Cam Brown & Haris Bajramovic & Gyaan Antia
Mar 14, 2023
CE 495: FPGAs and Whatnot

FM Radio Final Report

Background and Theory

Radio stations continuously emit radio waves for us to listen to at our leisure with any basic FM radio. Radio waves are always there, yet we cannot hear them since they have much higher frequencies than the limits of human hearing. All radio stations are constantly emitting radio waves at different frequencies, and we use FM radios to tune into those specific frequencies and listen.

In order for the radios to tune into these frequencies, they need to take an analog signal, the radio wave, sample it, process it, and play it back as accurately as possible. To sample the analog signal they use an ADC, or an analog to digital converter. This changes amplitudes from the analog receiver into binary numbers representing the amplitude of the wave at a given time. Once the signal is converted, the processing begins.

The radio uses a Discrete Fourier Transform (DFT) to convert the wave data in the time domain into the frequency domain. Each station has a range of frequencies that give a mono audio from about 30Hz to 15kHz. Above this, at 19kHz, may be a pure pilot tone sine wave. The presence of this pilot tone indicates that above this frequency is a range providing stereo audio for the station. Our radio assumed the presence of stereo audio and processed data based on that assumption. We use a 32-tap Finite Impulse Response (FIR) bandpass filter to extract the pilot tone and then square it to obtain a 38 kHz tone to divide the two stereo channels (Left - Right and Left + Right). Then a high-pass filter is used to remove the tone at 0Hz created by squaring the pilot tone. Next, the L-R channel is multiplied by the squared pilot signal and demodulated to baseband with a low-pass FIR filter. The filter also reduces the sampling rate by a factor of ten. Then we add the L+R to the L-R to get just 2 times the left side, and then subtract (L+R) - (L-R) to get 2 times the right side. Once we have the left and right channels isolated, we deemphasize them with a 2 tap Infinite Impulse Response (IIR) filter. Once this processing is done, the left and right channels are able to be heard when played, and we output the data to whichever audio driver we are using. The audio driver acts as the DAC, or digital to analog converter, and plays the analog sound waves back to us.

In this project, we designed, simulated, and tested this signal processing algorithm for use on a Cyclone IV E FPGA.

System Architecture

Our system architecture for this lab, like most other labs, was based on a streaming architecture. Each module with the exception of the most simple arithmetic operations (add and subtract) were implemented as FSMs. There are highly optimized FIFOs in between every module that ensure synchronicity across the design. This is particularly important, since there are portions of filtering and decimation in the design and there is not a one-to-one relationship between input and output size. Here is a block diagram of the top level of our architecture:

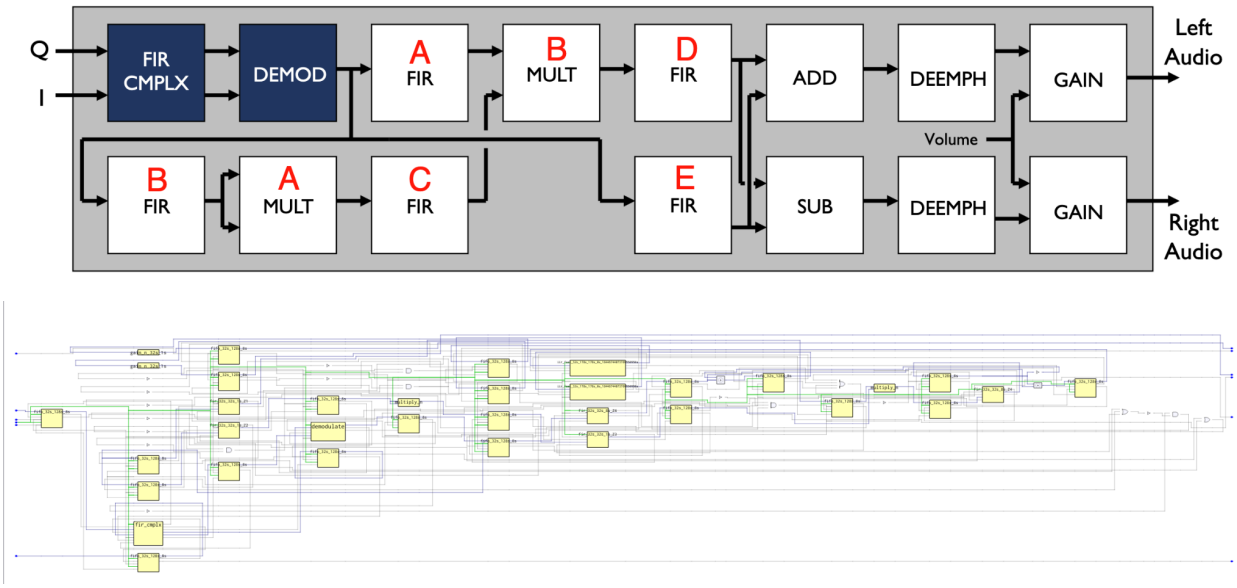

Figure 1: Top-level block diagram and RTL view

FIR Complex Module:

