

CSE125 / Spring 22

Lab 5: Is This Prime?

Due Date: 2022-06-02

Version 2021-05-12

Introduction

In this lab exercise, you will build a prime number checker module (**is_prime**) that takes an input number and checks if it is prime. A module (**modulo_is_zero**) that checks if the modulo of two numbers is zero is given, and you may use this circuit to build your prime checker. Your design needs to receive and transmit data with the valid-ready handshake protocol. After passing the testbench, build the **prime_checker** IP with **is_prime** and integrate the IP with the ARM processor core. Finally, run the test application on the development board to verify the functionality of the **prime_checker** IP.

Getting Started

Pull the latest commit on the master branch from the [Labs](#) repository (or directly download it from the web):

```
git remote add template git@https://git.ucsc.edu/cse125/spring22/labs.git
git pull template master
```

In **lab5/**, You should see the following folders:

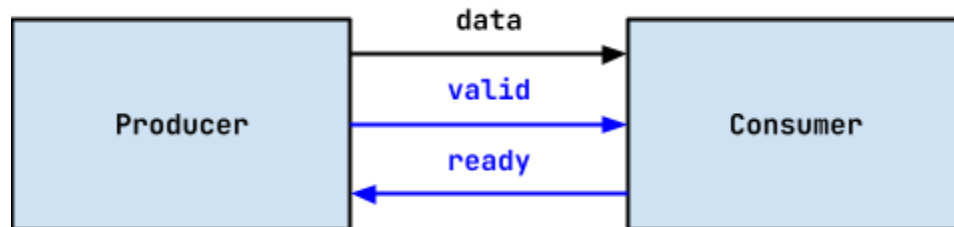
- **pynq-z2/**
 - Board files for PYNQ-Z2
- **skeleton/**
 - Skeleton code of **is_prime** (**modulo.v** and **prime.v**)
- **unit_test/**
 - Testbench for **is_prime** implementation (**prime_tb.v** and **prime_test.txt**)
- **prime_checker/**
 - Vivado IP package
- **sdk_code/**
 - C code to run on the development board

Note that you will need to come to BE-162 to finish the second half of the lab with the board. Try to complete the first part early so that you have enough time to test it on the board.

Part 1 → is_prime

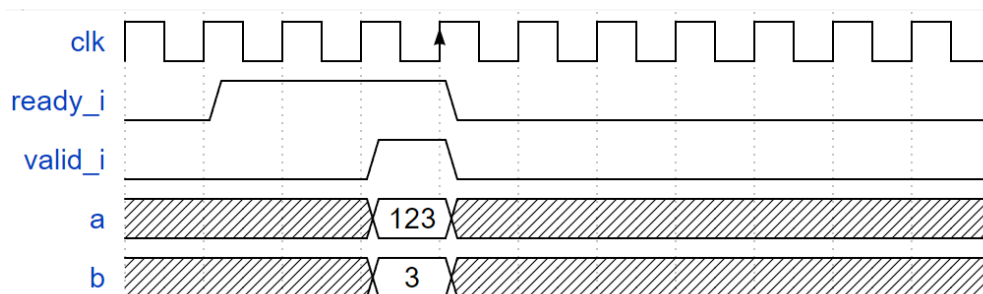
In this part, you will implement **is_prime** based on **modulo_is_zero**.

First, both modules use the valid-ready handshake protocol, which is described below:

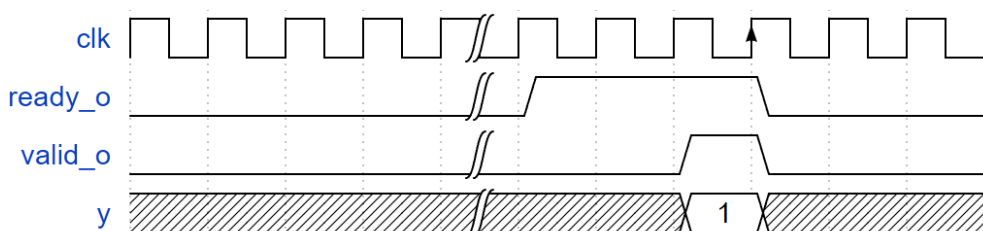


- The producer asserts the **valid** signal when the data is valid.
- The consumer asserts the **ready** signal when it is ready to read data from the producer.
- The transaction occurs when **valid** and **ready** are both asserted at the same clock edge.

For example, **modulo_is_zero** takes **a** and **b** from the input port when itself asserts **ready_i** and the producer asserts **valid_i**, indicating that input **a** and **b** are valid.

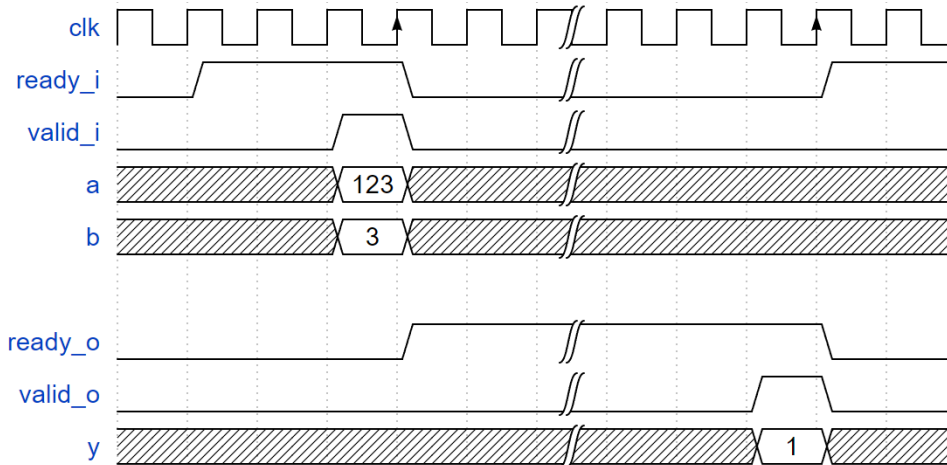


Similarly, the result of **modulo_is_zero** is returned to the upper module through another handshake with **ready_o** and **valid_o**.



