CSSE4010 A4

David Gaul

s4671313

Lab Session: Wednesday 10-12

Submission Date: 16th-ish October

1 Introduction

The purpose of this practical was to develop a finite impulse response (FIR) filter using Xilinx's Model Composer and test it's functionality. The filter was a low pass filter of the form

$$y(n) = \sum_{i=0}^{17} a_i x(n-i)$$

This filter was first developed using a filter developed through Matlab's built-in functions. Following this, two pipelines were developed for hardware equivalents: one with and the other without optimisation strategies. From here, the performance metrics of both filters were compared against each other and the software implementation.

I'm going to acknowledge that the lab reports have reduced in quality as the semester has progressed. My apologies. The quality of said reports is directly proportional to my will to live. But today we're going to have fun.

2 Word Length Selection

Before any hardware implementations could be commenced, it was necessary to determine the words lengths being used for fixed point decimal values. Fixed point decimal values are preferred over floating point representations because they save on space, while also improving time complexity. A properly selected fixed point word length can boost the running speed, reduce resource consumption and result in a minimal loss of accuracy. For the purposes of prac 5, two word lengths had to be decided: the word length for the input signal and filter co-efficients.

Deciding the word length for the input signal required close analysis of the input itself. As was observed, the input was the combination of two sinusoidal signals with the functions:

$$y_1 = \sin(400\pi t)$$
$$y_2 = 4\sin(4000\pi t)$$

The minimum and maximum values of the signal were found using Matlab's min() and max() functions. These were found to be -4 and 4 respectively. In order to fully express these inputs, 4 bits is required: three bits to express '4,' and a final bit for the 2's complement.

Similarly, the minimum and maximum values of the filter co-efficients were analysed. These were found to be -0.002 and 0.1634. Only a single bit is required to before the floating point to express these: the bit for 2's complement signing. Ultimately, it was decided that the input signal would require 4 bits on the integer value of its word length, whereas the co-efficients required only 1 bit. The total word lengths of each was tested within hardware filter 1 implementation, and will be discussed within that section.

3 Hardware Filter 1

Although both hardware implementations show many similarities, it is worth beginning with the initial software filter to explain each architecture. In implementing the software filter via Matlab, it returned a 17th degree filter, where the filter co-efficients can be found in figure 1.

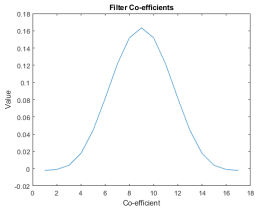


Figure 1: Polynomial co-efficient values for FIR Filter

Transferring this simple model over to Model Composer was more or less a drag and drop process. The entire block diagram for the un-optimised filter can be found in figure 2.

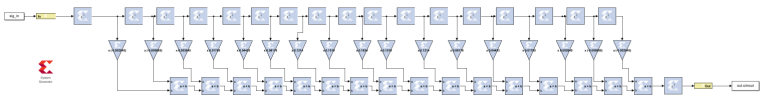


Figure 2: Block diagram of the untuned filter

Here, it's worth mentioning that the design architecture perfectly mimics that of the provided equation at the introduction of the report. Each delay block represents a delay of Z^{-1} in the Z-domain, which correlates to $x(n-1)$ in the time domain. The stacking of these delays, in combination with the summation of filter co-efficients, results in a hardware equivalent to the software implementation within the provided Matlab script. Figure

3 shows a zoomed-in view of the left-most co-efficients, which correspond with the first and second filter polynomials.

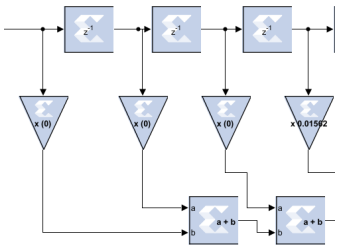


Figure 3: Zoomed in view of the first two co-efficients for the untuned filter

With the design successfully set up, the full word lengths for the input and filter co-efficients could be determined. To determine the accuracy of hardware filter implementations, a signal to error ratio (SER) function was used. The most common SER function is:

$$SER = 10 \times \log_{10}(\frac{(HW - SW)^2}{SW^2})$$

Where HW represents the hardware signal, and SW represents the software signal.