The FIR module is a Finite Impulse Response filter. The FIR complex module takes in the full signal consisting of the real and imaginary part of the radio wave. Its feedback system uses past inputs to calculate the current output. The current output is a discrete convolution of the input signal with a set of coefficients based on the past inputs. The module also takes care of decimation. It shifts in new inputs until it fulfills the required decimation, and then convolves the contents of its two internal buffers (real and imaginary) with two constant buffers full of coefficients, one for the real part and one for the imaginary part. These convolutions are done in parallel and are the outputs of the module.

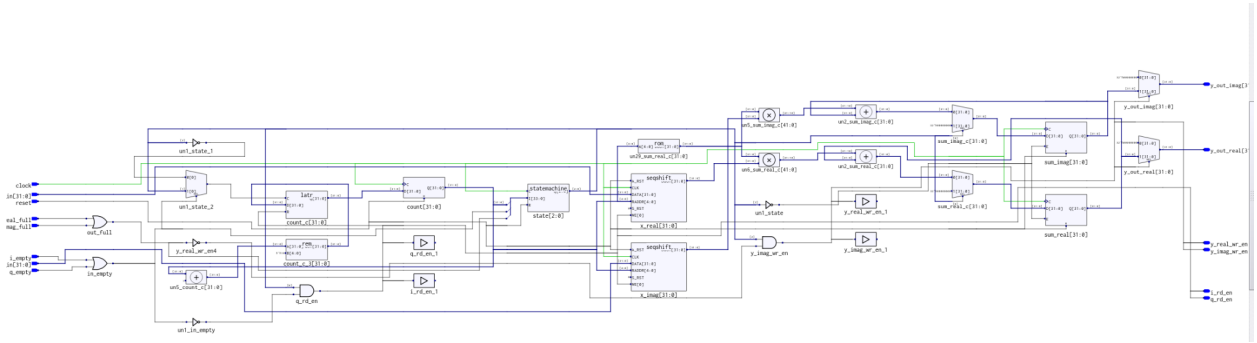

Figure 2: FIR-Complex RTL view

Demodulation:

Demodulation involved 3 modules. The first was the outer level demodulation, which was tricky to implement because it required two states that were initially executed and then never returned to, in order to handle the first two iterations of the FM datastream. This is because there are two sets of flip-flops (current and previous for both real and imaginary input) that contribute to the demodulation core computation (gain * qarctan(imaginary, real)). Qarctan was also non trivial to implement, as it required both careful signed arithmetic, as well as variable division that necessitated the inclusion of a divider. We initially tried to use the behavioral divider written in CE 355, but this proved to be too difficult to troubleshoot. Instead, Gyaan took it upon himself to write one in SystemVerilog that was based directly on the VHDL version, but that could be more easily integrated into our design. All three of demodulate, qarctan and divider are FSMs.

