ECE574 Final Project Report: Reconfigurable Partially Pipelined RAM-based FIR Filter

Brian B Cheng
Rutgers University Department of Electrical and Computer Engineering

1 Abstract

This report presents a Verilog implementation of a reconfigurable Finite Impulse Response (FIR) filter designed for heterogeneous FPGAs. The model is highly flexible, with customizable data width, filter depth, and the number of parallel multiply-accumulate (MAC) pipelines, all configurable through top-level parameters. The filter modifies the traditional direct-form FIR structure by replacing the input shift register with RAM elements and the stationary weights with ROM elements. Input samples are written to RAM as new samples arrive, with the write address counting downward. Concurrently, weights and samples are read from the ROMs and RAMs into their respective MAC units, with read addresses counting upward. This counter-directional address counting achieves the "shift register" effect of a traditional direct-form filter. The method is demonstrated with a 256-tap FIR filter featuring a 24-bit data width and eight parallel MAC pipelines. A testbench simulation provides a noisy sine wave as input, demonstrating the filter's performance in producing a clean sine wave output, verified in the waveform diagrams.

2 Introduction

Finite Impulse Response (FIR) filters are commonly used in digital signal processing to remove unwanted frequencies from a noisy signal while preserving the desired frequency bands. They are particularly well-suited for implementation on FPGAs due to their unique balance between low latency, parallelism, and hardware flexibility. When implementing FIR filters on FPGAs, several key design constraints must be considered. Hardware constraints include the number utilization of multipliers (which are costly), adders, and registers used. Performance constraints include throughput, output latency, and placement area. Efficient design requires careful trade-offs between these constraints to meet the application's performance and resource requirements. This report presents an FIR filter that prioritizes placement area by opting for

RAM and ROM elements instead of registers, and sacrifices low-latency for higher throughput via extensive usage of parallel pipelining. At the same time however, the usage of RAM and ROM instead of registers hinders the higher throughput as memory access throughput is slower than register throughput. Thus, the main benefit of this implementation is the placement profile on the FPGA chip. A smaller placement profiles can yield fewer wire nets and less congested routing, which may also contribute to lower power usage.

3 Data Description

As mentioned in the Introduction, the purpose of a FIR Filter is to filter out unwanted frequency bands from an input signal and provide an output signal containing only the desired frequency bands. This project presents a low pass filter with a cutoff frequency of 2KHz. Thus, to verify the filter, we supply a synthetic noisy signal containing frequencies below and above 2KHz and observe the output signal to see if the filter correctly filters out frequencies above 2KHz. To create this synthetic signal, we use Python to generate a Verilog header file containing the sine signal samples in hexadecimal format. We use the `` `include "filename.vh" `` compiler directive in our testbench SystemVerilog code to import the Python-generated input samples. All data signals in this implementation are 24-bit signed with fixed point representation Q1.23, meaning 1 sign bit, zero whole bits, and 23 fractional bits. Thus, the range of all data signals, including the weights, is between negative one and positive one. Shown in Equation 1 and Figure 1 is the formula and graph of the noisy sine wave, with the desired 200Hz sine wave added with an unwanted 5KHz sine wave noise. The sampling frequency is 44 kHz, meaning the sampling period is 22.273×10^{-6} seconds.

$$\text{input\_signal}(t) = 0.2 \cdot \sin(2\pi f_{\text{low}} t) + 0.1 \cdot \sin(2\pi f_{\text{high}} t) \tag{1}$$

$$f_{\text{low}} = 200\,\text{Hz}, \quad f_{\text{high}} = 5\,\text{KHz}$$

In similar fashion, we generate the 256 filter weights in Python using scipy's `firwin()` function. We store the weights in hexadecimal format in `.mem` files, one `.mem` file for each MAC pipeline. Each MAC operates on their unique subvector of the whole weight vector. We use Xilinx's XPM macro to instantiate ROM elements and initialize the memories with their respective `.mem` files. It is generally good practice to use vendor-specific macros instead of hand written memories to ensure correct memory behavior. Shown in Figure 2 below is the function call and plot of filter weights.

```
filter_depth = 256
cutoff_freq = 2000
sample_rate = 44000
normalized_cutoff = cutoff_freq / (sample_rate / 2)
```

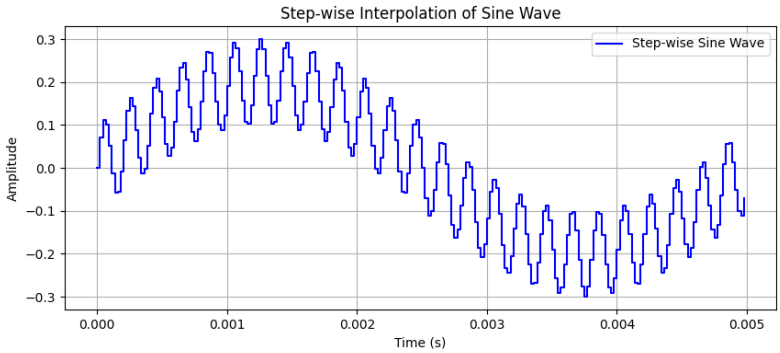

Figure 1: Noisy input signal over 220 samples

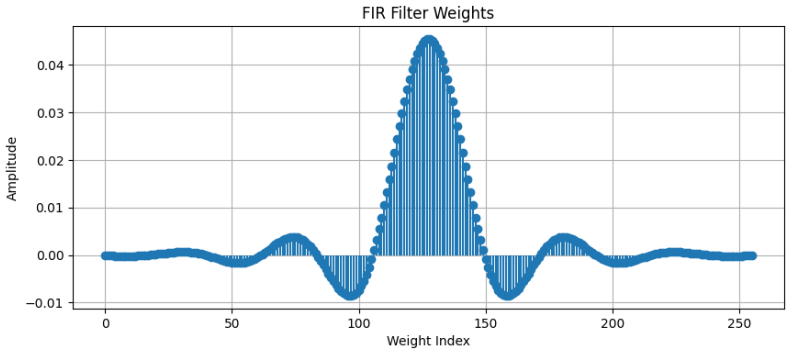

Figure 2: 256 filter weights for 44KHz sampling frequency and 2KHz cutoff frequency

```
firwin(filter_depth, normalized_cutoff)
```

