Parallel Programming for FPGAs is an open-source book aimed at teaching hardware and software developers how to efficiently program FPGAs using high-level synthesis (HLS). The goal is to train people to effectively use HLS tools. The book was developed over many years to serve as a primary reference for UCSD’s 237C — a hardware design class targeting first-year graduate students and advanced undergraduate students. While writing the book, we developed projects and labs for the students to learn the HLS tools and to build real systems using FPGAs. This serves a general one stop shop for those projects and labs.
Please follow this link for the Pynq setup.

You may also need the following link to connect to the board if the above steps don’t work: https://www.nengo.ai/nengo-pynq/connect.html#via-a-computer
Please follow this link for the textbook associated with these projects and labs.

2.1 Project: FIR Filter Design

2.1.1 1) Introduction

The goal of this project is to learn how the basics of an HLS tool. The learning outcomes are to gain a basic understanding of how the Vivado HLS tool works, to get exposed to the different types of HLS optimizations, to perform a guided design space exploration to obtain architectures with different tradeoffs in performance and resource usage, to generate a high quality FIR architecture, and to demonstrate the integration of that FIR on the Zynq FPGA using the Pynq infrastructure.

This project is designed to be paired with Chapter 2 from Parallel Programming for FPGAs book. The book directly covers many aspects of the optimizations in this project. We strongly encourage the reader to use this as a reference.

The project is divided into three parts:

- Design an 11 tap FIR filter
- Design and optimize a 128 tap FIR filter
- Prototype an FIR filter architectures on a Zynq FPGA

You should start this assignment by understanding the 11 tap FIR filter, and implementing a functionally correct design. Next, you modify the code and experiment different optimizations which are specified in the questions. Note that the 128 FIR filter is more complex and may have different trade-offs, and in the final report you need to answer the questions with regard to the 128 tap filter. Your answers should demonstrate your understanding of different optimization and their effects on throughput, latency and area. Finally, you will take one of your FIR filter designs, program that on a Zynq FPGA, and demonstrate its functionality with the Pynq infrastructure.
2.1.2 2) Preparation

Before you start, we strongly suggest that you go through these high-level synthesis tutorials: Lab 1, Lab 2 and Lab 3 in this document: ug871-vivado-high-level-synthesis-tutorial.pdf. You can find this document and lab files here. We will refer to these labs collectively as Lab 0 (with Parts 1, 2, and 3).

2.1.3 3) Materials

You can download the project files here:

- project1.zip

This contains:

- fir11 folder: 11 tap fir filter
  - fir.cpp - Implements top level function
  - fir.h - header file
  - fir_test.cpp - test bench
  - input.dat - input chirp signal
  - out.gold.dat - “Golden” output. When the testbench (from fir_test.cpp) is run through the file fir.cpp it should generate this result. If it does not, you did something wrong.

- fir128 folder: 128 tap fir filter
  - fir.cpp - Implements top level function
  - fir.h - header file
  - fir_test.cpp - test bench
  - input.dat - input chirp signal
  - out.gold.dat - “Golden” output. When the testbench (from fir_test.cpp) is run through the file fir.cpp it should generate this result. If it does not, you did something wrong.

- Tutorial folder:
  - ug871-vivado-high-level-synthesis-tutorial.pdf - Various tutorials for Vivado HLS

- demo folder: Demo folder for 11 tap filter
  - input.dat - input chirp signal

- Target Board: xc7z020clg400-1
- Software: Vivado 2019.1
- Time Period: 10 ns or 100MHz

2.1.4 4) Project Goal

The first goal of this project is to generate a functionally correct HLS design for an 11 tap FIR filter. Also you should start to gain an understanding of different HLS optimizations. For FIR128, you should modify the code to generate several optimized designs. Your goal is to create designs that provide tradeoffs between resource usage and execution time. This will require you to rewrite the code and/or insert pragmas. More specifically, you must do the following:

- Design an 11 tap FIR filter with HLS. In the rest of this document, we use the term FIR11 to refer this task.
- Design a 128 tap FIR filter with HLS and optimize it. We call this subtask FIR128.
2.1.5 5) FIR11

The first step for the project is to get a functionally correct design working for an 11 tap FIR filter. For this, you will need to use the Vivado HLS tool, and finish the function body of `void fir()` in the file fir.cpp to implement the filter. You can test the correctness of your code by using the provided testbench. This code does not need to be highly optimized; you will work on creating optimized code later. It just needs to work correctly. Use the provided script.tcl to create your project.

2.1.6 6) FIR128 Instructions

You must complete the following tasks:

1. First, implement a functionally correct, but not optimized, 128 tap FIR filter. This is your baseline implementation. Use the provided script.tcl to create your project. As you attempt each individual optimization according to the questions below, try to think about what other optimizations would well in conjunction with them.

2. Next, generate one or multiple designs that will help you answer the questions in your report. In your answers, you should reference the design that you generated for your experiment. You can reference the same design from multiple answers. Your resulting code must always be functionally correct (i.e. match the golden output). In your report you need to clearly answer the effect of following optimizations on your design. You can test other optimizations as you wish, but you do not need to include these in your report. For every design you include in your report, be sure to report the corresponding throughput, instead of estimated clock period and latency.

3. Finally, for Q6, generate your best architecture by combining any number of optimizations that you wish. Use what you learned from your designs for Q1-Q5.

4. Your report should only include the answers to the following questions.

For each of the following questions you need to reference a design or multiple designs. The source code in your design should have all the necessary pragmas. Please refer to Chapter 2 in the PP4FPGAs textbook before starting this assignment.

Questions:

- **Question 1 - Variable Bitwidths:** It is possible to specify a very precise data type for each variable in your design. The number of different data types is extensive: floating point, integer, fixed point, all with varying bitwidths and options. The data type provides a tradeoff between accuracy, resource usage, and performance.

  Change the bitwidth of the variables inside the function body (do not change the bitwidth of the parameters). How does the bitwidth affect the performance? How does it affect the resource usage? What is the minimum data size that you can use without losing accuracy (i.e., your results still match the golden output)?

- **Question 2 - Pipelining:** Pipelining increases the throughput typically at the cost of additional resources. The initiation interval (II) is an important design parameter that changes the performance and resource usage.

  Explicitly set the loop initiation interval (II) starting at 1 and increasing in increments of 1 cycle. How does increasing the II affect the loop latency? What are the trends? At some point setting the II to a larger value does not make sense. What is that value in this example? How would you calculate that value for a general for loop?

- **Question 3 - Removing Conditional Statements:** If/else statements and other conditionals can limit the possible parallelism and often require additional resources. If the code can be rewritten to remove them, it can make the resulting design more efficient. This is known as code hoisting.

  Rewrite the code to remove any conditional statements. Compare the designs with and without if/else condition. Is there a difference in performance and/or resource utilization? Does the presence of the conditional branch have any effect when the design is pipelined? If so, how and why?

- **Question 4 - Loop Partitioning:** Dividing the loop into two or more separate loops may allow for each of those loops to be executed in parallel (via unrolling), enable loop level pipelining, or provide other benefits. This may increase the performance and the resource usage.
Is there an opportunity for loop partitioning in FIR filters? Compare your hardware designs before and after loop partitioning. What is the difference in performance? How do the number of resources change? Why?

- **Question 5 - Memory Partitioning:** The storage of the arrays in memory plays an important role in area and performance. On one hand, you could put an array entirely in one memory (e.g., BRAM). But this limits the number of read and write accesses per cycle. Or you can divide the array into two or more memories to increase the number of ports. Or you could instantiate each of the variables as its own register, which allows simultaneous access to all of the variables at every clock cycle.

  Compare the memory partitioning parameters: block, cyclic, and complete. What is the difference in performance and resource usage (particularly with respect to BRAMs and FFs)? Which one gives the best performance? Why?

- **Question 6 - Best Design:** Combine any number of optimizations to get your best architecture. A design with high throughput will likely take a lot of resources. A design that has small resource usage likely will have lower performance, but that could still be the best depending the application goals.

  In what way is it the best? What optimizations did you use to obtain this result? It is possible to create a design that outputs a result every cycle, i.e., get one sample per cycle, so a throughput of 100 MHz (assuming a 10 ns clock).

  It is possible that some optimizations may not have a big (or any effect). Some optimizations may only work when you use them in combination with others. This is what makes the design space exploration process difficult.

  - **Note:** You should use ap_int types if necessary for required bit width. You can read about ap_int from [here](http://kastner.ucsd.edu/hlsbook/) or from section 2.10 of the textbook.

### 2.1.7 7) PYNQ Demo

Following are steps to implement your FIR11 HLS design on the PYNQ board. You will provide the input data (chirp signal) from the Notebook, and get the output from the PL on PYNQ. To do that, you must write a `host_fir.ipynb` program.

The specific things you must do in this section are:

- Download an appropriate image for your board from PYNQ.io and write it to your SD Card (instructions).
- Go through Lab: Pynq Memory Mapped IO example and learn how to write an IP for PYNQ and interact with it.
- Implement your 11-tap FIR design on PYNQ board.
- Write a host program `host_fir.ipynb`. The expected output is as shown below:
2.1.8 8) Report Guidelines

FIR11

- Submit your code (only source code for synthesis) and tcl script.
- Submit synthesis reports (.rpt file and .xml document files, located in a “/syn/report” folder).

Demo

- Submit your code (only host code)
- Submit a screenshot of results
- Submit your bitstream

FIR128

- For each “interesting” architecture:
  - Submit your code and tcl script
  - Submit synthesis reports (.rpt and .xml)
  - “Interesting” is imprecise, but it is often difficult to say definitively that one design is the best. Typically there are different designs that Pareto optimal. Any design that you discuss in answers to your questions should be submitted. Often one performs a lot of design space exploration by changing values, and this can lead to a lot of architectures, many of which are “bad” or “uninteresting”. We don’t need details on those. A good target is more than 5 and less than 20 “interesting” designs. Your report should only include the answer to the questions. Please clearly indicate where each question is answered in your report.

For each question, explicitly mention which design(s) you used in your answers. You are asked to discuss different performance and resource metrics. Make sure that the method used to calculate the performance and resource metrics is clear. Throughput calculation methods are described below. It is typically best to report performance metrics in terms of seconds (or frequency = 1/seconds) rather than some other interval, e.g., clock cycles. For this reason, we require you to state the corresponding throughput for every design, instead of estimated clock period and latency.
You are strongly encouraged to use figures and tables to explain an answer. Figure 1 and Figure 2 provide a typical way to compare different architectures. These are not the best figures, and can certainly be made better, but serve as a reference that can relatively easily be generated. You can also consider different figures, e.g., to provide an overview of a particular architecture or help explain how you determined the “best design” (i.e., your process of design space exploration).

**Throughput Calculation**

The throughput is reported in Hz using the formula from Equation 2. Note that you should use the “Estimated Clock Period (ns)” from HLS report instead of the specified clock period. The former is more accurate (though not totally accurate - to do that you must perform complete synthesis to bitstream) than the latter, which is the user-specified target rate clock period. Often the tools can do better than the estimated clock period.

The throughput in Hertz can be calculated as:

\[
\text{Throughput}(Hz) = \frac{1}{(\text{ClockPeriod}(s) \times \text{ClockCycles})}
\]

The throughput in Mhz can be calculated as:

\[
\text{Throughput}(MHz) = \frac{1000}{(\text{ClockPeriod}(ns) \times \text{ClockCycles})}
\]

You should always present your results using units (Hz, KHz, MHz, etc.) that make “sense”. For example, you should not do 10000 Hz rather 10 KHz. Or not 0.02 MHz rather 20 KHz.

**Example Figures**

Figure 1 shows an example graph of resource usage for 8 designs. Figure 2 shows the performance of these 8 designs in terms of throughput.

*Figure 1. Area results of different FIR designs. Note that these are only for reference and do not necessarily correspond exactly to results that you can/should obtain.*

*Figure 2. Example throughput results for different FIR designs. Note that these are only for reference and do not necessarily correspond exactly to results that you can/should obtain.*

### 2.1.9 Submission Procedure

You should submit a report as described in the report instructions for this project.

You must also submit your code (.cpp, *.h files, and *.tcl, but nothing else). Your code should have everything in it so that we can synthesize it directly. This means that you should use pragmas in your code, and not use the GUI to insert optimization directives. We must be able to only import your fir.cpp/h file and directly synthesize it. You can assume that we have correctly set up the design environment (fir_test.cpp, etc.). **DO NOT SUBMIT THE ENTIRE HLS FOLDER.**

You must follow the file structure below. We use automated scripts to pull your data, so DOUBLE CHECK your file/folder names to make sure it corresponds to the instructions.

Your repo must contain a folder named “fir” at the top-level. This folder must be organized as follows:

- Report.pdf
- Folder fir11_baseline: fir.h | fir.cpp | script.tcl | fir_csynth.rpt | fir_csynth.xml
• Folder fir128_baseline: fir.h | fir.cpp | script.tcl | fir_csynth.rpt | fir_csynth.xml
• Folder fir128_optimized1: fir.h | fir.cpp | script.tcl | fir_csynth.rpt | fir_csynth.xml
• Folder fir128_optimized2: fir.h | fir.cpp | script.tcl | fir_csynth.rpt | fir_csynth.xml
• Folder fir128_optimized3: fir.h | fir.cpp | script.tcl | fir_csynth.rpt | fir_csynth.xml
• Folder fir128_optimized4: fir.h | fir.cpp | script.tcl | fir_csynth.rpt | fir_csynth.xml
• Folder fir128_optimized5: fir.h | fir.cpp | script.tcl | fir_csynth.rpt | fir_csynth.xml
• Folder fir128_best: fir.h | fir.cpp | script.tcl | fir_csynth.rpt | fir_csynth.xml
• Folder Demo: (WES students only) host_fir.ipynb | .bit | .hwh

fir128_optimizedN corresponds to the architectures that you generated to answer the questions. You can have one or multiple, just make sure the code is readable (i.e., do not put multiple optimizations commented out in the same file).

fir128_best is the folder containing your best architecture.

Submission

Place your code on your private Bitbucket or GitHub repository. Give collaborator or read-only access to the TAs, whose email addresses and usernames will be made available on Piazza. Put separate assignments in separate folders; name each folder according to the project. Place your report directly under your project folder.

2.1.10 10) Grading Rubric

Your grade will be determined by your answers to the questions. Your answers should be well written and clearly delineated (for example: by copying the questions into the report before answering them, or placing each question under a separate subheading). Additional points (up to 20) will be subtracted for poor formatting and/or answers that are hard to understand. Examples of issues include any spelling errors, multiple/egregious grammar errors, poor presentation of results, lack of written comparison of the results, etc. Report throughput and resource usage for each design you discuss in your report, and include the files for these designs in your submission. We encourage the use of tables for stating results and the changes that produced them, and figures to draw comparisons between different designs. A well-written report is informative but not overly verbose. You will be deducted points if you do not follow the instructions on directory naming and file structure.

If you are submitting a report made in LaTeX, you might find this link that generates tables from spreadsheets helpful.

The report comprises of 80% of your grade. The remaining 20% is for the performance of the best version of your fir128 filter. If your design achieves a throughput of greater than 0.5MHz but less than 1MHz then you will be awarded 10 points. If you achieve 1MHz and higher than you will get complete 20 points. Try to make resource usage as small as possible. The resource usage must be within the resources provided by the Pynq board. Similarly the timing has to be fulfilled, i.e. the clock achieved should be within 10ns (100 MHz).

2.2 Project: CORDIC

2.2.1 1) Introduction

The goal of this project is to design a COordinate Rotation DIgital Computer (CORDIC). You are tasked with building one version of the CORDIC from scratch. This will likely take the majority of the time spent working on this project. A CORDIC is an efficient method for calculating trigonometric and hyperbolic functions. CORDIC can do a lot of
different functions; here we will specifically use it to convert Cartesian coordinates (x, y) to the polar coordinates (r, theta).

### 2.2.2 2) Materials

You can download the project files here:

- **cordic.zip**

The provided zip file has a number of subfolders and files related to implementing the CORDIC. This contains the documents necessary to build the project. You will start from HLS folder to design a CORDIC using Vivado HLS. Use the provided script.tcl to create your project.

