CE387 FM RADIO FINAL PROJECT

Luc Ah-Hot, Mitch Weng, Weihe Gao

March 8, 2024 (Winter 2024)

Table of Contents

1. FM Radio Background & Theory

Frequency Modulation (FM) radio represents a remarkable synthesis of physics, engineering, and design that collectively create a robust platform for audio transmission. By allocating a band of frequencies between 88 to 108 MHz, FM radio provides a broad spectrum for numerous channels to operate concurrently without interference. Each FM station is assigned a unique carrier frequency that is modulated, or varied, in accordance with the audio content that the station broadcasts. When tuning to a station, one is actually selecting the station's modulated carrier wave. The FM receiver then demodulates this wave to extract the audio signal, enabling it to be heard through the speakers.

FM's immunity to noise stems from its method of encoding audio information within the frequency variations of the carrier wave, rather than its amplitude. This approach inherently reduces susceptibility to static and interference that commonly plague Amplitude Modulation (AM) systems, thereby enhancing sound quality. At the broadcast end, the audio signal causes the frequency of a stable high-frequency carrier wave to vary; this variance is proportional to the audio signal's amplitude at any given moment. The frequency deviation during this process is controlled and is within a specific range, which is integral for ensuring clarity and consistency of the signal received.

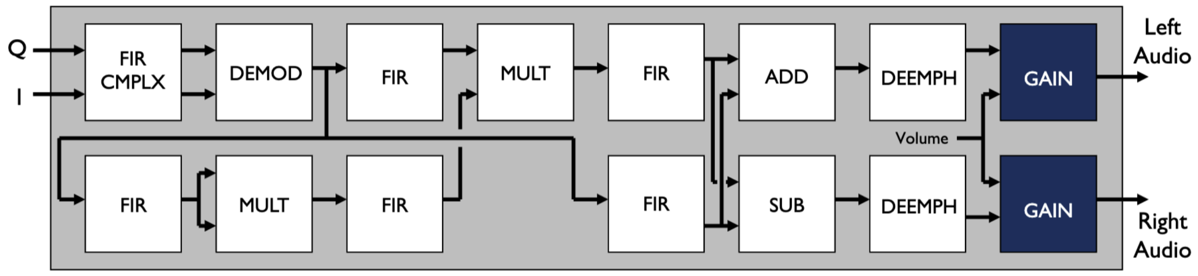
Upon reception, the FM signal undergoes several stages of processing within the radio receiver. An antenna first captures the signal, after which it enters a tuner. Contemporary FM radio tuners often utilize a phase-locked loop (PLL) for precise selection of the desired frequency. Following tuning, a bandpass filter is employed to isolate the carrier frequency while eliminating undesired signals and noise from adjacent channels.

The core of FM reception is the demodulation stage, which is pivotal in recovering the original audio signal. Utilizing either a discriminator or a PLL-based demodulator, the FM radio converts frequency variations of the carrier wave back into voltage variations, which correspond to the audio signal's amplitude. Discriminators achieve this by contrasting the incoming FM signal against a reference, producing a voltage that reflects the frequency deviation. Conversely, a PLL demodulator adjusts its own oscillator to lock onto the phase of the incoming FM signal, faithfully tracing the frequency shifts to reconstruct the audio signal.

To finalize the reception process, any remaining unwanted high-frequency components or noise are filtered out by a low-pass filter, ensuring only the desired audio frequency range is retained. The signal then progresses to an amplifier, and ultimately, to the speakers where the electrical impulses are transformed back into audible sound waves.

Through this intricate and technical sequence of frequency manipulation, signal filtration, and precise demodulation, FM radio continues to be a staple in broadcasting, providing high-fidelity audio transmission that is less susceptible to interference, establishing itself as the dominant medium for music and entertainment broadcasting.

2. System Architecture



The system architecture very closely follows the diagram from the slides above. It has an extra read_iq module to parse the inputs from the raw data file in the beginning, and also FIFOs in between each module to stream the data. It is worth noting that there is extra logic for the 3 fir modules that take input from the demod module. These three fir modules communicate with each other to sync up their read from the FIFO containing demod out. The multiply module used to square the bp_pilot signal has also been changed to a square module, which only takes in one input and squares the value. The FIFOs before the add and sub modules also have some logic to synchronize their reads. Both output FIFOs for the two FIRs must have values that are ready to be read before the add and sub modules can perform their calculations. The two deemph modules in the diagram are replaced with iir modules within our architecture.

