Optimizing Finite Impulse Response Filter Design

This paper was downloaded from TechRxiv (<https://www.techrxiv.org>).

LICENSE

CC BY 4.0

SUBMISSION DATE / POSTED DATE

18-01-2022 / 24-01-2022

CITATION

Liebeskind, Alexander; Asensio, Rafael (2022): Optimizing Finite Impulse Response Filter Design. TechRxiv. Preprint. <https://doi.org/10.36227/techrxiv.18665621.v1>

DOI

[10.36227/techrxiv.18665621.v1](https://doi.org/10.36227/techrxiv.18665621.v1)

Optimizing Finite Impulse Response Filter Design

Alexander Liebeskind
Computer Engineering
Columbia University
New York, United States
a.liebeskind@columbia.edu

Rafael Asensio
Electrical Engineering
University College London
London, United Kingdom
rafael.asensio.19@ucl.ac.uk

Abstract—In this study, we explored a potential design for a 64-tap 16-bit Finite Impulse Response (FIR) Filter, including several optimizations. We used RTL code, synthesized gate-level netlist, and ran PrimeTime analysis for each component and the overall circuit. We present the methodology associated with this approach, and its operational results. The design chosen uses a 16-bit floating point ALU with a dual-clock, first in first out memory architecture. The final FIR core was functional, as measured via timing and power analysis, throughput, maximum clock frequency, area, error rate, and accuracy. The source files are also available for further consultation.

Index Terms—FIR (Finite Input Response), ALU (Algorithmic Logic Unit), FIFO (First In First Out), FSM (Finite State Machine)

I. INTRODUCTION

A FIR filter is a filter whose response to a finite length input is a finite length impulse. Mathematically, its output can be described by the following:

$$y[n] = \sum_{i=0}^N b_i \cdot x[n - i] \quad (1)$$

Where $y[n]$ is the output signal, $x[n]$ is the input signal, N is the filter order, and b_i is the filter coefficient. For our specified 64-tap 16-bit filter design, N is 64.

In this study, we first wrote MATLAB code to simulate FIR filter behavior [1]. This allowed for initial testing and modelling with the FIR design, as well as an error free source to calculate the ideal outputs of the mathematical operations within the filter, to contrast to the results obtained through the Verilog design.

The FIR filter specification chosen for the design was the floating point, dual-clock FIFO memory, which interprets all 16-bit variables as half-precision floating point binary numbers according to IEEE format [2]. The dual-clock feature allows for the use of two clocks, one external (driving the input) and one internal (managing the operation of the core). This dual-clock behaviour is managed through the input memory, which has a FIFO design that allows for more a streamlined operation, as the incoming data stream from the input is passed to the core of the module.

There are five parts within the design of the overarching FIR filter module: the floating point ALU, the FIFO input memory, the CMEM (coefficient memory), the registers, and the FSM.

Together, these components comprise a functional FIR core replicates the function above.

II. METHODS

A. Filter Model

Before working on the RTL FIR filter design, a functional model of the filter was designed in MATLAB, using the underlying mathematical theory to replicate the outputs of the filter given a set of 16-bit inputs, x , and coefficients, b .

The two input files were saved with the purpose of applying them as inputs to the RTL design through the corresponding testbench, as can be seen further through in this report. Moreover, the output file can be used to compare against the results of the fully realised Verilog design, to test for accuracy and relevant measurements.

For the overall filter, as well as each individual component, synthesis was also performed using the design compiler to create a gate-level netlist. The functionality of this model was validated against the testbench, and then PrimeTime analysis was used to analyze timing performance for both the minimum and maximum path and power demand [3].

B. Core FIR Module

The objective of the FIR module is to instantiate and connect all the underlying modules under a single RTL design. This is the "fir.v" Verilog file.

```
`timescale 1ns/1ps

module fir(clk, resetn, din, addr, dload, cload, dout);

    input clk, resetn, dload, cload;
    input [15:0] din;
    input [13:0] addr;
    output wire [15:0] dout;

    wire [15:0] xout;
    wire [15:0] cin, cout;
    wire [15:0] Y0, X0;
    wire [15:0] yout;
    reg [63:0] CEN;
    reg WEN;
    reg clk2;

    parameter clkper2 = 8; //internal clock

    //imem imem_dut(.Q(xout), .CLK(clk), .WEN(dload), .A(addr), .D(din)); (if we
    //wish to revert to imem instead of the advanced fifo)
    cmem cmem_dut(.Q(cout), .CLK(clk), .CEN(CEN), .WEN(cload), .A(addr[5:0]), .D(
    din));
    alu alu_dut(.clk(clk), .X(X0), .B(din), .Y0(Y0), .Y(dout));
    fifo fifo_dut(.resetn(resetn), .dout(xout), .CLK1(clk), .CLK2(~clk2), .WEN(
    dload), .din(din));

    memreg mem_dut(.D(dout), .CLK(clk), .Q(yout));
    prodreg prod_dut(.D(cout), .CLK(clk), .Q(Y0));
    sumreg sum_dut(.D(din), .CLK(clk), .Q(X0));

    always # (clkper2/2) clk2 = ~clk2;

    always@(clk, addr) begin // MUX
        CEN = 64'hFFFFFFFFFFFFFFFF;
    end
endmodule
```

```

        CEN[addr[13:8]] = 1'b0;
    end
endmodule

