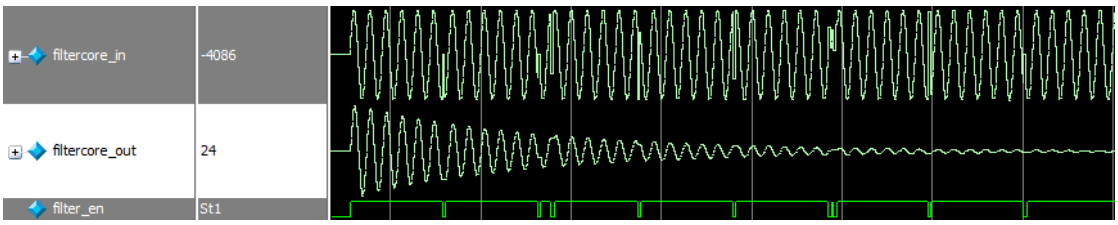

Figure 14: Simulation of filter in operation.

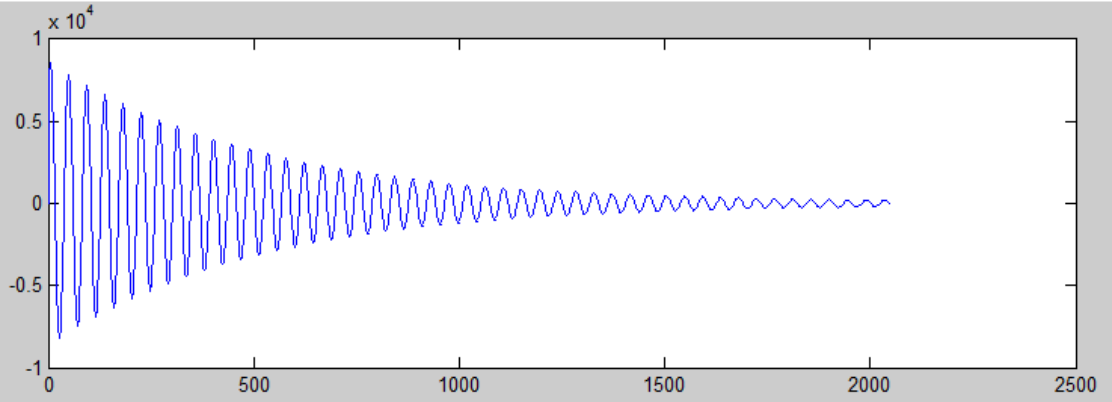

Figure 15: First 2048 output samples generated on hardware.

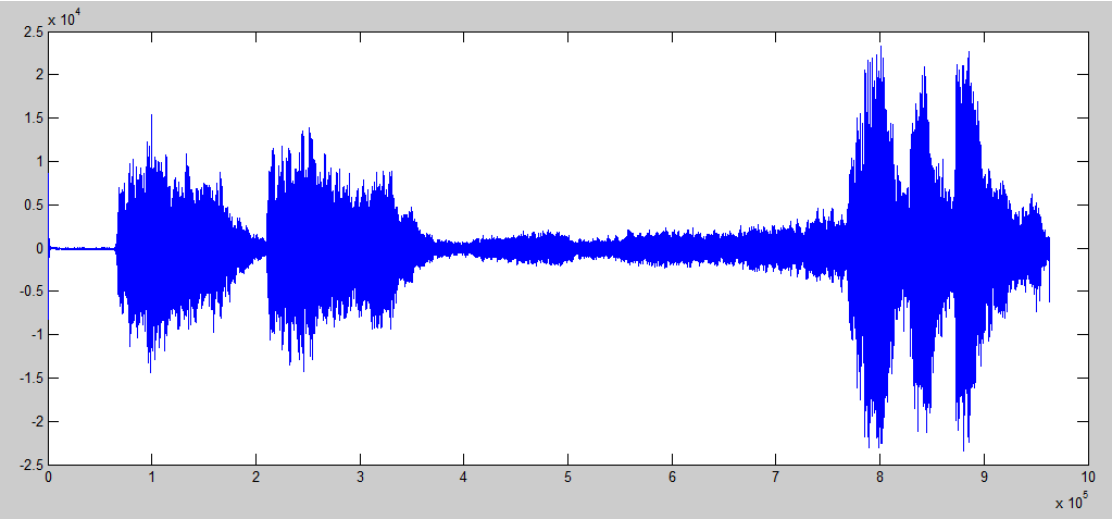

Figure 16: Output after filtering on hardware.

3.12 Future Improvements

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`