4 Method Description

4.1 Why Choose a RAM-Based Shift Register Over a Flip-Flop-Based Shift Register?

A critical decision in this design is replacing a traditional flip-flop-based shift register with a RAM-based implementation. This choice is motivated mainly by reducing CLB usage in favor of BRAM and reducing the area profile of the shift register. This, I theorize, improves the placement profile on the FPGA chip, and reduces the total number of wire nets which can alleviate routing

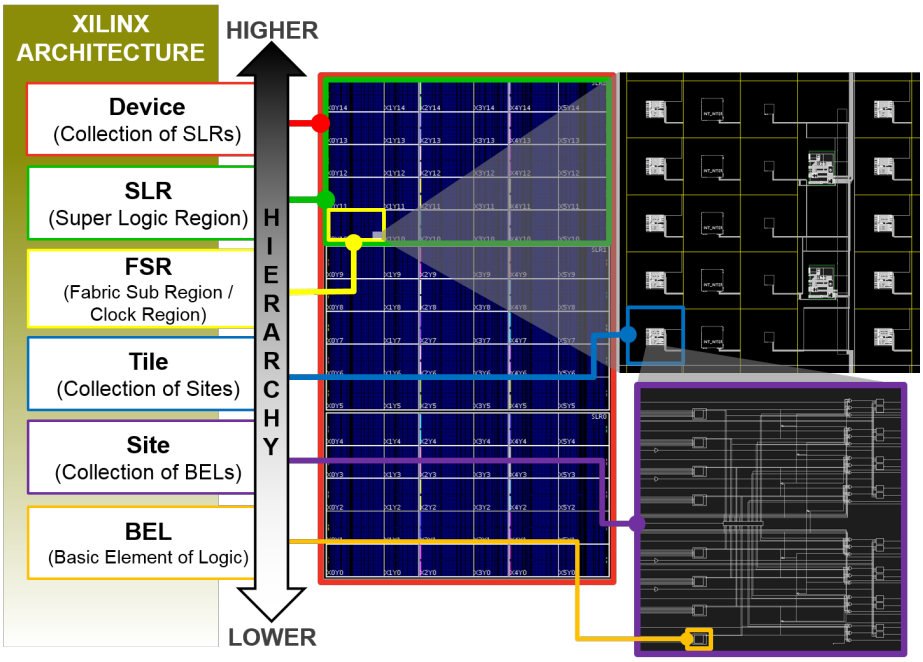

Figure 3: Xilinx FPGA Architecture Hierarchy

congestion in the programmable interconnects. These two benefits overall contributes to a better Quality of Result (QoR) given that the performance specifications are met. The usage of BRAMs over registers can also theoretically reduce power usage as vendor-specific memory macros are highly optimized in modern FPGAs.

For a 24-bit signal, each filter tap requires 24 registers. For instance, implementing a 256-tap FIR filter with 24-bit signals demands:

$$256\,\text{taps} \times 24\,\text{bits/tap} = 6144\,\text{flip-flops}.$$

In a typical low-end Xilinx FPGA, such as the xc7z020, the available resources include **6650 configurable logic block (CLB) tiles**. Each CLB contains two logic sites, which can either be both SLICELs or a combination of one SLICEL and one SLICEM. The key difference between SLICEL and SLICEM is that SLICEM can reconfigure its lookup tables (LUTs) to function as small local RAMs, known as "distributed RAM." The xc7z020 provides a total of **8950 SLICELs** and **4350 SLICEMs**, summing to **13,300 logic sites** across all CLBs. Each logic site, regardless of type, contains **8 flip-flops**, meaning each CLB tile has **16 flip-flops**. To implement the shift register using flip-flops:

$$\frac{6144\,\text{flip-flops}}{16\,\text{flip-flops/CLB}} = 384\,\text{CLBs}.$$

On the other hand, block RAM (BRAM) tiles provide a more resource-efficient alternative. Each BRAM tile contains **3 memory sites** ($2 \times$ 18Kb banks and $1 \times$ 36Kb bank) that can be configured as RAM, ROM, or FIFO. A single BRAM tile occupies approximately the same space as **10 CLBs**. For an 8-pipeline FIR filter, the design requires **8 RAMs** (for input samples) and **8 ROMs** (for weights), totaling **16 memory elements**. With **3 memory sites per BRAM tile**, this requires:

$$\frac{16\,\text{memory elements}}{3\,\text{memory sites per BRAM}} = 6\,\text{BRAM tiles.}$$

In terms of CLB-equivalent space, these 6 BRAM tiles occupy:

$$6\,\text{BRAM tiles} \times 10\,\text{CLBs/BRAM} = 60\,\text{CLBs.}$$

By opting for a RAM-based shift register, the design trades the **384 CLBs** required for a flip-flop-based implementation for just **60 CLBs**, resulting in a significant reduction in logic site utilization. This trade-off demonstrates the value of leveraging the specialized memory resources available on FPGAs, such as BRAM, to implement resource-intensive designs efficiently. With a smaller floorplanning profile (also known as a PBlock size), we can dedicate the freed up logic sites to other parts of the system design. This also allows us to pack more FIR filters in parallel should the system design operate on multiple signal channels. Refer to Figure 4 to see a space profile comparison between BRAMs and CLBs. All of this of course comes with the tradeoffs mentioned earlier, and obviously, if the system is memory intensive elsewhere in the design, the RAM-based FIR filter might not be feasible.