In this case, **modulo_is_zero** acts as the producer, and hence it asserts **valid_o** when the result **y** has been generated.

The following is the waveform diagram of a request to **modulo_is_zero** and a reply to that request. **modulo_is_zero** only takes one request at a time, so **ready_i** is only asserted after the active request has been processed and the output transaction (**ready_o** & **valid_o**) is finished.





Once you understand the handshake protocol, you can start building `is_prime`. Here are some tips and hints that might help you design the module:

- Write down the pseudo-code for the prime checking algorithm, i.e., what you would do in a software program. For example:

```
is_prime(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    for i = 2 to n-1:
        if (modulo_is_zero(n, i)):
            return 0
    return 1
```

The first two if statements deal with special cases like 1 and 2. The for loop iterates through all possible factors of input `n` and returns `0` when `n` has a factor. Otherwise, `n` is a prime, and the function returns `1`. You may start with a simple algorithm like this and refine it after the first implementation has passed the testbench.

- Build an FSM where states represent which part of the pseudo-code the module is executing. For example, you may start with an IDLE state and a PRECHECK state that checks if the input is a special case. For iterating the loop, you may have a START state to send a request to `modulo_is_zero`, and a WAIT state to wait for the completion of the computation. Lastly, you may need a DONE state that asserts `valid_o` and waits for the output handshake.
- Note that the design should be parameterized by the **WIDTH** of the input number.
- Add `prime_tb.v` and `prime_test.txt` as simulation sources to your Vivado project

- Run **prime_tb** to check your design. This testbench tests **every 16-bit prime number** and will print all the error cases to the console. You may start by simulating the module by a fixed time step . Once you are confident with your code, hit  to run the whole test set. (This may take some time. In the meantime, try to improve the runtime by reducing the loop iterations in the algorithm or using multiple **modulo_is_zero** to increase parallelism.)

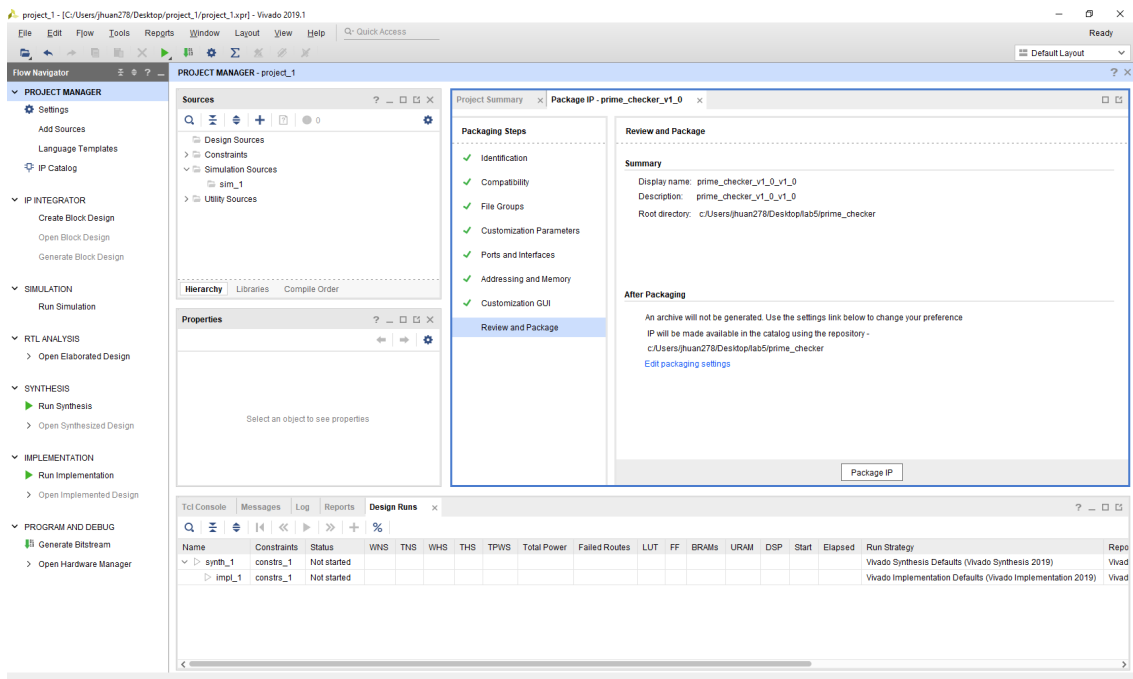
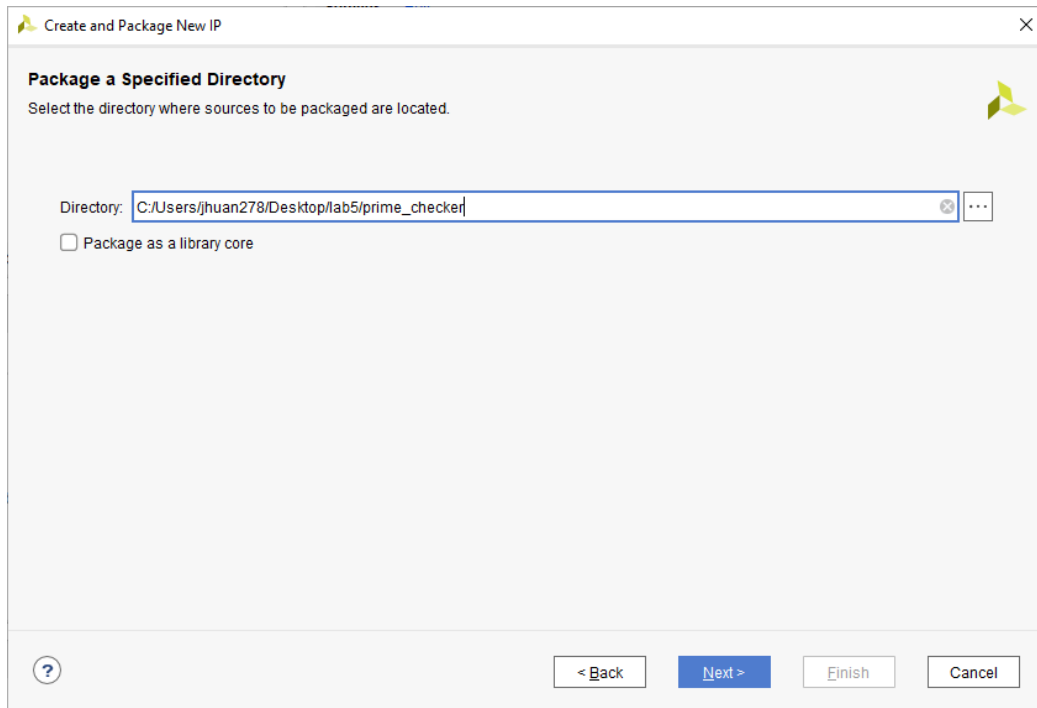
Part 2 → prime_checker IP