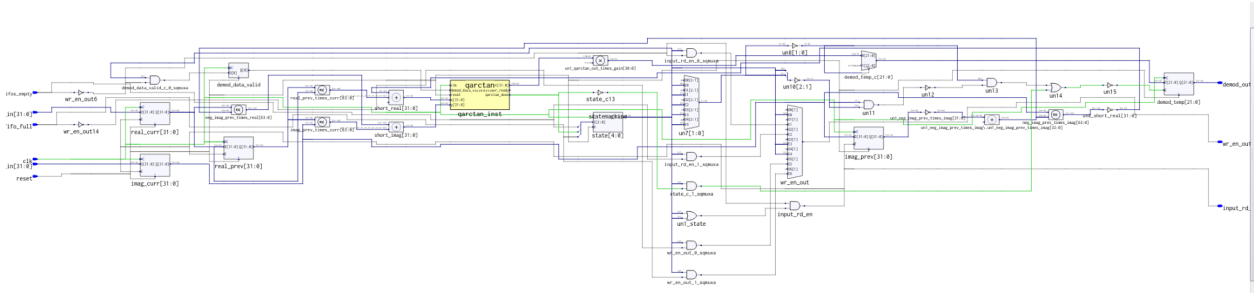

Figure 3: Demodulation module RTL View

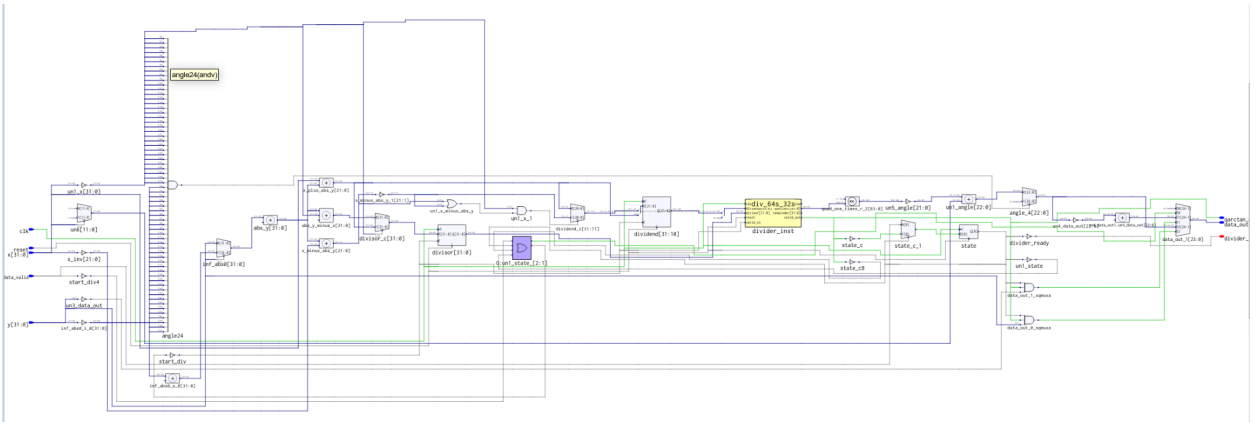

Figure 4: quantized arctan module RTL view

Multiplication:
Multiplication was massively straightforward in that it just was a dequantization of the product of its inputs.

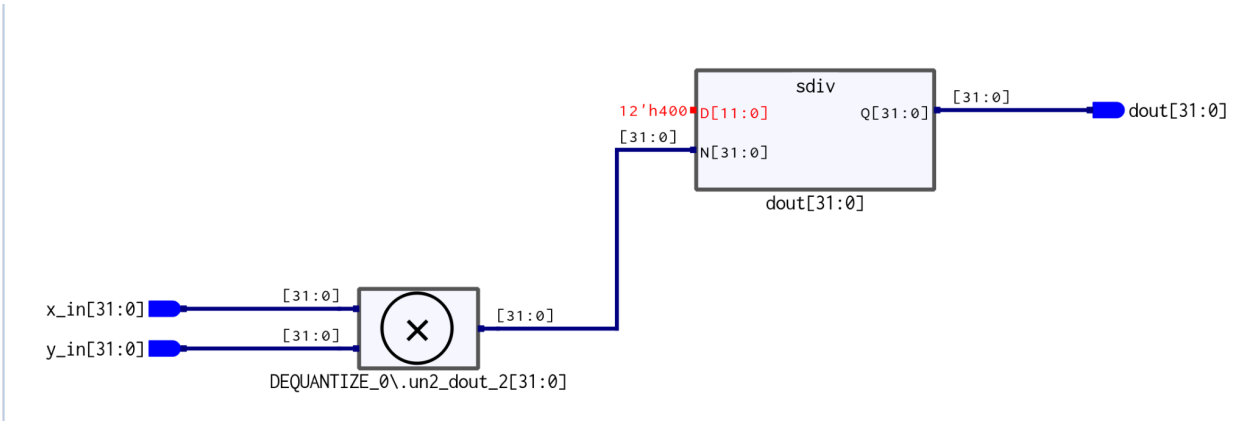

Figure 5: Multiplication module featuring a divider from dequantize

FIR:
The FIR module is a Finite Impulse Response filter. It uses a feedback system that uses past inputs to calculate the current output. The current output is a discrete convolution of the input signal with a set of coefficients based on the past inputs. The module also takes care of decimation. It shifts in new inputs until it fulfills the required decimation, and then convolves the contents of its internal buffer with a constant buffer full of coefficients. This convolution is the output of the module.