- **HLS cordic folder:**
  - cordiccart2pol.cpp - The place where you write your synthesizable code. Currently, it only contains the function prototype.
  - cordiccart2pol.h - header file with various definitions that may be useful for developing your code.
  - cordiccart2pol_test.cpp - test bench
  - script.tcl - Use this to create your project

- **HLS cordic_LUT folder:**
  - cordiccart2pol.cpp - The place where you can find the synthesizable code. Currently, it contains a simple implementation.
  - cordiccart2pol.h - header file with various definitions that may be useful for developing your code. Here you can modify values and types of the design parameters to explore their impact.
  - cordiccart2pol_test.cpp - test bench
  - script.tcl - Use this to create your project

- **Demo folder:**
  - Cordic.ipynb - Jupyter notebook host file

### 2.2.3 3) Tasks

1. Design and verify a functionally complete CORDIC IP core using HLS: **cordic_basline**. You are provided a testbench that you can use though that the testbench does not cover all cases. You are encouraged to create a more extensive testbench to ensure that your code is correct.

2. The ultimate goal is to create an efficient CORDIC that only uses simple operations, i.e., add and shift. You should not be using divide, multiply, etc. in your CORDIC core. First design your code using float variables. Once you have a functionally correct CORDIC, then change data types to fixed-point types incrementally while checking to see that the test-bench still passes. This should result in your multiplications being synthesized into shifts and adds (you should verify that this is indeed happening). Do not change your function header/definition/interface (the input and output variables and their datatypes), only change variables inside the body of your **cordiccart2pol** function. You may use new typedefs if you wish. Try to perform as few operations with float as possible, meaning there will likely be a lot of typecasting involved.

3. Explore the architectural tradeoffs of a CORDIC core architecture based upon a lookup table. We have provided a fully functional base implementation of this. You should read and understand this CORDIC code. You should analyze the design-space of this lookup table IP core by changing the parameters of the look-up tables, e.g., by varying the data type of the input data, and changing the number of entries. This should allow you to find architectures with different resource usage, performance, and accuracy results.
4. A major design tradeoff for the CORDIC revolves around the precision or accuracy of the results. For example, changing the number of rotations effects the accuracy, performance, and resource usage. Another important tradeoff is the data type of the variables. Using large, complex data types (like floating point) is typically most accurate, but not good with respect to performance and resource usage. Using fixed-point types is more performant, but may reduce the accuracy of the results. Perform design-space exploration to create a wide range of implementations using various data types for different variables, modifying the number of rotations, and performing other optimizations to find the Pareto optimal designs.

2.2.4 4) PYNQ Demo

The final task integrates a CORDIC IP core onto the programmable logic (PL) using PYNQ. The provided notebook gives a skeleton for running the CORDIC using memory mapped IO for communication. The Lab: Pynq Memory Mapped IO serves as an example you can follow. The notebook passes data to the CORDIC IP, starts the process, reads the result, and compares it with the result computed in Python.

2.2.5 5) Report

Your report should answer the following questions. Make it very clear where you are answering each of these questions (e.g., make each question a header or separate section or copy/paste the questions in your report and add your answer or simply put a bold or emphasized Question X before your answer). Your report will be graded based on your responses.

• **Question 1**: One important design parameter is the number of rotations. Change that number to numbers between 10 and 20 and describe the trends.

  What happens to performance? Resource usage? Accuracy of the results? Why does the accuracy stop improving after some number of iterations? Can you precisely state when that occurs?

• **Question 2**: Another important design parameter is the data type of the variables.

  Is one data type sufficient for every variable or is it better for each variable to have a different type? Does the best data type depend on the input data? What is the best technique for the designer to determine the data type(s)?

• **Question 3**: What is the effect of using simple operations (add and shift) in the CORDIC as opposed to multiply and divide? How does the resource usage change? Performance? Accuracy?

• **Question 4**: Thinking from a logic-design hardware perspective, what operation does the ternary operator ‘?’ perform? What function would you expect this operator to synthesize as? Do you notice a difference between using it and an if-else block? Is it useful in this project?

• **Question 5**: These questions all refer to the LUT-based CORDIC: Summarize the design-space exploration that you performed as you modified the data types of the input variables and the LUT entries.

  In particular, what are the trends with regard to accuracy (measured as error)? How about resources? What about the performance? Is there a relationship between accuracy, resource usage, and performance? What advantages/disadvantages does the regular CORDIC approach have over an LUT-based approach?

  The cordiccart2pol.cpp file in the cordic_LUT project has a comment asking what happens if you direct HLS to use RAM_1P_LUTRAM for the my_LUT_r and my_LUT_th variables. **You don’t have to answer these questions in the report**, but try them out for an understanding of LUTRAMs.

• **Note**: You should use ap_int or ap_fixed types if necessary for required bit width. You can read about ap_int and ap_fixed from here.
2.2.6 6) Submission Procedure

You must submit your code (and only your code, not other files). Your code should have everything in it so that we can synthesize it directly. This means that you should use pragmas in your code, and not use the GUI to insert optimization directives. We must be able to use what is provided (*.cpp, *.h files, and *.tcl) and directly synthesize it. We must be able to only import your source file and directly synthesize it. If you change test benches to answer questions, please submit them as well. You can assume that we have correctly set up the design environment (cordic_test.cpp, cordic.h, etc.).

You must follow the file structure below. We use automated scripts to pull your data, so DOUBLE CHECK your file/folder names to make sure it corresponds to the instructions.

Your repo must contain a folder named “cordic” at the top-level. This folder must be organized as follows (similar to the structure in other projects):

- Report.pdf
- Folder cordic_baseline: cordiccart2pol.h | cordiccart2pol.cpp | script.tcl | <report rpt/xml>
- Folder cordic_optimized1: cordiccart2pol.h | cordiccart2pol.cpp | script.tcl | <report rpt/xml>
- Folder cordic_optimized2: cordiccart2pol.h | cordiccart2pol.cpp | script.tcl | <report rpt/xml>
- ...
- Folder cordic_LUT: cordiccart2pol.h | cordiccart2pol.cpp | cordiccart2pol_test.cpp | script.tcl | <report rpt/xml>
- Folder Demo: Cordic.ipynb | .bit | .hwh
- Note: <report rpt/xml> references both the .rpt and the .xml files in the /syn/report folder. Please include both.
- Note: Provide the architectures that you used to answer the questions.

2.2.7 7) Grading Rubric

Unlike the FIR project, it is not explicitly necessary to come up with an optimally efficient solution for the CORDIC core using pipelining, unrolling, and other HLS directives to meet a certain frequency. This project is more about precision and accuracy of data using bit widths. However, the larger goal of this class is to understand how to obtain resource-efficient designs while still achieving the level of functionality you desire. So there’s no performance target to hit, but do your best.

The provided cordic_LUT does not pass the test. This is expected and fine. The idea is for this portion of the project is design-space exploration. It is possible to run synthesis for a design that doesn’t pass C-Sim, but for this question alone you may modify the threshold to make it pass. Do not do this for other designs or projects without checking with us first; we want to be able to compare all your designs as if they met or surpassed a certain standard of accuracy.

In this project and in future projects, you may find it necessary to edit the test-bench. Either the test-bench doesn’t cover enough cases, or (as in the case of cordic_LUT) the threshold for error of theta may be too high. We may even instruct you to edit the test-bench. In these cases, you are welcome to change this file, and if you do, please submit the altered test-bench file too, and explicitly state in your report what changes you made and why.

50 points: Response to the questions in your report. Your answers should be well written and clearly delineated (for example: by copying the questions into the report before answering them, or placing each question under a separate subheading). Additional points (up to 20) will be subtracted for poor formatting and/or answers that are hard to understand. Examples of issues include any spelling errors, multiple/egregious grammar errors, poor presentation of results, lack of written comparison of the results, etc. Report the throughput, resource usage, and accuracy for each design you discuss in your report, and include the files for these designs in your submission. We encourage the use of tables for stating results and the changes that produced them, and figures to draw comparisons between different designs. Use these figures and tables in your discussion. A well-written report is informative but not overly verbose. You will be deducted points if you do not follow the instructions on directory naming and file structure.
50 points: Correct working project on PYNQ.

2.3 Project: Phase Detector

2.3.1 1) Introduction

The goal of this project is to design a simple phase detector. This is done by combining a complex FIR filter and a CORDIC. You build a complex FIR filter by hierarchically instantiating four “real” FIR filters similar to what you developed in the FIR filter project. In this part, you use CORDIC IP core from the previous part.

The complex FIR filter is used to correlate to a known complex signal. We use Golay codes which have some great properties related to orthogonality and auto-correlation. This is not important to this lab, but is some really amazing math. We hope you look into it.

In the end, you will combine all of these modules into a phase detector. This is a common block used in a digital communications receiver. The goal is to do simple synchronization and discover the phase of the signal. The output of the CORDIC (r, theta) gives you these results. It is a simple phase detector, but should provide you with a basic understanding of the problem, and you should come away with knowledge on how to develop two new important hardware blocks (CORDIC and a complex FIR filter).

We provide a Simulink file that models a transmitter, channel, and receiver. You are building an equivalent receiver in HLS in this project. The Simulink file is provided for your information only. You do not have to edit or do anything with this file though it could be useful for understanding the overall application better.

2.3.2 2) Materials

You can download the project files here:

- phase_detector.zip

The provided zip file has a number of subfolders and files corresponding to the different parts of the phase detector. This contains the documents necessary to build the project. You will start from HLS folder to design your phase detector using Vivado HLS. Use the provided script.tcl to create your project.

- HLS fir_top folder: This folder contains .cpp, .h, and script files for a complex FIR filter. This is a particular type of filter called a matched filter. You are matching the incoming signal to complex I and Q Golay codes that are provided for you. In the fir.cpp file, there are four sub functions firI1, firI2, firQ1, and firQ2. These functions are real FIR filters i.e., the same that you designed in Project 1. You can use your favorite code from Project 1 in these four sub functions. In the complex FIR filter, four of these functions are used in the fir function. In that function, you need to connect the four FIR filters firI1, firI2, firQ1, and firQ2 to an adder and subtractor to create the complex matched filter. This structure is demonstrated in the Simulink file.

- HLS phasedetector folder: After you design the cordic and the complex fir, you will use them to design the phase detector.
  - fir.cpp - complex fir filter you designed previously.
  - cordiccart2pol.cpp - cordiccart2pol function you designed previously.
  - phasedetector.cpp - Top level skeleton for the phase detector. You use fir.cpp and cordiccart2pol.cpp to design the phasedetector function.
  - phasedetector_test - test bench
  - phasedetector.h - header file
  - script.tcl - Use this to create your project
• Simulink:
  – Project2_model.slx: Simulink model file. You do not have to edit or do anything with this file though it could be useful for understanding the overall application better.

• Demo Folder: Demo folder for Phase Detector.
  – host.ipynb - Jupyter notebook host file
  – input_i.dat - input I signal
  – input_q.data - input Q signal
  – out_gold.dat - golden output

### 2.3.3 3) Tasks

In this project, you will build a phase detector to process the given a complex signal (I and Q or real and imaginary parts) demonstrated in the figure below.

![Input I](image1)

![Input Q](image2)

The final goal is to implement this phase detector. To achieve this goal, you will need to finish the following tasks:
1. Implement the complex matched filter and verify it with the given testbench. This matched filter consists of four FIR filter modules which are similar to the ones in your Project 1. For the purpose of debugging, if you plot the outputs of this step, you should expect the waveforms as shown in the figure below. Or more simply just make sure that that testbench passes.

![Waveform of Match Filter Output I (i.e. X)](image1)

![Waveform of Match Filter Output Q (i.e. Y)](image2)

2. Connect the complex matched filter and CORDIC modules to implement the receiver. Verify this overall design with the given testbench. For the purpose of debugging, if you plot the outputs of this step, you should expect the waveforms as shown in the figure below.

Note: You are encouraged to modify this implementation code to gain better utilization or throughput. Remember to

2.3. Project: Phase Detector
submit modified .cpp and .h files

2.3.4 4) PYNQ Demo

Again, the final task integrates the phase detector onto a PYNQ. Implement the receiver design on the board. This process is mostly similar to Lab: Axistream Multiple DMAs, but you will need to modify your HLS code for the streaming interface.

Note that the DTYPE struct in this project is almost identical to the axis_t typedef we used in the multiple DMA lab, here containing a float (data) and an int (last).

When streaming the output structs, the last bit should be set to 1 for the last struct to be streamed, indicating end of stream. You may need to explicitly set the other last bits to 0, otherwise your stream may terminate early and without warning since there may be garbage data at the memory addresses of the struct you create that are streamed out. You do not need to do this for inputs, as the tool takes care of it for you. Sometimes, the output streaming’s last bit is also handled by the tool, but sometimes it may not be, which will cause the DMA to hang (corresponding to a forever-running Jupyter cell) and it is better to hard code it.

Another point worth discussing here is why we use pointers for inputs and outputs, and why we have to post-increment the pointer manually (like we did in the multiple DMA lab) when we stream inputs and outputs, but why it is a bad idea to use pointers in your code. You cannot use pointers in HLS; pointers are dynamic memory and Vivado HLS will not be able to synthesize it since it is not a deterministic thing (the datapath could change depending on inputs). Arrays, on the other hand, are fixed memory locations and therefore they can be synthesized to vectors in RTL. You can use pointers only as ports and even then you have to specify axistream, otherwise that will lead to synthesis issues as well.

In Vivado, the HP ports are High Performance ports which can be accessed by several interfaces. It is something like dynamic channel (also known as memory) which can access the entire channel at one go. Therefore it is not necessary to enable more than one HP port. This link says to use two HP ports if you value performance. If you use multiple HP ports, in the memory map you can see this will give you more space to access (like 512M instead of 256M). So it is always safer to use separate ports although not required. You should have both DMAs be write-enabled (the lab had only one output, but here you have two outputs, so we’ll need both). If you choose to use more than one HP port, HP0 and HP1 should have different masters. So HP0 will have the first DMA as its master, and HP1 will have the second DMA. Two DMAs can point to a single HP port, but two HP ports cannot have the same DMA as master.

You also should see these outputs:

<table>
<thead>
<tr>
<th>Thetas at the R peaks are:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.015529</td>
</tr>
<tr>
<td>0.047509</td>
</tr>
<tr>
<td>0.079485</td>
</tr>
<tr>
<td>0.111526</td>
</tr>
<tr>
<td>0.143491</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

These are the rotated phases that have been detected by your design.
2.3.5 5) Report

Your report should answer the following questions. Make it very clear where you are answering each of these questions (e.g., make each question a header or separate section or copy/paste the questions in your report and add your answer or simply put a bold or emphasized Question X before your answer). Your report will be graded based on your responses.

• Question 1: What is the throughput of your Phase Detector? How does that relate to the individual components (FIR, CORDIC, etc.)? How can you make it better?

2.3.6 6) Submission Procedure

You must submit your code (and only your code, not other files). Your code should have everything in it so that we can synthesize it directly. This means that you should use pragmas in your code, and not use the GUI to insert optimization directives. We must be able to use what is provided (*.cpp, *.h files, and scripts) and directly synthesize it. We must be able to only import your source file and directly synthesize it. If you change test benches to answer questions, please submit them as well. You can assume that we have correctly set up the design environment (cordic_test.cpp, cordic.h, etc.).

You must follow the file structure below. We use automated scripts to pull your data, so DOUBLE CHECK your file/folder names to make sure it corresponds to the instructions.

Your repo must contain a folder named “phase_detector” at the top-level. This folder must be organized as follows (similar to other projects):

• Report.pdf
  • Folder fir_top_baseline: fir.h | fir.cpp | script.tcl | report.rpt and .xml
  • Folder phasedetector_optimized1: phasedetector.h | phasedetector.cpp | cordiccart2pol.cpp | fir.cpp | script.tcl
    | <report rpt/xml>
  • Folder phasedetector_optimized2: phasedetector.h | phasedetector.cpp | cordiccart2pol.cpp | fir.cpp | script.tcl
    | <report rpt/xml>
  • ...
  • Folder Demo: host.ipynb | .bit | .hwh
• Note: <report rpt/xml> references both the .rpt and the .xml files in the /syn/report folder. Please include both.
• Note: Provide the architectures that you used to answer the questions. You may optimize individual components (FIR/CORDIC), or the phase detector directly.