In analysing the array of filter co-efficients, it was found that the smallest value was $8.914e^{-}4$. Such a number would require 14 bits to fully represent, which would indicate that a 16 bit word length would provide the most accurate results possible. However, it was acknowledged that smaller word lengths could be used with minimal loss in accuracy. To fully test the effect of word length on filter accuracy, tests were conducted over the number of bits representing the decimal value. Bit values from 3 through to 15 were used, and the results can be found in figure 4

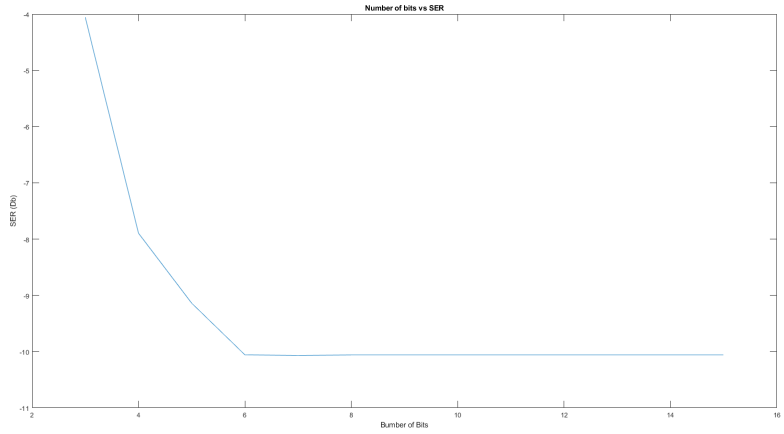


Figure 4: Line plot showing the reduction in SER as the number of decimal bits on the CMUL blocks increases

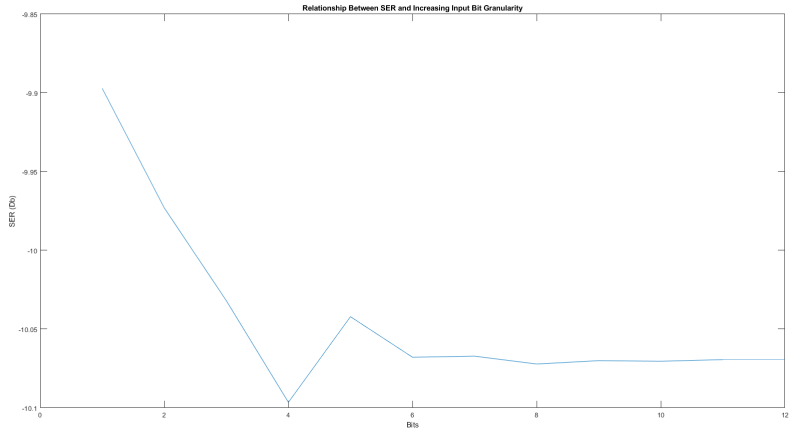


Figure 5: Line plot showing the reduction in SER as the number of decimal places on the input signal in increases

As can be seen, minimal accuracy is gained by adding more values past 6 decimal places. Conversely, it does add more complexity and runtime. For this reason, the word-length of the filter co-efficients was determined to be 7: 1 bit for the signed value, and 6 bits for the decimal value. Through similar testing, it was found that the optimum word length for the input signal was 8 bits: 4 for the integer value and 4 for the decimal value. The derivation of SER values for the input word length can be found in figure 5

The optimisation of these final values can be verified by comparing the obtained hardware signal to the original software filter. Figure 6 shows that the solutions are nearly identical.