In this part, you will integrate your **is_prime** module into the **prime_checker** IP. This IP interfaces with the AXI4 bus protocol (AXI4-lite), which is widely used in ARM systems for communications between IP blocks and CPUs. The skeleton of the IP is provided. You only need to follow the steps below to complete this part.

1. Use the 2019.1 version of Vivado (2021 version does not work)

If you cannot find a link to Vivado 2019.1 on Desktop, run
C:\Xilinx\Vivado\2019.1\bin\vivado

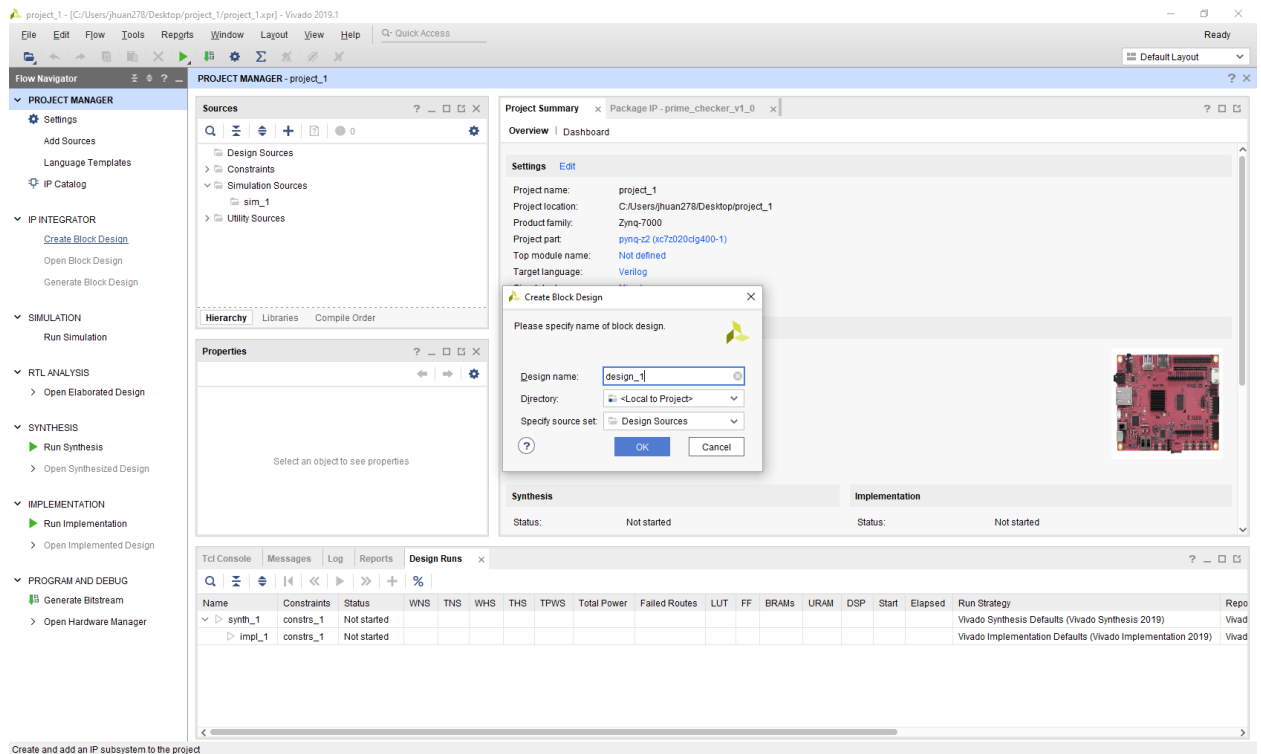
2. Copy **pynq-z2/** into .../Xilinx/Vivado/2019.1/data/boards/board_files/
3. Copy your **prime.v** to **prime_checker/src/prime.v** (overwrite it)
4. Open Vivado > Create Project > Project name: **lab5** > RTL Project > Select **pynq-z2**
5. Top Bar > Tools > Create and Package New IP... > Next > Package a specified directory > (Select the path to **prime_checker/**) > Next > Open > Package IP
6. Review and Package > Re-Package IP



