**EECS 395/495 Final Lab - FM Stereo Radio**

**3/20/2019**

**By Grant Yu**

**Introduction**

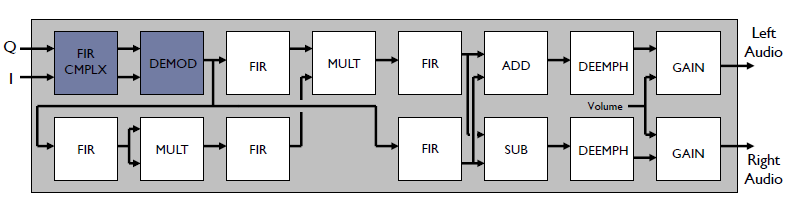
In this lab, we build an FM Stereo Radio in VHDL. The purpose of the design is to parse out a desired input radio signal using filtering and modulation, then transforming the signal into an audio output for a left-right stereo system. Therefore, the Stereo Radio design has two main components: the receiver and the transmitter.

An FM receiver must pick up a signal from the air, and then parse out the desired input signal, which is enveloped inside of a carrier frequency. The incoming signal is assumed to arrive in frequency domain form; rather than an amplitude and its sample time, we receive a complex value. These complex numbers can then immediately be sent into a complex FIR filter that removes frequencies above 80 kHz. This filtering is crucial since the relevant parts of the signal are contained below this frequency, like the Mono audio, Stereo audio, DirectBand, and Pilot tone.

We take advantage of the fact that for FM, the instantaneous value of the signal determines the frequency; therefore, it would be of favor for us to compute instantaneous frequencies of our baseband signal. We may do so by computing the instantaneous derivatives of the frequency between every 2 samples. Then, we attempt to identify if a stereo signal exists by extracting a pilot tone at the 19 kHz band. We square the frequency to get a squared pilot tone, which allows us to move the L-R channel over to baseband. Only then can we reconstruct the left and right channels. IIR filters at the end then improve the signal-to-noise ratio, and the gain stage at the end provides volume control. The extra feedback provided by the IIR filters (from output) strengthens its ability to smoothen the signal.

**System Architecture**

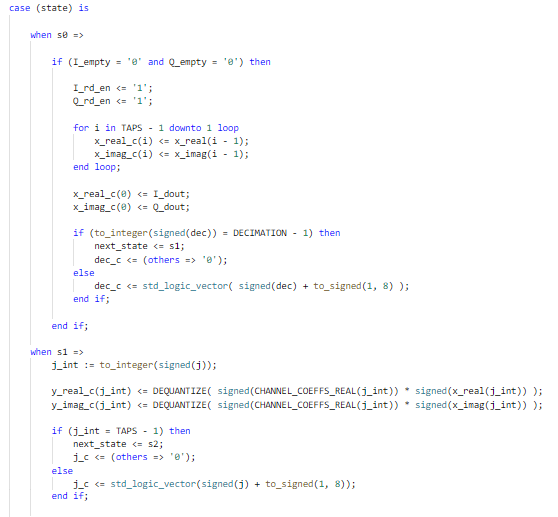
The entire system runs on streaming architectures with FIFOs. The design generally follows the figure presented in lecture:

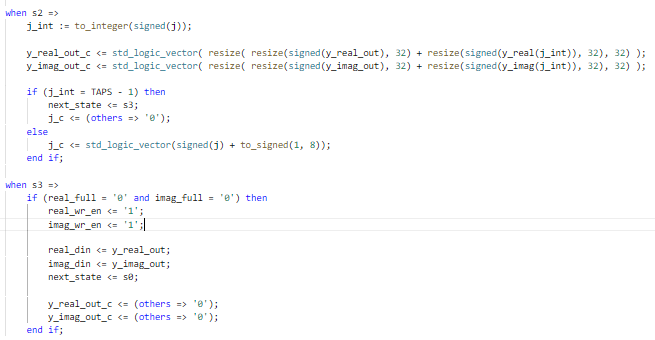


While these blocks above are the main components of the system, the highest level entity for our purposes is the testbench. From the testbench, we read in text data and feed it into a readIQ entity that appropriately parses out the byte stream for the I and Q values. This entity is part of a wrapper entity called radio\_top, which instantiates all of the above blocks and wires them all together.