2.3.7 7) Grading Rubric

50 points: Response to the questions in your report. Points will be deducted based upon poor presentation, grammar, formatting, spelling, etc. Results should be discussed succinctly but with a enough detail to understand your architectures and tradeoffs. Tables and figures should be properly labeled, well thought out, and described in the text with comments on both the design that produced each entry, and the observable trends between entries. Spelling errors are unacceptable.

50 points: Correct working project on PYNQ.
2.4 Project: Discrete Fourier Transform (DFT)

2.4.1 1) Introduction

The goal of this project is to design architectures that implement the Discrete Fourier Transform (DFT). DFT is a common operation in signal processing which generates a frequency domain representation of the discrete input signal. We start with the direct implementation of the DFT which is a matrix-vector multiplication. The input signal is a vector of samples and the matrix is composed of a set of basis functions corresponding to discrete cosine and sine waveforms of different frequencies. The multiplication of the input signal with these basis functions describes how well the input signal correlates with those waveforms, which is the value of the Fourier series at that frequency.

2.4.2 2) Materials

The files necessary for this project can be found here.

You can find the following files in the zip file: they are divided into five folders, dft_8_precomputed, dft_32_precomputed, dft_256_precomputed, dft_1024_precomputed, and Demo. Each of the first four folders has its own testbench, out.gold.dat file, and coefficients.h file.

Each dft_xx_precomputed folder contains following files:
- dft.cpp - the baseline implementation for the dft function.
- dft.h - header file
- dft_test.cpp - test bench
- coefficientsX.h - a file containing the values of corresponding to one sine/cosine period sampled based upon the DFT points. For example, an 8 point DFT has 8 samples across both the sine and cosine function evenly spaced across one period. This is equivalent to dividing one rotation in the complex plane equally by the number of points in the DFT.
- out.gold.dat - “Golden” output. The testbench (dft_test.cpp) generates a sample input and calls the function dft in dft.cpp with that sample input. This output of the function is compared to the expected output. This will indicate PASS or FAIL. If it fails, then the code in the folder Demo: dft.bit | dft.tcl | project3_host.ipynb dft is incorrect. There are four different versions of depending on the DFT size and way in which the DFT coefficients were generated.

- script.tcl and directives.tcl file to create the project

Demo folder contains one file:
- DFT.ipynb - notebook for demo

2.4.3 3) Project Goal

You should modify the code to create a number of different architectures that perform tradeoffs between performance and resource utilization. For dft_256_precomputed and dft_1024_precomputed designs, you need to use precomputed values from coefficients256.h and coefficients1024.h

For 256-point and 1024-point DFTs, you will create a report describing how you generated these different architectures (code restructuring, pragmas utilized, etc.). For each architecture you should provide its results including the resource utilization (BRAMs, DSP48, LUT, FF), and performance in terms of throughput (number of DFT operations/second), latency, clock cycles, clock frequency (which is fixed to 10 ns). You can do most of the design space exploration on the 256 point DFT. You should pick your “best” 256 architecture and synthesize that as a 1024 DFT.
The 8 and 32 point folders are provided for your convenience. If you would like, you can do some of your initial design space optimization on these smaller architectures. But it is not necessary to use these at all.

The key in this project is to understand the tradeoffs between loop optimizations (unrolling and pipelining) and data partitioning. Therefore you should focus on these optimizations.

### 2.4.4 4) Optimization Guidelines

- You should use a clock period of 10 ns.
- The output of your architecture must closely match the golden output. Be sure to generate a working function before performing any optimizations. If the result does not match exactly, but is close, please explain why in the report. You should use float for all data types. You do not need to perform bitwidth optimization in this project.
- You may want to start implementing DFT by using HLS math functions for \( \cos() \) and \( \sin() \). Then you can optimize your code based on this baseline code.
- There are many different ways to generate the DFT coefficients including using HLS math functions. These can be implemented as constants when the DFT size is fixed. We have given you the coefficients for both 256 (in coefficients256.h) and 1024 (in coefficients1024.h). They each have two constant arrays, sin_table and cos_table. You can use these coefficient arrays directly as memories in your architectures. You are also free to create your own arrays using a different structure (e.g., 2D array, reordering of elements in the given 1D arrays, etc.). Or you could dynamically generate the coefficients.
- There is significant amount of parallelism that can be exploited by (partially) unrolling the for-loops. Pipelining these (partially) unrolled for-loops should lead to higher throughputs. However, you may find that pipelining doesn’t make a difference once you have loop unrolling and array partitioning handled well. When you try to incorporate pipelining, the major issue you will face is data dependencies. You can read more about them here. Since you have some data dependencies, accessing memory will be the major overhead. This is why your estimated clock period may go beyond 10ns; to perform some task within N clock cycles each clock cycle needs to be high for the task to be completed. For example, at clock cycle 1 you might write a[1] and at clock cycle 2 you will need to read a[1] which would not be ready as the operation might take 4 clock cycles. This is a data dependency issue which is very common in pipelining. This is why pipelining does not seem to work well even though your loop unrolling and partitioning is the best you can find. Another reason is that the overhead might not be because of the task (it might take only 1 clock cycle), but the memory it is stored in might have only 2 ports or 1 port and this would mean that the memory cannot be accessed in parallel. You can see the critical path in the Synthesis log itself. Otherwise you can open the Analysis view and view which operation(s) or data path is critical and causing this delay, which in turn limits the performance of pipelining. There is a slightly different conversation here that may be helpful to read through. This paragraph might make more sense after you complete the project, so be sure to read through it again when you’re finalizing your report.
- One of the questions will ask you to read about dataflow and apply it to your design. You can think of dataflow as task-level or function-level pipelining. It uses functions (with their limited variable scopes, and clearly defined inputs and outputs) to parallelize some tasks with others. Therefore, roughly speaking, the more function calls you make the better. Or said another way, break your code up so that the functionality doesn’t change but your code is more modular: there will be more function calls happening and therefore more opportunities for function-level pipelining will exist. Now, there are some other things you need to ensure. Data flow implements FIFO buffers for variables (you can read this in the documentation linked in the question below) and because of this you should limit your reuse of old variables. Each memory address should only be written to once and read from once, but sometimes it is preferable to write back to the memory address repeatedly. It’s up to you to determine which is best (or even necessary) for each variable in your design. This means in most variable scopes you will have some extra variables whose entire purpose is to be intermediate variables between functions. There’s also an example code linked below that may be helpful to see how intermediate variables and passing variables by reference between functions is done.
- There are more efficient methods for performing the DFT that exploit the symmetries of the Fourier constants, e.g., the Fast Fourier Transform (FFT). **Do not use these symmetries.** In other words, treat this like a matrix-
vector multiply with unknown matrix values. Don’t worry, we will implement FFT architectures soon enough that will fully take advantage of these symmetries in Project: Fast Fourier Transform (FFT).

• You do not need to report your optimizations for your 8 point and 32 point DFT; these folders are provided for your convenience. Since these will very likely synthesize much faster than larger point DFT functions, it may be useful to use these to debug your code or in your initial design space exploration.

• Your report must explicitly state how you calculated the throughput results. Note that this is often not simply a function of the latency and the clock period, and involves using the initiation interval (II). Hint: Think about how many outputs you obtain from one run of your DFT operation.

• Here are samples for throughput results achieved by previous students for the DFT project:

<table>
<thead>
<tr>
<th>Examples of max throughput:</th>
<th>DFT256 Hz</th>
<th>DFT1024 Hz</th>
</tr>
</thead>
</table>

2.4.5 5) Questions

• **Question 1:** What changes would this code require if you were to use a custom CORDIC similar to what you designed for Project: CORDIC? Compared to a baseline code with HLS math functions for \( \cos() \) and \( \sin() \), would changing the accuracy of your CORDIC core make the DFT hardware resource usage change? How would it affect the performance? Note that you do not need to implement the CORDIC in your code, we are just asking you to discuss potential tradeoffs that would be possible if you used a CORDIC that you designed instead of the one from Xilinx.

• **Question 2:** Rewrite the code to eliminate these math function calls (i.e. \( \cos() \) and \( \sin() \)) by utilizing a table lookup. How does this change the throughput and resource utilization? What happens to the table lookup when you change the size of your DFT?

• **Question 3:** Modify the DFT function interface so that the input and outputs are stored in separate arrays. How does this affect the optimizations that you can perform? How does it change the performance? And how does the resource usage change? Modify your testbench to accommodate this change to DFT interface. **You should use this modified interface for the remaining questions.**

• **Question 4:** Study the effects of loop unrolling and array partitioning on the performance and resource utilization. What is the relationship between array partitioning and loop unrolling? Does it help to perform one without the other? Plot the performance in terms of number of DFT operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for resources (showing LUTs, FFs, DSP blocks, BRAMs). What is the general trend in both cases? Which design would you select? Why?

• **Question 5:** Please read the dataflow section in the HLS User Guide pages 145-154, or the summary at this page, and apply dataflow pragma to your design to improve throughput. You may need to change your code and make submodules so that it aligns with the task-level or function-level modularity that dataflow can exploit; an example of dataflow code is available here. How much improvement can you make with it? How does your dataflow design affect resource usage; how did it change compared to without dataflow? What about BRAM usage specifically? Please describe your architecture with figures on your report.

• **Question 6:** (Best architecture) Briefly describe your “best” architecture. In what way is it the best? What optimizations did you use to obtain this result? What is tradeoff you consider for the best architecture?

• **Question 7:** (Bonus; streaming architecture) If you create a design using hls::stream, you will get bonus points for Project 3. We do not provide a testbench for this case since this is optional. You must write your own testbench because we expect you to change the function prototype from DTYPE to hls::stream. Please briefly describe what benefit you can achieve with hls::stream and why? **NOTE:** To get the extra credit, your design must pass Co-Simulation (not just C-Simulation). You can learn about hls::stream from the HLS User Guide.
2.4.6 6) PYNQ Demo

For this demo, you will create an IP for the DFT 1024, and run it from the Jupyter notebook using two DMAs. You need to follow the Lab: Axistream Multiple DMAs example, with the major difference being that you will have 2 inputs and 2 outputs instead of 2 inputs and 1 output. Therefore you will have to enable read and write for both the DMAs, which is different from the lab instructions. You will additionally need to change the depth of your variable interface ports (you can read more about that here). For the single s_axilite port, you can either choose to do port=length like we did for the lab (in which case you will need to add a constant to your block diagram like we do in the lab, and you will need to write the length from Jupyter to the appropriate address), or you can choose to do port=return (in which case ap_start will not appear in your HLS IP, and you will need to write 1 to the appropriate address from Jupyter to start the process like in previous projects and labs).

Note that the DTYPE struct in this project is almost identical to the axis_t typedef we used in the multiple DMA, here containing a float (data) and an int (last).

Unlike the lab here you cannot start computation immediately after you stream an input struct. You must stream in all struct inputs, then compute the DFT using their float components, and finally stream all outputs as structs. When streaming the output structs, the last bit should be set to 1 for the last struct to be streamed, indicating end of stream. You may need to explicitly set the other last bits to 0, otherwise your stream may terminate early and without warning since there may be garbage data at the memory addresses of the struct you create that are streamed out. You do not need to do this for inputs, as the tool takes care of it for you. Sometimes, the output streaming’s last bit is also handled by the tool, but sometimes it may not be, which will cause the DMA to hang (corresponding to a forever-running Jupyter cell) and it is better to hard code it.

Another point worth discussing here is why we use pointers for inputs and outputs, and why we have to post-increment the pointer manually (like we did in the multiple DMA lab) when we stream inputs and outputs, but why it is a bad idea to use pointers in your code. You cannot use pointers in HLS; pointers are dynamic memory and Vivado HLS will not be able to synthesize it since it is not a deterministic thing (datapath could change depending on inputs). Arrays, on the other hand, are fixed memory locations and therefore they can be synthesized to vectors in RTL. You can use pointers only as ports and even then you have to specify axistream, otherwise that will lead to synthesis issues as well.

In Vivado, the HP ports are High Performance ports which can be accessed by several interfaces. It is something like dynamic channel (also known as memory) which can access the entire channel at one go. Therefore it is not necessary to enable more than one HP port. This link says to use two HP ports if you value performance. If you use multiple HP ports, in the memory map you can see this will give you more space to access (like 512M instead of 256M). So it is always safer to use separate ports although not required. You should have both DMAs be write-enabled (the lab had only one output, but here you have two outputs, so we’ll need both). If you choose to use more than one HP port, HP0 and HP1 should have different masters. So HP0 will have the first DMA as its master, and HP1 will have the second DMA. Two DMAs can point to a single HP port, but two HP ports cannot have the same DMA as master. Pay attention to which DMAs have been assigned to which interface variables, so you know what values are coming out of the fabric.

2.4.7 7) Submission Procedure

You must submit your code (and only your code, not other files). Your code should have everything in it so that we can synthesize it directly. This means that you should use pragmas in your code, and not use the UI to insert optimization directives. We must be able to use what is provided (*.cpp, *.h files, and *.tcl) and directly synthesize it. We must be able to only import your source file and directly synthesize it. If you change test benches to answer questions, please submit them as well. You can assume that we have correctly set up the design environment (dft_test.cpp, dft.h, etc.).
You must follow the file structure below. We use automated scripts to pull your data, so **DOUBLE CHECK** your file/folder names to make sure it corresponds to the instructions.

Your repo must contain a folder named “dft” at the top-level. This folder must be organized as follows (similar to the structure in other projects):

**Contents:**

- Report.pdf
- Folder dft256_baseline
  - Folder dft256_optimized1
  - Folder dft256_optimized2
  - ...
- Folder dft256_dataflow
- Folder dft256_best
- Folder dft1024_best
- Folder Demo: DFT.ipynb | dft.bit | dft.hwh

**Note:** Provide every architecture that you used to answer the questions: make sure each folder contains the source code (*.cpp, *.h, *.tcl only) and the reports (.rpt and .xml).

**Note** Do not submit DFT 8 and 32.

### 2.4.8 8) Grading Rubric

**50 points:** Response to the questions in your report. Your answers should be well written and clearly delineated (for example: by copying the questions into the report before answering them, or placing each question under a separate subheading). Additional points (up to 20) will be subtracted for poor formatting and/or answers that are hard to understand. Examples of issues include any spelling errors, multiple/egregious grammar errors, poor presentation of results, lack of written comparison of the results, etc. Report the throughput and resource usage for each design you discuss in your report, and include the files for these designs in your submission. We encourage the use of tables for stating results and the changes that produced them, and figures to draw comparisons between different designs. Use these figures and tables in your discussion. A well-written report is informative but not overly verbose. You will be deducted points if you do not follow the instructions on directory naming and file structure.

**50 points:** Correct working project on PYNQ.

### 2.5 Project: Fast Fourier Transform (FFT)

#### 2.5.1 1) Introduction

In this project, you will work on the Fast Fourier Transform (FFT) The project guides you through the process of building an efficient FFT module one submodule at a time. It is an improvement of DFT in terms of computational efficiency. You will be implementing this with AXI4 burst mode.

#### 2.5.2 2) Project Goal

The FFT implementation is divided into multiple stages. The first stage of the FFT reorders the input data using a bit reversal scheme. This gets added into a “software” version of the code which we have provided for you (minus
the bit reversal portion). After that, you will create a more hardware friendly FFT architecture. We have provided a set of testbenches for individual functions in addition to the testbenches for the overall FFT. While the major goal of this project is create a functional core, you will also perform optimizations on the code. **In particular, you have to achieve a target throughput in a final 1024-size FFT design that is less than 2000 clock cycles; therefore with a 10 ns clock period that is 50KHz.** This can be achieved by optimizing the submodules properly and using dataflow pragma across the submodules.

### 2.5.3 3) Materials

Download.

You are given a zip file with four folders 0_Initial, 1_Subcomponents, 2_Skeleton_Restructured. Folder 0_Initial contains the files corresponding to the “software” version of the FFT. Folder 2_Skeleton_Restructured provides a framework for a more optimized FFT implementation. And folder 1_Subcomponents has a number of subfolders that allow you to create projects for individual functions that you will develop over the project. This is largely for your convenience for testing and development. All of the code developed here will eventually be placed into 0_Initial and 2_Skeleton_Restructured.

The structure of each of these folders is largely the same.

- `~.cpp` - The place where you write your synthesizable code.
- `~.h` - header file with various definitions that may be useful for developing your code.
- `~test.cpp` - test bench out.gold.dat - “Golden” output. The testbench (~test.cpp) generates a sample input and calls the function in ~.cpp with that sample input. This output of the function is compared to the expected output. This will indicate PASS or FAIL. If it fails, then the code in ~.cpp is incorrect.
- `script.tcl` and `directive.tcl` - These allow you to easily create a project.