5 Model Description

All Verilog source codes for the implementation can be found in the `/src/` directory. Refer to Figure 5 for an RTL elaboration schematic for the top_level module. The FIR Filter module on its own communicates via multiple 24-bit data buses. Since an FPGA package only has a limited amount of IO pins, we implement serializer-deserializer (SERDES) modules to communicate serially over a simple AXI-like handshake protocol. Observe from the top_level schematic from left to right: input ports which correspond to physical IO pins on the chip, the deserializer, the fir filter itself, the serializer, and finally, the output ports.

Shown in Figure 6 is a block diagram showing the expandable ROM-RAM-MAC structure for the MAC pipelines. In the figure are only the first 3 pipelines, but the pattern for how to expand it to the necessary number of pipelines via should be clear. The partial sums on the right-side are summed concurrently in an adder chain. Further improvements can be made here in the final summation such as using a binary adder tree.

The full RTL elaborated schematic of the exploded FIR filter module is too large to fit on one screen, but to give an impression, shown in Figure 7 is a closeup of the filter with one pipeline highlited in dark-blue border.

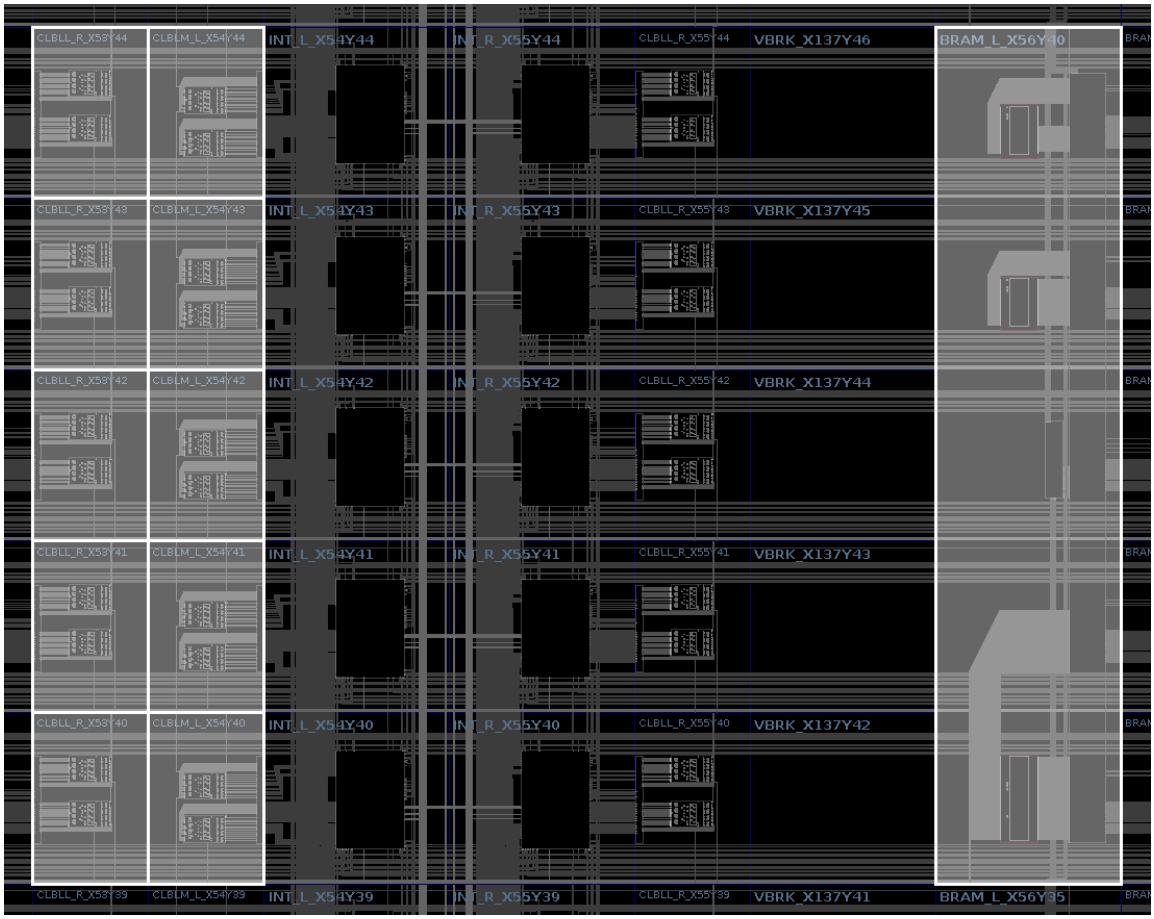

Figure 4: CLB vs BRAM area profile comparison on the xc7z020. 1 BRAM tile highlighted on the right, 10 CLB tiles highlighted on the left.