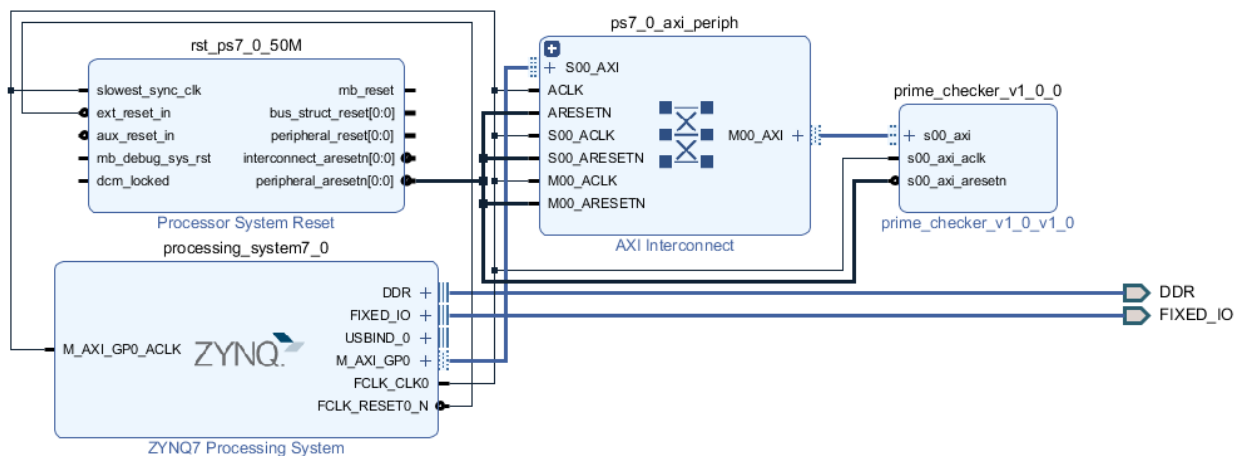
Part 3 → Complete the System!

In this part, you will build the full system on PYNQ that takes user input from a terminal, passes the number to the **prime_checker** IP, and returns the result.

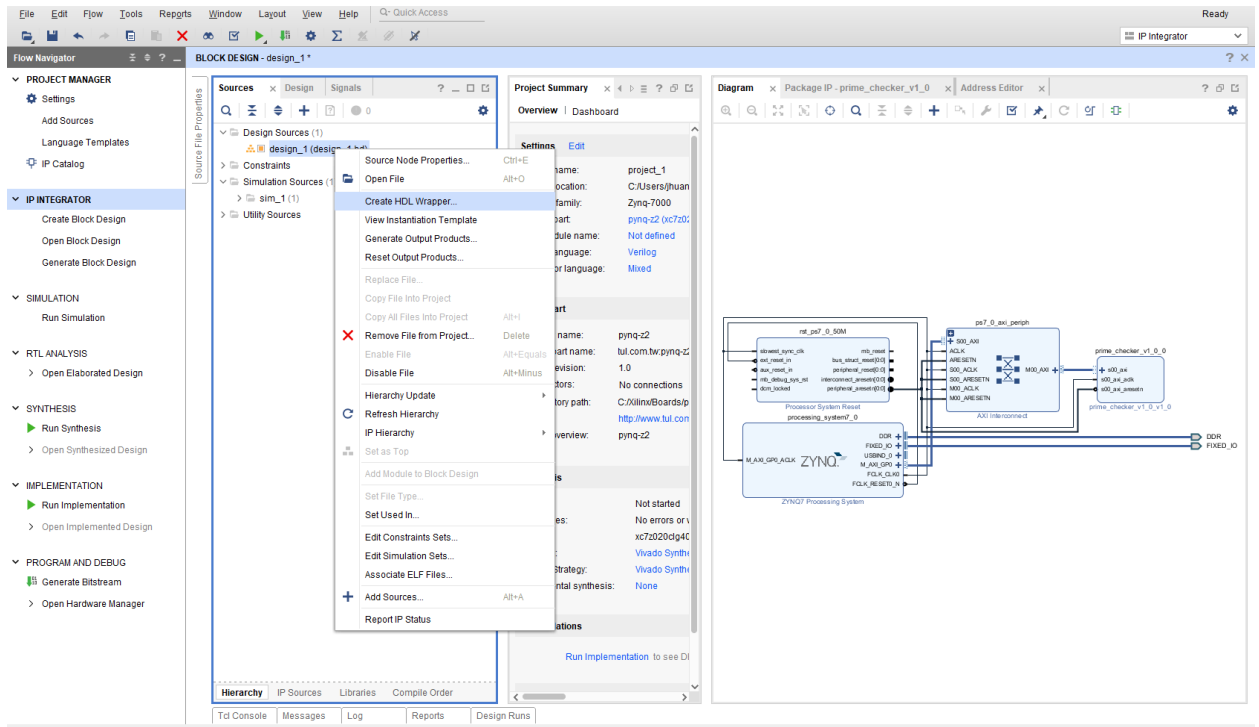
1. Flow Navigator > IP INTEGRATOR > Create Block Design > OK