After the readIQ parses the data, the entity sends the data to the complex FIR entity, which performs the decimation and weighting of the impulse response values, then sums them all together. As shown below in the fir\_cmplx entity, we have several states do each of these functions, as the C code did in for loops. In state s0, we perform a single decimation and input acceptance when we read in from the input FIFO. In s1, we perform the fixed-point multiplication and dequantization for every response of the impulse. The weights used for fir\_cmplx and the rest of the entities are located in the constants package in the work directory.

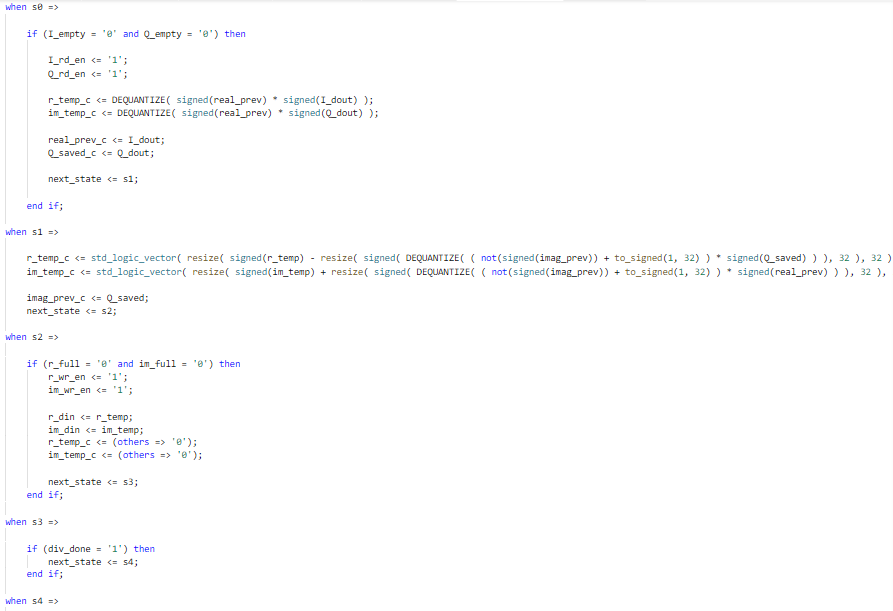
Once we compute these multiplications, twice per cycle (unrolling the multiplication further will unnecessarily use up multipliers), we move on to s2. Here, we sum up all of the impulse response terms for both the real and imaginary aggregates. The final sum becomes the output of the entity. Every accumulated value resets after it is done being used (for example, the counter ‘j’ and the accumulated real and imaginary values).





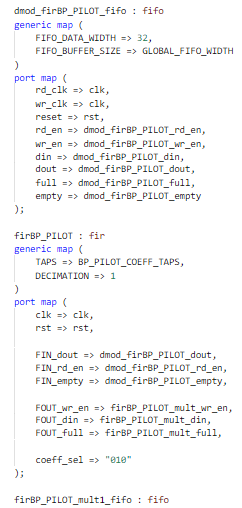
The fir\_cmplx feeds out a real and imaginary output, which streams down to the demodulator, which differentiates the frequency to ascertain the signal. At any time we multiply 2 fixed point numbers, we dequantize the result using an inline dequantize function. When we divide 2 fixed point numbers, we multiply the dividend by the quantization factor (done in the quantize function), then perform division. These macros are used throughout the entire design to maintain the integrity of fractional values.