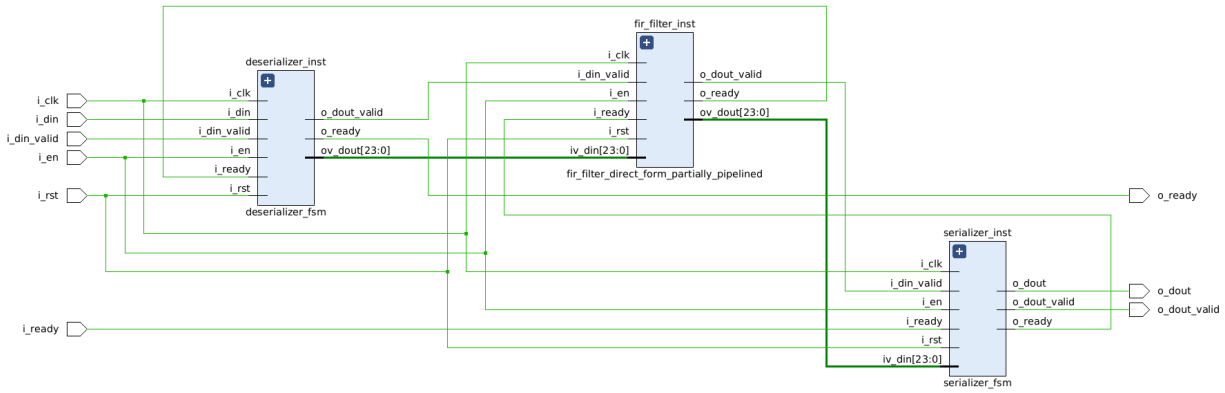

Figure 5: top_level RTL block diagram (auto-generated in Vivado: `synth -rtl`)

Shown in Figure 8 is a timing diagram illustrating the write operations for an 8-tap 2-pipeline FIR Filter. Every horizontal section divided by dotted lines represents a slice in time corresponding with each write operation. Note that t=1,2,.. only corresponds to write operations. In between each write operation are `PIPE_DEPTH` consecutive read operations, with:

$$\text{PIPE\_DEPTH} = \frac{\text{FIR\_DEPTH}}{\text{PIPELINES}}$$

For simple illustration, the diagram describes an 8-tap filter with just 2 MAC pipelines. Thus, PIPE_DEPTH = 4 in this illustration. Each MAC holds 4 input samples and 4 weights. The arrows between the RAM and ROM show the corresponding pairs of samples and weights that must be read into the MAC. The samples are written into the RAM starting at address = max_value, in this case, max_value = 2'b11.

Then, samples and weights are read consecutively into the MAC with the ROM read address starting at 2'b00 and the RAM read address starting at the write address. With each sample, the addresses increment upwards. The RAM read address resets to 2'b00 when it overflows and continues counting upward. Then, before the next input sample is received by the filter, the sample to be overwritten in each RAM is written into the RAM of the next pipeline. The sample to be overwritten at this point is already on the RAM's data_out bus and is immediately ready to be written into the next RAM. The newly received input sample is then written into the first pipeline RAM. Shown in Figure 8 at t=4 illustrates this write-into-next-ram-then-overwrite operation. Also note that the RAMs have a separate read address and write address which are multiplexed into its address port.

This illustration should hopefully suffice in showing how to expand this operation from 8-taps with 2-pipelines to 256-taps with 8-pipelines.

