CMPEN 417 – Final Project Report

Nathan Litzinger, Andre Mitrik

Contents

Overview

Project Introduction

In this work, we explore the functionality of a 25-tap (coefficient kernel of size 25) complex frequency impulse response (FIR) filter in conjunction with a coordinate rotational digital computer (CORDIC) rotator that converts cartesian outputs to polar form. The design not only focuses on functional correctness but also optimizing digital signal processor (DSP) resources usage for compact and efficient implementation on a Zynq-7000 FPGA.

High-Level Structure

Our project utilizes separate sections; the first takes care of frequency impulse response calculations given complex inputs and a filter of a desired kernel size which yields cartesian coordinates. Then, these cartesian coordinates are fed into a CORDIC rotator which converts the cartesian coordinates to polar form.

Function Design

`void complex_mult(int ar, int ai, int br, int bi, int* o_r int* o_i)`

This function takes in two complex numbers, multiplies them together, and then stores the output in a placeholder variable found though a pointer sent as a parameter.

`void fir(int input_r, int input_i, int filter_r[KERNEL_SIZE], int filter_i[KERNEL_SIZE], int* output_r, int* output_i)`

This function performs a single step of a convolution (i.e., a dot product with the kernel and the current values of the input within the sliding window). We create a static shift register holding each pushed input that is received, and then create a for loop to compute the dot product starting from the right side of the shift register/filter. This function is called multiple times (dependent on the kernel size) within the function below, ***`top_fir()`***.

`void top_fir(int* input_real, int* input_img, int kernel_real[KERNEL_SIZE], int kernel_img[KERNEL_SIZE], hls::stream<int>&output_real, hls::stream<int>&output_img, int length)`

This function is the overseer for the frequency impulse response process. We perform partial convolutions with the input data and filter coefficients in a for loop, and then write the output of each convolution step (dot product) to the input that was just fed in to produce the next output.

void cordic(int cos, int sin, float *output_mag, float *output_angle)

This function performs each CORDIC rotation based on the current cosine/sine value of size FIXED_POINT. After a declaration of constants, the function determines what quadrant the input coordinates fall into. Based on this, the direction to rotate is determined in order to get closer to the x-axis. This is repeated for the desired number of CORDIC rotations, and then the magnitude is computed as the product of the CORDIC gain times the current cosine value.

void top_cordic_rotator(hls::stream<int>&input_real, hls::stream<int>&input_img, float* output_mag, float* output_angle, int length)

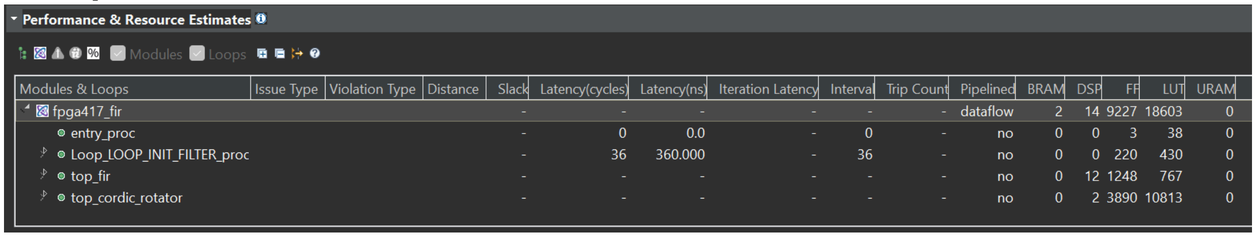
This function serves as the prototype for the CORDIC rotator and contains the process transforming cartesian outputs from ***top_fir()*** to polar form. We define variables intended to hold values popped from the queue that the FIR filter feeds, and then enters the primary CORDIC rotator loop. At each iteration, we read the real and imaginary outputs from the streams the FIR filter wrote to; then, ***cordic()*** is called to convert the complex number in cartesian form to polar form before the resulting magnitude and angle are then stored in their respective output arrays.

void fpga417_fir(int* input_real, int* input_img, int* kernel_real, int* kernel_img, float* output_mag, float* output_angle, int input_length)

This function connects everything together by allowing sequential calls to ***top_fir()*** and ***top_cordic_rotator()*** after loading the kernel values onto the FPGA. Thereafter, FIFO buffers are initialized as HLS streams to store the values between the FIR filter and the CORDIC rotator, opening up the ability to call the above functions.