The demodulator must be able to compute the inverse tangent of the imaginary and real components of the wave (in order to find the phase angle). To compute the inverse tangent, we stream the data into the qarctan entity, which computes the arctan and sends out a ‘done’ signal as a handshake to the demodulator entity. This handshake allows the demodulator to proceed to the next state. Within the qarctan entity, instead of using a trigonometric look-up table, we use an arithmetic algorithm instead—one that requires division. We use a combinational divider from a previous course to complete the task. This divider operates on the rising edge of a start signal, allowing our qarctan entity to control the divider as necessary. A difficult bug was found in the divider in the debugging process; the divider only worked on unsigned numbers, so additional logic was implemented to first divide the absolute values of the operands, and then to determine the resulting sign. The demodulator’s computations and divider handshake are shown here:

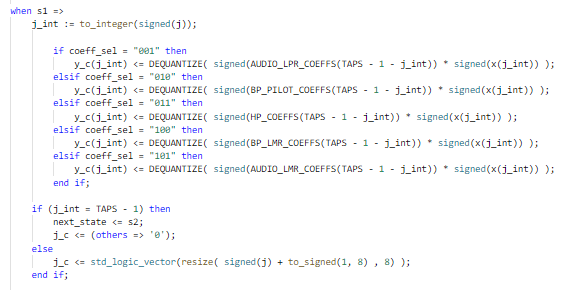


The demodulator’s output is passed to three different FIRs: the stereo pilot filter, L+R filter, and the L-R filter. After writing the demodulator entity, we modify it to have 3 ports for FIFO outputs, to avoid read contention. The FIR entities are very similar to the complex FIR; the one major difference is fewer lines of code. The overall idea of these FIRs is identical to the complex FIR.

The following screenshot demonstrates the top-level connections between fifos and the band-pass FIR for the pilot tone.



Of special note is the coeff\_sel input to the fir entity, as seen above by the “010” input. Because we have 5 instantiations of the fir filter that all use different impulse response weights, we must have a parameter that selects the appropriate array. This feature is shown in the below screen capture:

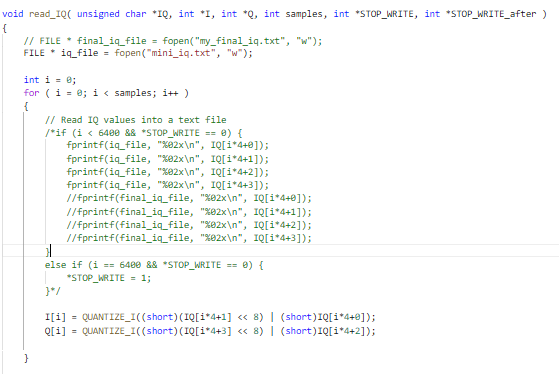


The remaining entities are mult, add, sub, iir, and gain\_n. The former three are straightforward and were easily done. The iir entity is extremely similar with the fir, with the accumulation incorporating feedback from the output. The data splits into a left and a right stream, with the left stream channeling through the add, iir, and gain\_n, and the right stream channeling through the sub, iir, and gain\_n. After the 2 sets of data stream out of gain\_n, they are fed into output fifos and can be played via compilation of audio code.

**Design Process**

The first thing to consider before even tackling the main blocks of the design was how to implement an IQ reader. The IQ reader is the first entity to process the input byte stream, so it was to be done first. After the IQ reader was done, a top-level wrapper (bare bones at this point) and a general testbench were created as a means to interface with the IQ reader.

At this point of the design process, we can start to verify our design. We have an IQ reader entity, and the C code has an IQ reader function, so it made sense to find a way to get the data from the C code. Thus, we modify the C code to print out test values:



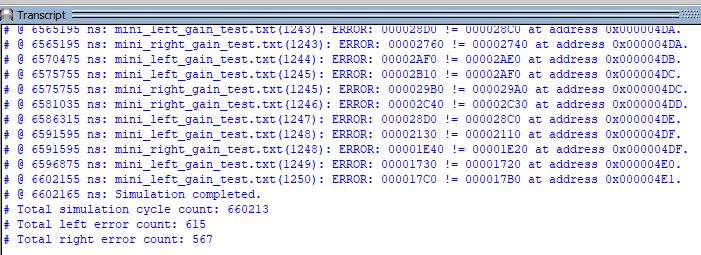
Similarly, we do the same for fir\_cmplx, demodulate, etc. We build the design, then verify the functionality by comparing values with the C code generated output. This iterative design process allowed the project to be completed at a steady pace, but the decision to implement the signed division logic, as mentioned in the previous section, was very difficult. We were not aware that the divider could only operate on unsigned numbers. The issue was solved by performing the computations on paper and comparing them with the waveform, and then realizing that the common denominator between all incorrect demodulator outputs was that they were produced from signed inputs.

**Optimizations**

An idea for optimization is to reduce the FIFO buffer sizes. After getting the design to work with a data size of 16 (for all FIFOs), we tried 8, then 4, then 2. Even at a size of 2, the design works. Loop unrolling was skipped over as an optimization technique because the design has already used an exorbitant number of multipliers (See Analysis and Synthesis section).

**Performance**

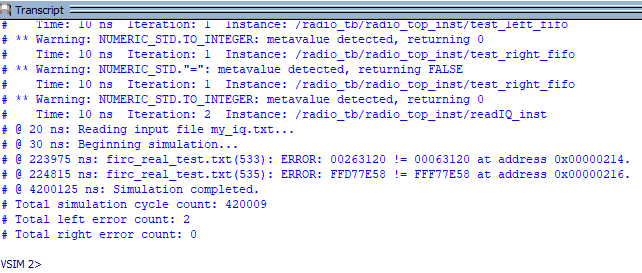
We tested the data with a text file of 40k bytes that was printed out by the C code when it processed the usrp.dat file. Because we decimate by a factor of 8 inside of the L-R and L+R filters right before the stereo portion, the output data will have one eighth of the data of the input (5000 bytes). When we run the simulation on ModelSim, we get the results:



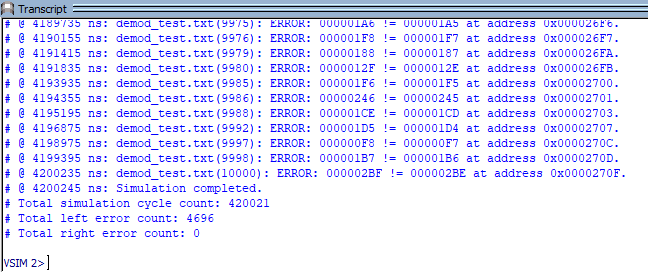
For 1250 output words, the idea of approximately half of them being errors (as seen above) is alarming. However, it is clear that all the discrepancies are simply off by insignificant magnitudes of x“10” or x“20.” We hypothesize that this miniscule error came from the demodulator’s divider, whose rounding could differ from the C code’s. Up until the demodulator was tested, the outputs were all correct, but after the divider, all values were just 1-3 off. At the gain\_n, these errors become amplified as the data themselves amplify. The data is not perfect, but it is very close.

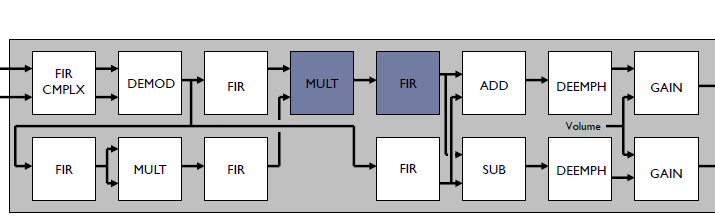
The total simulation cycle count is 660213, left audio error is 615, and right audio error is 567. We have 5000 b outputted in 660213 \* 10 ns. At this rate, we have 757331 bytes/second, or 757 Mb/second.