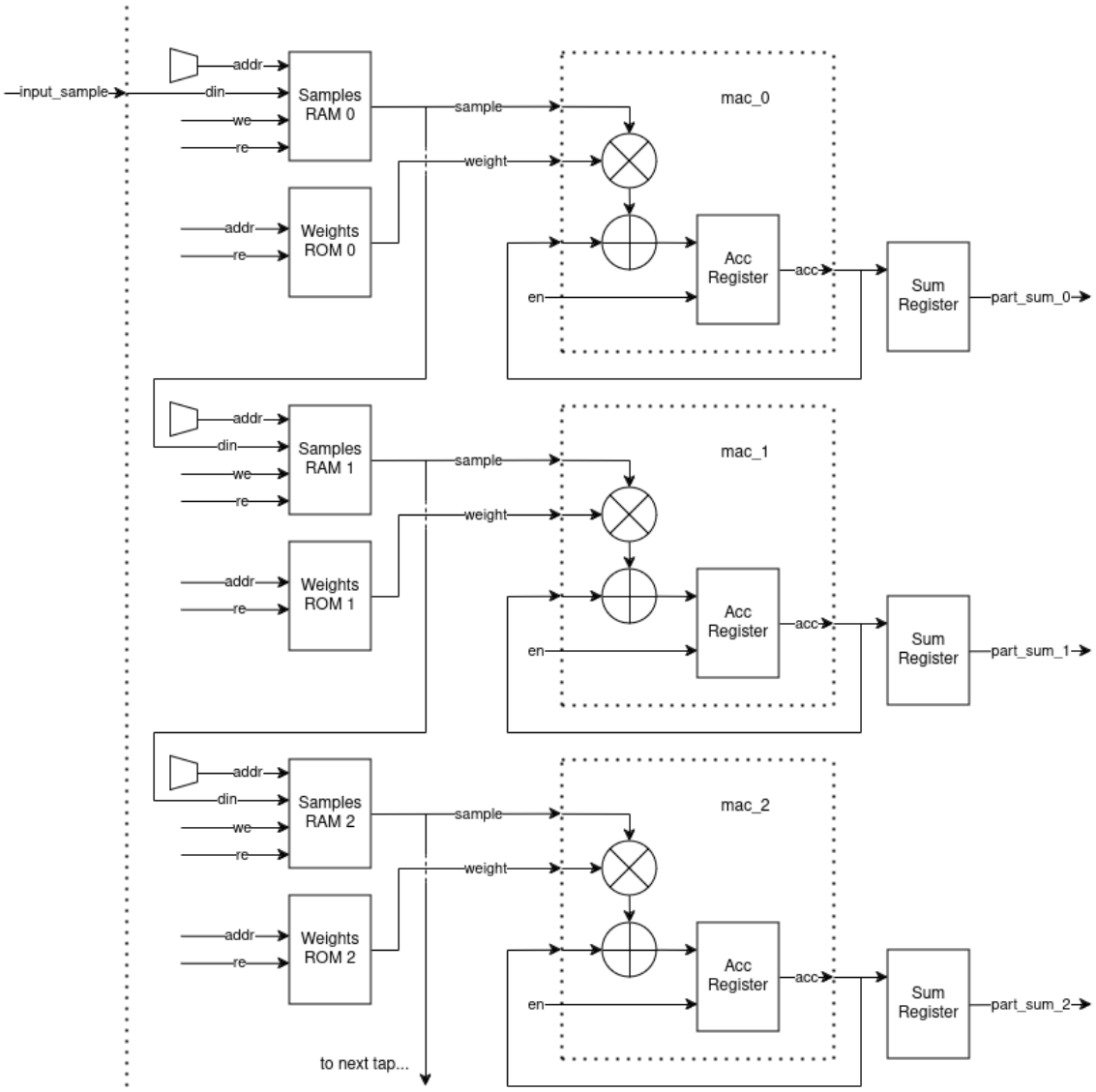

Figure 6: Block diagram describing the expandable ROM-RAM-MAC pipeline structure (manually drawn via Draw.io).

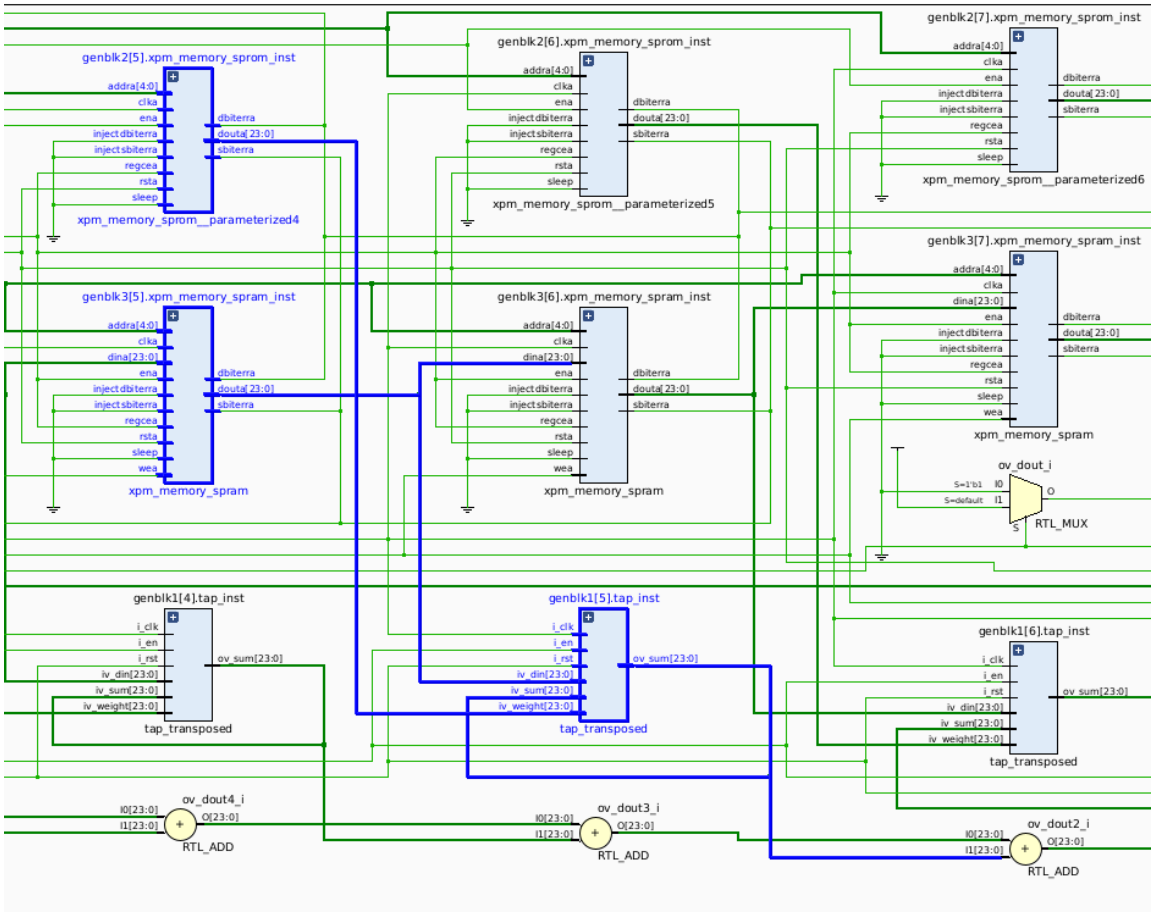

Figure 7: Closeup of RTL elaboration schematic highlighting one ROM-RAM-MAC pipeline in dark-blue border (auto-generated in Vivado: `synth -rtl`).