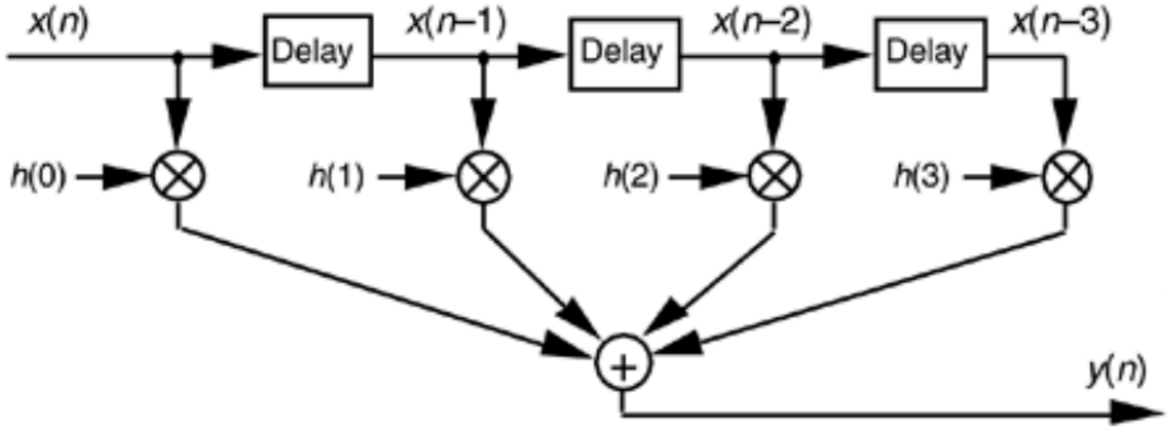
There is also a lot of extra logic and calculations done within the demodulate function, specifically hardware division and qarctan. The demodulation part begins with two parallel input FIFOs that buffer the incoming real and imaginary components of the signal. These FIFOs serve to synchronize the flow of data to the demodulation module, ensuring that the real and imaginary parts of the signal are processed in tandem.

Once the data is ready, the demodulation module, which operates under a state machine control, starts processing the inputs. The state machine governs the behavior of the module through various phases: reading input values, executing complex multiplication, and sending data to a dedicated qarctan module for arctangent computations—necessary for frequency demodulation. The arctangent computation is crucial as it helps to determine the instantaneous phase difference, which is then used to extract the original information from the frequency-modulated signal.

3. Design process

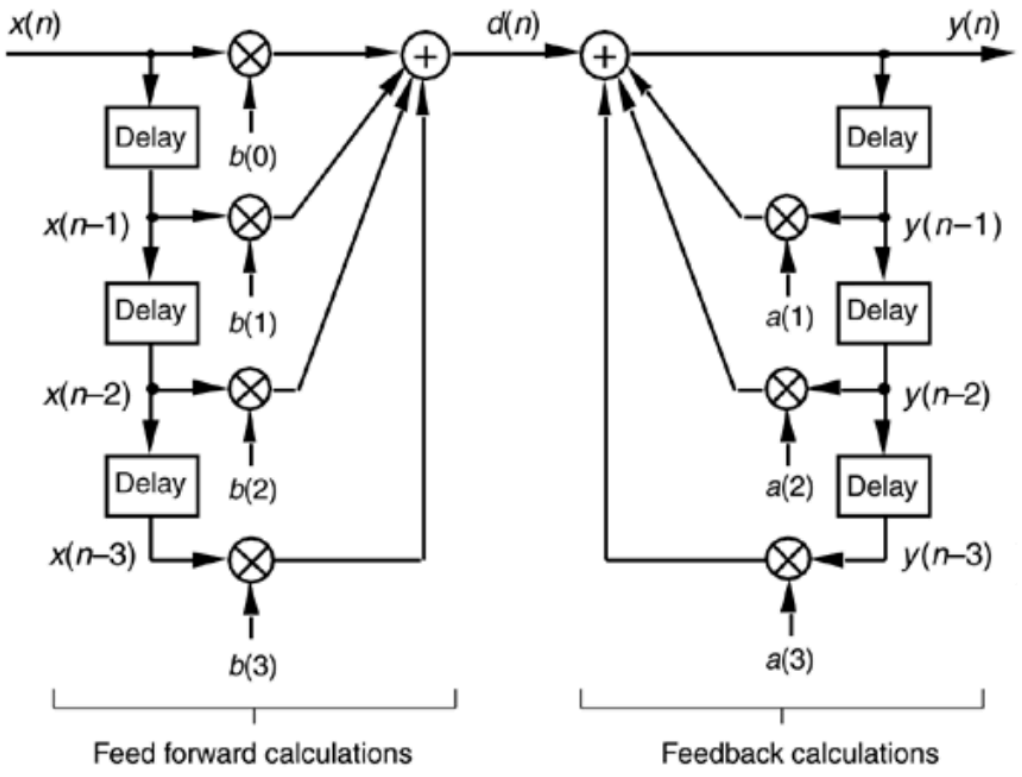
Filters:

All the filters, including fir, iir, and fir_cmplx were built with a similar FSM architecture. Starting with fir, the first stage for each new computation of a single output starts with shifting in new values from the input FIFO and decimating according to the DECIMATION parameter. The two firs that take the output from the demodulate module have a DECIMATION parameter of 8 which means they shift in 8 new values at a time before starting a new computation. Once 8 new values have been shifted in (the last 8 values in the shift register get shifted out), then it moves on to start computing the finite impulse response. This process is split up into 2 steps that are pipeline together which include multiplying each element inside the shift register by a corresponding constant that determines the type of filter, and then dequantizing it (dividing by 2^{10}), and accumulating these products into a sum like a dot product. We split this up into 2 states to increase the performance frequency and pipelined it to save clock cycles. After the sum is calculated, it is outputted and written into an output FIFO.



fir_cmplx is very similar in nature but has no decimating so the number of outputs equals the number of inputs and calculates two sums. These are the real and imaginary outputs of read_iq that are fed into the demodulating module. Everything's the same except that it only shifts in 1 new value into the shift register before starting the whole MAC computation.

iir or infinite impulse response takes into account feedback from the calculated sum. After shifting in the correct amount of inputs as the fir, it also has a shift register to hold the past sums to be used in the current calculation. Every new computation, all the values inside this past sums shift register are also multiplied by another set of coefficients, and then that sum is added to the sum calculated taking in new inputs (essentially the same sum as a normal fir MAC). These 2 partial sums are added together and stored as the first element of the past sums shift register to be used in the following calculations. The output of the entire iir module is the value at the end of the part sums shift register.



Demodulation:

The demodulation module takes in a real and imaginary value from the read_iq module and differentiates frequency by finding the difference in angle of phase between consecutive I/Q samples. This means that this module needs to keep track of previous real and imaginary values to be used in the next computation. A new real and imaginary value is calculated using a combination of current and previous real/imaginary values, then this is fed into a qarctan function before being multiplied by a gain constant. The following equation was used in the demodulate module:

$$\text{demod} = k * \text{atan}(\, IQ_1 * \text{conj}(IQ_0)\,)$$

This qarctan calculates the difference in angle of phase between consecutive I/Q samples and requires a divide by variable. Since this is extremely expensive in hardware (versus a divide by a constant), we also had to create a dedicated division module. The division was based off of the divide module we made in CE355, where we were essentially implementing a iterative long division process to find the quotient and remainder of a division. We also used a recursive get_msb_pos() function to reduce the amount of iterations this function had to go through to find the master bit position. Since divisions have to be done over many cycles, this was definitely a part of the design that created many bottlenecks until we split up its long states into multiple smaller states.

Read_IQ:

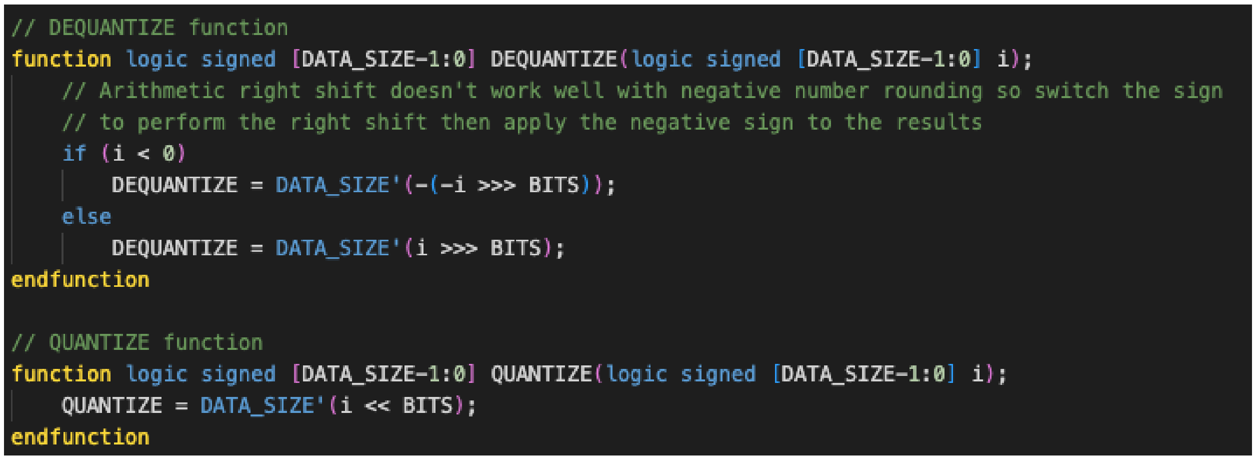
The Read_IQ module parsed the inputs into I and Q components that could be processed by the fir_cmplx module. Since the raw data file consisted of one chunk of IQ data being 4 bytes, the module reads in 4 bytes at a time, differentiates between I and Q, then outputs I and Q to fir_cmplx as 4 byte values.