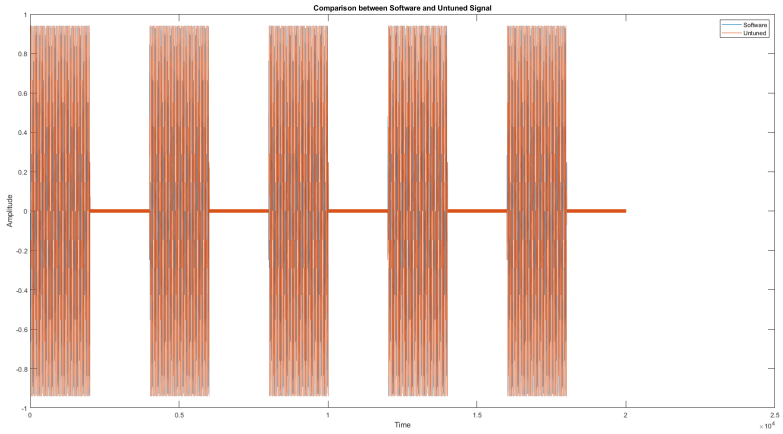


Figure 6: Comparison of software and untuned hardware filters

Close inpection of these signal reveals that there is a delay on the hardware signal. This is due to the implementation of delay blocks within the Simulink model. As these are mandatory to retain system synchronicity, this delay is unavoidable. Figure 7 shows a zoomed-in view of this delay.

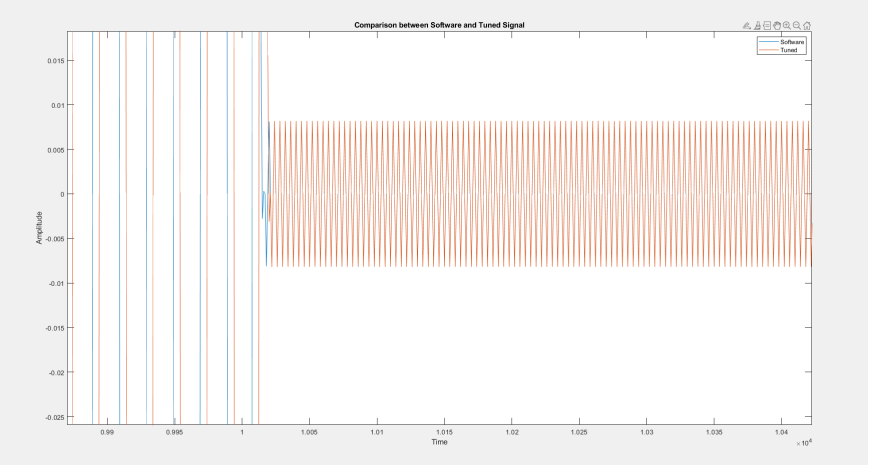
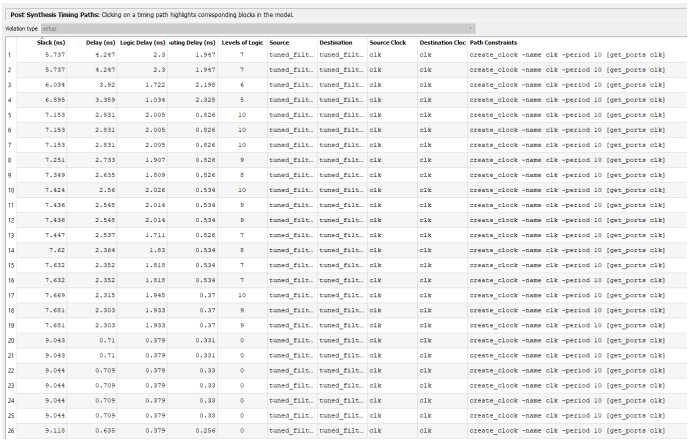


Figure 7: Zoomed in view of software and untuned filter comparison. This highlights the slight delay present for the hardware filter

With successful verification of the hardware implementation's functionality, its runtime metrics could be recorded. Figure 8 shows the timing schedule for model synthesis. As could be seen, all pathways were able to run in under 5ns and model synthesis was successful. The critical path can be observed in figure 9



Post Synthesis Timing Paths: Clicking on a timing path highlights corresponding blocks in the model.