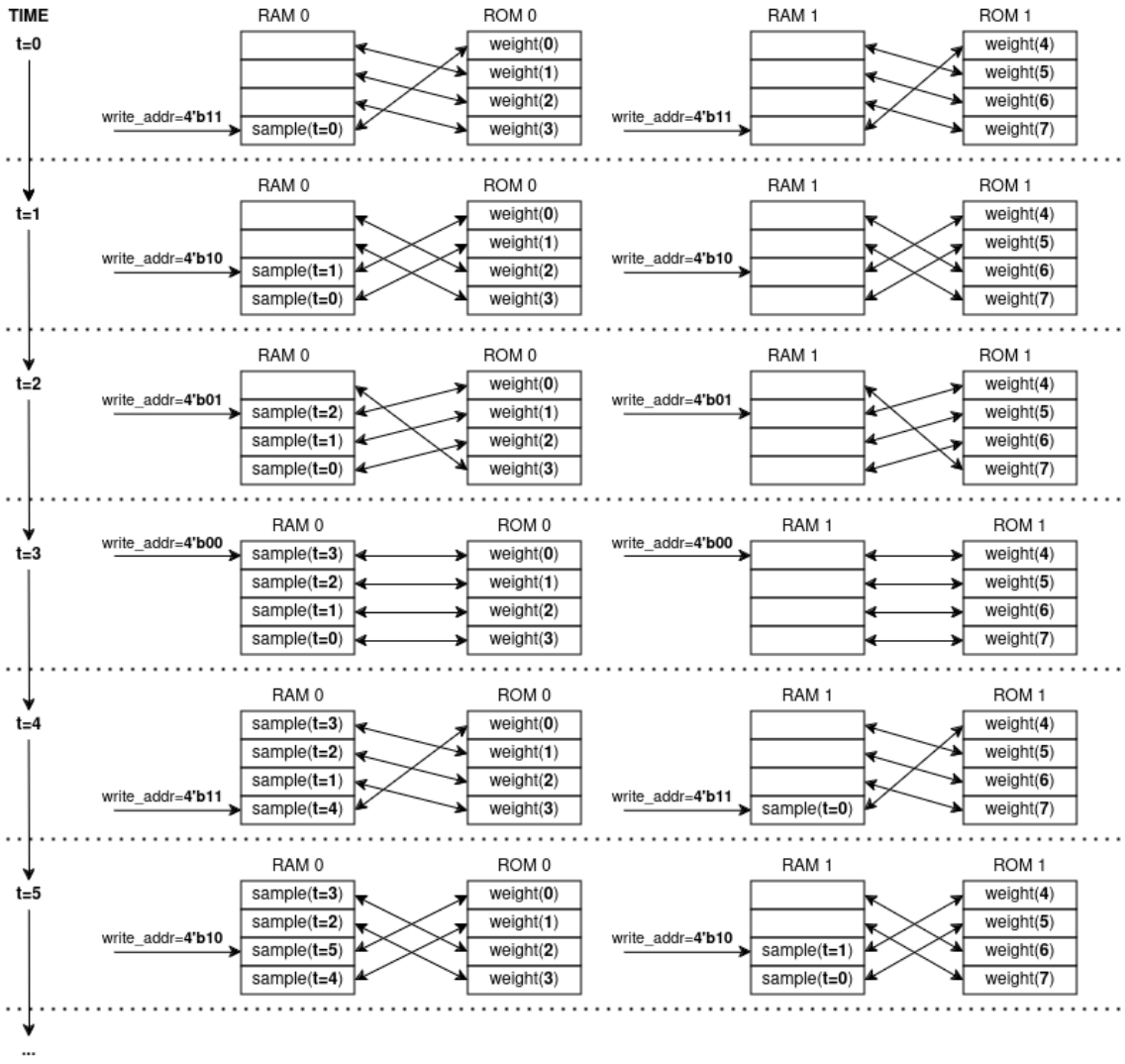

Figure 8: Timing diagram describing the write operation with each incoming input sample for an 8-tap 2-MAC-pipelines FIR Filter (manually drawn via Draw.io).

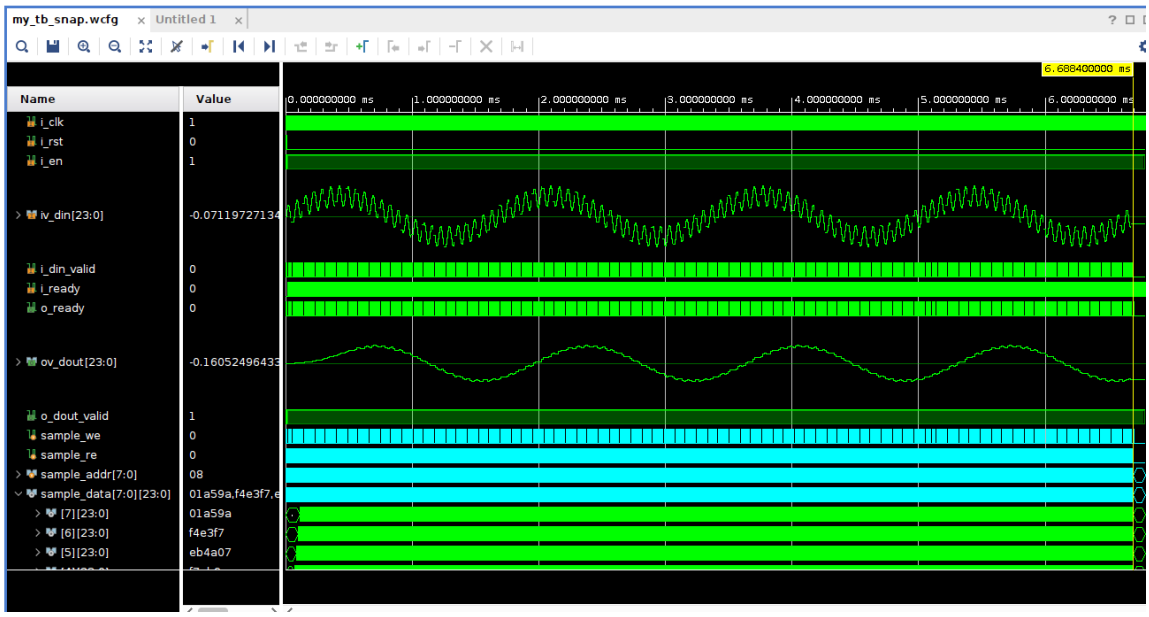

Figure 9: Testbench waveform simulation showing the noisy input signal iv_din and clean output signal ov_dout.