Optimizations

Pre-Optimization Performance Metrics


Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fpga417_fir				-	-	-	-	-	-	dataflow	2	14	9227	18603	0
entry_proc				-	0	0.0	-	0	-	no	0	0	3	38	0
Loop_LOOP_INIT_FILTER_proc				-	36	360.000	-	36	-	no	0	0	220	430	0
top_fir				-	-	-	-	-	-	no	0	12	1248	767	0
top_cordic_rotator				-	-	-	-	-	-	no	0	2	3890	10813	0

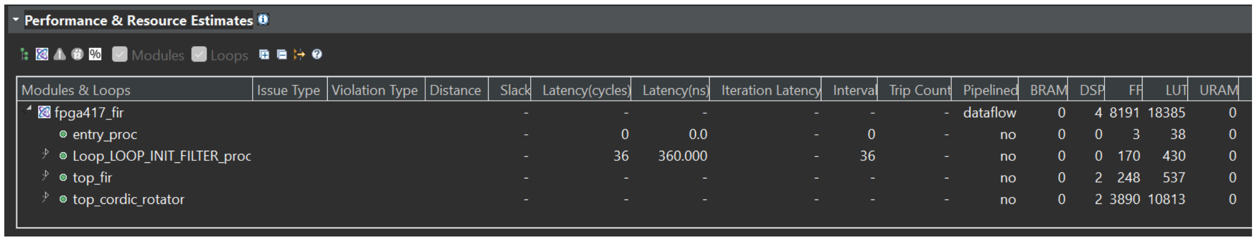
Optimization 1 – 32-bit int to ap_int<7>

```
// Custom Fixed Point types for FIR filter.
typedef ap_int <7> INP_INT; // 7 bit integer needed to represent input values and kernel coefficients.
typedef ap_int <14> OUT_INT; // 14 bit integer needed to represent FIR output values.
```

This optimization followed the recommendation to convert data types of ap_int. By knowing that the range of our inputs falls between -50 and 50, this allowed us to specify the optimal number of bits used to cover all numbers. In our case, we use $k = 7$ bits.

We know that the range of signed binary numbers for a given bit number k is $-(2^{k-1} - 1)$ to $2^{k-1} - 1$. When utilizing a value of $k = 7$, we yield a range of -63 to 63. If $k = 6$, the range would only be -31 to 31 due to the signed bit.

Post-Optimization Performance Metrics



Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fpga417_fir				-	-	-	-	-	-	dataflow	0	4	8191	18385	0
entry_proc				-	0	0.0	-	0	-	no	0	0	3	38	0
Loop_LOOP_INIT_FILTER_proc				-	36	360.000	-	36	-	no	0	0	170	430	0
top_fir				-	-	-	-	-	-	no	0	2	248	537	0
top_cordic_rotator				-	-	-	-	-	-	no	0	2	3890	10813	0

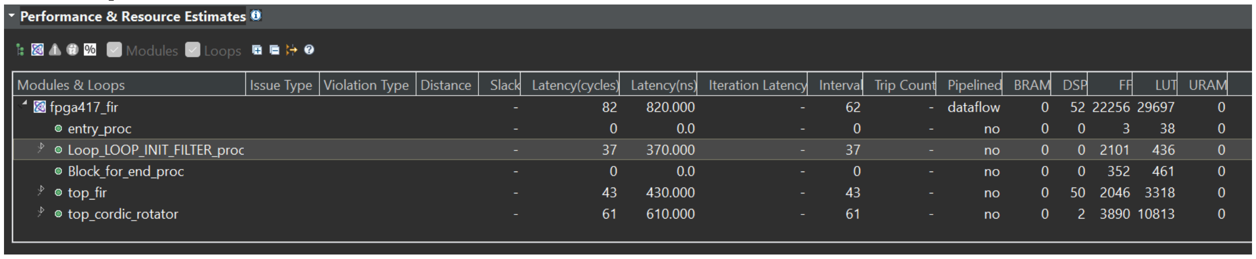
Analysis

We started with 32-bit integers and then condensed it to 7-bit integer types to hold the inputs and kernel data (between -50 and 50), and our DSP usage went from 14 to 4 because we effectively divide our operand size by a factor of over 4. The DSP usage going down is still a positive sign here because we have yet to activate pipelining. In general, we did the same number of multiplications as before but with less DSPs for efficiency (implying we can scale this up and do more in parallel).