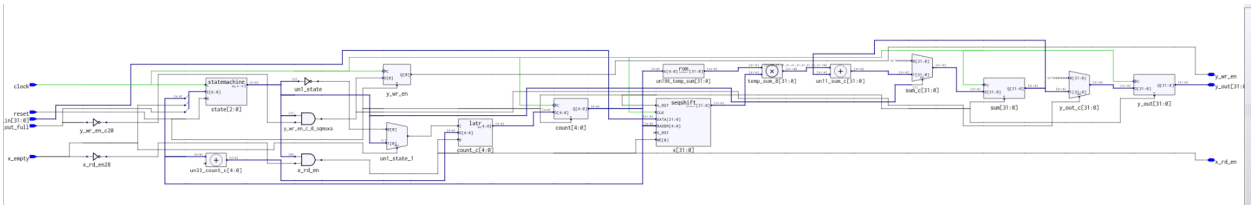

Figure 6: FIR RTL view

Deemphasis:
Deemphasis consists of an IIR, a feedback feedforward system. Values would shift in, be multiplied by a constant and then added together to produce an output. This was created using states to maximize resource sharing. Two 2 element shift registers hold values for x and y. The incoming x and previous x are multiplied by a predefined constant and added together. This was then added to the product of the previous y value and a constant to produce an out value. This out value would be shifted into the register for the next calculation.

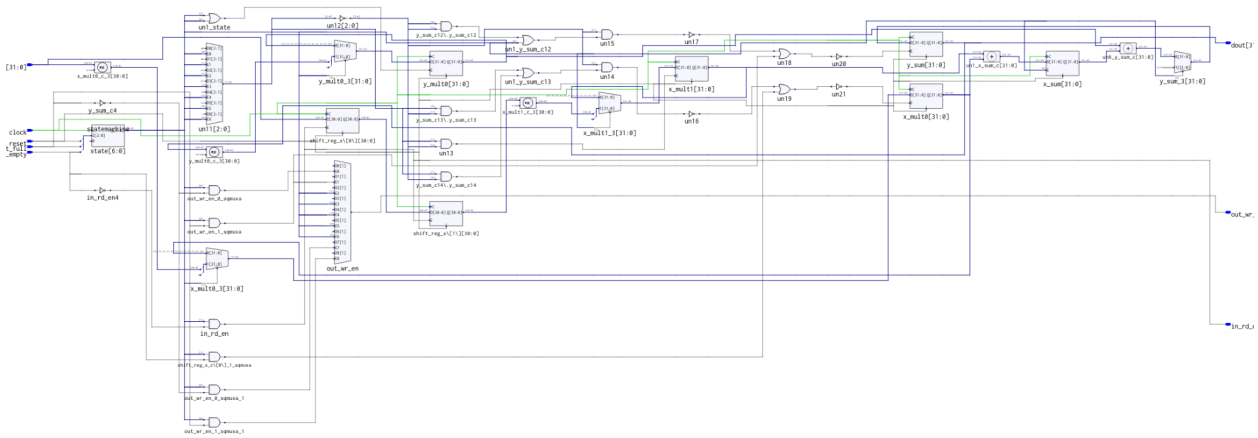

Figure 7: Deemphasis module RTL view (which implements IIR)

Design Process

Our design process consisted of modularity and rigorous testing. The first thing we did was get test data for each step of the fm radio software. This way we can test each part for functionality. We did this by streaming values out of the functions into text files.

```
void demodulate( int real, int imag, int *real_prev, int *imag_prev, const int gain, int *demod_out, std::ofstream& demodstream )
{
    // k * atan(c1 * conj(c0))
    int r = DEQUANTIZE(*real_prev * real) - DEQUANTIZE(-*imag_prev * imag);
    int i = DEQUANTIZE(*real_prev * imag) + DEQUANTIZE(-*imag_prev * real);

    *demod_out = DEQUANTIZE(gain * qarctan(i, r));

    char demod_data[9];
    sprintf(demod_data, "%08x", *demod_out);
    demodstream << demod_data << endl;

    // update the previous values
    *real_prev = real;
    *imag_prev = imag;
}
```

Figure 8: Code to get test data for functions

Then we would look at each of the necessary functions in the C++ code. After understanding the process in each of the functions, we would plan out the logic and states at a

high level. Drawing out schematics and state machines got us thinking how we wanted the system to work. Afterwards we would write the module and create a testbench for it. The testbench would have input data from the previous function and then output data would be the output of the corresponding function.