6 Experimental Procedure and Results

As described in the Data Description section, we design a testbench that feeds a noisy signal containing a 200Hz sine wave + 5KHz noise sine wave into the design and expect a clean 200Hz sine wave from the output. Shown in Figure 9 is the testbench waveform simulation showing the noisy input signal passed into the FIR filter and the clean signal output. In an industry setting, a more mature verification approach would be to do a spectral analysis on the output signal, likely via Python or Matlab, to see what frequency bands have passed the filter. Verifying an analog signal via waveform diagrams is just a quick-and-dirty way to demonstrate the basic principles of the low-pass filter.

Shown in Figure 10 is the resource utilization report summarizing the number of LUTs, FFs, DSPs, and BRAMs used. Note that BMEM count shows 8, but after inspecting the netlist, I can confirm that 16 BRAM BELs are indeed used. I am not entirely sure what BMEM is or how it is counted.

Shown in Figure 11 is the design placed and routed onto the xc7z020 using the default Vivado implementation strategy. Shown in Figure 12 is the timing report with the clock constraint set to 10MHz. With a worst negative slack of +87ns which represents the critical path delay of the design, we can likely increase our clock speed without penalty if higher performance is needed.

I have not used the Power report summary in Vivado before and not familiar with the report methodology, so I cannot offer concrete commentary on

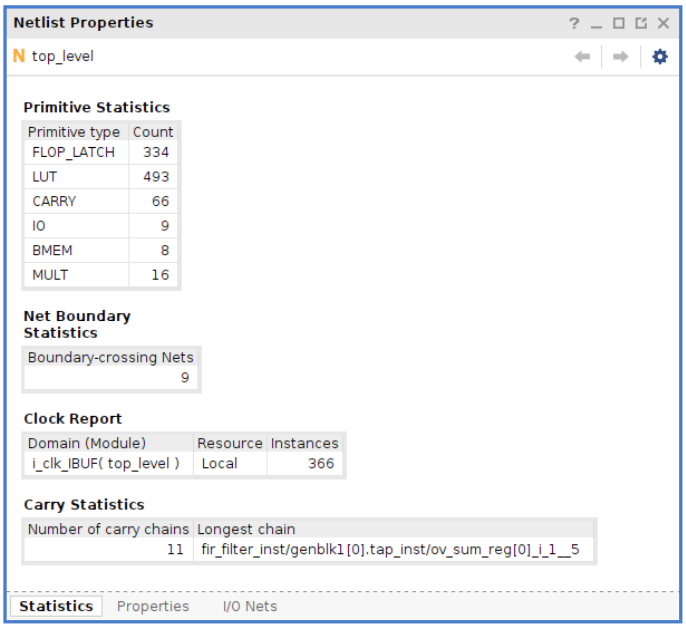

Figure 10: Synthesis resource utilization report.

the power usage. However, to give some impression, shown in Figure 13 and Figure 14 are the power breakdown and environment settings. At a glance, it seems that BRAM usage takes up a lionshare of 44% of the power usage. Using 16 BRAM sites in our partially pipelined implementation replaced 6144 flip-flops of a traditional direct-form implementation as discussed in the Method Description section. From the utilization report, 334 FLOP_LATCH (Flip Flops) were used. From the power report breakdown, I assume flip flop power usage is represented by Clocks and Signals which use 5% and 16% respectively, summing up to 21%. Had we used the traditional direct-form implementation and added 6144 additional flip-flops to the design, I suspect they may have used a larger percentage of the power.

7 Conclusion

This paper presented a RAM-based partially pipelined Verilog model of an FIR Filter whose data width, filter depth, and number of pipelines can be easily reconfigured via the top_level parameters. We have also demonstrated a 24-bit 256th order low-pass filter with 8 parallel MAC pipelines and performed a simple testbench simulation to verify its functionality. There is certainly many improvements that could be done with this Verilog model in the future.