Optimization 2 - Adding Trip Count and Complete Partitions

Added trip count for analysis clarity (could not previously see the iteration latency in cycles, so explicitly defined the number of iterations as the input length) and did a complete partition of the kernel coefficients. These were added in the ***top_fir()*** loop and the ***top_cordic_rotator()*** loops.

Post-Optimization Performance Metrics



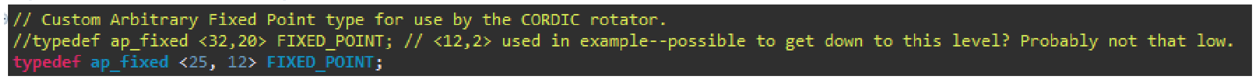
Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fpga417_fir				-	82	820.000	-	62	-	dataflow	0	52	22256	29697	0
entry_proc				-	0	0.0	-	0	-	no	0	0	3	38	0
Loop_LOOP_INIT_FILTER_proc				-	37	370.000	-	37	-	no	0	0	2101	436	0
Block_for_end_proc				-	0	0.0	-	0	-	no	0	0	352	461	0
top_fir				-	43	430.000	-	43	-	no	0	50	2046	3318	0
top_cordic_rotator				-	61	610.000	-	61	-	no	0	2	3890	10813	0

Analysis

The DSP usage went up, but the flip flop and LUT usage also went up (by a pretty dramatic amount). This makes sense because now we are dedicating individual memory banks for each element in the kernel coefficient arrays.

Optimization 3 – ap_fixed<25,12>



```
// Custom Arbitrary Fixed Point type for use by the CORDIC rotator.
//typedef ap_fixed <32,20> FIXED_POINT; // <12,2> used in example--possible to get down to this level? Probably not that low.
typedef ap_fixed <25, 12> FIXED_POINT;
```

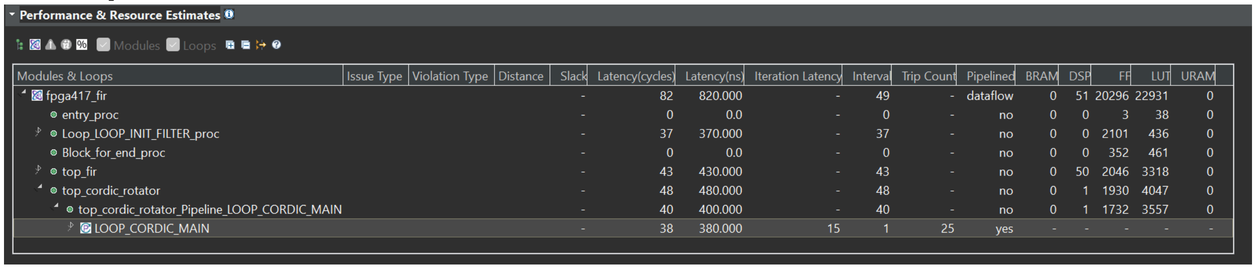
The idea behind this optimization was to reduce the usage of unneeded bits and increase the efficiency of our calculations by tailoring our data type to being 25 bits long. This was calculated through the following:

We know the maximum value for these CORDIC computations come from the output of the FIR, and the FIR yields a maximum value of 5000. In order to create a data type that is optimal for this, we note that (for unsigned bits), if we use a factor $k = 13$, our whole number range becomes $-(2^{13} - 1)$ to $2^{13} - 1$ which yields a span of -8191 to 8191.

Additionally, this means there is a decimal point degree of specificity of $\frac{1}{2^{25-13}}$, meaning that our decimal point extends to $n = 4$ places. One necessary implication is to also adjust the number of CORDIC iterations based on the decimal degree specificity n, where the CORDIC iterations = $n + 1 = 5$. We implemented this change in the header file.

This type may have actually caused some issues revealed during our demonstration, but those issues may have also been due to a forgotten line of code in the CORDIC rotator related to adjusting for quadrants.