After getting functionality correct we would try to synthesize the module to see the layout, resource utilization, and frequency. Following that we would try to optimize the modules to boost performance. Afterwards we plugged them into a top level design. The top level was written to include a fifo between each module, and the signals were created prior, making plugging in the new module simple.

Following that we wrote a testbench for the top level design. After making sure that was correct, we ran a uvm testbench and double checked correctness. Overall, our process was to plan, code, test, optimize, and repeat for every part of this design.

Optimizations

One of the optimizations we used was pipelining and resource sharing. This was done in the Deemphasis module as shown below. In this module the longest possible delay would be a dequantize and then two consecutive multiples and an add. However in each of the states there would only be 1 multiply and 1 dequantize, decreasing the latency and allowing for a higher clock speed. If all the multiplies were done in a single state it would have to instantiate several multiple units and dequantize units which would take more resources.

```
(s0): begin
    if (in_empty == 1'b0) begin
        in_rd_en = 1'b1;
        shift_reg_x_c[1] = shift_reg_x[0];
        shift_reg_x_c[0] = din;
        shift_reg_y_c[1] = shift_reg_y[0];
        shift_reg_y_c[0] = dout;
        x_mult0_c = $signed(din)*$signed(b0);
        state_c = s1;
    end
end
(s1): begin
    x_mult0_c = DEQUANTIZE(x_mult0);
    x_mult1_c = $signed(shift_reg_x[1]) * $signed(b1);
    state_c = s2;
end
(s2): begin
    x_mult1_c = DEQUANTIZE(x_mult1);
    y_mult0_c = $signed(dout) * $signed(a0);

    state_c = s3;
end
```

```
(s3): begin
    y_mult0_c = DEQUANTIZE(y_mult0);
    x_sum_c = $signed(x_mult0) + $signed(x_mult1);
    state_c = s4;
end
(s4): begin
    y_sum_c = $signed(x_sum) + $signed(y_mult0);
    state_c = s5;

end
(s5): begin
    if (out_full == 1'b0) begin
        out_wr_en = 1'b1;
        state_c = s0;
end
end
```

Figure 9: Example of pipelining in state machine

Simulation Results

For the simulation we were attempting to verify the functionality of each of the blocks and the top level system. This was achieved through testbenching with the proper input and output data. Results of our UVM are below.

Our throughput bottleneck was in the FIR and FIR complex modules. These modules include feedback loops and decimation. These modules have very long latencies and do a lot of processing. This combination makes the throughput low for these modules. For other modules like add or multiply, they have much higher throughput, but the data coming in is bottlenecked by previous modules.

Throughput is calculated using the number of cycles until all the data was outputted. This was then multiplied by the reciprocal of our synthesized frequency. This was multiplied by the number of lines in our text file of output data. Each line had 4 bytes of data and there are 8 bits in a byte. Lastly there are two audio channels, doubling the data throughput.

Cycles (UVM)	12,431,592
Throughput	12,431,592 cycles * (1/30100000 Hz) * 32768 lines * 4 bytes/line * 8 bits/byte * 2 channels = 866144 Mb/s

Synthesis Results

Our ultimate goal for this project was to successfully synthesize a design that could run near 100 MHz. Even after making several attempts at optimization, we were still only able to reach ~35 MHz. The following table details our synthesis results:

LUTs	5582
I/O Pins	136
DSP Blocks (dsp_used)	62 (360)
DSP.Simple_Multipliers_9_bit	4
DSP.Simple_Multipliers_18_bit	52
DSP.Simple_Multipliers_36_bit	2
Non I/O Registers	3118
Memory Bits	88704
fm_radio_top|clk	35.4 MHz / 30.1 MHz (req_freq/est_freq)

Our clock bottleneck can be found inside of the divider that we wrote. In the divider, it is necessary to extract the most-significant bits from the dividend and divisor signals as part of the incremental computation of the quotient in the FSM. This necessitated us writing a function that has a worst-case delay path of iterating over the entire signal. We think that if we added more states to break up this function into per-clock-cycle steps, we could increase frequency, but it is not clear that this would improve throughput, since it adds ~15 clock cycles per amplitude value for a total of (15 * 262144) = ~3 million extra cycles to the design.