Other Modules:

Other modules include add, sub, multiply and square. Add and sub are simple addition and subtraction functions that are clocked to wait for the outputs from lpr_fifo and lmr_fifo. Multiply and square also takes inputs from different fir filters and clocks the multiplication process.

4. Optimizations and Impact

One of the major bottlenecks in our design at the start was being able to dequantize multiplication products correctly with no error and without causing huge delays. Since we are quantizing with a fixed-point bit value of 10, we could equivalently divide by 2^{10} or 1024 to dequantize a value correctly. While this logically works, it tanked timing performance when we synthesized any module. Instead, we decided to use an arithmetic right shift of 10 bits to achieve the same results. Because of the fact that this did not work with negative numbers, we added some extra logic to make sure that an arithmetic right shift would always yield the same results as a divide by 1024. Although this increases delay, it is still much faster than using a divide by 1024. Below is a screenshot of the DEQUANTIZE and QUANTIZE functions we used throughout the entire design.



```
// DEQUANTIZE function
function logic signed [DATA_SIZE-1:0] DEQUANTIZE(logic signed [DATA_SIZE-1:0] i);
    // Arithmetic right shift doesn't work well with negative number rounding so switch the sign
    // to perform the right shift then apply the negative sign to the results
    if (i < 0)
        DEQUANTIZE = DATA_SIZE'(-(-i >>> BITS));
    else
        DEQUANTIZE = DATA_SIZE'(i >>> BITS);
endfunction

// QUANTIZE function
function logic signed [DATA_SIZE-1:0] QUANTIZE(logic signed [DATA_SIZE-1:0] i);
    QUANTIZE = DATA_SIZE'(i << BITS);
endfunction
```

After writing up the RTL for the filter modules, the demodulate module became the biggest bottleneck for the design. The initial iteration of demodulate (which includes qarctan and div) had a maximum frequency of 14.3MHz. After changing the dequantization function from a divide by 1024 to the above function, we were able to run demodulate at 31.3MHz. Changing the get_msb_pos() function inside of the division module from a linear function to a binary tree function with log(N) versus O(N) time complexity then increased our frequency to 37.8MHz. It became clear that the states in both qarctan and div had to be broken up in order to achieve higher clock speed. After doing so, this jumped our demodulate frequency to 81.3MHz. Since the target frequency was still around 100MHz, we decided to re-write demodulate using simpler states and logic, balancing out combinational logic accordingly. This resulted in a final demodulate frequency of 106.8MHz. When redesigning the module, we understood the importance of writing minimal and clean code to decrease unnecessary delays and resource utilization. The same rationale was used in the other modules at this point and we saw increases in performance as well: 95MHz to 125MHz on the fir modules for example.

Very basic pipelining was also employed for the filter modules as reading the values from the shift registers took a lengthy delay so we chose to split up the reading of values from the shift register from the multiplication and also from the dequantization and accumulation. This resulted in pipelining multiplications with the accumulation + reading in new value from the shift register in the 3 filter modules. We were able to save on clock cycles while not bottlenecking the design with this expensive operation if done in one clock cycle.

Lastly, loop unrolling was used in the fir and fir_cmplx modules as these required 32 and 20 inlined multiplications respectively if no loop unrolling was performed. We were able to greatly reduce our cycle count (mentioned in the next section) through this process.

5. Simulation and Performance Results

We tested our architecture using UVM by comparing our left and right audio outputs to the outputs of the C file that was printed out into txt files. The inputs were byte values directly from the usrp.dat file that were read into the read_iq module which configured them into 32-bit I and Q values. These inputs were run through our fm_radio module and the output was the outputs of left and right gain modules at the end of the block diagram we covered in class. We decided to run the simulation for 32000 inputs, expecting back 1000 outputs since our decimation or downsampling parameter was 8 and each byte input was configured into a 4-byte output.

Without loop unrolling, the simulation time required was 2775515 ns, and since the clock period was kept at 10ns, that results in 277551 cycles being required to process 32000 inputs. This comes down to requiring about 277/278 cycles to calculate 1 output including downsampling by a factor of 8 each time.

