CSE 237C: Project 2 - Phase Detector

Will Zegers

October 20, 2016

Question 1

The first version of the phase detector (*phasedetector_optimized1*) was implemented by simply calling the baseline `fir` and optimized `codriccart2pol` functions in the main body of the phase detector, then connecting inputs and outputs from the top module as well as between modules. The throughput of this design, extracted from simulation, was 2.51 GHz. For *phasedetector_optimized2*, I added a few additional optimizations to the detector itself, such as function pipelining and inlining to the sub modules. This nearly doubled the throughput of the synthesized design to 4.26 GHz.

	Optimzation 1	Optimization 2
FIR latency (cycles)	8	9
CORDIC latency (cycles)	21	10
Detector latency (cycles)	34	20
Throughput (GHz)	2.51	4.26

Table 1: Latencies of sub-module and top phase detector module. Note that the detector latency is approximately just the sum of the latencies of its submodules.

Ultimately, the performance of the phase detector was a function of the throughput through each submodule. Total latency for data to move through the phase detector was just the sum of the latency through each module. Table Question 1 illustrates this with a breakdown of each optimization, and the latencies for its submodules. For the first architecture, total latency is slightly longer due to the latency of the complex FIR components being, itself, of function of *its* submodules (i.e. the four simple FIR filters for each term in the complex FIR equations); so total latency of the complex FIR is the length of a pipeline of the four submodules. In detector architecture 2, these modules were in some way subsumed by the complex FIR as a result of additional optimization pragmas.

This means that optimizing the phase detector and reducing its latency is a matter of optimizing and reducing the latencies of its individual components. Additionally, this also includes taking bottlenecks into account and maintaining a constant data flow between modules. By keeping throughputs at about the same levels for submodules, data producing modules (e.g. the complex FIR) are able to keep the data consumers (e.g. the CORDIC) fed with a stable flow of data.

Question 2

Question 3

Output accuracy of the CORDIC design stops improving due to two factors: the constant oscillating over the x-axis coupled with lack of additional precision (i.e. decimal places) in the input data. If the input widths are constrained to less and less precision, the computations for the output will not be able to improve without more precise data; outputs will start to get "stuck," improvement-wise, as the precision of the inputs fall off.

Varying the bidwidths played a key role in improving both throughput and layout area (as well as, through the use of fixed-point numbers, allow for shift operations on decimal data). Table Question 3 shows a comparison between the baseline CORDIC module and a first round of optimization that reduced the bit widths of variables. For this optimization, I reduced bidwidths of `data_t` type variables from float to a 16-bit fixed-point number with three integer bits. This "optimal" bitwidth I determined experimentally, iteratively adjusting the size and integer width to a minimal amount that could still (just) fulfill testbench accuracy constraints.

Ideal, bitwidths were dependent largely on input data, whose level of precision would ultimately determine the accuracy of the output. Using larger bitwidths permitted a greater accuracy (i.e. lower RMSE error) of output vs golden data; however, this also increases the layout area and degrades performance since more bits require more operations. On the other, too bits allocated to input data can result in low precision, leading to accumulated error in over many operations (i.e. a higher RMSE). Characterizing the input data, meant examining the range of

values (including whether or not can be negative) to determine the integer size for fixed-point numbers. Decimal bits were determined by using just enough bits to allow the module to pass testbench accuracy tests.
Finally, optimizations 4 and 5 (table Question 3 used smaller bit widths and a reduced number of iterations to still meet accuracy requirements, but further decrease area and boost throughput (with the help of loop pipelining and unrolling, respectively).

	Baseline	Optimization 1
Estimated clock (ns)	8.63	7.75
Throughput (GHz)	0.694	1.95
DSP48E	10	4
BRAM_18K	0	0
FF	1111	195
LUT	2259	385

Table 2: Comparison of the area and performance of the baseline CORDIC and and optimization that reduced the total bidwidths of `data_t` and `iter_t` variables

.