Post-Optimization Performance Metrics



Performance & Resource Estimates

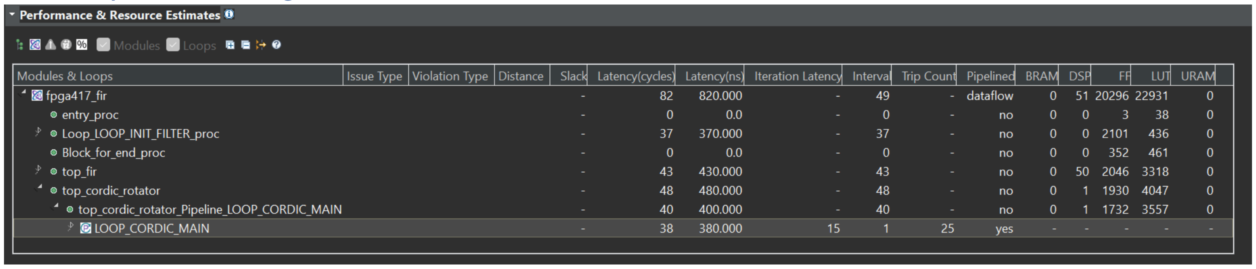
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fpga417_fir				-	82	820.000	-	49	-	dataflow	0	51	20296	22931	0
entry_proc				-	0	0.0	-	0	-	no	0	0	3	38	0
Loop_LOOP_INIT_FILTER_proc				-	37	370.000	-	37	-	no	0	0	2101	436	0
Block_for_end_proc				-	0	0.0	-	0	-	no	0	0	352	461	0
top_fir				-	43	430.000	-	43	-	no	0	50	2046	3318	0
top_cordic_rotator				-	48	480.000	-	48	-	no	0	1	1930	4047	0
top_cordic_rotator_Pipeline_LOOP_CORDIC_MAIN				-	40	400.000	-	40	-	no	0	1	1732	3557	0
LOOP_CORDIC_MAIN				-	38	380.000	15	1	25	yes	-	-	-	-	-

Analysis

So as expected, to fit on a DSP with an allowable size of 25x18, the DSP usage for our CORDIC rotator went from 52 to 51 because our CORDIC rotator function now only requires a single DSP due to us truncating the bit length.

Summary

Final Optimal Design



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fpga417_fir				-	82	820.000	-	49	-	dataflow	0	51	20296	22931	0
entry_proc				-	0	0.0	-	0	-	no	0	0	3	38	0
Loop_LOOP_INIT_FILTER_proc				-	37	370.000	-	37	-	no	0	0	2101	436	0
Block_for_end_proc				-	0	0.0	-	0	-	no	0	0	352	461	0
top_fir				-	43	430.000	-	43	-	no	0	50	2046	3318	0
top_cordic_rotator				-	48	480.000	-	48	-	no	0	1	1930	4047	0
top_cordic_rotator_Pipeline_LOOP_CORDIC_MAIN				-	40	400.000	-	40	-	no	0	1	1732	3557	0
LOOP_CORDIC_MAIN				-	38	380.000	15	1	25	yes	-	-	-	-	-

After implementing the optimizations listed above, we completed our design only utilizing 50 DSPs, ~20,000 FFs, and about 22,000 LUTs. We later learned, however, that because of our undersized ap_fixed type utilizing only 13 bits caused the HLS tool to implement some of our multiplications as LUTs, rather than using the discrete DSPs. Unbeknownst to us, the tool figures out that it must be more efficient to implement these in LUTs over DSPs under certain circumstances (like what we created, where are operands were significantly smaller than the acceptable input size for the onboard DSPs).

Additionally, while attempting to unroll our main FIR loop, we encountered seemingly unsolvable memory-dependency errors. While reviewing, however, we learned why this wasn't feasible. It basically came down to the fact that, because unrolling our main FIR loop resulted in 25 separate "hardware instances" of the FIR function/module, each of those 25 instances were accessing overlapping convolution coefficients from the S2P (single port) BRAM maintaining the kernel weights. This much parallelism effectively saturated the BRAM and ultimately resulted in delayed accesses to memory, leading it to take longer than the target time constraint.

Rather than unrolling the main loop, we learned that the optimal approach is to pipeline the main loop. Additionally, we particularly saw the impact this could have on operations where memory reads were involved. That is, the overall latency for operations when pipelined increased dramatically when pipelining was introduced, as memory read operations could take place while FIR operations were being performed on operands read on a previous clock cycle.

Best Running Frequency – 136.99 MHz

Optimized Latency Cycles – 82

Resource Utilization – (see above)