Violation type: setup

	Slack (ns)	Delay (ns)	Logic Delay (ns)	uting Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Cloc	Path Constraints
1	5.737	4.247	2.3	1.947	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
2	5.737	4.247	2.3	1.947	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
3	6.034	3.92	1.722	2.198	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
4	6.595	3.359	1.034	2.325	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
5	7.153	2.831	2.005	0.826	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
6	7.153	2.831	2.005	0.826	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
7	7.153	2.831	2.005	0.826	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
8	7.251	2.733	1.907	0.826	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
9	7.349	2.635	1.809	0.826	8	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
10	7.424	2.56	2.026	0.534	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
11	7.436	2.548	2.014	0.534	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
12	7.436	2.548	2.014	0.534	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
13	7.447	2.537	1.711	0.826	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
14	7.62	2.364	1.83	0.534	8	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
15	7.632	2.352	1.818	0.534	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
16	7.632	2.352	1.818	0.534	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
17	7.669	2.315	1.945	0.37	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
18	7.681	2.303	1.933	0.37	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
19	7.681	2.303	1.933	0.37	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
20	9.043	0.71	0.379	0.331	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
21	9.043	0.71	0.379	0.331	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
22	9.044	0.709	0.379	0.33	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
23	9.044	0.709	0.379	0.33	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
24	9.044	0.709	0.379	0.33	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
25	9.044	0.709	0.379	0.33	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
26	9.118	0.635	0.379	0.256	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]

Figure 8: Timing results of the untuned filter. Note that all paths have run in passing time

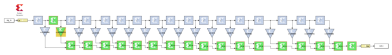


Figure 9: Critical path for the untuned model

The memory utilisation of the model was found by copying and synthesizing the generated .vhdl code created by model composer. The utilisation report can be found in figure 10

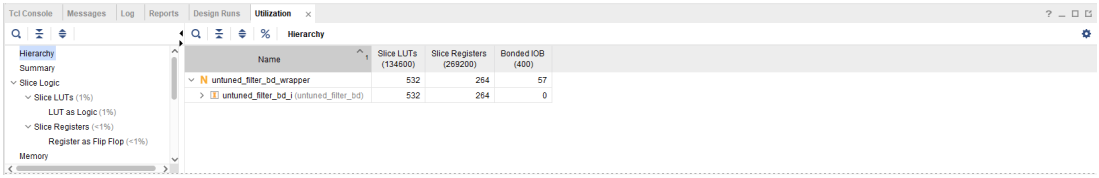


Figure 10: Memory utilisation of the untuned filter

At the end of the first hardware filter, it was found that it could almost perfectly replicate the software filter with a critical path delay of 4.3ns and a memory utilisation of 532 LUTs: less than 1% of the available resources.

4 Hardware Filter 2

Although it was established that the first pipeline worked, there were several optimisation issues. Due to the lack of optimisation, the filter ran slower and consumed more resources than was necessary. Careful analysis showed that there were two main ways of optimising this initial filter:

- Pipelining - Used to parallelise computations and reduce reliance on delays.
- Data broadcasting - Using the same filter co-efficients in multiple areas.

Data broadcasting was possible due to the nature of the filter co-efficients. Figure 1 shows that a mirror image of co-efficient values is created by the software filter. Computed values could therefore be recycled and implemented later in the FIR filter. Finally, delay blocks could also be removed by integrating them into the CMUL and AddSub blocks. The fully optimised hardware filter can be found in figure 11.

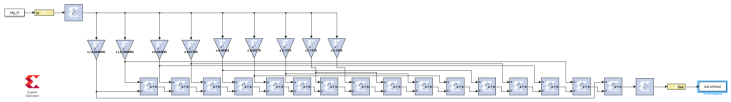


Figure 11: Block diagram of tuned filter

Figure 12 shows a close-up view of the first two co-efficients for this block diagram.

