ECE6775 High-Level Digital Design Automation, Fall 2024
School of Electrical and Computer Engineering, Cornell University

---

Lab 1: CORDIC Design

Due Friday, September 13, 2024, 11:59pm
Late submission: 4% penalty per day; cannot be late by more than 6 days

---

1 Introduction

COordinate **R**otation **DI**gital **C**omputer (CORDIC) is a method for calculating a variety of functions including trigonometric and hyperbolic. The various functions are calculated through an iterative set of vector rotations. At the end of these rotations, the value of the function is easily determined from the (x, y) coordinate. A CORDIC is often used to achieve low-cost multiplierless sine/cosine implementations in FPGA as well as ASIC designs.

To obtain a good understanding of CORDIC for the purpose of this lab, **please read Chapter 3 of *Parallel Programming for FPGAs* [1]** (in particular, Ch. 3.1-3.3). This book chapter provides a detailed tutorial of the CORDIC algorithm. It also gives a reference C/C++ code for the HLS implementation. You are encouraged to write your own program, but allowed to reuse and modify the reference code from the book. The main purpose of this lab is to help you get familiar with the HLS toolflow and practice the basic concepts of fixed-point design.

2 Materials

You are given a zip file named *lab1.zip* in */classes/ece6775/labs* on the ecelinux server[1], which contains the following files for you to build the project.

- *cordic.cpp*: an **incomplete source** file where you write your synthesizable code.
- *cordic.h*: the header file with various macro and type definitions that may be useful for developing your code.
- *cordic_test.cpp*: a test bench that helps verify your code.
- *Makefile*: a Makefile for you to easily (1) compile source code into an executable and (2) generate results to measure error (e.g., `make fixed-sw` or `make float-hw`).
- *run_{float/fixed}.tcl*: the Tcl scripts that create a Vivado project and synthesize the CORDIC design to RTL. For this assignment, it is sufficient to run the Vivado HLS via the Makefile with the `fixed-hw` and `float-hw` targets.

[1] You may directly log into specific ecelinux nodes such as `ecelinux-01,02,...,20`, if you feel the current node is slow or unstable.

- *run_opt.tcl*: an empty Tcl script that you will write to explore optimizations in the fixed-point design.

Before starting your assignment, please **copy and unzip the zip file to your working directory**. Be sure to **source the class setup script** using the following command before compiling your source code: `source /classes/ece6775/setup-ece6775.sh`.

Please refer to the Vivado HLS user guide [2] for more detailed descriptions of the Vivado HLS synthesis flow and the Tcl commands used in *run_{float/fixed}.tcl*.

3 Goal

The goal of this assignment is to create and optimize a CORDIC core that calculates the sine and cosine values of a given input angle. You will write the code in C++ for the CORDIC core, perform design space exploration using Vivado HLS, and explore trade-offs between area, performance, and accuracy.

The first part of this assignment is to write a functional CORDIC core using the double-precision floating-point type. With this baseline design, you will explore the design trade-offs by varying the iteration count of the main computation loop. Since CORDIC is an iterative algorithm, the number of iterations will affect the output accuracy as well as the performance of the synthesized hardware.

The second part is to use the fixed-point data type to optimize the CORDIC core for area, performance, and accuracy. The primary design space exploration goal is to understand how the bitwidth setting affects accuracy, as well as area and performance.

The third part asks you to maximize the throughput of the CORDIC core using optimization directives in Vivado HLS. This exercise will help you familiarize with common HLS optimizations and understand the effect of each optimization on the microarchitecture, performance, area, and timing of the design.

You will create a report describing the various trade-offs that you would make and how you maximize the throughput of the CORDIC core. For each design point (or architecture) you should provide its results including the area in terms of **resource utilization** (number of BRAMs, DSP48s, LUTs, and FFs), and performance in **throughput** in terms of number of CORDIC operations / second (i.e., number of input angles processed / second). **The throughput can be calculated based on the reported interval (in clock cycles) and the target clock period (fixed to 10 ns in this assignment).**

4 Guidelines and Hints

4.1 Coding and Debugging

- The input arguments to the `cordic` function are typed `theta_type` and `cos_sin_type`. These are currently set as double-precision floating-point type (i.e., `double`)[2]. In the

[2]You can use a floating-point division, `x/(double)(1ULL<<SHIFT_AMOUNT)`, to perform a right shift on variable `x` of type `double`. Note that `ULL` means unsigned long long in C++ (usually 64 bits); when `SHIFT_AMOUNT` $\geq$ 64, the result may overflow. An alternative is using a for loop to realize the shift in an iterative fashion: `for (int i=0; i<SHIFT_AMOUNT; i++) {x=x*0.5;}`

second part of this assignment, you are expected to change them to a fixed-point type to optimize your design. **Please carefully consider the number of integer bits necessary for representing the range of required values. Your fixed-point design should be free of multiplication and division (i.e., NO usage of DSP48 in the synthesis report**).