2. Add “ZYNQ7 Processing System” and “prime_checker” to the block diagram
3. Run Block Automation and Connection Automation (Use the default values)



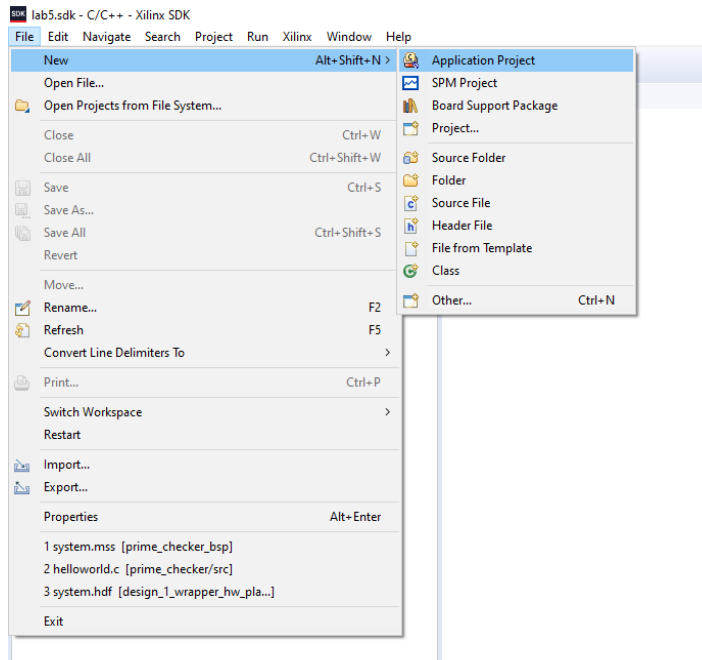
4. Sources > Right click your block design > Create HDL Wrapper... > OK



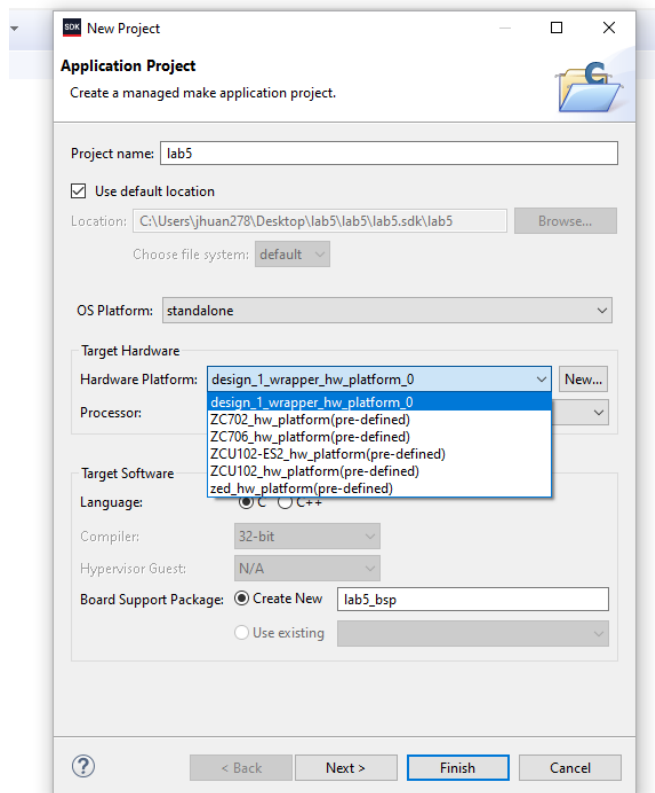
5. Generate Bitstream > Yes > OK
6. Top Bar > File > Export > Export Hardware... > Select "Include bitstream" > OK
7. Top Bar > File > Launch SDK > OK

Part 4 → SDK

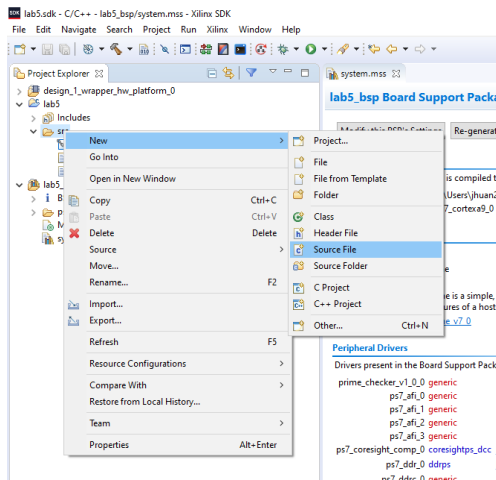
1. Top Bar > New > Application Projects...