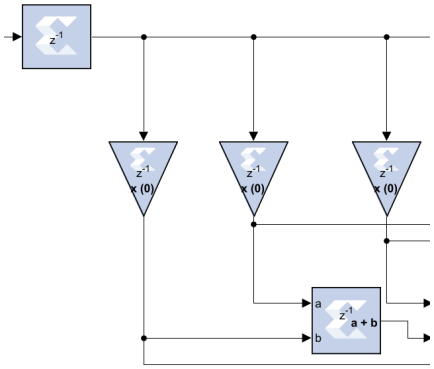


Figure 12: Close up view of the first two filter co-efficients for the tuned filter

Since the values for the input signal and co-efficients remained unchanged between filter iterations, updating word lengths was unnecessary.

Which can be further verified by figure 13 which compares the tuned filter to the original software implementation.

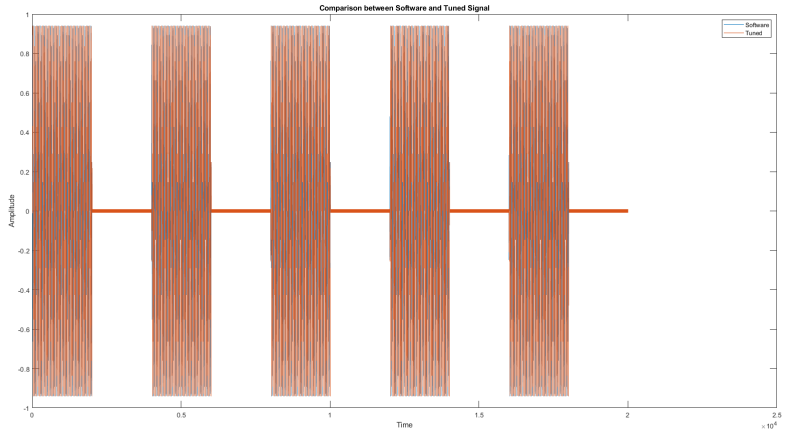


Figure 13: Output comparison between the software and tuned hardware filters.

However, a zoomed in view of the output indicates that there is more delay in the output. This is due to the integration of latency within the CMUL and AddSub blocks. Because of this, the critical path delay had been changed. Figure 14 shows the timing report of Model Composer synthesizing.

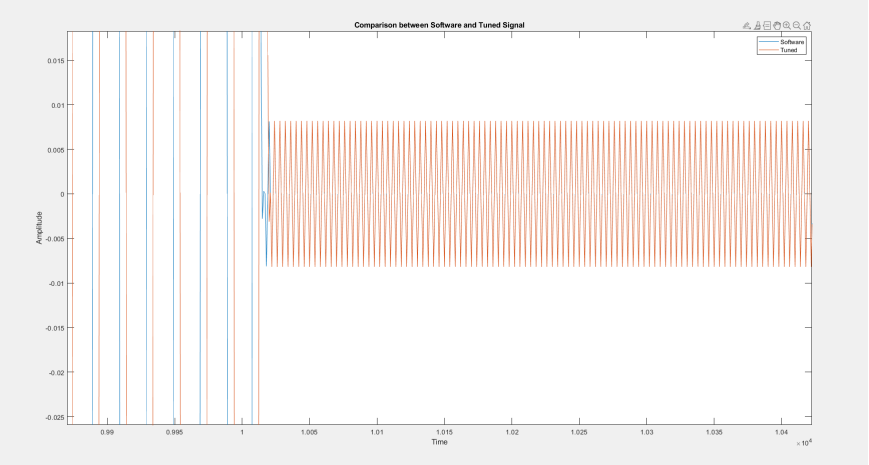
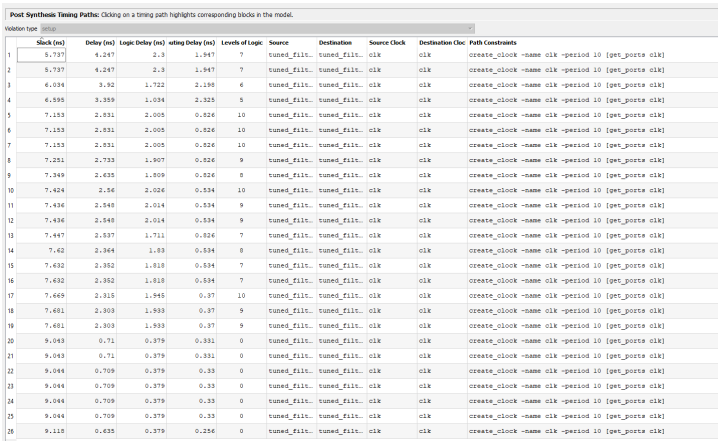


Figure 14: Zoomed in view of the software and tuned hardware filters highlighting the delay between them

Although the critical path remained unchanged for the tuned filter, the average running speed for each path was increased. The critical path delay for the tuned pipeline could be found within figure 15.



