ECE6775 High-Level Digital Design Automation, Fall 2025
School of Electrical Computer Engineering, Cornell University

---

Tutorial on Software-Hardware Codesign with CORDIC

---

1 Introduction

So far in ECE6775 you have worked with Vivado HLS to automatically compile C/C++ programs into hardware in the form of RTL. You would then check the quality of the generated RTL by examining the HLS reports produced by the tool and/or performing some kind of simulation. This means that up until now in the coursework, you have not actually executed anything on real hardware. Furthermore, you have not had to deal with how to interface a software program with a hardware accelerator.

In this tutorial you will learn how to take a modified version of the CORDIC design from Lab 1 and implement a full-fledged software-hardware system. You will then test the system on ZedBoard, which is an FPGA development board that features the Xilinx Zynq-7000 programmable system-on-chip (SoC) device. Specifically, the Zynq SoC integrates an ARM microprocessor and reconfigurable logic on a single chip.

The goal of this tutorial is to prepare you for Lab 3, where you will be designing a similar system, except targeting a different application in digit recognition. In addition, some of you may choose to use the same system setup for the final project as well.

2 Materials

You are given a zip file named *zedboard_tutorial.zip* on *ecelinux* under */classes/ece*6775/*labs*, which contains the following directories:

- *ecelinux*: contains a Vivado HLS project to synthesize the CORDIC hardware module. This should be done on `ecelinux`.
- *zedboard*: contains a C++ project to build the CORDIC host program. The host program should be run on the `ZedBoard` after programming the FPGA.

3 System Overview

A diagram of the system used in this tutorial is shown in Figure 1. The system roughly consists of three parts:

- **Host Program** — The C/C++ program running on the ARM processor which sends data to the FPGA and receives results. The host must ensure the data exchange is in a format consistent with the FPGA interface.

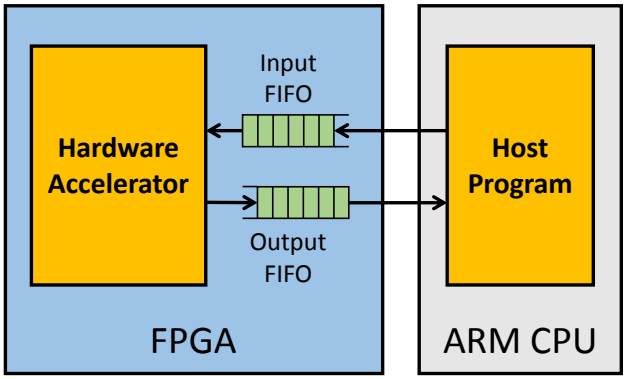

Figure 1: Block diagram of the software-hardware system.

- **Hardware Accelerator** — The hardware module implemented on the FPGA reconfigurable fabrics.
- **Input & Output FIFOs** — Two **32-bit** FIFOs are used as the (latency-insensitive) communication media between the hardware and software components. These two FIFOs are exposed as two Linux I/O devices on the host.

For this tutorial, all three components have been provided for you. The application we are targeting is a modified version of CORDIC from Lab 1. The hardware platform is the ZedBoard. Your task is to synthesize the accelerator, generate a **bitstream**, and use it to program the Zynq FPGA. Then you will execute the host program on the ARM processor to invoke the FPGA accelerator. Please carefully read through the explanations in Section 5 to understand the provided code.

4 ZedBoard Environment and Login

There are 10 ZedBoard units accessible from `ecelinux` via `ssh`. You **must** login to `ecelinux` before you login to the Zedboards even if you use the lab computers. Since each ZedBoard runs Linux using the embedded ARM processor, you should be comfortable with the work environment once you are logged in. The hostnames of the boards are:

```
zhang-zedboard-xx.ece.cornell.edu
```

where `xx` can be 01, 02, ..., 10. All students have an account on each ZedBoard with NetID as the both the username and initial password. You will be prompted to change your password on your first login to each board. **Be sure to `ssh` into every ZedBoard to set your permanent password before using the board in any way**. **In addition, the user accounts and file systems are not linked.** That means your files on one ZedBoard will not be automatically synchronized with other boards. You can choose to either stick to the same ZedBoard or back up your files using git or scp before you log out.

To ensure two students are not simultaneously attempting to program one FPGA (Linux reboot is required after each FPGA reconfiguration), we have restricted each ZedBoard such that only ONE student can login at a time. **Before you attempt to login, please first check the occupancy of the available boards using the following link:**

https://www.csl.cornell.edu/courses/ece6775/zedboard.html

This also means that **you cannot open multiple ssh sessions for the ZedBoard** and that **you cannot scp to a ZedBoard while logged into it from another terminal**. If you get strange error messages from running these commands please first check for this issue.

When you log out from a ZedBoard, it will be possible for another student to log in, at which point you will no longer be able to access your code. We **heavily** recommend you back up your files using version control (such as git) or by copying the files back to `ecelinux` **every time before you log out**.

5 Source Code Explanation

This section provides additional comments on several important source files to help you develop a better understanding of the overall system setup.

typedefs.h

Most of the `typedef` statements here should be familiar to you from Lab 1. However, note that we changed the bitwidth of the fixed-point types to 40. This is to demonstrate how to transfer wide data to the FPGA.

ecelinux/cordic.cpp

First note that the name of the top-level function has been changed to `dut`, which stands for "design under test". This is to make scripting the flow simpler. The arguments of `dut` are a pair of `hls::stream<bit32_t>` objects. These templated streams are part of Xilinx's `hls_stream` library; they represent the input and output FIFOs in Figure 1 and provide `read()` and `write()` methods as seen inside the function. `dut` reads the input angle through `strm_in`, calls the cordic function, and write the results to `strm_out`. The `<bit32_t>` indicate the data type which is stored in the buffers. A 32-bit integer is chosen to simplify the CPU-FPGA interface, as such types are standard in software programming.

