Will Covington and Arthur Leigh-Wood

March 12, 2025

CE387

Professor Zaretsky

FM Radio Design

Section I: Background & Radio Theory

In this project, we implemented the baseband processing of an FM radio in system verilog. We assume that before our system receives the FM signal, the carrier wave has already been removed through a mixing circuit and the input to our system is the unseparated in-phase (I) and quadrature (Q) signals, ranging from 30Hz to 53kHz. This baseband signal is composed of four relevant bands; a 30Hz to 15kHz band containing the left and right mono audio added together, a 19kHz tone containing information of whether or not the stereo signal exists, and 23kHz to 38kHz and 38kHz to 53kHz bands, each containing the right stereo audio subtracted from the left.

The first stage of our receiver system involves separating the I and Q signals and performing the corresponding frequency demodulation operation. To ensure signals outside of the relevant range do not impact our demodulation, we low pass filter the signal to remove frequencies above 80kHz. Then, to demodulate the waveforms, we compute the frequency difference between consecutive signals with the inverse tangent function, as shown in equation (1). The inverse tangent function computes the phase difference between consecutive signals, which is related to the modulated frequency by a differentiation.

$$f = k * arctan(IQ_1 * conj(IQ_0)) \qquad (1)$$

Once the demodulated values are computed, they are copied and separately filtered to isolate the individual bands. The first filter isolates the 20Hz to 15kHz band with a 16kHz lowpass cutoff frequency, while downsampling the signal by a factor of 10. The downsampling operation helps us optimize our system’s resources, as the human ear can only process audio up to around 20kHz. Thus, a downsampling from 320kHz to 32kHz allows 16kHz worth of audio to be represented in samples at the appropriate Nyquist rate without losing much of the 20kHz. The second filter is a bandpass around the 19kHz, followed by a squaring of the signal and a high pass filter to remove the tone at DC that is created from the previous squaring operation. This three-stage process results in an isolated tone at 38kHz. Finally, a bandpass filter isolates the 23kHz to 53kHz band and down samples the signal by a factor 10.

Once the three signals are isolated, the left and right stereo channels are extracted, filtered from noise, and passed through a gain stage to change the output volume. This process occurs through the following steps: first, the signal from the second filtering operation, which is now a tone shifted to 38kHz, is multiplied by the 23kHz to 53kHz band from the third filter. This multiplication shifts the band by 38kHz such that it is now centered around DC. This shifted stereo band is now in the same frequency range as the output of the first filtering operation, which contains the mono left and right audio signal. The mono and stereo bands are combined with addition and subtraction stages to yield the left and right audio, respectively. They are then passed through IIR filters to improve the signal to noise ratio (SNR) and multiplied by a gain factor to control the volume of the output signal. The volume control is the last stage of the receiver pipeline, and the fully processed output of this stage would be fed into a speaker or headphones for listening.

Section II: System Architecture

The implemented receiver architecture is displayed in Figure 1. Each box corresponds to an individual RTL module, each of which communicates through FIFO buffers. FIFO buffers allow the system to stream data asynchronously such that each module can read and write information when they have completed their task. When the previous module has finished its operation on a 32 bit word, the current module reads the word from a FIFO buffer and outputs an updated word to an output FIFO, which can then be read by the succeeding FIFO. The modules are therefore able to process information in a pipelined manner, improving the overall efficiency of the system.

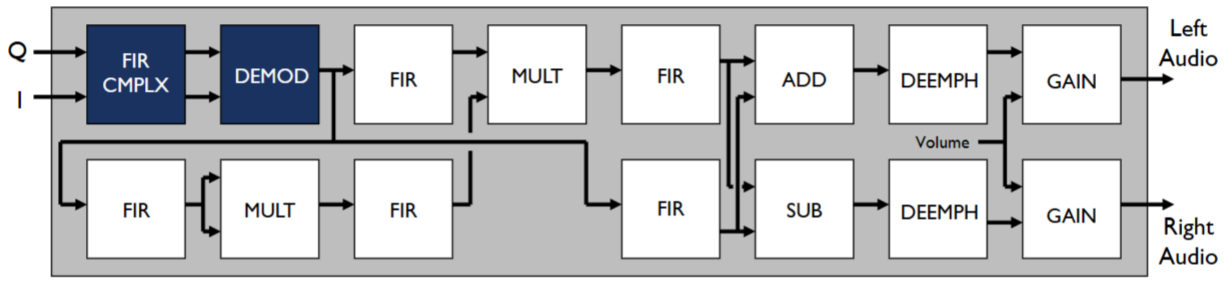


Figure 1: Receiver System Architecture.

