Tiny Tapeout Verilog Project Template

- Read the documentation for project

What is Tiny Tapeout?

Tiny Tapeout is an educational project that aims to make it easier and cheaper than ever to get your digital and analog designs manufactured on a real chip.

To learn more and get started, visit https://tinytapeout.com.

Set up your Verilog project

1. Add your Verilog files to the `src` folder.
2. Edit the info.yaml and update information about your project, paying special attention to the `source_files` and `top_module` properties. If you are upgrading an existing Tiny Tapeout project, check out our online info.yaml migration tool.
3. Edit docs/info.md and add a description of your project.
4. Adapt the testbench to your design. See test/README.md for more information.

The GitHub action will automatically build the ASIC files using OpenLane.

Enable GitHub actions to build the results page

- Enabling GitHub Pages

Resources

- FAQ
- Digital design lessons
- Learn how semiconductors work
- Join the community
- Build your design locally

What next?

- Submit your design to the next shuttle.
- Edit this README and explain your design, how it works, and how to test it.
- Share your project on your social network of choice:
    - LinkedIn #tinytapeout @TinyTapeout
    - Mastodon #tinytapeout @matthewvenn
    - X (formerly Twitter) #tinytapeout @tinytapeout

Quick Start Journal (for agneya)

- remember to `wsl --shutdown`

- to generate a .vcd, instantiate the virtual python environment `source venv/bin/activate` IN WSL and run `make -B` in the test folder
- if starting a new project, open WSL, run `python -m venv venv` in the test folder. Then, `source venv/bin/activate` to start the virtual environment. Then, `pip install -r requirements.txt` to install the necessary packages.
- start wsl
- python3 -m venv venv (create virtual environment. MUST BE MADE IN WSL)
- source venv/bin/activate (activate virtual environment)
- pip install -r requirements.txt (install necessary packages)
- change the module name in tb.v
- make -B (run the testbench)
- spi is main project, template is the buildable project
- uart_to_spi.v is the module file i put into vivado
- below code im supposed to put into my project.v file?

```
12430000.00ns INFO     cocotb.regression                  test_matrix_mult_uart passed
12430000.00ns INFO     cocotb.regression                  ***************************************************************************
****************

                                                          ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)
 RATIO (ns/s) **

                                                          ***************************************************************************
*************

                                                          ** test.test_matrix_mult_uart     PASS    12430000.00          0.12
 105499893.20  **

                                                          ***************************************************************************
****************

                                                          ** TESTS=1 PASS=1 FAIL=0 SKIP=0           12430000.00          0.49
 25311027.44  **

                                                          ***************************************************************************
****************
```

- UART finally works! i managed to fix it by first instantiating the reciever, and then setting up the transmitter. two independent files.
- UART WORKS WITH ALL BAUD RATES NOW YAYYYY

```
uart_to_spi uart (
        .clk(clock),
        .resetn(porb_h),        // "_h" is only valid for FPGA!

        .ser_tx(ser_tx_out),
        .ser_rx(ser_rx_in),

        .spi_sck(mprj_io_in[4]),
        .spi_csb(mprj_io_in[3]),
        .spi_sdo(mprj_io_out[1]),
        .spi_sdi(mprj_io_in[2]),
```

```
        .mgmt_uart_rx(mprj_io_in[5]),
        .mgmt_uart_tx(mprj_io_out[6]),

        .mgmt_uart_enabled(uart_enabled)
    );
```

OPERATION SERIAL TO USB TO UART TO SPI

- Top arty file recieves data from the computer via USB, sends it to the UART module, which sends it to the SPI module, which sends it to the FPGA. The FPGA then sends the data to the SPI module, which sends it to the UART module, which sends it to the Top arty file, which sends it to the computer via USB.
- to run the `interface_fpga.py` file, i just set my python interpreter to the pythonw.exe file in the scripts folder. i have no idea what this does. but somehow it worked. do on windows, not wsl instance.

Project Overview

This project focuses on creating a UART-to-SPI interface using Verilog. The design enables communication between a computer and an FPGA through a series of protocol conversions: USB to UART to SPI. The primary module, `uart_to_spi.v`, handles the UART-to-SPI conversion and is integrated into the top-level design.

Experimentation and Observations

1. **Module Integration**:

- Integrated the `uart_to_spi` module into the top-level Verilog file (`project.v`).
- Verified the connections for UART and SPI signals, ensuring proper mapping to the `mprj_io` pins.

2. **Simulation**:

- Used the provided testbench (`tb.v`) to simulate the design.
- Generated `.vcd` waveforms to analyze signal transitions and validate the UART-to-SPI functionality.
- Observed that the SPI clock (`spi_sck`) and data signals (`spi_sdo`, `spi_sdi`) behaved as expected during data transfers.