However, this presents two problems: (1) our interface is only 32-bit wide while the input angle is 40 bits; (2) our input is in integer format (`ap_uint`) while the input angle to cordic is of fixed-point type which contains fractional bits. This is resolved using the bit slicing methods provided by the `ap_int` and `ap_fixed` data types. We use these methods to assign the appropriate (raw) bits from input to `theta_type`, and similarly, assign `cos_sin_type` to output.

zedboard/host.cpp

This file contains the program responsible for invoking the hardware module. The ZedBoard system is set up so that the FIFO channels connecting the FPGA are exposed to the Linux OS as two device files, which can be written to and read from like any regular files.

```
/dev/xillybus_write_32
/dev/xillybus_read_32
```

We begin by opening the two files. To access them we simply use the C functions `write` and `read`, which take in a file handle, a pointer to the data, number of bytes to send/receive, and returns the number of bytes successfully communicated. We again use bit assignment to convert the input angle from `theta_type` to a 64-bit integer (`int64_t`). Unlike in *cordic.cpp*, we can send the 64-bit value with a single function call instead of two separate transactions.

This is because the `write` and `read` functions actually break up the data for you and transfers one byte at a time.

In *host.cpp* we send one angle at a time, wait for the hardware module to produce a result, and read the result before sending the next angle. This is actually very inefficient — in our system setup the communication channel between the ARM processor and the FPGA has a long latency but high throughput. Because we send only one piece of data each time and wait for the result, we incur this latency for every angle.

zedboard/host_batch.cpp
This file provides a much more efficient implementation of the host program. The only difference is that we first write all 180 angles to the input FIFO of the CORDIC hardware module, then read all 180 sets of results. This allows us to take advantage of the high throughput of the processor-FPGA communication channel and greatly reduce the runtime.

Both *host.cpp* and *host_batch.cpp* use a timer object to measure the runtime of invoking the hardware module (including data transfer). The timer is straightforward to use, and will print the total time recorded when the program ends. This allows us to see the difference in execution time between the two host programs.

6 Tutorial Instructions

6.1 Generating the Bitstream

As in previous labs, you will begin by synthesizing the C++ source files into Verilog using the Vivado HLS tool. On `ecelinux` do:

```
unzip zedboard_tutorial.zip                # unzip the archive
cd zedboard_tutorial/ecelinux
source /classes/ece6775/setup-ece6775.sh   # setup env
make                                       # builds and runs the csim
vivado_hls -f run.tcl                      # generate Verilog for CORDIC
```

Compiling the csim is optional, but it will allow you to verify that the FPGA is producing the correct results.

You can examine the directory *cordic.prj/solution1/syn/verilog* to check that the Verilog files have been generated.

The next step is to implement the hardware circuit described by the Verilog on the FPGA. This is done by Vivado (not Vivado HLS), and is usually quite complicated. Fortunately, we provide a script which automates this process.

```
source run_bitstream.sh                    # generate FPGA bitstream
```

Vivado will first **map** the circuit logic to components like wires, gates, and registers. It will then perform **placement and routing**, which places each component into a physical look-up table (LUT), and makes connections between LUTs using the FPGA's programmable interconnect. Finally, Vivado generates the bitstream — this file contains the configuration bits which are sent to the LUTs and switches on the FPGA in order to (re)configure the FPGA to become the CORDIC circuit.

Due to how complex this process is, this command may take upwards of 15 minutes. When it finishes, it will copy the bitstream file *xillydemo.bit* to the current directory.

6.2 Programming the FPGA

To program the FPGA, we first copy the bitstream file onto the ZedBoard. Then we will mount the SD card on the board (exposing it to Linux as a writeable directory) and move the bitstream file onto the card. When the ZedBoard boots up, it will read the bitstream from the SD card and reconfigure the FPGA.

```
# Copy the bitstream from ecelinux to ZedBoard
scp xillydemo.bit <user>@zhang-zedboard-xx.ece.cornell.edu:~

# Login to the ZedBoard
ssh <user>@zhang-zedboard-xx.ece.cornell.edu

# Mount SD card. Try rebooting if there is a device busy error
mount /mnt/sd

# Copy the bitstream file to the SD card and reboot the ZedBoard
cp xillydemo.bit /mnt/sd
sudo reboot
```

The ZedBoard will restart. Wait about 30 seconds and login again — the FPGA will be fully programmed after the reboot.

If you encounter any issues with `scp` or `ssh`, please check the status of the ZedBoard and try another ZedBoard. If you encounter any issue copying the bitstream to the SD card, please first reboot the board and attempt the commands again.

6.3 Running The CORDIC Example

Use `scp` to copy *zedboard_tutorial.zip* to the ZedBoard, then do:

```
unzip zedboard_tutorial.zip          # unzip the archive
cd zedboard_tutorial/zedboard
make fpga                            # build FPGA host program
./cordic-fpga                        # run the baseline host
make fpga_batch                      # build FPGA host program (batch mode)
./cordic-fpga-batch                  # run the batched host
```

You should see the same accumulated error as when you ran the csim on `ecelinux`. Furthermore, `host_batch` should run much faster than `host`.

7 Acknowledgement

The baseline FPGA+Linux setup used in this tutorial is based on the Xillinux distribution provided by Xillybus (https://xillybus.com/xillinux).