### 2.5.4 4) Design Instructions

The FFT is a more efficient version of the Discrete Fourier Transform (DFT). The FFT utilizes symmetry in the DFT coefficients to provide a recursive implementation that reduces the runtime from $O(N^2)$ to $O(N \log N)$ where $N$ is the number of samples in the input signal.

Your tasks for this part of the lab are:

1. Implement a working FFT module that passes the testbench in HLS.
2. Optimize the FFT module to achieve a target throughput

FFT input data reordering via bit reversal

The first step in most optimized FFT implementation is to reorder the input data by performing “bit reversed” swapping. This allows for in-place computation of the FFT, i.e., the resulting “frequency domain” data (as well as the intermediate results) can be stored into the same locations as the input “time domain” data. In addition, the output frequency domain data will be in the “correct order” at the end of the computation.

An example of the bit reversed data for an 8 point FFT is as follows:
In other words, the input data that was initially stored in the array at location 1 is stored in location 4 after the bit reversal is completed. The input data stored in the array at location 4 will be put in array location 1. The input data stored in locations 0, 2, 5 and 7 stay in those locations. Note that this is only true for an 8 point FFT. Other sizes of FFT will have different reordering of the data though it is still based on the bit reversed pattern. For example, in a 16 point FFT, the input data stored in location 1 (binary 0001) will be relocated into location 8 (binary 1000).

You should create an architecture that, efficiently as possible, transforms the input data into a bit reversed order. Note that there are many “software” implementations of this that will not effectively map to “hardware”. While the first goal is to get a working function, you should also consider the performance of the architecture.

We have given you a set of files that allows you to develop and test this bit reversal code in isolation. This includes a simple testbench that exercises this function directly. You should develop and optimize your bit reversed code here. You will later copy this code into the FFT code.

This code is in subfolder 1_bit_reverse in the folder 1_Subcomponents. You should develop your code here to insure that it matches the expected result. Note that this testbench is exercising only one input/output result. In other words, even if it passes this, it may not pass all results. Feel free to add additional testbenches to insure your code is correct.

The bit reverse function has the following prototype: void bit_reverse(DTYPE X_R[SIZE], DTYPE X_I[SIZE])

You should perform the swapping “in place” on the data in both of the real and imaginary portions of the data. That is the input data in both X_R and X_I will be reordered when the function completes. Focus on how you modified your code in order to make it more “hardware friendly”.

**Hint:** Logical operations map well to hardware. Calculating the indices of the arrays that should be swapped can be done with logical operations.

### Optimizing the “Software” Version of the FFT

The next portion of this project performs optimization on a typical software implementation of the FFT. You are given typical three nested loop implementation of the FFT in the folder 0_Initial. First, you should understand in detail what this code is doing. It is worth spending time on this now as you will have to rewrite the FFT in a more hardware friendly manner in the next steps. You can reuse some of this code in those steps.

You should optimize this code as much as possible. The results of the code will be poor; it will likely have > 250 million cycles. The throughput here is likely much worse than running this in software on a microprocessor. This
often happens when we put the initial software versions of an application into a high level synthesis tool. And it should not be all that surprising. The code is optimized to run quickly in software, which runs largely in a sequential model of computation. The code must typically be carefully optimized with the final hardware architecture in mind to get good results. This involves exploiting parallelism and pipelining.

You will also notice that the first loop has function calls to sine and cosine. This code will synthesize quickly with these function calls. However, you may wish to replace these function calls (which will synthesize into CORDIC cores), into table lookups. We have provided two tables in the header file, \( W_{\text{real}} \) and \( W_{\text{imag}} \) which contain the precomputed twiddle factors for our 1024 FFT, i.e., \( W_{\text{real}}[i] = \cos(2.0 \times \pi \times i/\text{SIZE}) \) and \( W_{\text{imag}}[i] = \sin(2.0 \times \pi \times i/\text{SIZE}) \) where \( i \) is in \([0,512)\).

Some potential optimizations include:
- Using the \( W_{\text{real}} \) and \( W_{\text{imag}} \) tables
- Pipelining
- Loop unrolling
- Memory partitioning

**Hardware Friendly FFT Implementation**

A good architecture will selectively expose and take advantage of parallelism, and allow for pipelining. Your final FFT architecture will restructure the code such that each stage is computed in a separate function or module. There will be one module for bit reversal that you have already developed, and then \( \log N \) stages (10 in our case) for the butterfly computations corresponding to the 2-point, 4-point, 8-point, 16-point, … FFT stages.

The skeleton code for this final FFT implementation can be found in the 2_Skeleton_Restructured folder. This creates code connects a number of functions in a staged fashion with arrays acting as buffers between the stages. Figure 1 provides a graphical depiction of this process.
Figure 1: A staged implementation of a 1024 FFT. Bit reversal is followed by 10 stages of butterfly computations. This architecture is capable of pipeline both within the stages and across the stages.

The first step in this process is to create code that computes the first and last stages of the FFT. The hope is that this will allow you to get a better understanding of exactly how memory accesses and the butterfly computations are performed in a general case. You can develop these two functions `fft_stage_first` and `fft_stage_last` in isolation. They both have subfolders in the 1_Subcomponents folder. Once these are working correctly, you can copy and paste the code directly in the same functions in the 2_Skeleton_Restructured project.

The next task is to create code that can implement “generic” function, i.e., one that can compute any stage of the FFT. This is the function `fft_stages` which also has its own project in the 1_Subcomponents folder. Note that this function prototype is similar to `fft_stage_first` and `fft_stage_last` with one major difference: it has a stage argument. This code will used to implement stages 2 through 9 in the 2_Skeleton_Restructured project.

**Hints:**

- These stages are performing the same calculation as one iteration of the outer for loop in the 0_Initial project.
- The major difference between the stages is what data elements you are performing the butterfly functions on, i.e., in what order do you pull data from `X_R` and `X_I`.
- Test each of the functions in isolation with the provided projects. Make sure that the code compiles and passes the testbench before attempting any optimizations.

Once you have a correctly functioning set of functions, you should copy and paste them in the 2_Skeleton_Restructured project and make sure that it passes the testbench. Since our testbenches only perform one check, which is far from comprehensive, it is possible, though hopefully unlikely, that you have some error that the 2_Skeleton_Restructured testbench exposes and was not exercised in the individual testbench. If your code passes the 2_Skeleton_Restructured project you can assume it is correct (though again since it is only one test, it may be wrong; you would need to perform significantly more testing in any “non-class” situation).

Now onto the final part of the project, optimizing of this restructured code. You should perform the typical tricks here:
pipelining, memory partitioning, unrolling, etc. Some of these may not make sense depending on how you wrote your code. This final architecture should be orders of magnitude better than the 0_Initial project. Highly optimized FFT architectures can easily have less than 10000 cycles. Here are sample results achieved by previous students for the FFT project:

<table>
<thead>
<tr>
<th>Past bests: FFT1024</th>
<th>Latency (cycles)</th>
<th>BRAMs (%)</th>
<th>DSPs (%)</th>
<th>FFs (%)</th>
<th>LUTs (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1027</td>
<td>17</td>
<td>97</td>
<td>29</td>
<td>78</td>
</tr>
<tr>
<td>B</td>
<td>1033</td>
<td>35</td>
<td>100</td>
<td>43</td>
<td>96</td>
</tr>
</tbody>
</table>

Your name here :)  

Optimization Guidelines

- You must always use a clock period of 10 ns.
- The output of the various architectures that you generate must match the golden output. We have broken down the project into subcomponents to allow you to develop and test them individually. You would be wise to do it in such a manner.
- You should not change the data types as given to you. You do not need to perform bitwidth optimization of this project.
- It is OK to rewrite the code if it helps you with optimizations. For example, you can change the function interfaces. There are some variables defined in the header files for your convenience. These include $SIZE = 1024$, $SIZE2 = 512$, and $M = 10$ (i.e. $\log SIZE$). Feel free to use these in your code. They are defined in every header file across all of the different folders. The software version has a nested for loop structure that does not allow Vivado HLS to provide an exact number of cycles. The `tripcount` directive can help with this. You should be able to understand the reported results. For example, while Vivado may give you a best, worst and average case numbers, your algorithm for a fixed size FFT should be a fixed number of cycles.

2.5.5 5) PYNQ Demo

For this demo, you will create an IP for the FFT 1024, and run it from the Jupyter notebook using AXI4. You need to follow the Lab: AXI4-Burst Mode example, with the major difference being you will have 2 inputs and 2 outputs instead of 1 input and 1 output.

2.5.6 6) Submission Procedure

You must also submit your code (and only your code, not other files, not HLS project files). Your code should have everything in it so that we can synthesize it directly. This means that you should use pragmas in your code, and not use the GUI to insert optimization directives. We must be able to only import your source file and directly synthesize it. If you change test benches to answer questions, please submit them as well. You can assume that we have correctly set up the design environment (fft.cpp, fft.h, etc.).

You must follow the file structure below. We use automated scripts to pull your data, so DOUBLE CHECK your file/folder names to make sure it corresponds to the instructions.

Your repo must contain a folder named “fft” at the top-level. This folder must be organized as follows (similar to previous projects):

Contents:

- `Report.pdf`
- Folder `fft1024_best`
  - Source code (*.cpp, *.h, *.tcl only) and reports (.rpt and .xml).
• Folder Demo
  – .bit and .hwh files
  – FFT.ipynb host file

Report:
For this project, you need to submit an 1-page report to explain only your final architecture. You can add figures, diagrams, tables, or charts to describe your architecture with a short paragraph explaining them. A 2-page report at max is allowed if it is necessary. No questions; no answers. Just explain your design. We will check if (1) your final FFT design is functionally correct and (2) your final FFT design achieves the target performance. The report will help us to understand your design.

2.5.7 7) Grading Rubric

50 points: Functionally correct design
40 points: Achieving target performance
10 points: Report

2.6 Project: OFDM Receiver

2.6.1 1) Introduction

In this project, you will learn the basic idea behind an orthogonal frequency-division multiplexing (OFDM) system by implementing a simple OFDM receiver in programmable logic. A major part of OFDM is a Fast Fourier Transform (FFT), and thus you will be working with your FFT implementation from the previous project. You are given a set of test benches for the different submodules. You should design and test each individual submodule individually and integrate them into the FFT module. In the final part of the project, you will complete the OFDM receiver by combining the FFT module with a QPSK symbol decoder.

2.6.2 2) Project Goal

The OFDM receiver is divided into two parts – the FFT and the QPSK decoder. You will be working with your FFT module from the FFT project. You must design and implement a QPSK decoder, and integrate it with the FFT to complete the receiver. While the major goal of this project is create a functional core, you will also perform optimizations on the code.

2.6.3 3) Materials

Download.

You are given a zip file with four folders 0_Initial, 1_Subcomponents, 2_Skeleton_Restructured, and 3_OFDM. Folder 0_Initial contains the files corresponding to the “software” version of the FFT. Folder 2_Skeleton_Restructured provides a framework for a more optimized FFT implementation. Folder 3_OFDM gives a basic structure for the OFDM receiver with QPSK decoder. And folder 1_Subcomponents has a number of subfolders that allow you to create projects for individual functions that you will develop over the project. This is largely for your convenience for testing and development. All of the code developed here will eventually be placed into 0_Initial and 2_Skeleton_Restructured.

The structure of each of these folders is largely the same.
  • ~.cpp - The place where you write your synthesizable code.
• ~.h - header file with various definitions that may be useful for developing your code.

• ~test.cpp - test bench out.gold.dat - “Golden” output. The testbench (~test.cpp) generates a sample input and calls the function in ~.cpp with that sample input. This output of the function is compared to the expected output. This will indicate PASS or FAIL. If it fails, then the code in ~.cpp is incorrect.

• script.tcl and directive.tcl - These allow you to easily create a project.

2.6.4 4) Design Instructions

Your tasks for this part of the lab are:

1. Implement the QPSK decoder.

2. Integrate the FFT and decoder into a complete OFDM receiver.

The decoder takes the output of the FFT (complex values) and translates them into data. This is essentially undoing the effect of the QPSK encoder which takes input data for transmission and encodes it into a complex exponential i.e., an I/Q complex number. You can think of this as a translation from the input data into a complex number.

We used the QPSK encoding scheme shown in the below figure. The plot shows four points in the complex plane at (+ 0.707, +- 0.707). This is called a constellation. Each of these points is labeled with an integer value 0, 1, 2 or 3. These integer values correspond to the input data being encoded. You can also think of these as two bit values if you want to consider binary input data. The complex values are the I/Q data that is encoded onto a specific frequency (e.g., one of 1024 frequencies when using a 1024 point FFT). The decoder performs the opposite – it takes a complex number and translates it into an integer. You can look at the Simulink file for more information. This figure is taken directly from that file. The output of your encoder should be the exact data that was given to the OFDM receiver in the Simulink file.
Receiver Integration

You should connect the FFT and the QPSK decoder together to form the complete OFDM receiver. The input to the receiver is the data from the channel. The output of the receiver should match the transmitted data.

Optimization Guidelines

- You must always use a clock period of 10 ns.
- The output of the various architectures that you generate must match the golden output. We have broken down the project into subcomponents to allow you to develop and test them individually. You would be wise to do it in such a manner.
- You should not change the data types as given to you. You do not need to perform bitwidth optimization of this project.
• It is OK to rewrite the code if it helps you with optimizations. For example, you can change the function interfaces. There are some variables defined in the header files for your convenience. These include \( \text{SIZE} = 1024 \), \( \text{SIZE2} = 512 \), and \( M = 10 \) (i.e. \( \log \text{SIZE} \)). Feel free to use these in your code. They are defined in every header file across all of the different folders. The software version has a nested for loop structure that does not allow Vivado HLS to provide an exact number of cycles. The \textit{tripcount} directive can help with this. You should be able to understand the reported results. For example, while Vivado may give you a best, worst and average case numbers, your algorithm for a fixed size FFT should be a fixed number of cycles.

\section*{2.6.5 5) PYNQ Demo}

The final part is to integrate the receiver onto the Zynq’s processing system (PS) using a proper interface to transmit data to the OFDM receiver, and receive the decoded data back from your hardware implementation on the Zynq’s programmable logic (PL).

We provided the general framework for creating different PL-PS interfaces and Jupyter Notebook host applications in previous labs. You should use that to create this demo. We will not be providing you with anything more than what was given in previous labs. You are free to modify the function interface in Vivado HLS however you please, and to use whichever streaming method you like.

\textbf{The end of your Jupyter notebook must contain code that verifies your output.} Your can either use the provided input/output, and plot the received data against the golden output, or compare custom signals against a software version of OFDM (similar to what was done in the FFT project).

\section*{2.6.6 6) Submission Procedure}

You must also submit your code (and only your code, not other files, not HLS project files). Your code should have everything in it so that we can synthesize it directly. This means that you should use pragmas in your code, and not use the GUI to insert optimization directives. We must be able to only import your source file and directly synthesize it. If you change test benches to answer questions, please submit them as well. You can assume that we have correctly set up the design environment (fft.cpp, fft.h, etc.).

You must follow the file structure below. We use automated scripts to pull your data, so \textbf{DOUBLE CHECK} your file/folder names to make sure it corresponds to the instructions.

Your repo must contain a folder named “ofdm_receiver” at the top-level. This folder must be organized as follows (similar to previous projects):

\textbf{Contents:}

• Report.pdf
• Folder fft1024\_best
  – Source code (*.cpp, *.h, *.tcl only) and reports (.rpt and .xml).
• Folder OFDM\_receiver
  – Folder containing all of the HLS files necessary to build the complete OFDM receiver.
• Folder Demo
  – .bit and .hwh files
  – .ipynb host file
• You are welcome to include multiple OFDM architectures if you please, in additional folders.

\textbf{Report:}
For this project, you need to submit an 1-page report to explain only your final architecture. You can add figures, diagrams, tables, or charts to describe your architecture with a short paragraph explaining them. A 2-page report at max is allowed if it is necessary. No questions; no answers. Just explain your design. We will check if (1) your final OFDM design is functionally correct and (2) your final FFT design achieves the target performance. The report will help us to understand your design.