As mentioned before, the final summation of the partial sums from each

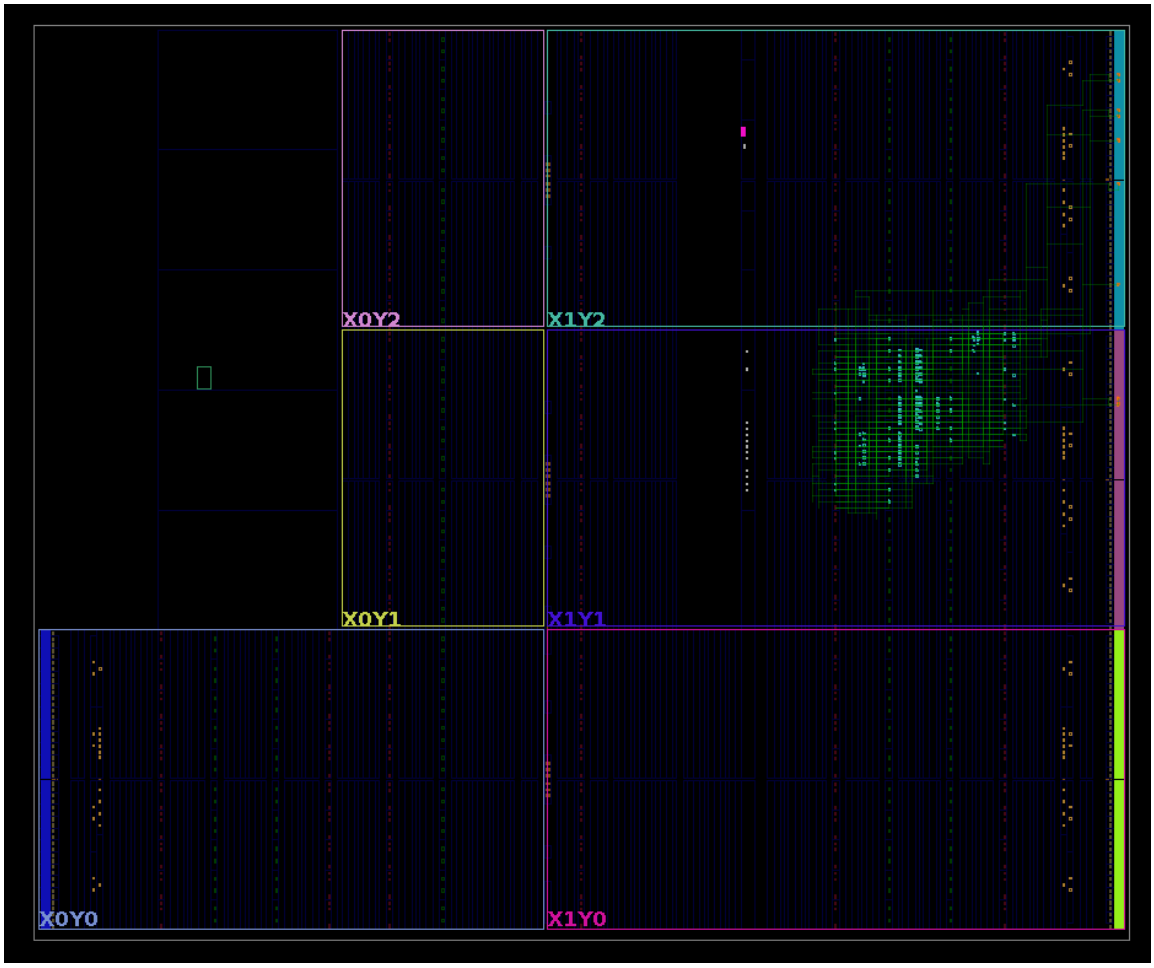

Figure 11: The final design placed and routed onto the xc7z020 via Vivado default implementation strategy.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	87.103 ns	Worst Hold Slack (WHS):	0.181 ns	Worst Pulse Width Slack (WPWS):	3.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1244	Total Number of Endpoints:	1244	Total Number of Endpoints:	366

All user specified timing constraints are met.

Figure 12: Timing report shoing critical path delay (Worst Negative Slack)

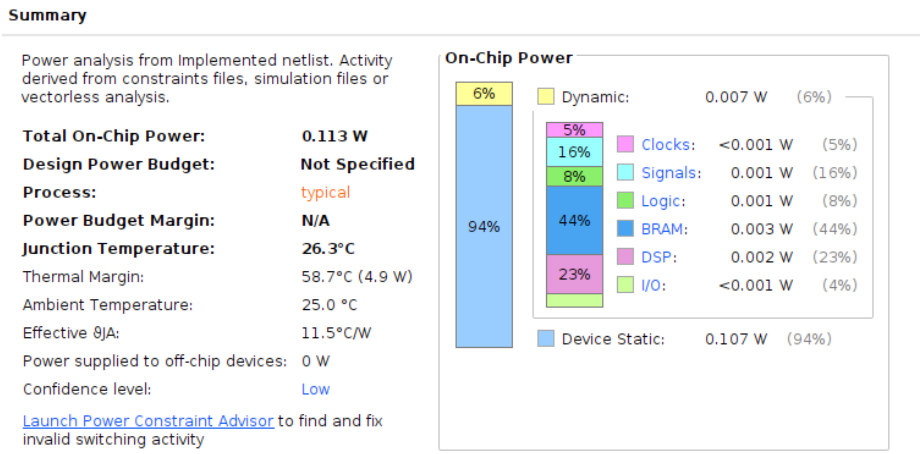

Figure 13:

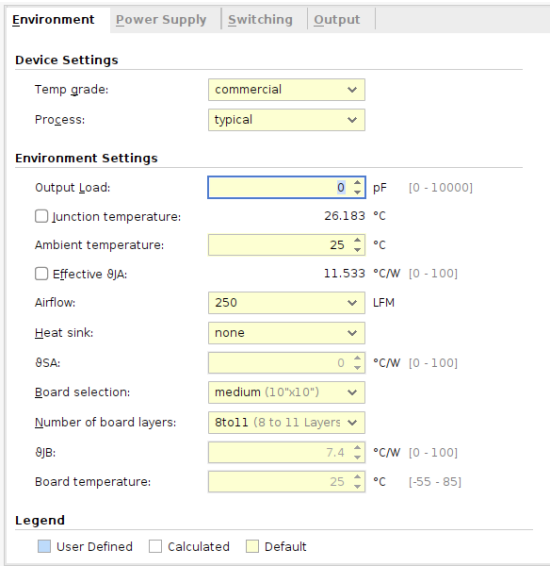

Figure 14:

MAC pipeline are summed in one clock cycle via adder chain. One could improve upon this by using a binary adder tree and introducing some pipelining along the tree stages. However, the FPGA CARRY4 BELs in modern 7-series FPGAs are very optimized, and the fact that our throughput is likely limited by BRAM memory access suggests that this is probably not the bottleneck. I suspect the limiting factor in terms of throughput is indeed the BRAM memory access. If BRAM access becomes a problem, then the designer will have to increase the number of MAC pipelines to exploit the parallel time-slice processing.

Another consideration would be to explore the usage of Distributed RAMs mentioned previously to forego expensive BRAMs entirely. The SLICEM logic sites have special LUTs that can be reconfigured as small local LUTRAMs. Some more implementations can be set up to see if this could keep CLB usage low while avoiding BRAM.

As mentioned in the Experiments and Results section, a more mature verification environment can be employed to really stress-test the filter and to run batches of tests with various random input signals to really see the band of frequencies that passes through the filter. This could also help us in verifying the frequency response and to judge the signal to noise ratio of the signal.

A more in-depth exploration of the power usage and the QoR of the placement and routing is also due.

8 References

References

[1] All About Circuits, “Pipelined direct-form fir versus the transposed structure,” 2024.