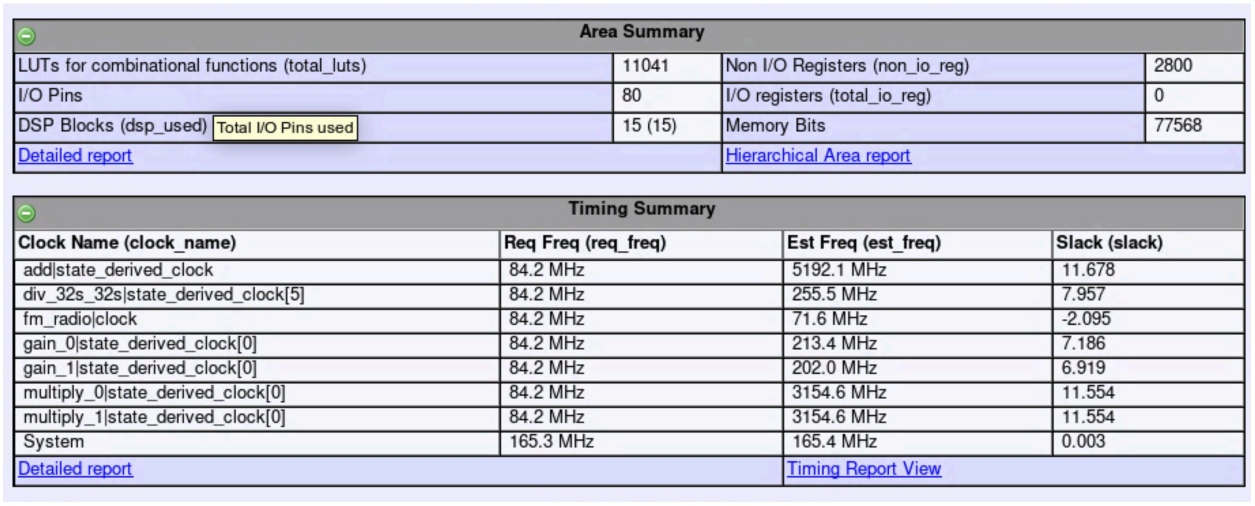
After unrolling the fir and fir_cmplx modules by 4, the total cycle count was 195423 for 32000 inputs. This comes down to 195 cycles to calculate 1 output. This is a 29.6% decrease in cycle count which translates to a 29.6% decrease in process time given that both tests use a clock of 10ns. We tried unrolling more i.e. by 8 or 16 but the total cycle count remained the same over the entire fm_radio. We suspect this means that the fir or fir_cmplx do not become the bottlenecks after a while as the FIFOs filling up may be what is restricting our cycle performance.

With regards to the FIFO widths, we chose the smallest width to limit our design size and delay. This came down to 16 elements per FIFO. If this were to increase, there is a possibility that unrolling some of the modules by a factor of greater than 4 would have yielded better cycle counter results.

Final throughput = 0.032 MB / 0.001954235 seconds = **16.37 MB/s** with the bottleneck being the demodulate module as it has the heaviest combinational logic required out of all the blocks.

6. Synthesis Results & Discussion

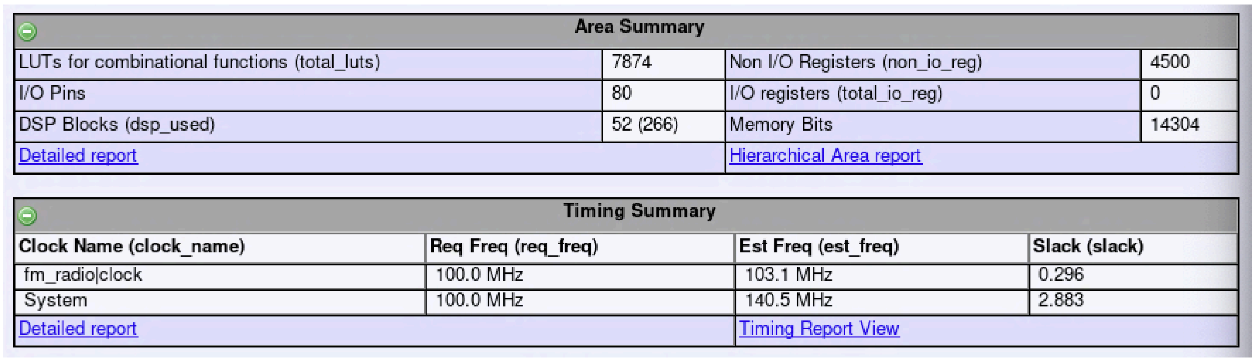
Before optimizing all are modules, we were getting the following synthesis results:



Area Summary			
LUTs for combinational functions (total_luts)	11041	Non I/O Registers (non_io_reg)	2800
I/O Pins	80	I/O registers (total_io_reg)	0
DSP Blocks (dsp_used) Total I/O Pins used	15 (15)	Memory Bits	77568
Detailed report		Hierarchical Area report	

Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
add|state_derived_clock	84.2 MHz	5192.1 MHz	11.678
div_32s_32s|state_derived_clock[5]	84.2 MHz	255.5 MHz	7.957
fm_radio|clock	84.2 MHz	71.6 MHz	-2.095
gain_0|state_derived_clock[0]	84.2 MHz	213.4 MHz	7.186
gain_1|state_derived_clock[0]	84.2 MHz	202.0 MHz	6.919
multiply_0|state_derived_clock[0]	84.2 MHz	3154.6 MHz	11.554
multiply_1|state_derived_clock[0]	84.2 MHz	3154.6 MHz	11.554
System	165.3 MHz	165.4 MHz	0.003
Detailed report		Timing Report View	

Even while using the wrong FPGA board, we can see that there are a lot of latches being created, denoted by the many inferred clocks. Our maximum frequency here is also only 71.6Mhz. After optimizing the bottleneck critical path, we were able to achieve the following results:

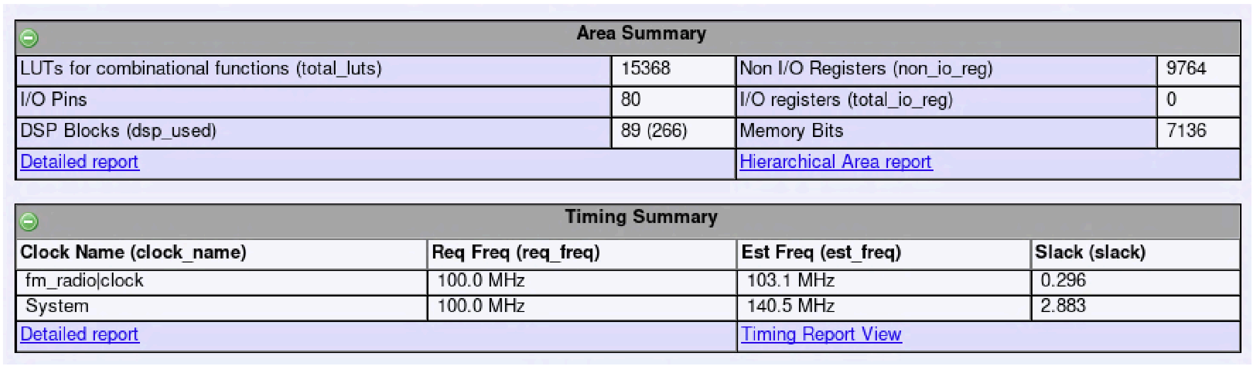


Area Summary			
LUTs for combinational functions (total_luts)	7874	Non I/O Registers (non_io_reg)	4500
I/O Pins	80	I/O registers (total_io_reg)	0
DSP Blocks (dsp_used)	52 (266)	Memory Bits	14304
Detailed report		Hierarchical Area report	

Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
fm_radio|clock	100.0 MHz	103.1 MHz	0.296
System	100.0 MHz	140.5 MHz	2.883
Detailed report		Timing Report View	

Since there are more than enough dsps or multipliers for our entire design, LUT usage is able to go down. Total combinational usage is at 6%, with only 3% of total register usage.

To reduce our cycle count, loop unrolling was used but this created problems with Synplify's interpretation of shift registers. When implementing loop unrolling, the tool was not able to infer a "seqshift" element for the shift registers and used instead a chain of flip flops. While this is not ideal, the near 30% reduction in total cycle count was worth the increase in LUTs, dsps, and registers. Below are the synthesis results from the unrolled design.