```

The code works by connecting the memory blocks and the ALU through the registers, using one clock driven from the testbench (*clk*) and another kept internally (*clk2*), maximising the utility of the FIFO and managing the difference between external and internal clock frequencies.

To verify the functionality of the module, an appropriate testbench was designed.

```

`timescale 1ns/1ps
`define QSIM_OUT_FN "qsim.out"
`define MATLAB_OUT_X "matlab_in_fir.txt"
`define MATLAB_OUT_B "matlab_b_fir.txt"

module testbench();
    reg[15:0] X, B, Y0, din;
    reg clk, resetn, dload, cload;
    wire[15:0] Y, dout;
    reg[13:0] addr;
    parameter clkper = 10;
    integer i, j;
    integer fir_out_matlab;
    integer fir_out_qsim;
    integer qsim_out_file;
    integer matlab_x_file;
    integer matlab_b_file, ret_read;

    //imem imem_dut(.Q(xout), .CLK(clk), .WEN(dload), .A(addr), .D(din));
    //cmem cmem_dut(.Q(cout), .CLK(clk), .CEN(CEN), .WEN(cload), .A(addr), .D(cin))
    ;
    fir_dut(.clk(clk), .resetn(resetn), .din(din), .addr(addr), .dload(dload), .
        cload(cload), .dout(dout));

    always # (clkper/2) clk = ~clk;

    initial begin
        clk = 0;
        resetn = 0;
        B = 0;
        //din = 0;
        addr = 0;
        dload = 0;
        cload = 0;
        //dout = 0;

    end

    initial begin
        for (j = 0; j <= 16384; j = j + 1)
            #10 addr = j;
        for (j = 1; j <= 16384; j = j + 1)
            #10 addr = j;
    end

    initial begin
        $dumpfile("./fir.vcd");
        $dumpvars(0, testbench.dut);

        qsim_out_file = $fopen("QSIM_OUT_FN","w");
        if (!qsim_out_file) begin
            $display("Couldn't create the output file.");
            $finish;
        end

        matlab_x_file = $fopen("MATLAB_OUT_X","r");
        if (!matlab_x_file) begin
            $display("Couldn't open the Matlab file.");
            $finish;
        end

        matlab_b_file = $fopen("MATLAB_OUT_B","r");
        if (!matlab_b_file) begin
            $display("Couldn't open the Matlab file.");
            $finish;
        end

        clk = 0;
        //@ (posedge clk); // start the first cycle
        for (i = 0; i < 256; i = i+1) // test 64 times since we will use 64 taps
            begin
                @(negedge clk) ret_read = $fscanf(matlab_x_file, "%d", din); //din
                = $random;
                @(posedge clk) begin
                    ret_read = $fscanf(matlab_b_file, "%d", B);

                    fir_out_qsim = dout;
                    $fwrite(qsim_out_file, "%0d\n", fir_out_qsim);
                    //B = $random;
                    #14.5
                    $display("din: %b B: %b dout: %b", din, B, dout);

                end
                //@ (posedge clk); // next cycle
            end
    end

#10

```

```

        $fclose(qsim_out_file);
        $fclose(matlab_x_file);
        $fclose(matlab_b_file);
        $dumpall;
        $dumpflush;
        $finish;
    end
endmodule

```

The testbench works by first opening the two files containing the 16-bit inputs and coefficients used in the MATLAB model, as specified in the previous subsection. This allows for testing the reliability of the module against the results obtained from the mathematical model, which can be found in Section III of this report.

C. ALU

The ALU employed in our design was specialized to handle 16-bit floating point inputs. Each 16-bit input was managed in three parts, as dictated by IEEE 754: the first bit containing the sign of the number, the next five bits containing the size of the exponent, and finally the remaining ten bits displaying the mantissa.

Fig. 1. Half-precision floating point (IEEE 754) [4]

Hence, for every term in the summation, the coefficient, b_i , needs to be multiplied to its respective input, $x[n-i]$. This is done by multiplying the sign bits together (bit 15), adding the exponent bits (bits 14-10), and multiplying the mantissas (bits 9-0).

Moreover, after every product operation the result must be added to the previous as per (1). This is achieved by equalising the exponents through bit shift operations and adding up the results. More on this process and the full RTL code for the ALU can be found in the submission file.