2.6.7 7) Grading Rubric

50 points: Functionally correct OFDM design

40 points: Achieving target performance of FFT (exactly the same performance benchmark suggested for the FFT project)

10 points: Report

2.7 Project: Matrix Multiplication on Intel DevCloud Using DPC++

2.7.1 1) Introduction

This project provides an introduction to hardware acceleration using Intel DevCloud. DevCloud provides access to Intel oneAPI - a set of hardware acceleration development tools. We focus on the Data Parallel C++ (DPC++) programming methodology that is a core part of oneAPI. DevCloud provides access to different Intel hardware platforms including multicore CPUs, GPUs, and FPGAs. DPC++ aims to provide a single source programming methodology that covers these different hardware accelerators.

2.7.2 2) Project Goal

We use matrix multiplication as an example as it is relatively simple, yet fundamentally important computation used in a wide variety of applications. As such, it is very well-studied in the hardware acceleration literature with a substantial number of technical reports and research papers. The project develops a block matrix multiplication architecture and discusses some common methods to optimize it. This enables a design space exploration process to determine the best architecture.

Before we begin, please complete Lab: DPC++ on Intel DevCloud. We will use this code as your baseline architecture.

2.7.3 3) Tasks and Questions

Optimize Load Transfers

As discussed in the lab, the load store unit (LSU) in the baseline implementation requires hundreds of cycles and is the major bottleneck in the c_calc kernel. DPC++ uses a Burst/Coalesced LSU by default. Burst/Coalesced buffers contiguous memory requests until it reaches the maximum burst size. Change the LSU type to lower the latency for the load operations. At this link, you can find examples of LSU types being used in code.

- Modify the LD operation to a different LSU style to achieve a lower latency. This should bring the c_calc.B2 latency to less than 25 cycles.
- Question 1: Describe your modification and discuss why it achieves a lower latency.

References:

- oneAPI Samples
Loop Unrolling

Loop unrolling is a common optimization to expose parallelism across iterations of a loop.

- Perform loop unrolling using `#pragma unroll`. Change the unroll factor by 2, 4, and 8. You will need to change the widths and heights of the matrices to be powers of two; the default values are multiples of 150, which are not cleanly divisible by every unroll factor stated. Mention the matrix sizes in your report.

- Question 2: What are the effects and general trends of performing unrolling using the pragma? Are the results as expected?

Manually unroll the loop by replicating the reductions into separate variables.

- Perform loop unrolling manually. Change the unroll factor by 2, 4, and 8. You will need to use the same change as in the previous question.

- Question 3: What are the effects and general trends of performing manual unrolling? Are the results as expected?

References:

- The Schedule Viewer typically provides good insight about how DPC++ synthesizes the code.
- oneAPI FPGA Optimization Guide
- dpcpp FPGA loop unroll example

2.7.4 4) Block Matrix Multiplication (BMM)

Block matrix multiplication is a common way to expose parallelism by loading and operating on blocks of the A and B matrices. Here is a BMM implementation in DPC++ BMM_DPCPP.zip. It is based on this OpenCL implementation, which provides good background on blocking and how the design leverages it for parallel execution. This is different from the implementation you were using earlier; this implementation has the additional ability to change the block size and unrolling factor. We call these “knobs” since they can be changed to “tune” the design to the problem at hand. You can upload the zip file to DevCloud using the Jupyter interface and unzip it via the Jupyter terminal.

Your goal is to change these knobs and observe their effects. After a few steps, you should see trends for each knob. You should use the results from previous steps to make better adjustments. You may want to adjust the LSU to something appropriate for the knobs you adjust, like you did for the previous questions. By focusing on a particular knob, try to maximize throughput (while adjusting the other knobs as necessary to help achieve this maximum). After you’ve done this for a few designs for each knob, you should be able to gain an understanding of which knobs are important in this design, and how/why. With this knowledge, we ask you to find points on the Pareto frontier (if you didn’t already find them from the experimentation earlier). However, you don’t have to achieve the absolute best maximum throughput for any particular knob; this is a design-space exploration problem and the goal is to gain an understanding of what possibilities for improvements exist and what trade-offs come with them. Briefly discuss these trends you noticed and what your thought process was as you moved from one design to another in your report, and reference the chart to explain your thoughts.

You also have to show the result of each knob adjustment in a normalized throughput vs. normalized hardware utilization plot. Data points in that plot with the maximum throughput and minimum resource utilization are your plateau points (red dots in the following example). At minimum, we ask for just 1 plot for all of the tuning experiments, preferably annotate each point with the knobs you used to produce that design (maybe even with color coding the points to classify them). You may, however, want to plot a separate chart for each knob that you adjust so you can identify the trends for yourself. But the idea of question 4 is to compare every design you synthesize to find the Pareto frontier (in the example chart below, draw a line between the red points to roughly visualize the frontier), so ideally that would be by being able to compare every design on a single chart. Do whatever helps you best distinguish the trends.
After you decide on a design you think is best, try this design with different matrix sizes: 128, 256, 512, 1024, 2048, 4096. Report the performance of your design on these input sizes. You can report additional sizes if you like.

Note: Notice that the provided code prints throughput in kb/s on the command line output. This number is dependent on the fact that the code is running on a server, sharing resources with other programs submitted by other users. Therefore, multiple runs of the same program will print different throughputs, depending on who is running what and when. To avoid this, there are a few strategies you can choose from. You could run the program multiple times and take the average of throughputs over runs, or (preferably) you can use the normalized 1/(loop latency) from the synthesis report as a proxy for throughput.

Requirements

1. **Knobs**: You should define a set of variables (knobs) to change your optimizations for monitoring their effects on your design’s performance and hardware utilization. You should use the following knobs:
   - Block size
   - Matrix size (we use square matrices)
   - Unrolling factor for the unroll pragma
   - Unrolling factor for the manual unrolling

References: Spector is a good example to start with DSE; sample codes are available here.

Bonus

The *OpenCL implementation* is simpler than the matrix multiply implementation used in Spector. As a bonus, you can implement the matrix multiply implementation used in Spector, in DPC++. A functionally correct code is enough for this section.

2.7.5 5) Submission Procedure

You must also submit your code (and only your code, not other files). Your code should have everything in it so that we can synthesize it directly. We must be able to only import your source file and directly synthesize it. You can
assume that we have correctly set up the design environment.
You must follow the file structure below. We use automated scripts to pull your data, so **DOUBLE CHECK** your file/folder names to make sure it corresponds to the instructions.

Your repo must contain a folder named “matrix_multiplication” at the top-level. This folder must be organized as follows (similar to previous projects):

**Contents:**

- Report.pdf
- Folder **mm_optimized1**
  - Source code (matrix_mul_dpcpp.cpp) and reports (screenshots).
- Folder **mm_optimized2**
  - Source code (matrix_mul_dpcpp.cpp) and reports (screenshots).
- Folder **mm_optimized3**
  - Source code (matrix_mul_dpcpp.cpp) and reports (screenshots).
- Folder **bmm_optimized**
  - Source code (matrix_mul_dpcpp.cpp) and reports (screenshots).

**Report:** For this project, you must submit a report that answers the questions on this page. You may add figures, diagrams, tables, or charts to describe your architectures with a sufficient explanation of how they were achieved and what they demonstrate. You can submit the synthesized report screenshots as image files or include them as (properly labeled) figures in your report.

### 2.7.6 6) Grading Rubric

**100 points:** Your grade will be determined by your answers to the questions. Your answers should be well written and clearly delineated (for example: by copying the questions into the report before answering them, or placing each question under a separate subheading). Additional points (up to 20) will be subtracted for poor formatting and/or answers that are hard to understand. Examples of issues include any spelling errors, multiple/egregious grammar errors, poor presentation of results, lack of written comparison of the results, etc. Report throughput and resource usage for each design you discuss in your report, and include the files for these designs in your submission. We encourage the use of tables for stating results and the changes that produced them, and figures to draw comparisons between different designs. A well-written report is informative but not overly verbose. You will be deducted points if you do not follow the instructions on directory naming and file structure.

### 2.8 Project: FM Demodulator

#### 2.8.1 1) Introduction

In this project we use the RTL2832 RF tuner to sample RF signals and will build a FM Demodulator and implement it on the Pynq Board.

#### 2.8.2 2) Project Goal

In this project, you will use your knowledge from previous projects to implement an FM Demodulator in programmable logic. The project is divided into two parts.
In the first part, you develop different functions to implement the scikit-dsp-comm mono_FM Demodulator in Vivado HLS. This FM Demodulator consists of a linear filter, downsampler, and a discriminator. The second part is to integrate the Demodulator onto the Pynq Board. You should be able to listen to local FM radio channel using your MonoFM implementation in programmable logic.

2.8.3 3) Materials

Download.
This contains a python notebook which explains the working of a Mono FM Demodulator.
For this project the following will not be provided:
• .cpp - The place where you write synthesizable code
• .h - header file with various definitions that may be useful for developing the code
• -test.cpp - testbench
You will have to build the entire project from scratch

2.8.4 4) Design Instructions

The FM Demodulator has 3 main parts: downsampler, linear filter and discriminator.

downsampler

This part consists of a straight forward downsampler. We have to downsample by a factor of N, that is keep every Nth sample. The implementation of downsampler can be found here.

linear filter

Build a linear filter whose function is implemented as a direct II transposed structure.
This means that the filter implements:


More information about the linear filter implementation can be found here.

discriminator

To demodulate FM we require a discriminator circuit, which gives an output which is proportional to the input frequency deviation.

```python
def discrim(x):
    """
    function disdata = discrim(x)
    where x is an angle modulated signal in complex baseband form.
    """

    Mark Wickert
    """
    X=np.real(x)  # X is the real part of the received signal
```

(continues on next page)
Y = np.imag(x) # Y is the imaginary part of the received signal
b = np.array([1, -1]) # filter coefficients for discrete derivative
a = np.array([1, 0]) # filter coefficients for discrete derivative
derY = signal.lfilter(b, a, Y) # derivative of Y
derX = signal.lfilter(b, a, X) # " " X,
disdata = (X*derY-Y*derX)/(X**2+Y**2)
return disdata

The above code is the scikit-dsp-comm implementation of an FM baseband discriminator.

To understand the operation of `discrim()` start with a general FM modulated signal and obtain the complex envelope:

\[
X_c(t) = A_c \cos(\omega_c t + \phi(t))
\]
\[
= Re\{A_c e^{j\phi(t)} e^{j\omega_c t}\}
\]
\[
= A_c Re\{[\cos(\phi(t)) + j \sin(\phi(t))] e^{j\omega_c t}\}
\]

The complex envelope is:

\[
\hat{X}_c(t) = \cos(\phi(t)) + j \sin(\phi(t))
\]
\[
= X_R(t) + j X_I(t)
\]

A frequency discriminator obtains the derivative of the modulated angle:

\[
\phi(t) = \tan^{-1}\left(\frac{X_I(t)}{X_R(t)}\right)
\]

And its derivative is:

\[
\frac{d\phi(t)}{dt} = \frac{X_R(t)X'_I(t) - X'_R(t)X_I(t)}{X_R^2(t) + X_I^2(t)}
\]

Optimization Guidelines

- You must always use a clock period of 10 ns.
- The latency for demodulation should be less than the sample time; i.e. less than 1 second for 1 second sample time.

2.8.5 5) PYNQ Demo

This project is different from your previous projects in the sense that it has to achieve a real time performance, with a processing time less than the sampling time. You are highly encouraged to modify the code to achieve a better performance and observe the throughput by changing the way you transmit data between PS and PL. Make use of the “RTL 2832” USB tuner in-order to receive the input RF Samples.
2.8.6 6) Submission Procedure

You have to submit your code (and only your code, not other files, not HLS project files). Your code should have everything in it so that we can synthesize it directly. This means that you should use pragmas in your code, and not use the GUI to insert optimization directives. We must be able to only import your source file and directly synthesize it.

You must follow the file structure below. We use automated scripts to pull your data, so DOUBLE CHECK your file/folder names to make sure it corresponds to the instructions.

Your repo must contain a folder named “mono_fm” at the top-level. This folder must be organized as follows (similar to previous projects):

Contents:

- Report.pdf
- Folder fm-demodulator
  - Source code (*.cpp, *.h, *.tcl only) and reports (.rpt and .xml).
- Folder Demo
  - .bit and .hwf files
  - FM.ipynb host file

Report: For this project, you must submit a report with 1 page for each function from section 4. You may add figures, diagrams, tables, or charts to describe your architectures with a short paragraph explaining them. No questions; no answers. Just explain your design. We will check if (1) your final FM Demodulation functions are functionally correct (they pass their test benches) and (2) achieves target latency. The report will help us to understand your design. You also can use this report to explain your work for bonus part (check the grading section).

2.8.7 7) Grading Rubric

30 points: Functionally correct design. You will get full credit if we are able to build your blocks without any effort. You need to report the throughput of the final design.

60 points: Pynq Demo. You will get full credit for clear audio output and an RMSE less than 1e-6.

10 points: Report.

Bonus: Integrate your design with the base overlay to make use of audio instance on the Pynq Board. More information about the audio module on Pynq Z2 can be found here and here. Detailed description of the BaseOverlay can be found here.

2.9 Lab: Pynq Memory Mapped IO (s_axilite)

This lab describes how to use Pynq to develop an application on the Zynq SoC. The application performs a simple hardware accelerated function on the programmable logic. We first create the IP core that performs the function \( f(x) = 2x \) using high level synthesis. We synthesize it to the programmable logic using the Vivado tools. Using the PYNQ infrastructure, we talk to the IP core from ARM processor using memory mapped I/O. We develop a Pynq notebook that sends data to the IP core, executes the core, and receives the computed results.

2.9.1 1) Vivado HLS: C/C++ to RTL

In this section, you will write your code in C/C++, test it, and convert it to RTL using Vivado HLS.
1.1) Write your code

Open Vivado HLS, create a new project, and name it **pynq_mul**

Set top function name to **mul_test**
Do not add any files to your project and proceed to part selection and select xc7z020clg400-1
In Explorer section, right click on Source, select New file and create mul_test.cpp. Complete the body of mul.cpp as following:

```cpp
void mul_test(int* out, int in){
  *out = 2*in;
}
```

Create a test bench file by right clicking on Test Bench in Explorer section and create a new file named mul_tb.cpp. Complete the body of this file as following:

```cpp
#include <iostream>

using namespace std;

void mul_test(int* out, int in);
```

(continues on next page)
```c
int main(int argc, char *argv[]){
    int x=5;
    int y;
    mul_test(&y, x);
    if(y!=2*x){
        cout << "Test Failed: output(" << y << ") is not equal to 2x" << x << endl;
    }else{
        cout << "Test Passed" << endl;
    }
    return 0;
}
```

1.2) Test your code

Run C simulation and make sure your code passes your test bench.

1.3) Set port types

Make sure that `mul_test.cpp` is open. Open Directive on right side and set all the ports to `s_axilite` by right clicking on available options in Directive window.

1.4) Synthesis and export your design

Run C Synthesis and after finished, click on export RTL and export your design.

At this point, you can exit and close Vivado HLS.

2.9.2 2) Vivado: RTL to bitstream

In this section, you will import your RTL code to Vivado and generate a bitstream.

2.1) Create a new project

Open Vivado and create a new project and Name your project as `mul_test`
Select **RTL Project** and check **Do not specify sources at this time**
Set default part to xc7z020clg400-1
Under IP Integrator, click on Create Block Design
2.2) Import your design

Under Project Manager, click on IP Catalog. Right click inside the newly open ‘IP Catalog’ tab and select Add Repository. In the open window navigate to your Vivado HLS project folder and select `<pass_to_vivado_hls_folder>solution1implip`
In IP Catalog search for **mul_test**, double click on it and add it to your block design.
2.3) Add connections

Go back to IP Catalog and add ZYNQ7 Processing System to your block design.

Your diagram should look like the following:
On top of Diagram window, first click and complete Run Block Automation and then Run Connection Automation with default settings. Your diagram should change and show connections and a couple of extra IPs:
2.4) Generate bitstream

In Sources, right click on design_1 and select Create HDL Wrapper
Under **Project Manager**, click on **Generate Bitstream** to build the .bit and .hwh files.

### 2.5) Bitstream, .hwh, and addresses

Before closing Vivado, we need to note our IP and its ports addresses.