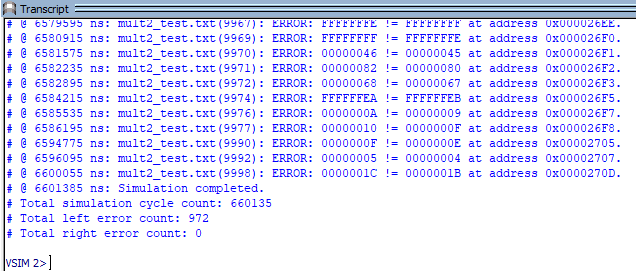
For the function fir\_cmplx, we have a throughput of 40k bytes in 420009 cycles. We have a rate of 9523605 bytes/second, or 9.52 Gb/second.



The fir\_cmplx is directly pipelined via FIFO to the demodulator. The results are shown as below. There is only a slight increase in the total simulation cycle count, which is attributed to the pipelining start and end (fill and drain). The rate is still at 9.52 Gb.



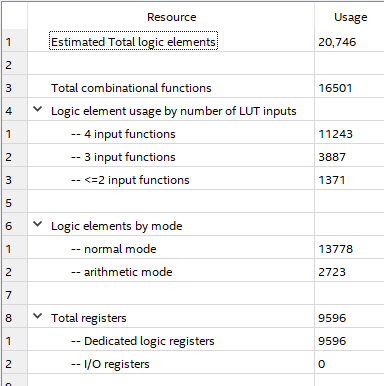
We skip over the next FIR because it is just another stage in the pipeline. The next interesting stage is the 2nd mult, shaded in blue:

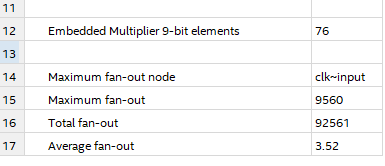


We see that there is a significant increase in the cycles needed. The three stages before the second mult (fir, first mult, fir), are the bottleneck in this path, and clearly delay the pipeline immensely. The new cycle count is 660135. The throughput is now just 6 Gb/s. Note that our cycle count 660135 is extremely close to how many cycles the design takes overall, 660213. Therefore, the remaining 78 cycles completely use up the rest of the blocks: the add, sub, iir, and gain\_n. We conclude that the pilot filter triplet stage is the culprit of the bottleneck, and if future improvements for performance were to be made, these 3 stages should be targeted. An idea for future development would be to further divide these stages into more pipelines.

**Analysis and Synthesis**

The following screenshot captures the synthesized resource utilization summary.

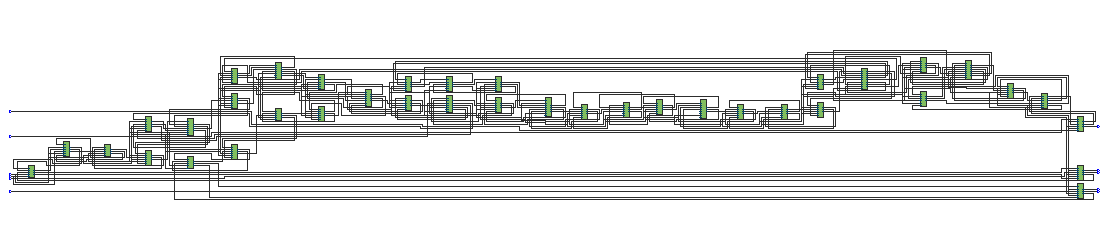




Of particular note is that the reported memory bit usage is 0. However, we can calculate the memory usage manually. We have 27 FIFOs, each holding 2 32-bit elements. We have 27 \* 2 \* 32 = 1728 memory bits.

We have 14846 total registers, and 76 multipliers (although none are unrolled, and quantization involves sll, not multiplies).

The RTL viewer provides the schematic:



**Conclusion**

The FM Radio Stereo has been a challenging exercise in VHDL, DSP, planning, and debugging. The implementation of digital filtering is a powerful tool to extract and process signals to our advantage, and will have applications in many domains and capacities. A difficult but rewarding lab, the FM Radio Stereo packages together a potpourri of advanced VHDL concepts into a powerful application.

Note: To run simulation, type: “do sim.do”.