The receiver follows the general demodulation and digital processing as described in section I. In-phase and Quadrature signals are separated by the first module (not shown) and are passed into a low pass finite impulse response (FIR) filter to remove high frequency components of the waveform. This filtered signal is then passed through the demod module, which performs the arctan demodulation. The demodulated signal is then copied into three separate FIR filters that isolate the relevant frequency bands; the first FIR (to the right of demod) isolates the 23kHz to 53kHz band, the second FIR (bottom left) isolates the 19kHz tone, and the third FIR (bottom middle, connected to demod) isolates the 20Hz to 16kHz signal and perform the 10x downsampling. In the bottom left, the output of the second FIR feeds into a multiplication module that shifts the 19kHz signal to 38kHz, followed by another FIR filter to remove the generated DC signal. This signal, along with the output of the first FIR, are fed into a second multiply module that centers the 23kHz to 53kHz band at DC. This signal is passed through another FIR filter that performs the 10x downsampling and removes high frequencies, and then is combined with the output of the third FIR (bottom middle) through the add and subtract modules. The add and subtract modules, which extract the left and right audio signals, are each followed by de-emphasis and gain modules, which perform SNR filtering and volume changes.

Section III: Design Process

We decided to break down the system into its individual modules. Will completed the readIQ, demodulate, multiply, add/sub and gain modules; meanwhile Arthur completed the fir, fir complex and deemphasis modules. When developing each module, we analysed the C software code, generated test inputs and outputs (stored in .txt files), wrote the system verilog for the module, top module (buffered by input and output FIFOs) and testbench (bit-true validation). Once each module was working as desired we worked on one computer to stitch all of the modules together. Since about 24 FIFOs were needed in the whole system, we numbered each one and instantiated them using the generate function.

We stitched the whole system together sequentially, at the start we combined readIQ and complex fir, we then added demod, and so on. At each addition of a new module we checked the output was correct before adding the next. This way we could easily identify and fix errors. We also developed our uvm with a placeholder module that mimicked the structure of the whole system, allowing us to address errors in the uvm interface separately from our system verilog modules. Individual modules were tested with testbenches that were all included in the same sim file, allowing us to switch between testing different modules with a change of only a few lines of code.

Once we verified that our top level system implementation was correct, we focused on synthesis and optimizations as a team. Our changes to the demod module, which will be discussed at length in Section IV, uncovered several issues in our reading and writing to FIFOs in other modules. This was largely due to the extended number of cycles that the optimized demod required, and forced changes to the FIR, complex FIR, and IIR modules. Our previous implementation read the input in a combinational block and the output in a clocked block, which we suspect causes a timing issue when the output FIFO becomes full. After a substantial amount of debugging, we found that the system is capable of processing a small number of samples correctly before this reading and writing issues causes errors in our pipeline. We are confident that our system works outside of this error, as the number of correct outputs scales with the FIFO sizes we assign.

It is important to note that the demod module included in the submitted radio code is the unoptimized code, which completes the simulation without error. The optimized demod code is called “demod_fast.sv”, which correctly outputs when simulated outside of the radio pipeline, but creates FIFO clobbering issues when implemented with the full radio pipeline. The “demod_fast.sv” code is used for our synthesis results to showcase the speedup we achieved through optimization. We are confident that the area and timing results are accurate even without a fully functioning, optimized pipeline, as the issues are related to our FIFO communications, and not data processing.

Section IV: Optimizations & Throughput Analysis

Pipelining is a technique used to increase the throughput of a system by dividing a process into multiple stages, with each stage performing a part of the process. This technique was used in the fir, complex fir, and deemphasis modules. When calculating the sum of all the taps in these modules, instead of taking a looping sum, we pipelined all the taps: so the sum could be taken in one cycle: maximising frequency.

The primary optimizations of our system were related to the multiply and demod stages. Due to our use of pipelining in the FIR, IIR, and complex FIR modules, our goal of 100MHz was met in all modules with the exception of multiply and demod. For the multiply module, the bottleneck was a result of a one output per cycle system, causing a path that included reading from the input FIFOs, multiplying, and reading from the output FIFOs. By clocking the data after the multiplication, we are able to increase our frequency by ~13MHz while adding another cycle per operation. We chose not to further optimize the module due to the demod module bottleneck, which determined the operating frequency of our entire system.

The demod module includes several multiplications and a divide operation in succession, and very clearly is the bottleneck of our system. We initially implemented the demod module with three states; one state to read in the input, one to perform the arctan function (including the multiplies and divides), and a third state to output data. Our divides were implemented with a custom divide function, but were not pipelined. As shown in Figure 2, our first system synthesis achieved an operating frequency of 1.8MHz, which was defined exclusively by the demod module operating at 1.8MHz. The longest path in the system was the custom divide function, leading us to our first demod optimization.

To speed up the demod module, we added a state to perform the loop unrolling of the divide function. This change moved our operating frequency from 1.8MHz to around 56MHz, while increasing the number of cycles per output to around 13 operations per second, on average. While this frequency improvement is significant, it is still not sufficient for the requirements of the system. To further improve the operating frequency, we analyzed the new worst paths in the module, which were related to sequential multiplication in the calculations of the “r” and “i” values, the sequential subtraction, xor, and multiplication in computing the output angle, and the unrolled divide itself. We chose to split the output angle computation and the divide unrolling each into two states, which roughly granted us another 16MHz. We also considered the path of sequential multiplications in the calculations of “i” and “r”, which left us with an interesting tradeoff. To keep the computation in a single state, we would achieve a frequency of 69.3MHz with 16 cycles per output. Conversely, splitting the computation into separate states resulted in 72.2MHz and 17 cycles per output. With a brief throughput analysis, we determined that keeping the computation the same resulted in 4.33 MSamples/cycle, while changing resulted in 4.25MSamples/cycle. Therefore, we chose to keep the computation in one state.