	Baseline	Optimization 4	Optimization 5
Throughput (GHz)	0.694	10.32	16.32
DSP48E	10	0	0
BRAM_18K	0	0	0
FF	1111	134	480
LUT	2259	533	1648

Table 3: Comparison of the area and performance of the baseline CORDIC and and optimization that reduced the total bidwidths of `data_t` and `iter_t` variables

.

Question 4

Add and shift operations are computationally cheap, whereas floating-point multiplies and divides require significantly more cycles and chip area to complete; consequently, substituting add-shift operations in for these costly operations, where possible, can generally give a tremendous boost to performance. The *cordic_optimized1* architecture uses variable fixed-point bitwidths, but still relies on multiply operations (with help from precomputed values in the array `Kvalues`) to compute the final `r` value and update `x` and `y` through each iteration of the *Rotate* for loop. Contrast this to the *cordic_optimized2* architecture, which just uses shift and add operations to compute values for `x`, `y`, and `r`.

Furthermore, add-shift operations are, in terms of hardware, much easier to implement; that is, they generally require fewer fundamental components (e.g. DSPs and flip-flops) and, therefore, can reduce the total layout area of the design. Table Question 4 shows the difference of the two architectures, both in terms of performance and chip area (in terms of the number of components used in the layout).

	Optimization 1	Optimization 2
Throughput (GHz)	1.98	4.25
DSP48E	4	0
BRAM_18K	0	0
FF	195	124
LUT	385	504

Table 4: Comparison of the area and performance of multiply (optimzation 1) vs. add-shift (optimization 2) architectures.

Here, it is evident that not only do add-shift offer a significant improvement (over 2x), but also eliminated the need for DSPs entirely as well as the number of flip flops.

Question 5

According to Professor Kastner's and Stephen book *Parallel Programming for FPGAs*, the ternary operator is implemented in hardware as a multiplexer "that performs a selection between two input operations based upon

an input condition." In other words, the ternary operator will use the hardware and control signals to select from the operands in the statement during execution, rather than a control statement that dictates flow. Consequently, the ternary operator determines during execution, via a select line, which value a variable will take on; execution does not need to stop and "make a decision" at an `if/else` control statement — this decision is made as soon as the variable to the left of the "`?`" is updated.

	Optimation 2	Optimization 3
Estimated clock (ns)	6.92	8.68
Throughput (GHz)	4.25	3.38
DSP48E	4	0
BRAM_18K	0	0
FF	124	108
LUT	504	506

Table 5: Comparison of the area and performance of `if/else` (optimzation 2) vs. ternary (optimization 3) architectures

Interestingly, this optimization had little beneficial impact on the stand-alone CORDIC module, reducing the number of flip-flops only marginally, while *increasing* the estimated clock period of the design (see table Question 5). So I was initially going to abandon using the ternary operator altogether, but while experimenting with architectures to optimize the phase detector, using the ternary operator in the CORDIC *sub*module ended up giving a significant boost to throughput. In fact, substituting the ternary operator in for existing `if/else` statements in `cordiccart2pol` was the overwhelming reason for the reduction in throughputs from optimzation 1 to 2 (see the last row in table Question 1 for a comparison of throughputs).

Question 6

Examining the LUT-based CORDIC was a matter of adjusting the data types and widths in the header file, as well as commenting in the pragmas in the *.cpp* file.

Overall, the relation between precision, accuracy, and performance could be characterized as a "choose two" design problem. For example, accuracy and performance could be improved, but at the cost of an (at times, exponential) increase in area; reducing the area of the layout meant sacrificing either performance or area, or a fraction of both. This is true for the LUT-based, and also for the regular CORDIC.

One noticeable advantage to the LUT-based CORDIC came after including the pragmas at the beginning of the *.cpp* file, which essentially explicitly defined the LUT resources and gave a throughput of 163 GHz! This came at the cost of larger area needed for the LUT resources.