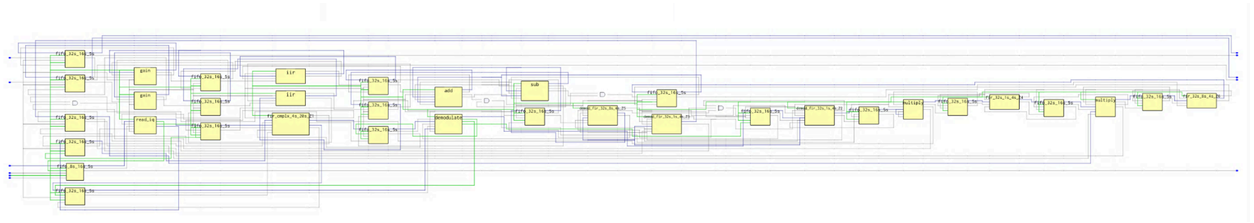


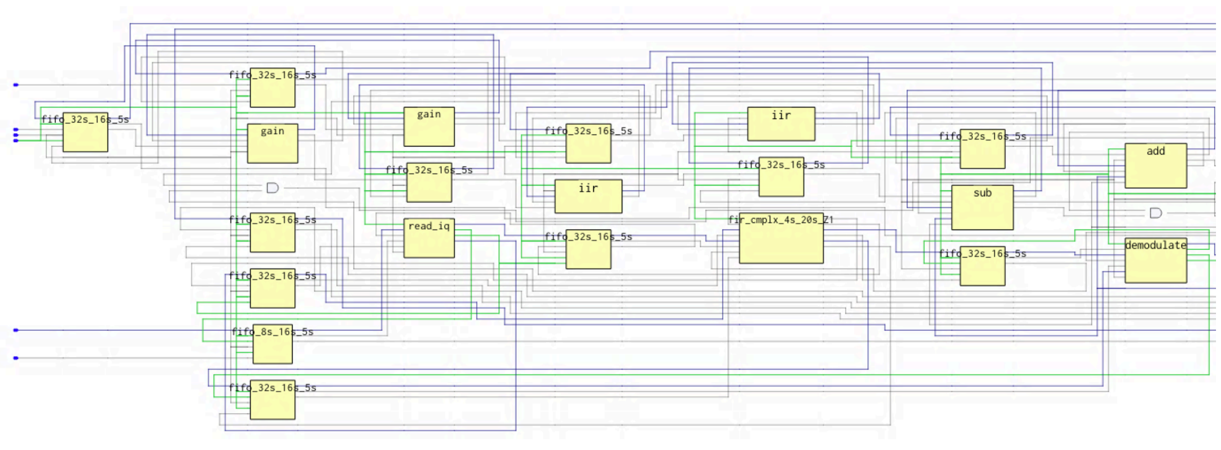
Area Summary			
LUTs for combinational functions (total_luts)	15368	Non I/O Registers (non_io_reg)	9764
I/O Pins	80	I/O registers (total_io_reg)	0
DSP Blocks (dsp_used)	89 (266)	Memory Bits	7136
Detailed report		Hierarchical Area report	

Timing Summary			
Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
fm_radio|clock	100.0 MHz	103.1 MHz	0.296
System	100.0 MHz	140.5 MHz	2.883
Detailed report		Timing Report View	

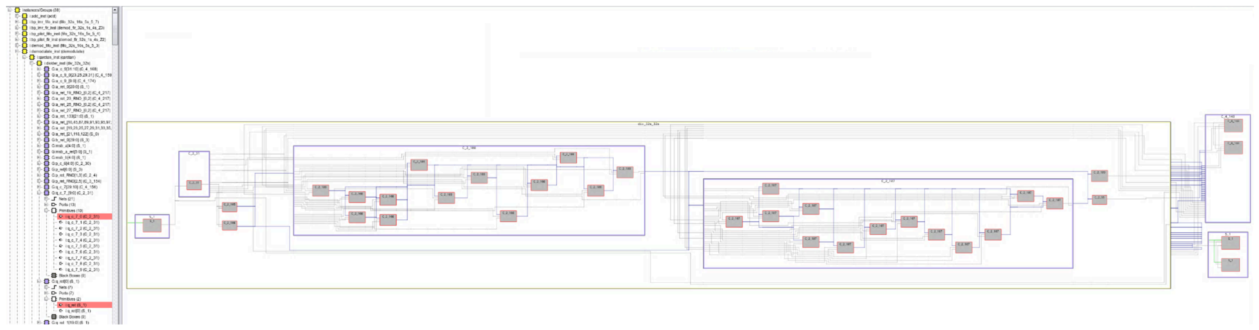
We increased our LUT usage up to 13% and register usage to 8%. We can also see an increase of dsp usage from 20% to about 33.5%. Since this did result in a cycle count decrease of 30%, we found this increase in resource utilization well worth it. It's also important to note that we are still way below the limit on how many resources we are using given our relatively low resource utilization rates. Our final frequency achieved was **103.1MHz** which is perfectly at our target of 100MHz. As mentioned in the previous section, the limiting module was the demodulate block as it required the heaviest amount of combinational computations.