D. FIFO

The FIFO handles the input data stream, storing and accessing the x values to perform the necessary calculations in the ALU. The main feature of the block is its first in first out operation, working as a queue by enqueueing inputs and dequeuing them sequentially to the rest of the core filter.

This FIFO functionality is particularly useful for implementing the dual-clock setup. Due to the differing clock frequencies between the incoming data stream and the core calculations, the queue-like behaviour synchronises the two clocks by writing into memory at the external frequency, and reading to the filter at the internal frequency.

E. Coefficient Memory (CMEM)

CMEM is the module that handles the 64 b_i coefficients. Due to its small size, it can be used as directly generated by the memory compiler tool, without needing an overarching file to instantiate several blocks. Its RTL design can therefore seem

rather complicated, since it is automatically generated, but this makes the module more streamlined and efficient, performing its function as requested by the specifications.

F. Registers (Memory, Product, and Sum)

The registers are the cohesive element in the FIR filter design that connect the memory blocks and ALU together to access the full capabilities of the system. There are three kinds: memreg, prodreg and sumreg.

One of the main ways in which they do this is by connecting the FIFO and CMEM with the ALU for the product calculation. Moreover, they feedback the result to the ALU for the sum calculation, helping join all the variables to drive the output.

G. FSM

The FSM is handled implicitly through the filter core, which maintains the connections between components and the logic underlying FSM behavior. This is essential to ensure that each portion of the FIR is functioning correctly within the overall design.

III. RESULTS AND DISCUSSION

A. RTL Code Functionality

In order to test the functionality of the FIR core module, a testbench was designed, as explained in the previous section. This testbench was run through the LINUX terminal using helper files such as "runsime.do" and "waveformat.do", which specified included waveforms and other linking details. This yielded the following output in QuestaSim.

Fig. 2. FIR Testbench Results

As can be seen from the console output and waveform, the testbench correctly verifies the functionality of the FIR filter block.

B. Filter DC Synthesis

Once the functionality of the RTL code is verified, the module can be synthesized to obtain the gate-level netlist. To do this it is necessary to include appropriate "fir.tlc" and "timing.tcl" files, please refer to these in the submission files. Once the synthesis is completed we obtain the desired files.

Fig. 3. Gate-level netlist file snapshot

Running this synthesised file through the testbench, we can verify the correct operation of the gate-level netlist, displaying certain glitches along the waveform.

Fig. 4. Gate-level netlist testbench results

C. Overall Performance

Analysis through PrimeTime yielded the following output for the overall FIR core. As can be observed in the figure below, no slack violations occurred. Additionally, all timing values seem reasonable when compared to expected timing performance, indicating that the FIR core produces viable timing results.

Fig. 5. PrimeTime analysis of timing for minimum path

Fig. 6. PrimeTime analysis of timing for maximum path

Power analysis further demonstrated that the FIR core performed as envisioned. The report below shows that total power also takes on an acceptable value, which is necessary for proper functioning of the FIR core.

Fig. 7. PrimeTime analysis of power

In short, both timing and power analysis show that the FIR core is performing as desired. The optimized FIR design therefore passes validation via PrimeTime analysis.

D. Further Metrics

To more deeply analyze the performance of the FIR core, we also computed a number of metrics quantifying specifics associated with the design.

One such metric is throughput, which was 10kS/s for the FIR. This can be ascertained from the usage of a 10MHz clock (noting that frequency = 1/period, and total time = clock cycles * clock period). Since throughput is samples per unit time, using a 10MHz clock at 1 sample per cycle dictates that we will have a throughput of 10kS/s.

$$Throughput = \frac{Samples}{Time} = \frac{10k}{10k * (1/10MHz)} \quad (2)$$

Maximum supported clock frequency was also calculated as another performance metric. We should also note that the internal clock frequency was 8MHz, to maintain speed of access while minimizing energy usage within reason. Maximum clock frequency was calculated using the following relationship, noting that the minimum clock period to accommodate the

maximum path (i.e. support the circuit) was 6.8653ns. The determined value was 145.7MHz.

$$f_{max} = \frac{1}{6.8653ns} = 145.7MHz \quad (3)$$

The total area was derived directly from synthesis, and was measured at 1029.6 mm².

Fig. 8. FIR area report

Energy efficiency was also calculated, to help assess the energy usage of the final FIR core. The derived value was 4.212pJ/S, using the calculation here. Note that total power and throughput are valued as aforementioned.