Under **Sources**, open `mul_test_mul_io_s_axi.v`, scroll down and note addresses for in and out ports. We need these addresses for our host program.

In the example below for the `streamMul`, the addresses to pay attention to are 0x00 (control bus ap_ctrl), 0x10 (output), and 0x18 (input). These are the addresses you will need to use to write data to the fabric from the ARM core, start the fabric to run your design and generate your outputs, and then read your outputs from the fabric into the ARM core on the Pynq board.
Under **Address Editor** note IP’s address

```
Cell | Slave Interface | Base Name | Offset Address | Range | High Address
-----|-----------------|-----------|----------------|-------|----------------
p processing_system7_0

Data (32 address bits: 0x40000000 [1G])

  - mul_test_0 s_axi_mul_io Reg 0x43C0_0000 64K 0x43C0_FFFF
```

2.9. Lab: Pynq Memory Mapped IO (s_axilite)
2.9.3 3) PYNQ board and Host program

Using SMB or SCP or the Jupyter interface, copy `design_1_wrapper.bit` from `vivado_project_path/mul_test.runs/impl1` and copy `design_1.hwh` from `vivado_project_path/mul_test.srcs/sources_1/bd/design_1/hw_handoff` to your PYNQ board at `/home/xilinx/jupyter_notebooks/mul_test`. Make sure to name the .bit file and the .hwh file with the same name. In this case, we name them “design_1_wrapper.bit” and “design_1_wrapper.hwh”.

Open a new Notebook and run the following code to test your IP

```python
from pynq import Overlay
from pynq import MMIO

ol = Overlay("/home/xilinx/jupyter_notebooks/mul_test/design_1_wrapper.bit") # designate a bitstream to be flashed to the FPGA
ol.download() # flash the FPGA

mul_ip = MMIO(0x43C00000, 0x10000) # (IP_BASE_ADDRESS, ADDRESS_RANGE), told to us in Vivado
inp = 5 # number we want to double

mul_ip.write(0x18, inp) # write input value to input address in fabric
print("input:", mul_ip.read(0x18)) # confirm that our value was written correctly to the fabric
mul_ip.write(0x00, 1) # set ap_start to 1 which initiates the process we wrote to the fabric
print("output:", mul_ip.read(0x10)) # read corresponding output value from the output address of the fabric
```

2.10 Lab: Axistream Single DMA (axis)

2.10.1 Simple streaming example

In this example we learn how to use Xilinx AXI_DMA to create streaming interfaces for and IP. This class will not go too deep into AXI protocols and Vivado, but a nice tutorial of the AXI Direct Memory Access (DMA) exists here.

1) Vivado HLS: Generating RTL code from C/C++ code

In this section you learn how to create a project in Vivado HLS, synthesize your code, and generate RTL.

1.1) Download code and create a Vivado HLS project

Download and unzip `streamMul.zip`. Generate your project using the provided `script.tcl` file:

Linux: open a terminal, make sure your environment is set, navigate to `streamMul` folder, and run the following

```
$ vivado_hls script.tcl
```

Windows: open vivado_hls command line and run the following
Now can open you your project in Vivado HLS. **Your code is not complete!**, modify your code to become same as the following:

```c
#include "streamMul.hpp"

void smul(axis_t *INPUT, axis_t *OUTPUT, unsigned

for(unsigned int i=0; i<br>
  axis_t cur = *INPUT;
  cur.data = cur.data;
  OUTPUT++; cur;

```

INPUT and OUTPUT ports are set to axis interfaces for streaming and length is set to s_axilite for a non-streaming interface. axis_t is a struct defined in the header file that is composed of an int data and an ap_uint<1> last. The 1-bit last is required for axis interfaces, and signals the last struct of the stream, ending the stream. In the pragmas, depth is set to 50 because that’s the maximum number of values we are streaming in and out of the fabric.

Note that

```c
*OUTPUT++ = cur;
```

is performing two separate operations. Breaking it down:

```c
*OUTPUT = cur;  // write the output struct to the address in OUTPUT
OUTPUT++;        // post-increment the address in OUTPUT for the next write operation
```

In this lab, since we are reusing an input struct cur to generate an output struct, the last bit is handled for us. However, if you must construct your own axis_t struct, you must ensure you set last to 1 when the struct is the last one to be streamed out, else explicitly set it to 0 (otherwise there may be garbage data in the memory address of last that
terminates your stream early, leaving you scratching your head about why the output error on Pynq’s Jupyter interface is so high).

You can do so like this:

```
axis_t curr;
curr.data = ...; // write data
curr.last = ...; // set to 1 if end of stream, else set to 0
*OUTPUT++ = curr; // make sure you only write to a particular address once, so do it after the struct is constructed
```

We must interact with them this way because we are dealing with an AXI stream, not an array.

1.2) Generate RTL code and export it

Click on Run C Synthesis to generate RTL code. After it is done, you can check your resource utilization and timing report. Your latency is unknown (?) because your loop size (length) is a variable.
Now you can export your RTL code by clicking on **Export RTL**.
After exporting is done, you can close and exit from Vivado HLS.

2) Vivado: Generating bitstream from RTL code

In this section we import our RTL code from last section, add some required IPs, and generate our bitstream
2.1) Create a new Vivado project

Open your Vivado tool and create a new project. Select an appropriate location for your project and leave the default project name as is (project_1).

Select RTL Project and check Do specify not sources at this time.

Select xc7z020clg400-1 for your part.
2.2) Import RTL code

Under **Flow Navigator**, click on **IP Catalog**. Right click on the opened window and select **Add Repository**. Navigate to your **Vivado HLS project > solution1 > impl > ip** and select it:

- streamMul
  - project_1
- streamMul
  - solution1
  - impl
    - ip
      - misc
      - verilog
      - vhdl
    - syn

2.3) Add IPs to your design

Under **Flow Navigator**, click on **Create Block Design**. Leave the design name as is (**design_1**). In the newly opened window you can add IPs by clicking on the plus sign.

Add **ZYNQ7 Processing System** to your design:
Double click on **ZYNQ IP** to customize it. In the opened window, double click on **High Performance AXI 32b/64b Slave Parts**: 
Select and check S AXI HP0 interface:

**ZYNQ7 Processing System (5.5)**

- Add a Smul to your design and rename it to smul:
Add a AXI Direct Memory Access to your design and rename it to smul_dma. Double click on your AXI DMA and change the following parameters: 1) uncheck Enable Scatter Gather Engine. 2) Change Width of Buffer Length Register to 23:
Add a **Constant** to your design

### 2.4) Manual connections

Connect the following ports:

- `smul::OUTPUT_r` to `smul_dma::S_AXIS_S2MM`
- `smul_dma::M_AXIS_MM2S` to `smul::INPUT_r`
2.5) Automatic connections

Now you can leave the rest of the connections to the tool. There should be a highlighted strip on top of your diagram window.

1. Click on Run Block Automation
2. Click on Run Connection Automation and select all
Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

3. **IMPORTANT!** you have to click again on Run Connection Automation
At this point your design should look like this:
2.6) Generate bitstream

1. Save your design CTRL+S or File > Save Block Design.
2. Validate your design: Tools > Validate Design
3. In Sources, right click on design_1, and Create HDL Wrapper. Now you should have design_1_wrapper.
4. Generate bitstream by clicking on Generate Bitstream in Flow Navigator

2.7) Note required addresses and copy generated files

After bitstream generating process is done, open Address Editor from window menu.

Note that smul address is 0x43C00000, we need this address in our host program for sending length data.
In sources, expand `design_1_wrapper::design_1::streamMul::smul::design_1_smul_0_0::inst : smul`, double click on `smul_CTRL_s_axi_U`, and note the address for `length_r` is `0x10`. We need this address in our host program.

Copy your `project directory > project_1 > project_1.runs > impl_1 > design_1_wrapper` to your project directory `project_1` and rename it to `smul.bit`.

Copy your `project directory > project_1 > project_1.srcs > sources_1 > bd > design_1 > hw_handoff > design_1.hwh` to your `project directory > project_1` and rename it to `smul.hwh`.

You should have both `smul.bit` and `smul.hwh`.

You can close and exit from Vivado tool.
3) Host program

In this section we use Python to test our design.

3.1) Move your files

Create a new folder in your PYNQ board and move both smul.bit and smul.hwh into it.

3.2) Python code

Create a new Jupyter notebook and run the following code to test your design:

```python
import time
from pynq import Overlay
import pynq.lib.dma
from pynq import Xlnk
import numpy as np
from pynq import MMIO
import random

ol = Overlay('/home/xilinx/jupyter_notebooks/smul/smul.bit')  # check your path
ol.download()  # it downloads your bit to FPGA
dma = ol.streamMul.smul_dma  # creating a dma instance. Note that we packed smul and...

sadd_ip = MMIO(0x43c00000, 0x10000)  # we got this IP from Address Editor
xlnk = Xlnk()

length = 11
in_buffer = xlnk.cma_array(shape=(length,), dtype=np.int32)  # input buffer
out_buffer = xlnk.cma_array(shape=(length,), dtype=np.int32)  # output buffer

samples = random.sample(range(0, length), length)
np.copyto(in_buffer, samples)  # copy samples to inout buffer
sadd_ip.write(0x10, length)  # we got this address from Vivado source

in_buffer.close()
out_buffer.close()

print('Hardware execution time: ', t_stop-t_start)
for i in range(0, length):
    print('i=i+2 = {}'.format(in_buffer[i], out_buffer[i]))
```

2.10. Lab: Axistream Single DMA (axis)
2.11 Lab: Axistream Multiple DMAs (axis)

2.11.1 Simple streaming example with multiple inputs

In this example we learn how to use Xilinx AXI_DMA to create a design with two streaming inputs and one streaming output.

1) Vivado HLS: Generating RTL code from C/C++ code

In this section you learn how to create a project in Vivado HLS, synthesize your code, and generate RTL.

1.1) Download code and create a Vivado HLS project

Download and unzip streamAdd.zip. Generate your project using the provided script.tcl file:

Linux: open a terminal, make sure your environment is set, navigate to streamMul folder, and run the following

```bash
$ vivado_hls script.tcl
```

Windows: open vivado_hls command line and run the following

```bash
$ vivado_hls script.tcl
```

Now you can open your project in Vivado HLS. It should look like this:
INPUT1, INPUT2 and OUTPUT ports are set to axis interfaces for streaming and length is set to s_axilite for a non-streaming interface. axis_t is a struct defined in the header file that is composed of an int data and an ap_uint<1> last. The 1-bit last is required for axis interfaces, and signals the last struct of the stream, ending the stream. In the pragmas, depth is set to 50 because that’s the maximum number of values we are streaming in and out of the fabric.

Note that

```c
*OUTPUT++ = cur1;
```

is performing two separate operations. Breaking it down:

```c
*OUTPUT = cur1;  // write the output struct to the address in OUTPUT
OUTPUT++;        // post-increment the address in OUTPUT for the next write operation
```

In this lab, since we are reusing an input struct cur1 to generate an output struct, the last bit is handled for us. However, if you must construct your own axis_t struct, you must ensure you set last to 1 when the struct is the last one to be streamed out, else explicitly set it to 0 (otherwise there may be garbage data in the memory address of last that terminates your stream early, leaving you scratching your head about why the output error on Pynq’s Jupyter interface is so high).
You can do so like this:

```c
axis_t curr;
curr.data = ...; // write data
curr.last = ...; // set to 1 if end of stream, else set to 0
*OUTPUT++ = curr; // make sure you only write to a particular address once, so do it after the struct is constructed
```

We must interact with them this way because we are dealing with an AXI stream, not an array.

### 1.2) Generate RTL code and export it

Click on Run C Synthesis to generate RTL code. After it is done, you can check your resource utilization and timing report. Your latency is unknown (?) because your loop size (length) is a variable.

Now you can export your RTL code by clicking on Export RTL:
After exporting is done, you can close and exit from Vivado HLS.

2) **Vivado: Generating bitstream from RTL code**

In this section we import our RTL code from last section, add some required IPs, and generate our bitstream
2.1) Create a new Vivado project

Open your Vivado tool and create a new project. Select an appropriate location for your project and leave the default project name as is (project_1).

Select RTL Project and check Do specify not sources at this time.

Select xc7z020clg400-1 for your part:

![New Project](image)

Default Part
Choose a default Xilinx part or board for your project. This can be changed later.

Select: Parts Boards

Filter

Product category: General Purpose
Speed grade: -1
Family: Zynq-7000
Temp grade: All Remain...
Package: clg400

Search: 

<table>
<thead>
<tr>
<th>Part</th>
<th>I/O Pin Count</th>
<th>Available IOBs</th>
<th>LUT Elements</th>
<th>FlipFlops</th>
<th>Block RAMs</th>
<th>Ultra RAMs</th>
<th>DSPs</th>
<th>Gb Transceivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>xc7z007sclg400-1</td>
<td>400</td>
<td>100</td>
<td>14400</td>
<td>28800</td>
<td>50</td>
<td>0</td>
<td>66</td>
<td>0</td>
</tr>
<tr>
<td>xc7z010clg400-1</td>
<td>400</td>
<td>100</td>
<td>17600</td>
<td>35200</td>
<td>60</td>
<td>0</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>xc7z014sclg400-1</td>
<td>400</td>
<td>125</td>
<td>40600</td>
<td>81200</td>
<td>107</td>
<td>0</td>
<td>170</td>
<td>0</td>
</tr>
<tr>
<td>xc7z020clg400-1</td>
<td>400</td>
<td>125</td>
<td>53200</td>
<td>106400</td>
<td>140</td>
<td>0</td>
<td>220</td>
<td>0</td>
</tr>
</tbody>
</table>
2.2) Import RTL code

Under Flow Navigator, click on IP Catalog. Right click on the opened window and select Add Repository. Navigate to your Vivado HLS project > solution1 > impl > ip and select it:

![File Structure]

2.3) Add IPs to your design

Under Flow Navigator, click on Create Block Design. Leave the design name as is (design_1). In the newly opened window you can add IPs by clicking on the plus sign.

Add ZYNQ7 Processing System to your design:

![Module Selector]

Designer Assistance available. Run Block Automation

processing_system7_0

ZYNQ7 Processing System
Double click on **ZYNQ7 IP** to customize it. In the opened window, double click on **High Performance AXI 32b/64b Slave Parts**:
2.11. Lab: Axistream Multiple DMAs (axis)
Select and check **S AXI HP0 interface** and **S AXI HP1 Interface**:

**ZYNQ7 Processing System (5.5)**

Add a `Sadd` to your design and rename it to `sadd`:
Add two **AXI Direct Memory Access** to your design and rename it to `sadd_dma1` and `sadd_dma2`.

Double click on your `sadd_dma1` and change the following parameters:
1) uncheck **Enable Scatter Gather Engine**.
2) Change **Width of Buffer Length Register** to 23:
Double click on sadd_dma2 and change the following parameters: 1) uncheck Enable Scatter Gather Engine. 2) Change Width of Buffer Length Register to 23. 3) uncheck Enable Write Channel.
The first DMA will be connected to one input port and one output port, but the second DMA only connects to one input port and that is why we disabled the write channel for the second DMA.

Add a **Constant** to your design

**2.4) Manual connections**

Connect the following ports:
xlconstant_0 to sadd::ap_ctrl::ap_start
sadd::OUTPUT_r to sadd_dma1::S_AXIS_S2MM
sadd_dma1::M_AXIS_MM2S to sadd::INPUT1
sadd_dma2::M_AXIS_MM2S to sadd::INPUT2
2.5) Automatic connections

Now you can leave the rest of the connections to the tool. There should be a highlighted strip on top of your diagram window.

1. Click on **Run Block Automation**
2. Click on **Run Connection Automation** and select all. Click on **S_AXI_HP1** and select **sadd_dma2/M_AXI_MM2S** as master:

3. **IMPORTANT!** you have to click again on **Run Connection Automation**
Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

At this point your design should look like this:
2.6) Create a Hierarchy

Select `sadd`, `sadd_dma1`, and `sadd_dma2`, right click on one of them, and select **Create Hierarchy**. Name it `streamAdd`. This will make our host code more organized. This step is optional, but it is good to know how to do. Note that, in the Jupyter notebook, we will have to access the hierarchy before accessing the DMA or the IP. You can see this in the Python code at the bottom of the page.
Your design should look like this:
2.7) Generate bitstream

1. Save your design CTRL+S or File > Save Block Design.
2. Validate your design: Tools > Validate Design
3. In Sources, right click on design_1, and Create HDL Wrapper. Now you should have design_1_wrapper.
4. Generate bitstream by clicking on Generate Bitstream in Flow Navigator
2.8) Note required addresses and copy generated files

After bitstream generating process is done, open Address Editor from window menu.