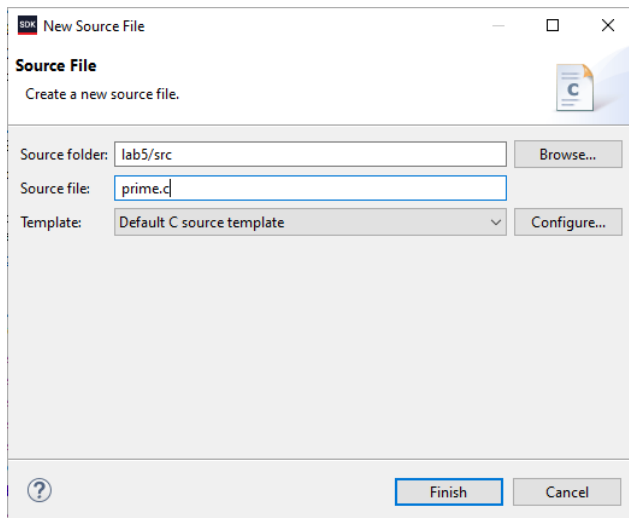
2. Select the exported hardware design as the hardware platform



- Next > Select “Empty Application” > Finish
- Right-click on **src** of the created application > New > Source File




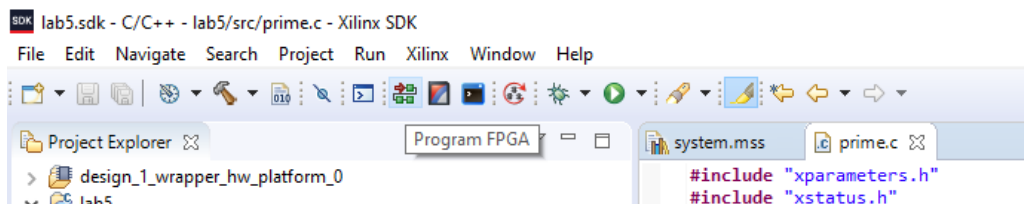
- Create the source code file



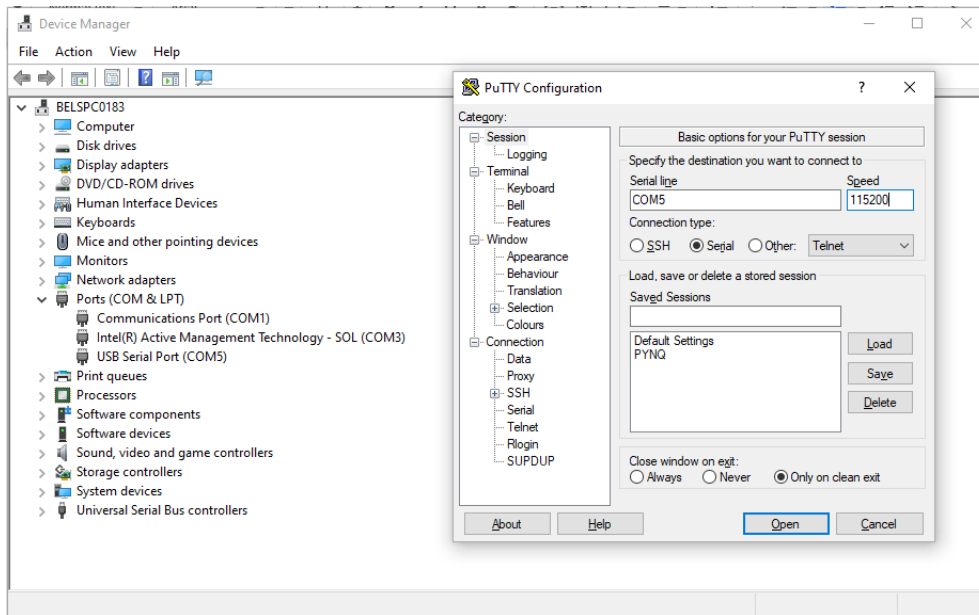
- Copy the code in sdk_code/prime.c to the source code file.
- Save the file.

(You need a connected, switched-on dev board in the following steps)

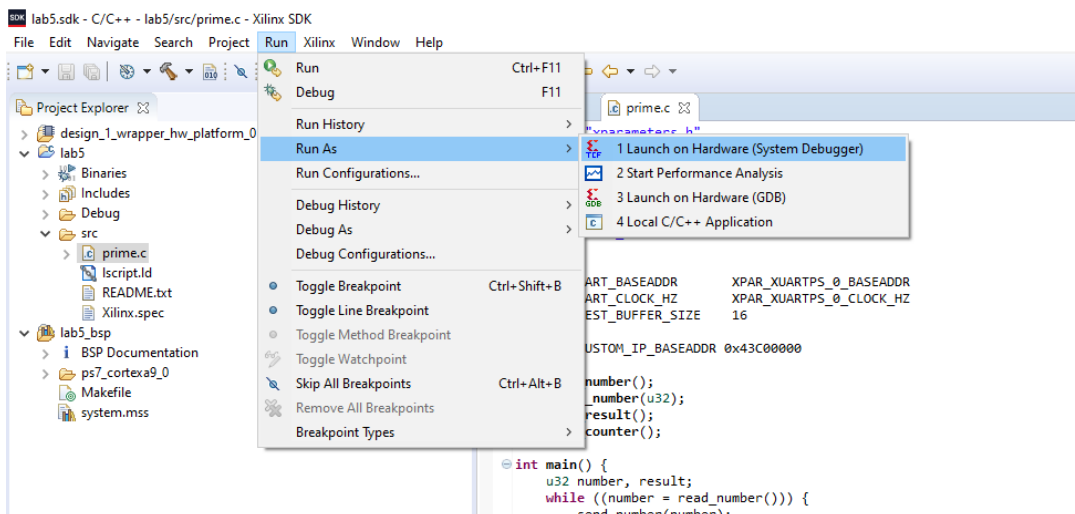
- Program the FPGA by clicking  > Program