Post Synthesis Timing Paths: Clicking on a timing path highlights corresponding blocks in the model.

Violation type: setup

	Slack (ns)	Delay (ns)	Logic Delay (ns)	eating Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Cloc	Path Constraints
1	5.737	4.247	2.3	1.947	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
2	5.737	4.247	2.3	1.947	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
3	6.034	3.92	1.722	2.198	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
4	6.595	3.359	1.034	2.325	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
5	7.153	2.831	2.005	0.826	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
6	7.153	2.831	2.005	0.826	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
7	7.153	2.831	2.005	0.826	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
8	7.251	2.733	1.907	0.826	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
9	7.349	2.635	1.809	0.826	8	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
10	7.424	2.56	2.026	0.534	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
11	7.436	2.548	2.014	0.534	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
12	7.436	2.548	2.014	0.534	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
13	7.447	2.537	1.711	0.826	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
14	7.62	2.364	1.83	0.534	8	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
15	7.632	2.352	1.818	0.534	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
16	7.632	2.352	1.818	0.534	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
17	7.669	2.315	1.945	0.37	10	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
18	7.681	2.303	1.933	0.37	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
19	7.681	2.303	1.933	0.37	9	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
20	9.043	0.71	0.379	0.331	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
21	9.043	0.71	0.379	0.331	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
22	9.044	0.709	0.379	0.33	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
23	9.044	0.709	0.379	0.33	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
24	9.044	0.709	0.379	0.33	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
25	9.044	0.709	0.379	0.33	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
26	9.118	0.635	0.379	0.256	0	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]

Figure 15: Total timing results for the tuned filter. All paths are able to evaluate within passing time.

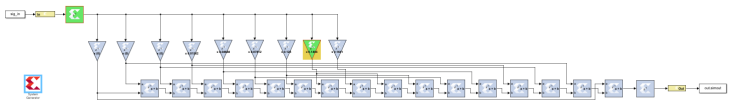
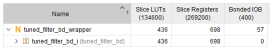


Figure 16: Critical path delay of the tuned system

Lastly, the memory utilisation for the tuned model can be found within figure 17



Name	Slice LUTs (134600)	Slice Registers (269200)	Bonded IOB (400)
tuned_filter_bd_wrapper	436	698	57
tuned_filter_bd_i (tuned_filter_bd)	436	698	0

Figure 17: Memory consumption of tuned filter

5 Model Comparison

After comparing both models, several notable observations surfaced. The first of these is that both the tuned and untuned filters were able to filter equally as accurately on the input signal. However, the tuned filter was able to outperform the untuned implementation in terms of runtime and area complexity. In implementing pipelining and broadcasting, the memory utilisation was able to be reduced from 532 to 436 LUTs; a 20% reduction in memory utilisation. Furthermore, although the critical path delay remained unchanged between implementations, the tuned model generally had on average less delay and more slack for each of its paths.

In total, optimising the model reduced memory utilisation while increasing running speed.

6 Conclusion

In conclusion, the system worked as expected. It was found that, quite unsurprisingly, a tuned filter can outperform un-optimised variants in terms of runtime and memory utilisation, with no loss in accuracy. Furthermore, hardware implementations can have the word lengths of their inputs adjusted to further increase this speed and memory optimisation. This, however, comes at the cost of reducing accuracy. Careful tuning is necessary to find the perfect balance between these parameters.

Anyway, job done. Product worked. This is the final prac report required for my time in CSSE4010. Thanks for marking. God bless.

xoxo