Note that **sadd address** is **0x43C00000**, we need this address in our host program for sending **length** data.

In sources, expand **design_1::design_1::streamAdd::sadd::design_1_sadd_0_0::inst : sadd**, double click on **sadd_CTRL_s_axi_U**, and note the address for **length_r** is **0x10**. We need this address in our host program.
Copy your **project directory > project_1 > project_1.runs > impl_1 > design_1_wrapper** to your **project directory > project_1** and rename it to **sadd.bit**.

Copy your **project directory > project_1 > project_1.srcs > sources_1 > bd > design_1 > hw_handoff > design_1.hwh** to your **project directory > project_1** and rename it to **sadd.hwh**.

You should have both **sadd.bit** and **sadd.hwh**.

You can close and exit from Vivado tool.

### 3) Host program

In this section we use Python to test our design.

#### 3.1) Move your files

Create a new folder in your PYNQ board and move both **sadd.bit** and **sadd.hwh** into it.

#### 3.2) Python code

Create a new Jupyter notebook and run the following code to test your design:

```python
import time
from pynq import Overlay
import pynq.lib.dma
from pynq import Xlnk
import numpy as np
from pynq import MMIO

ol = Overlay('/home/xilinx/jupyter_notebooks/sadd/sadd.bit')
# check this path
ol.download()  # this downloads your bitstream into FPGA
dma1 = ol.streamAdd.sadd_dma1  # first DMA. Note that we had to access the hierarchy
    # before accessing the DMA
dma2 = ol.streamAdd.sadd_dma2  # second DMA
sadd_ip = MMIO(0x43c00000, 0x10000)  # we got this address from
xlnk = Xlnk()

length = 8

in_buffer1 = xlnk.cma_array(shape=(length,), dtype=np.int32)  # input buffer 1
in_buffer2 = xlnk.cma_array(shape=(length,), dtype=np.int32)  # input buffer 2
out_buffer = xlnk.cma_array(shape=(length,), dtype=np.int32)  # output buffer

samples = random.sample(range(0, length), length)
np.copyto(in_buffer1, samples)
samples = random.sample(range(0, length), length)
np.copyto(in_buffer2, samples)
sadd_ip.write(0x10, length)  # we got this address from Vivado source. Since we didn’t
    # do port=return, and we set a constant for ap_start, we only have to write length.
t_start = time.time()
dma1.sendchannel.transfer(in_buffer1)
dma2.sendchannel.transfer(in_buffer2)
```

(continues on next page)
2.12 Lab: AXI4-Burst Mode (m_axi)

2.12.1 Simple example of AXI4-Burst Mode

This lab is an example of AXI4 data transfer in burst mode. It takes in a given sample of values and provides the square root.

1) Vivado HLS: Generating RTL code from C/C++ code

In this section you learn how to create a project in Vivado HLS, synthesis your code, and generate RTL.

1.1) Download code and create a Vivado HLS project

This is the code we will be using:

```c
#include "axi4_sqrt.hpp"
#include <string.h>
#include <math.h>

void axi4_sqrt(float *in, float *out, int len)
{
    #pragma HLS INTERFACE s_axilite port=return bundle=sqrt
    #pragma HLS INTERFACE s_axilite port=len bundle=sqrt
    #pragma HLS INTERFACE m_axi depth=50 port=out offset=slave bundle=output
    #pragma HLS INTERFACE m_axi depth=50 port=in offset=slave bundle=input
    #pragma HLS INTERFACE s_axilite port=in
    #pragma HLS INTERFACE s_axilite port=out

    float buff[100];
    memcpy(buff, (const float*) in, len * sizeof(float));

    for(int i = 0; i < len; i++)
        buff[i] = sqrt(buff[i]);

    memcpy(out, (const float*) buff, len * sizeof(float));
}
```

Note that we had to include `string.h` to be able to use `memcpy`. Additionally, we use `memcpy` instead of a for-loop (as used in AXI-streaming) to force Vivado HLS to infer an AXI4-Burst. Sometimes Vivado HLS will not infer this from a for-loop, but will for `memcpy`. 
Download and unzip `axi4_burst.zip` that contains the above code. Generate your project using the provided `script.tcl` file:

Linux: open a terminal, make sure your environment is set, navigate to `streamMul` folder, and run the following

```
$ vivado_hls -f script.tcl
```

Windows: open `vivado_hls` command line and run the following

```
$ vivado_hls -f script.tcl
```

Now you can open your project in Vivado HLS. It should look like this:
1.2) Generate RTL code and export it

Click on Run C Synthesis to generate RTL code. After it is done, you can check your resource utilization and timing report. Your latency is unknown (?) because your loop size (len) is a variable.

Now you can export your RTL code by clicking on Export RTL:
After exporting is done, you can close and exit from Vivado HLS.

2) Vivado: Generating bitstream from RTL code

In this section we import our RTL code from last section, add some required IPs, and generate our bitstream.
2.1) Create a new Vivado project

Open your Vivado tool and create a new project. Select an appropriate location for your project and leave the default project name as is (project_1).

Select RTL Project and check Do specify not sources at this time.

Select xc7z020clg400-1 for your part:
2.2) Import RTL code

Under **Flow Navigator**, click on **IP Catalog**. Right click on the opened window and select **Add Repository**. Navigate to your **Vivado HLS project > solution1 > impl > ip** and select it:

![IP Repositories]

2.3) Add IPs to your design

Under **Flow Navigator**, click on **Create Block Design**. Leave the design name as is (**design_1**). In the newly opened window you can add IPs by clicking on the plus sign.

Add **ZYNQ7 Processing System** to your design:
Double click on **ZYNQ7 IP** to customize it. In the opened window, double click on **High Performance AXI 32b/64b Slave Parts**.
Select and check **S AXI HP0 interface**:

**ZYNQ7 Processing System (5.5)**

<table>
<thead>
<tr>
<th>Page Navigator</th>
<th>PS-PL Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zynq Block Design</td>
<td>PS-PL Configuration</td>
</tr>
</tbody>
</table>

- **Peripheral I/O Pins**
- **MIO Configuration**
- **Clock Configuration**
- **DDR Configuration**
- **SMC Timing Calculation**
- **Interrupts**

<table>
<thead>
<tr>
<th>Name</th>
<th>Select</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AXI Non Secure Enablement</td>
<td>0</td>
<td>Enable AXI Non Secure Transaction</td>
</tr>
<tr>
<td>GP Slave AXI Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP Slave AXI Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S AXI HP0 interface</td>
<td>✔️</td>
<td>Enables AXI high performance</td>
</tr>
<tr>
<td>S AXI HP1 interface</td>
<td></td>
<td>Enables AXI high performance</td>
</tr>
<tr>
<td>S AXI HP2 interface</td>
<td></td>
<td>Enables AXI high performance</td>
</tr>
<tr>
<td>S AXI HP3 interface</td>
<td></td>
<td>Enables AXI high performance</td>
</tr>
<tr>
<td>ACP Slave AXI Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMA Controller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PS-PL Cross Trigger interface</td>
<td></td>
<td>Enables PL cross trigger signal</td>
</tr>
</tbody>
</table>

Add the `axi4_sqrt` IP to the design.
2.4) Automatic connections

1. Click on **Run Block Automation**
2. Click on **Run Connection Automation** and select all. Click OK.

3. Click on **Run Connection Automation** again and select all. Click OK.
Run Connection Automation

Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

This is how the final design should look
2.12. Lab: AXI4-Burst Mode (m_axi)
2.5) Generate bitstream

1. Save your design CTRL+S or File > Save Block Design
2. Validate your design: Tools > Validate Design
3. In Sources, right click on design_1, and Create HDL Wrapper. Now you should have design_1_wrapper
4. Generate bitstream by clicking on Generate Bitstream in Program and Debug

2.6) Post bitstream Generation

In sources, expand design_1_wrapper::design_1::design_1::axi4_sqrt_0::design_1_axi4_sqrt_0_0::inst : axi4_sqrt, double click on axi4_sqrt_sqt_s_axi_U, and note the address for in_r, out_r, len as 0x10, 0x18 and 0x20 respectively. We need this address in our host program.
2.12. Lab: AXI4-Burst Mode (m_axi)
You can close and exit from Vivado tool. Copy your project directory > project_1 > project_1.runs > impl_1 > design_1_wrapper.bit to your project directory > project_1 and rename it to axi4_sqrt.bit

Copy your project directory > project_1 > project_1.srcs > sources_1 > bd > design_1 > hw_handoff > design_1.hwh to your project directory > project_1 and rename it to axi4_sqrt.hwh

These files need to have the same name (except for their file extension).

3) Host program

In this section we use Python to test our design.

3.1) Move your files

Create a new folder in your PYNQ board and move both axi4_sqrt.hwh and axi4_sqrt.bit into it.

3.2) Python code

Create a new Jupyter notebook and run the following code to test your design:

```python
from pynq import Overlay
from pynq import Xlnk # replace with allocate for Pynq >= 2.7
import numpy as np

ol = Overlay('axi4_sqrt.bit')
sqrt_ip = ol.axi4_sqrt_0  # if you can't find the IP, type 'ol.' and hit Tab to see what options are available

length = 40
inpt = Xlnk().cma_array(shape=(length,), dtype=np.float32)
outpt = Xlnk().cma_array(shape=(length,), dtype=np.float32)
a = [i*i for i in range(length)]
np.copyto(inpt, a)
soft_op = np.sqrt(inpt)

sqrt_ip.write(0x20, length)
sqrt_ip.write(0x10, inpt.physical_address)
sqrt_ip.write(0x18, outpt.physical_address)
sqrt_ip.write(0x00, 1)

print("Hardware Output", "Software Output \n")
for i in range(length):
    print(outpt[i], "		", soft_op[i])
```

2.13 Lab: Interrupts

2.13.1 Interrupt Controller

In this example we implement $f(x) = x^2$ as an IP for PYNQ with interrupt controller.
1) Vivado HLS: C/C++ to RTL

In this section, you will write your code in C/C++ and convert it to RTL using Vivado HLS.

1.1) Create a project

Open the Vivado HLS tool, create a new project, and name it `pynq_fact`.

Set top function to `fact_intrpt`.
Select \texttt{xc7z020clg400-1} as your part number.
1.2) Write your code

In Explorer section, right click on Source, and select new file. Create a new file and name it fact_intrpt.cpp. Complete your code as following:

```cpp
void fact_intrpt(int* out, int in){
    int i, k=1;
    for(i=1; i<=in; i++){
        k = k*i;
    }
    *out = k;
}
```

Open Directive tab and set in, out, and return ports as s_axilite interfaces. Your code should look like this:
1.3) Synthesize your code and export RTL

You can synthesize your code by clicking on C Synthesis. After finishing C synthesis, click on Export RTL.
Now that you exported your RTL, you can close Vivado HLS.
2) Vivado: RTL to Bitstream

In this section we generate a bitstream for our IP.

2.1) Create a new project

Open Vivado tool and create a new project. Select an appropriate directory for your project and leave the project name as is, project_1.

Select RTL Project and check Do not specify sources at this time.

Set your part number to xc7z020clg400-1:
2.2) Import RTL

In Flow Navigator, under Project Manager, click on IP Catalog.

On the newly opened window, right click and select Add Repository.

Navigate to your Vivado HLS project directory and select solution1 > impl > ip.
2.3) Add blocks to your design

In Flow Navigator, under IP Integrator, click on Create Block Design. Leave the name as is, design_1.

You can add IPs to your design by clicking on the + sign. Add the following IPs:

Add ZYNQ7 Processing System to your design:
Double click on ZYNQ7 IP to customize it. Under page navigator, go to interrupts, and enable IRQ_F2P[15:0]
Add `fact_intrpt` and rename it to `fact_intrpt`:
Add AXI Interrupt Controller and rename it to `axi_intc`:
Double click on \texttt{axi\_intc} to customize it. Set \textbf{Interrupt Output Connection} to \textbf{Single}.
Add two Concat IPs to your design, double click on them for customization, and set Number of Ports to 1:
2.4) Connect your blocks

Connect the following ports:
fact_intrpt::interrupt to xlconcat_0::in0
xlconcat_0::dout to axi_intc::intr[0:0]
axi_intc::irq to xlconcat_1::in0

That is all the manual labor you have to do for the connections. Now, you can use the tool for the rest of the blocks and connections.

In the Diagram window, in the highlighted area, click on Run Block Automation. Then click on Run Connection Automation and select all:
Run Connection Automation

Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface pin on the left to display its configuration options on the right.

Your design should look like the following:
2.5) Generating bitstream

Save your block diagram and check your design from Tools > Validate Design. If it passed successfully, under Sources, right click on design_1, and select Create HDL Wrapper with default settings.

Now you can click on Generate Bitstream in Flow Navigator with default settings to generate your bitstream.

2.13. Lab: Interrupts
2.6) Export .tcl file and note addresses

Exporting the block design is an optional step. Pynq seems to prefer .hwh over .tcl. After finishing your bitstream
generation, you can export your block design from File > Export > Export Block Design, and name it fact_intrpt.tcl:

Copy your_vivado_project_directory > project_1.runs > impl_1 > design_1_wrapper.bit to your_vivado_project_directory > fact_intrpt.bit next to fact_intrpt.tcl.

Copy your project directory > project_1 > project_1.srcs > sources_1 > bd > design_1 > hw_handoff > design_1.hwh to your project directory > project_1 and rename it fact_intrpt.hwh.

In Sources, open fact_intrpt_cntrl_io_s_axi.v and note the port addresses. We need these addresses in our Python
code.
3) Host program

In this section we use a Jupyter notebook to interact with our IP.