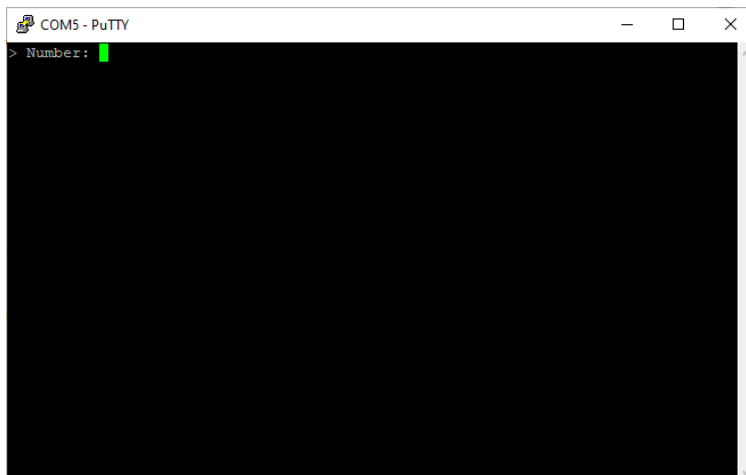
- Open PuTTY and Device Manager. Find the name of the USB port connected to the board (**COM5** in the example). Select Serial as the connection type. Enter the name and the baud rate **115200**. Click Open, and a terminal would pop up.



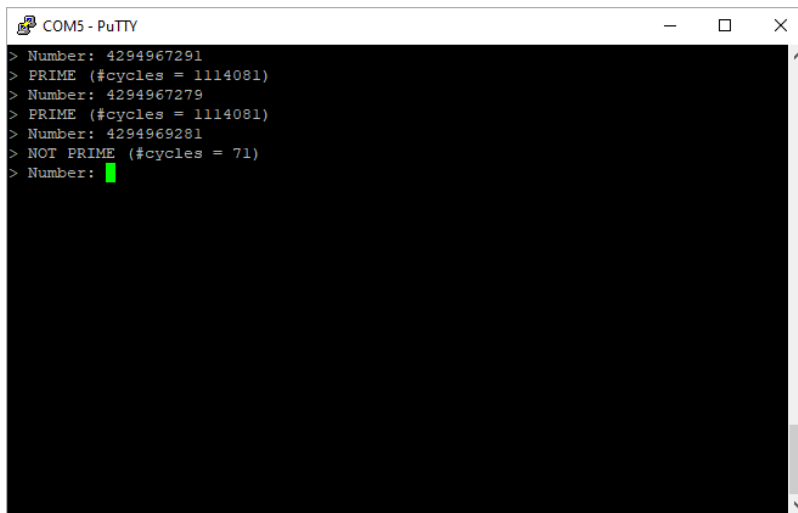
10. Run > Run As > Launch on Hardware (System Debugger).



11. You should see a prompt message in the terminal you just opened.



12. Run some tests!

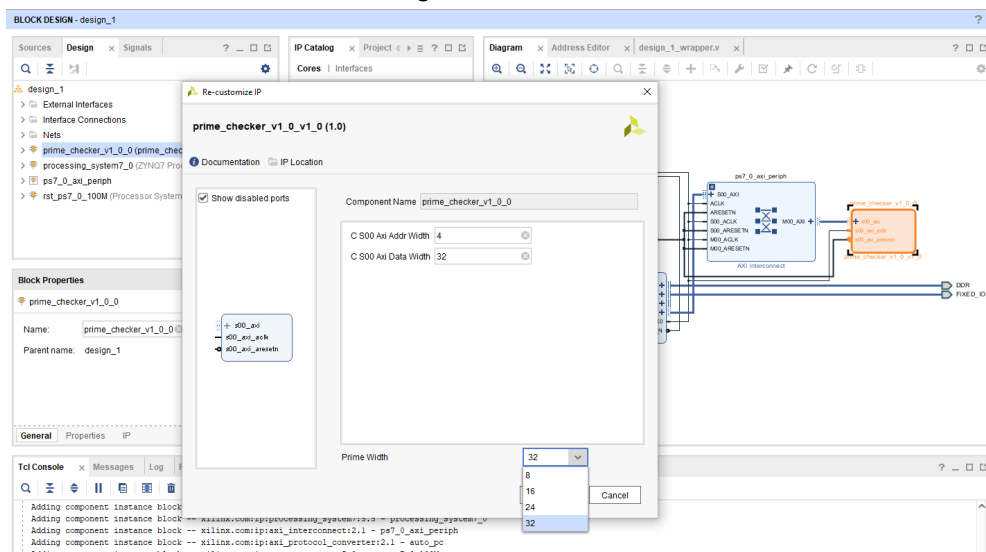


```
COM5 - PuTTY
> Number: 4294967291
> PRIME (#cycles = 1114081)
> Number: 4294967279
> PRIME (#cycles = 1114081)
> Number: 4294969281
> NOT PRIME (#cycles = 71)
> Number: 
```

Extra Credit

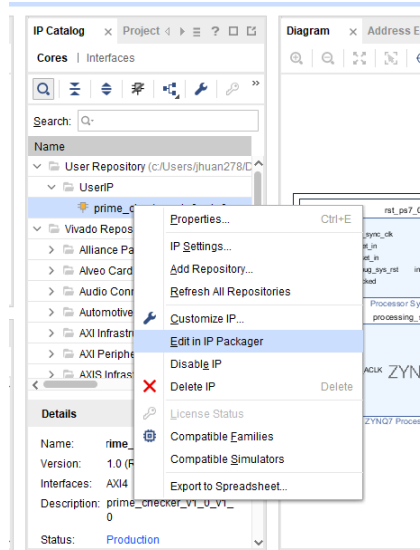
- Is your design fast enough to test 32-bit numbers?