To optimize our FIFO sizes, we analyzed the number of cycles each module took to complete one operation. We chose our standard FIFO size to be 32 elements, which included all FIFO instantiations with the exception of those leaving the demod module and entering the top and third bottom FIR, respectively. We selected a FIFO size of 96 for the input to the top FIR, as it must wait three cycles (one from FIR, two from multiply) in order to process data. We selected a FIFO size of 192 for the input of the second bottom FIFO, as it must wait six cycles to process data.

	Before optimization			After optimization		
	Cycles per Output	Frequency	Throughput (Output/sec)	Cycles per Operation	Frequency	Throughput (Output/sec)
Read IQ	4	196.4 MHz	48.6M	-	-	-
Complex FIR	1	126.8 MHz	126.8M	-	-	-
Demod	3	1.8 MHz	.6M	16	69.3MHz	4.33M
FIR	8	104.9 MHz	15.9M	-	-	-
Mult	1	78.4 MHz	78.4M	2	91.7MHz	45.9
Add	1	134.9 MHz	134.9M	-	-	-

Sub	1	134.9 MHz	134.9M	-	-	-
Deemph	1	106.7 MHz	106.7M	-	-	-
Gain	1	220.2 MHz	220.2M	-	-	-
Full pipeline	32	1.8 MHz	.056M	*	70.2MHz	*

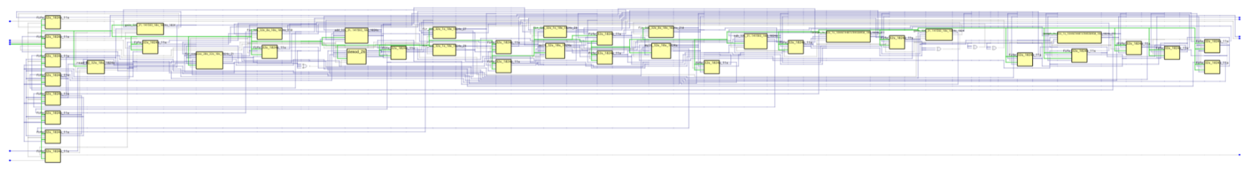
Figure 2: Throughput Table. Note: “-” indicates no change and “*” indicates not simulated (see section III).

Section V: Synthesis Results

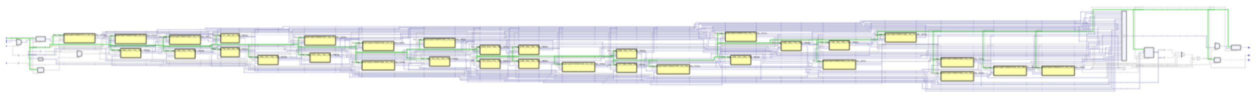
Our synthesis results are displayed in Figure 3. It is clear that our hardware pipelining of the complex FIR, FIR, and IIR units, while demonstrating an impressive operating frequency, utilized nearly all of the available DSP units. In our evaluation of the design, we determined that this is a clear tradeoff; so long as the resources are available, we should focus our attention on achieving the operating frequency requirement. The worst path of our system was within the demod module, as discussed above, and our final design achieved a frequency of 70.2MHz.

	Synthesis Results
LUTs	23370
DSPs	248 (266)
Memory Bits	690176
Frequency	70.2MHz
Longest Path	Demod: q -> a -> angle (angle computation)

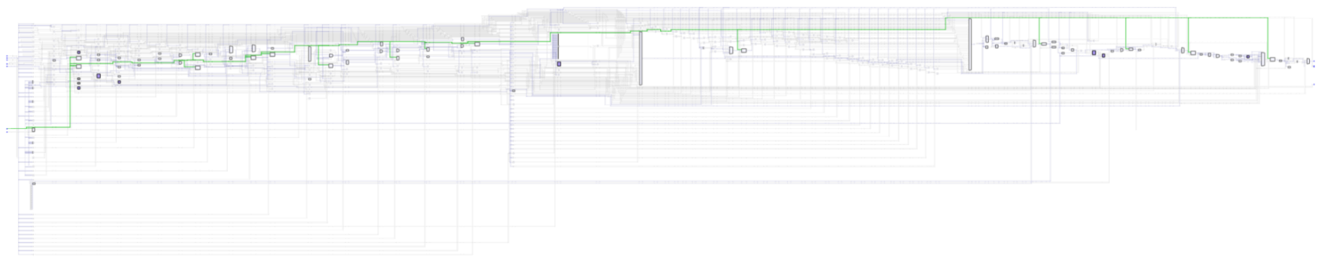
Figure 3: Synthesis Results.



(a)



(b)



(c)

Figure 4: (a) Top Level RTL. (b) Pipelined FIR module. (c) Demod Module.