3.1) Move files to PYNQ

Connect to your PYNQ board and create a new folder. Copy fact_intrpt.hwh and fact_intrpt.bit to this folder like we have done in previous labs. Create a new Jupyter notebook and complete it as following to interact with your design:
from pynq import Overlay
import asyncio
from psutil import cpu_percent

ol = Overlay("fact_intrpt.bit")
ol.download()

# IP's addresses
IP_CTRL = 0x00
AP_START = 0x1
GIER = 0x04
IP_IER = 0x08
IP_ISR = 0x0C
INTRPT_AP_DONE = 0x1
INTRPT_AP_READY = 0x2
OUT_REG = 0x10
INP_REG = 0x18
_INTRPT = INTRPT_AP_DONE

fact_ip = ol.fact_intrpt
fact_ip.write(GIER, 0x1)
fact_ip.write(IP_IER, _INTRPT)

# Coroutine that waits for an IP to be done.
async def read_ip(ip):
    while True:
        # Wait for the IP to finish.
        await ip.interrupt.wait()
        # Clear the interrupt and then print output's value.
        if (ip.read(IP_ISR) & _INTRPT):
            ip.write(IP_ISR, _INTRPT)
            print('interrupt received, out = {}
'.format(ip.read(OUT_REG)))

    # Task for IP using the coroutine
ip_task = asyncio.ensure_future(read_ip(fact_ip))

# Coroutine for writing input and starting the IP with delay
async def write_wait(interval):
    await asyncio.sleep(interval)
    # write to input
    fact_ip.write(INP_REG, 10)
    print("input = ", fact_ip.read(INP_REG))
    fact_ip.write(IP_CTRL, AP_START)  # You can comment it out to test the interrupt
    print("IP started")
    await asyncio.sleep(interval)

    # Run the event loop until the time interval expires
    time_interval = 2  # time in seconds
    loop = asyncio.get_event_loop()
    write_task = asyncio.ensure_future(write_wait(time_interval))

    # Using psutil to record CPU utilization.
    cpu_percent(percpu=True)  # Initializing the CPU monitoring.
    loop.run_until_complete(write_task)
    cpu_used = cpu_percent(percpu=True)

(continues on next page)
# Printing the CPU utilization
print('CPU Utilization = {cpu_used}'.format(**locals()))

# Removing the IP task from the event loop.
ip_task.cancel()

## 2.14 Lab: MicroBlaze

Let’s take a look inside the BaseOverlay class which corresponds to the Overlay below:

![Zynq PS and PL diagram](image)

Let’s load the base overlay and check the IO Processors: iop_arduino, iop_pmoda, and iop_pmodb

```
[21]: import pynq
    from pynq.overlays.base import BaseOverlay
    ol = BaseOverlay("base.bit")
```

```
[22]: ol.ip_dict
[22]: {'audio_direct_0': {'addr_range': 65536, 'driver': pynq.lib.audio.AudioDirect,

(continues on next page)
'fullpath': 'audio_direct_0',
gpio': {'sel_direct': {'index': 3,
  'pins': {'audio_direct_0/sel_direct', 'audio_path_sel/Dout'},
  'state': None},
'interrupts': {},
'phys_addr': 1136656384,
'state': None,
'type': 'xilinx.com:user:audio_direct:1.1'},
'btns_gpio': {'addr_range': 65536,
'driver': pynq.lib.axigpio.AxiGPIO,
'fullpath': 'btns_gpio',
gpio': {},
'interrupts': {'ip2intc_irpt': {'controller': 'system_interrupts',
  'fullpath': 'btns_gpio/ip2intc_irpt',
  'index': 11}},
'phys_addr': 1092681728,
'state': None,
'type': 'xilinx.com:ip:axi_gpio:2.0'},
iop_arduino/mb_bram_ctrl': {'addr_range': 65536,
'fullpath': 'iop_arduino/mb_bram_ctrl',
gpio': {},
'interrupts': {},
'phys_addr': 1140850688,
'state': None,
'type': 'xilinx.com:ip:axi_bram_ctrl:4.0'},
iop_pmoda/mb_bram_ctrl': {'addr_range': 65536,
'fullpath': 'iop_pmoda/mb_bram_ctrl',
gpio': {},
'interrupts': {},
'phys_addr': 1073741824,
'state': None,
'type': 'xilinx.com:ip:axi_bram_ctrl:4.0'},
iop_pmodb/mb_bram_ctrl': {'addr_range': 65536,
'fullpath': 'iop_pmodb/mb_bram_ctrl',
gpio': {},
'interrupts': {},
'phys_addr': 1107296256,
'state': None,
'type': 'xilinx.com:ip:axi_bram_ctrl:4.0'},
leds_gpio': {'addr_range': 65536,
'driver': pynq.lib.axigpio.AxiGPIO,
'fullpath': 'leds_gpio',
gpio': {},
'interrupts': {},
'phys_addr': 1092943872,
'state': None,
'type': 'xilinx.com:ip:axi_gpio:2.0'},
rgbleds_gpio': {'addr_range': 65536,
'driver': pynq.lib.axigpio.AxiGPIO,
'fullpath': 'rgbleds_gpio',
gpio': {},
'interrupts': {},
'phys_addr': 1092878336,
'state': None,
'type': 'xilinx.com:ip:axi_gpio:2.0'},
switches_gpio': {'addr_range': 65536,
'driver': pynq.lib.axigpio.AxiGPIO,
'phys_addr': 1124073472,
'state': None,
'type': 'xilinx.com:ip:axi_vdma:6.3'},
'video/hdmi_in/color_convert': {'addr_range': 65536,
'fullpath': 'video/hdmi_in/color_convert',
'gpio': {},
'interrupts': {},
'phys_addr': 1136984064,
'state': None,
'type': 'xilinx.com:hls:color_convert:1.0'},
'video/hdmi_in/frontend/axi_gpio_hdmiin': {'addr_range': 65536,
'fullpath': 'video/hdmi_in/frontend/axi_gpio_hdmiin',
'gpio': {},
'interrupts': {'ip2intc_irpt': {'controller': 'system_interrupts',
'fullpath': 'video/hdmi_in/frontend/axi_gpio_hdmiin/ip2intc_irpt',
'index': 4}},
'phys_addr': 1092747264,
'state': None,
'type': 'xilinx.com:ip:axi_gpio:2.0'},
'video/hdmi_in/frontend/vtc_in': {'addr_range': 65536,
'fullpath': 'video/hdmi_in/frontend/vtc_in',
'gpio': {},
'interrupts': {'irq': {'controller': 'system_interrupts',
'fullpath': 'video/hdmi_in/frontend/vtc_in/irq',
'index': 3}},
'phys_addr': 1136852992,
'state': None,
'type': 'xilinx.com:ip:v_tc:6.1'},
'video/hdmi_in/pixel_pack': {'addr_range': 65536,
'fullpath': 'video/hdmi_in/pixel_pack',
'gpio': {},
'interrupts': {},
'phys_addr': 1136918528,
'state': None,
'type': 'xilinx.com:hls:pixel_pack:1.0'},
'video/hdmi_out/color_convert': {'addr_range': 65536,
'fullpath': 'video/hdmi_out/color_convert',
'gpio': {},
'interrupts': {},
'phys_addr': 1137049600,
'state': None,
'type': 'xilinx.com:hls:color_convert:1.0'},
'video/hdmi_out/frontend/axi_dynclk': {'addr_range': 65536,
'fullpath': 'video/hdmi_out/frontend/axi_dynclk',
'gpio': {},
'interrupts': {},
'phys_addr': 1136721920,
'state': None,
'type': 'digilentinc.com:ip:axi_dynclk:1.0'},
'video/hdmi_out/frontend/hdmi_out_hpd_video': {'addr_range': 65536,
'fullpath': 'video/hdmi_out/frontend/hdmi_out_hpd_video',
'gpio': {},
'interrupts': {'ip2intc_irpt': {'controller': 'system_interrupts',
'fullpath': 'video/hdmi_out/frontend/hdmi_out_hpd_video/ip2intc_irpt',
'index': 5}},
'phys_addr': 1092812800,
'state': None,
Some PMODs already have drivers: list of plug and play PMODs

```python
from pynq.lib import Pmod_OLED
pmod_oled = Pmod_OLED(ol.PMODA)
```

```python
pmod_oled.clear()
pmod_oled.write('Welcome to 

PYNQ!')
```

Let’s take a look at it’s code on pynq github

### 2.14.1 Hello World on Microblaze

IO processors can also be used for running small applications.

**IPython magic** allow the execution of Non-Python code in a Python cell.

```python
from pynq.lib import MicroblazeLibrary
```

```python
%%microblaze ol.PMODA
#include <pyprintf.h>

int mb_print(){
    pyprintf("Hello World!");
    return 0;
}
```

```python
mb_print()
```

### 2.14.2 Task 1

Here you learn how to use provided python libraries.

Using the source code for RGBLED write a for loop to change the coloring of your rgb leds between red, blue, and white similar to a police light bar. You can use `time.sleep` as delay between flashes.
2.14.3 Task 2

Write a microblaze code to calculate factorial of an input on PL. Your result should be same as the python code output.

```python
int mb_fact(int in){
    //write code here
}
```

```python
inp = 10
print('out = {}'.format(mb_fact(inp))) # your function is called here, note the function prototype
```

```python
ff=1
for i in range(1,inp+1):
    ff = ff*i
print('ans = {}'.format(ff))
```

2.14.4 Task 3

Use the source code for OLED_PMOD to learn how to use the GPIO pins.

Write two microblaze functions: `mba_send(v)` and `mbb_recv()` to send 1-bit across the two PMODs A and B.

You need to connect pin 0 of the two PMODs using a jumper wire.

```python
#include <gpio.h>
int mba_send(int v){
    //write code here
}
```

```python
#include <gpio.h>
int mbb_recv(){
    //write code here
}
```

```python
mba_send(1)
print(mbb_recv())
1
```
2.14.5 Task 4

Read voltages from the Pmod_AD2. Since there are only 2 Pmod_AD2. Write the code to the best of your ability and ask one of the TA’s to check before testing on the Pmod_AD2.

- The first part of this task is to only use the pynq python libraries. You can reference the API here. Remember to close the pmod instance.

```python
# write code here
```

- For the second part of the task, we’ll see how to communicate with the Pmod_AD2 by programming the microblaze in C. While this is more work, most of the PMODS we have available do not have python libraries pre-built so you’ll need to search through the documentation and program the microblaze in order to use them.

microblaze libraries for pynq

Pmod_AD2 reference manual

```c
#include <i2c.h>
#include <pyprintf.h>

//TODO: Find the pmod_ad2 address value
#define AD2IICAddr 0x9A

//Configuration
#define CH3 7
#define CH2 6
#define CH1 5
#define CH0 4
#define REF_SEL 3
#define FLTR 2
#define BitTrialDelay 1
#define SampleDelay 0
#define BitMask 0xFFF

float read_i2c() {
    //TODO: open a new i2c device
    unsigned char WriteBuffer[1];
    unsigned char cfgValue = (1 << CH3) |
                           (1 << CH2)  |
                           (1 << CH1)  |
                           (1 << CH0)  |
                           (0 << REF_SEL) |
                           (0 << FLTR)  |
                           (0 << BitTrialDelay) |
                           (0 << SampleDelay);
    WriteBuffer[0]=cfgValue;
    //TODO: write the configuration to the pmod (1 byte)
    //Receiving data.
    unsigned char rcvbuffer[2];
    int rxData;
    //TODO: read from the pmod and format the raw data (2 bytes)
    // The first byte is MSB, while the second byte is LSB
```

(continues on next page)
rxData = //;

// Format as a voltage
int raw = (rxData & BitMask);
return (float)(raw * 2.00 / 4096.0); // 2.0 V is the reference voltage for the
→ AD2 Pmod.
}

2.15 Lab: DPC++ on Intel DevCloud

2.15.1 Getting Started

To complete this lab, you need access to Intel DevCloud. Log in to DevCloud and navigate to Get Started.

DevCloud provides several ways to access the software and hardware. We recommend setting up both JupyterLab and SSH since each has benefits:

• **JupyterLab** provides visual interfaces to navigate through your files and edit your code.

• **SSH** gives a simple command-line interface to the resources. You also can use SSHFS and SCP to access and transfer your files to/from the cloud. We strongly recommend Linux for this lab.
We suggest reviewing modules 1, 2, and 3 of *Introduction to oneAPI* and *Essentials of Data-Parallel C++*, available on the *Get Started* page.

### 2.15.2 Resources

oneAPI and DPC++ are open source and there are many resources available online to help you start your project:

- oneAPI
- Developer Reference
- DPC++ book (chapters 1 - 4)
- oneAPI Programming Guide
- oneAPI FPGA Optimization Guide
- Example Codes
- SYCL Specifications

### 1) Instructions

This section walks through some example matrix multiplication code to illustrate how to use dpc++ and the DevCloud environment. This requires access to JupyterLab (on the cloud) and a SSH connection to your machine.
1.1) Create a Project in JupyterLab

- Start a JupyterLab session.
- Click the + sign to open the Launcher.

- Use oneapi-cli to create the baseline matrix multiplication project.
- Open a Terminal and launch the tool:
$ oneapi-cli

- This tool has a basic interface that allows you to choose a project based on its various templates. Perform the following:
  1) Select Create a project
  2) Select cpp
  3) Select Matrix Multiplication under Get Started
  4) Leave Directory and Project Name with their default parameters and select Create

This creates several files inside the matrix_mul folder:

The source file is in the src folder and the README provides instructions to compile the code.

- Use the terminal to compile and run your code. Be sure to use the Jupyter terminal; SSH doesn’t seem to work for this operation:

  $ cd matrix_mul
  $ make all
  $ make run

The output should look like this:

$ ./matrix_mul_dpc
Device: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
Problem size: c(150,600) = a(150,300) * b(300,600)
Result of matrix multiplication using DPC++: Success - The results are correct!

It shows the device name used for the matrix multiplication, the matrices’ size, and the testbench result.
1.2) Modify the Project for FPGA

Open the file `src/matrix_mul_dpcpp.cpp`.

Line 55 uses the `default_selector` for the hardware device. See Section 4.3.8 of the Programming Guide for more details.

Modify the code to select a FPGA device. This can be accomplished in two steps:

- Include a header file by adding the line:
  ```cpp
  #include "CL/sycl/INTEL/fpga_extensions.hpp"
  ```

- Modify the queue initialization:
  ```cpp
  INTEL::fpga_emulator_selector _device;
  queue q(_device, dpc_common::exception_handler);
  ```

Re-compile and re-run the code. The device should now be an FPGA emulator.

```bash
$ ./matrix_mul_dpc
Device: Intel(R) FPGA Emulation Device
Problem size: c(150,600) = a(150,300) * b(300,600)
Result of matrix multiplication using DPC++: Success - The results are correct!
```

1.3) Generate the FPGA Report

Once your code is functionally correct, synthesize it to check your design’s performance and resource utilization.

Perform high level synthesis and create the report using `dpcpp`:

```bash
$ cd src
$ dpcpp -fintelfpga matrix_mul_dpcpp.cpp -c -o matrix_mul_dpcpp.o
$ dpcpp -fintelfpga matrix_mul_dpcpp.o -fsycl-link -Xshardware
```

These commands do not perform a full synthesis, i.e., they do not create an FPGA bitstream. They perform high level synthesis and simulate the FPGA design on a processor. This enables estimations of the design performance and resource usage. To generate a bitstream, remove `-fsycl-link`. Note that generating a bitstream can take a long time and thus should only be done sparingly, i.e., when the design optimizations are finalized.

The report is an HTML file in `matrix_mul_dpcpp.prj/reports`. At the time of writing this tutorial, JupyterLab has issues with some combinations of operating systems and web browsers. If the reports are not rendering, make sure to “Trust” the HTML. Safari seems to have the best compatibility with this, with Firefox coming second, but neither is guaranteed to work. If they still do not render, use SCP or SSHFS to copy or mount your files (including the `lib` folder from which the report draws its components) to your local drive, then open the report locally.

The report provides an overview of the design performance. Open `report.html`, click Summary button, click Compile Estimated Kernel Resource Utilization Summary in the Summary Content pane.

This shows the resource usages for the different kernels and other system resources.
The kernel names look almost random as these compiler generated. To make the report readable modify the code to name to the kernels:

Add three classes in your code as follows:

```cpp
class a_init;
class b_init;
class c_calc;
```

Now you can assign each class to one of your kernels. For example, `a_init` will be assigned to the kernel that initializes buffer `a` as follows:

```cpp
h.template parallel_for<a_init>(range(M, N), [=](auto index) {
```

Change the other kernels in a similar manner. Repeat the steps to generate a new report. Do not forget to remove the old files before generating a new report:

```
$ rm -rf matrix_mul_dpcpp.prj/ matrix_mul_dpcpp.o matrix_mul_dpcpp.a
$ dpcpp -fintelfpga matrix_mul_dpcpp.cpp -c -o matrix_mul_dpcpp.o
$ dpcpp -fintelfpga matrix_mul_dpcpp.o -fsycl-link -Xhardware
```

This will make your report more readable:
2) Analyzing the Report

Let’s take a deeper look at the report. Under System Viewer, open Graph Viewer. On the open page, select System under Graph List.

This graph shows that your code has three kernels: a_init and b_init are simple kernels that store values into the global memory. c_calc has two blocks: B2 calculates the matrix multiply, and B1 stores the results in the global memory.

Now let’s explain the red arrows. Click on the red circle titled LD (x2) inside c_calc.B2. These two LD operations correspond to line 126 in the source code. They have a latency of 223 cycles to read from DDR memory, which is the bottleneck for the loop iteration.
Check the results in *Loop Analysis*: under *Throughput Analysis*, select *Loop Analysis* and click on `c_calc.B2`:

The *Schedule Viewer* under *System Viewer* provides detailed information for each step of the process in the kernels:
Details for the load operation LD is available by clicking on their yellow bar.
The Load Store Unit (LSU) Style for the LD operations is shown as Burst-coalesced cached. Intel oneAPI DPC++ compiler generates different types of LSUs to manage data transfer between device and host. The compiler uses the Burst/coalesced LSU by default. In Burst/coalesced mode, the compiler optimizes for buffering contiguous memory requests for the largest possible burst. We can change the LSU type to achieve a lower latency for the load operations. More details are available at Intel® oneAPI DPC++ FPGA Optimization Guide.

<table>
<thead>
<tr>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>Width</td>
</tr>
<tr>
<td>LSU Style</td>
</tr>
<tr>
<td>Stall-free</td>
</tr>
<tr>
<td>Global Memory</td>
</tr>
<tr>
<td>Start Cycle</td>
</tr>
<tr>
<td>Latency</td>
</tr>
</tbody>
</table>