Recall that `is_prime` is parameterized. You can actually set a larger bitwidth to the `prime_checker` IP on Block Designer. Go back to Part 3, and double click on the `prime_checker` module. You may set this to 32 and re-run the steps. Note that you have to export the hardware information and bitstream and then reprogram the FPGA on the SDK interface to make the change effective.

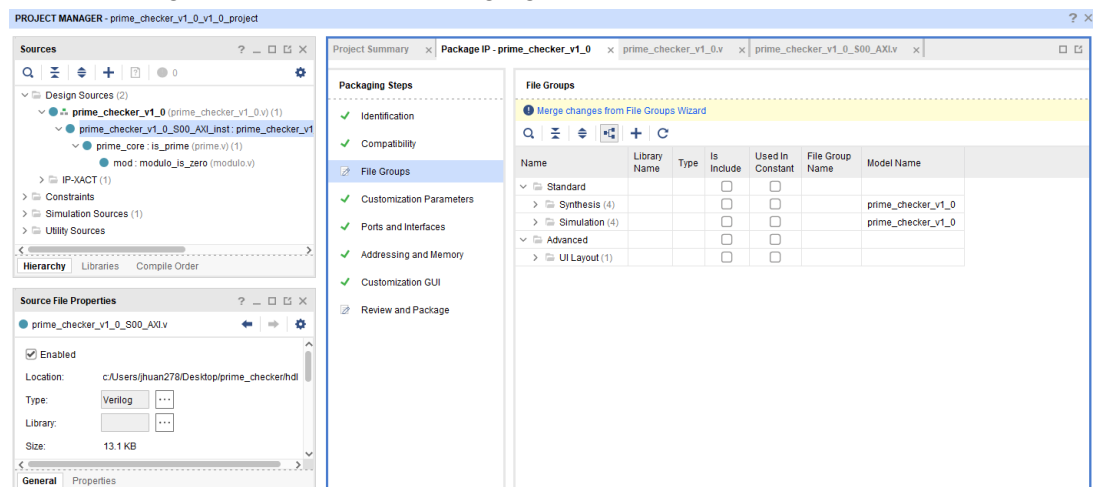


If your original design of `is_prime` is not fast enough, improve it. After the design is verified by the `prime_tb`, follow the steps below to make changes to the packaged IP

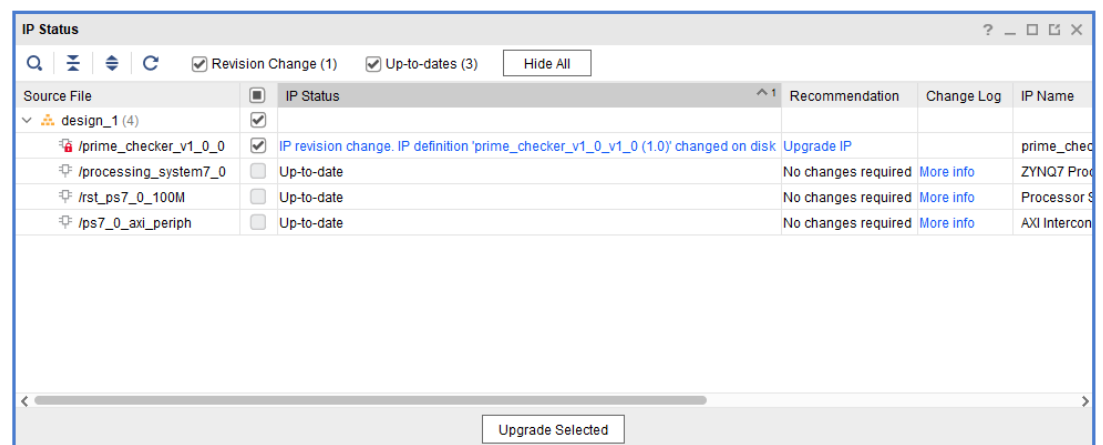
1. Go to IP Catalog > Right click on `prime_checker` > Edit in IP Editor Packager



2. Copy your `is_prime` code to `prime.v` in the IP
3. Go to Package IP and follow the merge guides



4. Upgrade the IP version in the main project (Upgrade Selected)



5. Rerun all the steps in Part 3 and Part 4

- Metrics: #cycles of a number randomly chosen from 1000 largest 32-bit primes
(Try this: **4,294,967,291**)
 - The reference design runs in **1,114,081** cycles with one **modulo_is_zero** instance.
 - You can get extra points if your design runs better than (or equal to) the reference design.

Lab Submission

- **prime.v**
- A write up in PDF format that includes
 - pseudo-code and/or state machine of the design
 - a short description of how the code/state machine works
 - total simulation time in milliseconds to run through all the test cases in **prime_tb**
 - a list of optimizations you can do or have done to reduce runtime or resources
 - #LUTs and #FFs used by the **prime_checker** IP with **PRIME_WIDTH=16**
(Find this in the submodule synthesis report in “Design Runs” window)
 - Screenshot of PuTTY terminal with tests