- The input angle (i.e., the `theta` argument of function `cordic`) is in radian.
- Enter `make` or `make float` under the project folder to compile and execute the floating-point program; Enter `make fixed` to run the fixed-point implementation (where the `FIXED_TYPE` macro is defined).
- The test bench creates an *out.dat* file which is useful for debugging[3]. This file lists the golden sine/cosine values from *math.h*, the sine/cosine values computed from your function, and the normalized difference (error). You will be able to assess the correctness and/or accuracy of your code based on the error reported by the test bench. Note that the errors are expected to be close to but NOT exactly zero even with the correct code. The accuracy should be improved by increasing the number of iterations. Otherwise, your code is not working.
- There is a constant array called `cordic_ctab` in *cordic.h*. You may find this useful although you do not necessarily have to use it.
- Please include meaningful comments in your code.

4.2 Design Exploration

- In this assignment, you will use a fixed 10 ns clock period targeting a specific FPGA device (i.e., Zynq). The clock period and target device have been specified in the *run_{float/fixed}.tcl* script.
- The number of iterations in your `cordic` function will play an important role in the accuracy and performance of the design. You should explore this aspect with your floating-point design. **You are expected to specify the list of iteration counts at line 23 of *run_float.tcl* to run simulation and synthesis in batch.** The script will also automatically collect important stats (i.e., accuracy, performance, and resource usage) from the Vivado HLS reports and generate a *float_result.csv* file under the *result* folder.
- The data types of the variables in your `cordic` function would also make a significant difference in area, accuracy, and performance. This should be another form of your design space exploration. **For this part, the number of iterations is fixed to 20.** You should experiment extensively with the data types and your report should show how different data types affect the accuracy as well as area and performance. You are expected to specify the list of bitwidth settings in *run_fixed.tcl* to run simulation and synthesis in batch. Similar to *run_float.tcl*, the script will also automatically collect important stats from the Vivado HLS reports and generate a *fixed_result.csv* file under the *result* folder.
- Although the synthesis tool takes some time to initialize, it should finish within 1-2 minutes for each design point based on our past experience with *ecelinux*. It is not

[3]You are welcome to use *gdb* as well.

normal if Vivado HLS runs for more than 10 minutes. You can use the `top` command in a separate shell to check the real-time system usage to see if the current *ecelinux* node is overloaded with other processes.

4.3 Performance Optimization

- In this part, please finish *run_opt.tcl* where we fix the design configuration to use **20 iterations and 32-bit signed fixed-point type with 8 integer bits**. Note that we use this configuration only for convenience instead of efficiency.
- The goal is to maximize the throughput of this design using optimization directives provided by Vivado HLS. The optimization directives serve the same purpose as the pragmas. While pragmas are embedded into the source code, directives are added in the *.tcl* file. Directives allow us to optimize the design without modifying the C++ source code so the same source code can be reused to synthesize different microarchitectures.
- For the purpose of this lab, you may use `set_directive_pipeline` and/or `set_directive_unroll` commands in *run_opt.tcl* to optimize your design instead of adding pragmas. Detailed descriptions of these directives can be found in Chapter 4, p.450 - 452 and p.458 - 460 of the Vivado HLS User Guide [2]. **You are recommended to carefully study sections relevant to these commands**.

4.4 Report

- Please write your report in a **single-column and single-space format with a 10pt font size. Page limit is ONE, including necessary figures and tables.** You may place figures side-by-side in one line to save space.
- The report should start with an overview of the document. This should inform the reader what the report is about, and highlight the major results. In other words, this is similar to an abstract in a technical document.
- There should be a section that summarizes your experiments by comparing and contrasting the various design points that you generated. **You are encouraged to create a table and additional plots to that clearly show the design choices, resulting performance, area/resource allocation, and accuracy.**
- All of the figures and tables should have captions. These captions should do their best to explain the figures (explain axis, units, etc.). Ideally you can understand the report just by looking at the figures and captions. Please avoid just putting some results and never saying anything about them.
- The report should only show screenshots from the tool when they demonstrate some significant idea. If you do use screenshots, make sure they are readable (e.g., not blurry). In general, you are expected to create your own figures. While more time consuming, it allows you to show the exact results, figures, and ideas you wish to present.

5 Deliverables

Please submit your assignment on CMS. You are expected to submit your report, code, and scripts in a single zipped file named *cordic.zip* that contains the following contents:

- *report.pdf*: the project report in pdf.
- A folder named *solution*: the set of source files and scripts required to reproduce your experiments; no need to inlcude the generated HLS projects and reports. We recommend you run `make clean` to remove all the automatically generated output files and copy the content of *lab1* into *solution*.

6 Acknowledgement

This document is adapted from a project description originally developed by Prof. Ryan Kastner for CSE 237C at UCSD.

References

[1] Ryan Kastner, Jannarbek Matal, and Stephen Neuendorffer, *Parallel Programming for FPGAs*, arXiv, 2018.

[2] AMD Xilinx Inc., *Vivado Design Suite User Guide: High-Level Synthesis UG902 (v2019.2)*, Available at `https://docs.amd.com/v/u/2019.2-English/ug902-vivado-high-level-synthesis`