This is the RTL module view of the synthesized design:

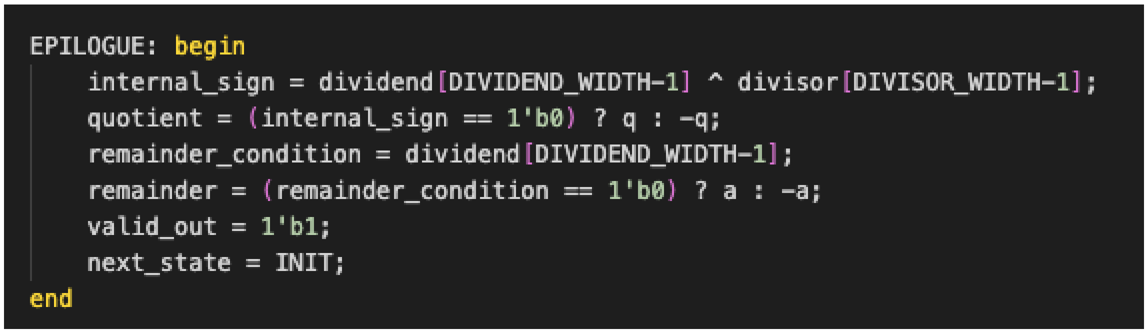




This is the critical path in the technology view:

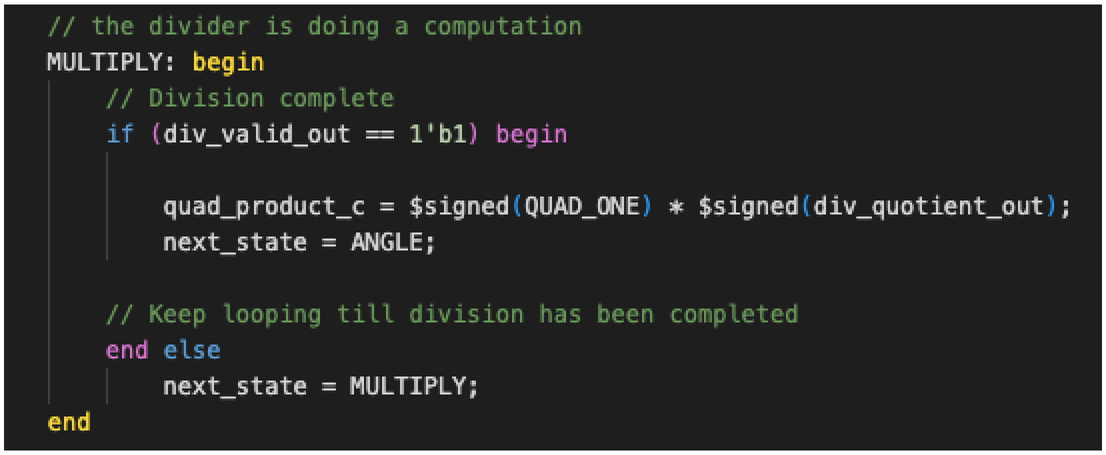


The path is through the divider and qarctan corresponding to the following code:



```
EPILOGUE: begin
    internal_sign = dividend[DIVIDEND_WIDTH-1] ^ divisor[DIVISOR_WIDTH-1];
    quotient = (internal_sign == 1'b0) ? q : -q;
    remainder_condition = dividend[DIVIDEND_WIDTH-1];
    remainder = (remainder_condition == 1'b0) ? a : -a;
    valid_out = 1'b1;
    next_state = INIT;
end
```

Start of the critical path in the divide epilogue state.



```
// the divider is doing a computation
MULTIPLY: begin
    // Division complete
    if (div_valid_out == 1'b1) begin

        quad_product_c = $signed(QUAD_ONE) * $signed(div_quotient_out);
        next_state = ANGLE;

    // Keep looping till division has been completed
    end else
        next_state = MULTIPLY;
end
```

End of the critical path in the qarctan module receiving the division.

7. Conclusion

This project was very different from the 355 final project as there were many more modules that we had to consider. We also had to consider optimizing our hardware division to improve the frequency of our design to be near 100 MHz. Quantization was another factor that led to some confusion as multiplication and division required values to be quantized and dequantized to make sure that the correct output was being achieved. We also had to think about synchronizing signals when streaming our data, since there were data paths of different lengths that required the use of the same data. The concept of loop unrolling to run computations in parallel was also another aspect of optimization that was used.

We also struggled during synthesis as we tried to synthesize with a smaller board that only had 15 dsps instead of 266. Since there weren't enough multiplies, it required the use of many more LUTs to compensate for the lack of multipliers and this resulted in not having enough resources on the FPGA for our design. This proved to be a great learning experience as it made us reconsider the size of our FIFOs and also be conscious about the resources available to us. Although we ultimately did not have to optimize our design since using the correct FPGA chip gave us a lot more resources to use, it did show the importance of resource management when designing for FPGAs as there is more than having a design run logically correctly and with good timing performance.

Compared to the previous assignments and the ones in CE355, this final project challenged us to consider all factors (except power) in hardware design. The tedious process of finding bottlenecks in our sub-modules to speed up the top module over and over again really honed our skills in analyzing critical timing paths and balancing the tradeoff between more cycles versus a higher clock speed.