Accuracy was quantified using normalized root mean squared error (NRMSE) as a measure, compared against MATLAB FIR output as a frame of reference. This employed the following equations.

$$RMSE = \sqrt{\frac{\sum_{i=0}^n (y_i - y_i^*)^2}{n}} \quad (4)$$

$$NRMSE = \frac{RMSE}{y_{max} - y_{min}} \quad (5)$$

Here, n is the total number of inputs, y_i is the full-precision MATLAB result, and y_i^* is the RTL output. However, this computation was somewhat tedious manually without the assistance of any software. To facilitate this operation, we also wrote MATLAB code for automatic computation. This script is included here for reference.

```

qsim = fopen('qsim.txt');
matlab = fopen('matlab_out_fir.txt');
max = -realmax;
min = realmax;
sum = 0;
count = 0;
while ~feof(fid) && ~feof(matlab)
    q_line = fgetl(qsim);
    m_line = fgetl(matlab);
    a = str2double(q_line);
    b = str2double(m_line);
    if b > max
        max = b;
    end
    if b < min
        min = b;
    end
    sum = sum + power((a-b), 2);
    count = count + 1;
end

disp('NRMSE:');
disp((sqrt(sum/count))/(max-min));

fclose(qsim);
fclose(matlab);

```

Fig. 9. MATLAB code for NRMSE calculation

The NRMSE rates were determined to be 11.2% and 15.8% for average and worst case respectively, meaning accuracy rates via this method were 88.8% and 84.2% at average and worst case. We also verified this against our inputs and outputs to ensure that these calculations were feasible.

E. Component Performance

As described above, synthesis and PrimeTime analysis allowed for the assessment of various performance metrics, including timing and power. Using these data, we can compare and observe the performance of each component involved in the design.

Timing performance was assessed for both minimum and maximum paths. Slack, or the difference between achieved time and desired time for a timing path (data required and data arrival) ranged across the values found below. No violations occurred for any of the components involved.

TABLE I
BREAKDOWN OF COMPONENT TIMING PERFORMANCE

Component	Timing Analysis	
	Minimum Path Slack (ns)	Maximum Path Slack (ns)
ALU	0.3746	0.0308
FIFO (clk1)	0.5715	5.0916
FIFO (clk2)	0.5714	0.0579
Memory Register	0.0629	9.2331
Product Register	0.0629	9.2331
Sum Register	0.0629	9.2331

Power analysis revealed the following results across components. All values were within reasonable range.

TABLE II
BREAKDOWN OF COMPONENT POWER PERFORMANCE

Power Analysis	
Component	Total Power (mW)
ALU	2.667e-05
FIFO	2.102e-04
Memory Register	3.561e-05
Product Register	7.143e-05
Sum Register	8.456e-05

Note that CMEM was generated from ARM as an independent memory unit.

IV. CONCLUSION

This study successfully explored the nuances of designing an FIR filter. Along with the necessary registers and memory units, a floating point ALU and a dual clock FIFO were employed to optimize the design. This investigation therefore not only reached a functional FIR core, but also an FIR core with adapted modifications to improve performance.

This corroborates prior studies [5] on FIR design using Verilog and other synthesis and analysis tools [6]. Future studies may examine other ways to design the FIR core, using novel components, or otherwise improve upon the components included here. Additionally, component and FIR core performance may be optimized based on the performance metrics quantified in this study. Overall, however, this study illuminated a functional FIR core design and implementation.

ACKNOWLEDGMENTS

We thank Yunran Zhou (Columbia University) and Ashish Jayant Shukla (Columbia University) for providing support and helping us understand better the different tasks along the project during our time in the lab. Moreover, we thank Mingoo Seok (Columbia University) for assisting with furthering our knowledge of the fundamental theory behind filter design.

REFERENCES

- [1] "FIR Filter Design - MATLAB & Simulink." <https://www.mathworks.com/help/signal/ug/fir-filter-design.html>.
- [2] "IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic." IEEE SA - The IEEE Standards Association - Home. <https://standards.ieee.org/standard/754-2019.html>.
- [3] "PrimeTime User Guide." <https://picture.iczhiku.com/resource/eetop/WYkreqGDkEtYbVX.pdf>.
- [4] "File:IEEE 754R Half floating point format.svg," Wikimedia Commons. [Online]. Available: https://commons.wikimedia.org/wiki/File:IEEE_754r_Half_Floating_Point_Format.svg.
- [5] Yuan, J., Feng, Q. Y., Wang, D. (2012). Design of High-Precision FIR Filter Based on Verilog HDL. In Advanced Materials Research (Vols. 433-440, pp. 5198-5202). Trans Tech Publications, Ltd. <https://doi.org/10.4028/www.scientific.net/amr.433-440.5198>.
- [6] W. Knitter, "DSP for FPGA: Simple FIR filter in Verilog," Hackster.io, 07-Mar-2021. [Online]. Available: <https://www.hackster.io/whitney-knitter/dsp-for-fpga-simple-fir-filter-in-verilog-91208d>.