3. **FPGA Testing**:

- Synthesized the design in Vivado and programmed it onto the FPGA.
- Connected the FPGA to the computer via USB and used the `interface_fpga.py` script to send and receive data.
- Verified that data sent from the computer was correctly transmitted to the FPGA and back.

4. **Challenges**:

- Faced initial issues with the SPI clock signal not toggling correctly. Resolved this by debugging the clock enable logic in the `uart_to_spi` module.
- Encountered difficulties setting up the Python environment for the `interface_fpga.py` script. Switching to the correct Python interpreter resolved the issue.

5. **Next Steps**:

- Optimize the Verilog code for better timing performance.
- Add error-checking mechanisms to handle communication failures.
- Explore extending the design to support additional protocols or higher data rates.

USB-to-UART Communication Challenges :

The USB-to-UART bridge introduces several complexities that must be addressed for reliable data transmission. Key considerations include:

1. **Baud Rate Synchronization**:

- USB operates at much higher data rates compared to UART. The USB-to-UART bridge must handle this disparity by buffering data and ensuring proper baud rate synchronization.
- Investigate how the bridge manages flow control (e.g., RTS/CTS or XON/XOFF) to prevent data loss during high-speed transfers.

2. **Latency and Throughput**:

- USB communication introduces latency due to its packetized nature. This can impact real-time applications where UART expects continuous data streams.
- Measure the end-to-end latency from USB to UART and analyze its impact on SPI communication timing.

3. **Error Handling**:

- USB and UART have different error detection mechanisms. USB uses CRC for error checking, while UART relies on parity bits or framing checks.
- Explore how the USB-to-UART bridge translates and reports errors, and implement mechanisms in the Verilog design to handle these errors gracefully.

Signal Integrity and Noise Considerations

1. **UART Signal Degradation**:

- Long USB cables or noisy environments can introduce jitter and signal degradation in the UART signals.
- Use an oscilloscope to analyze the signal integrity of the UART lines and identify potential issues such as overshoot, ringing, or crosstalk.

2. **SPI Signal Timing**:

- The SPI clock and data signals must meet strict timing requirements to ensure proper communication with the FPGA.
- Perform timing analysis to verify that the SPI signals remain within the setup and hold time constraints of the FPGA.

Debugging and Optimization

1. **Protocol Analyzer**:

- Use a USB protocol analyzer to capture and decode USB packets. This can help identify issues in the USB-to-UART conversion process.

- Similarly, use a logic analyzer to monitor UART and SPI signals during operation.

2. **Buffer Management**:

- Analyze the buffering strategy used in the `uart_to_spi` module. Ensure that the FIFO depth is sufficient to handle burst data transfers without overflow or underflow.

3. **Clock Domain Crossing**:

- The design involves multiple clock domains (USB, UART, SPI). Investigate how clock domain crossings are handled, and ensure that proper synchronization techniques (e.g., dual flip-flop synchronizers) are implemented.

Future Exploration

1. **High-Speed Alternatives**:

- Evaluate the feasibility of replacing UART with a higher-speed protocol such as USB 3.0 or PCIe for improved performance.
- Investigate the use of SPI over USB (e.g., USB-SPI bridges) to eliminate the intermediate UART step.

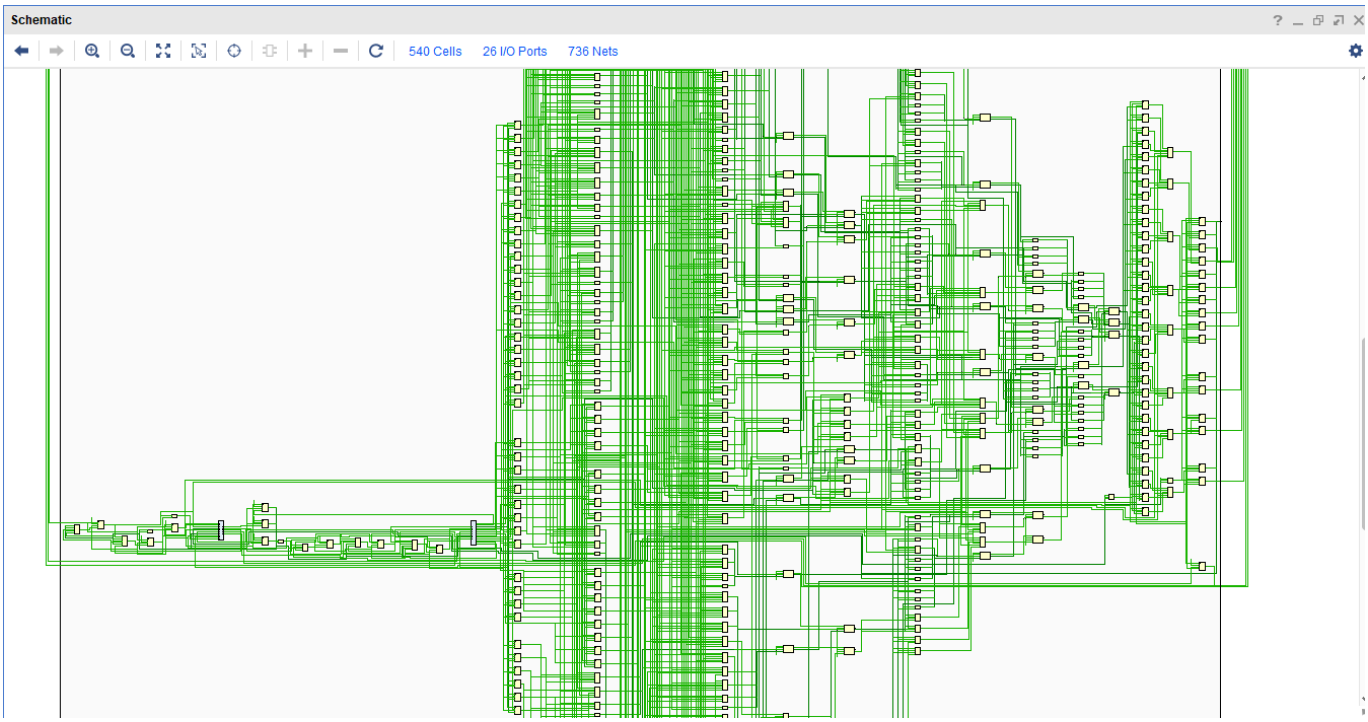
2. **Error Correction**:

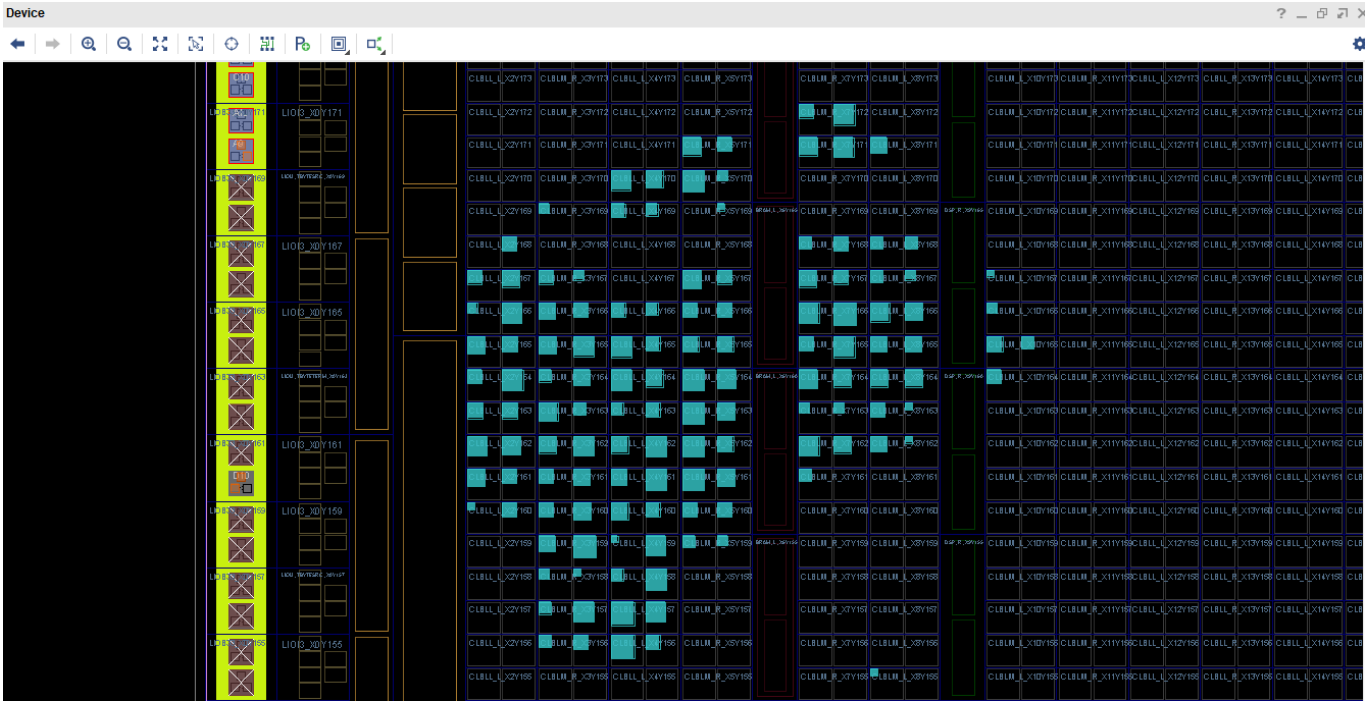
- Implement advanced error correction techniques, such as Hamming codes, to improve the reliability of UART-to-SPI communication.

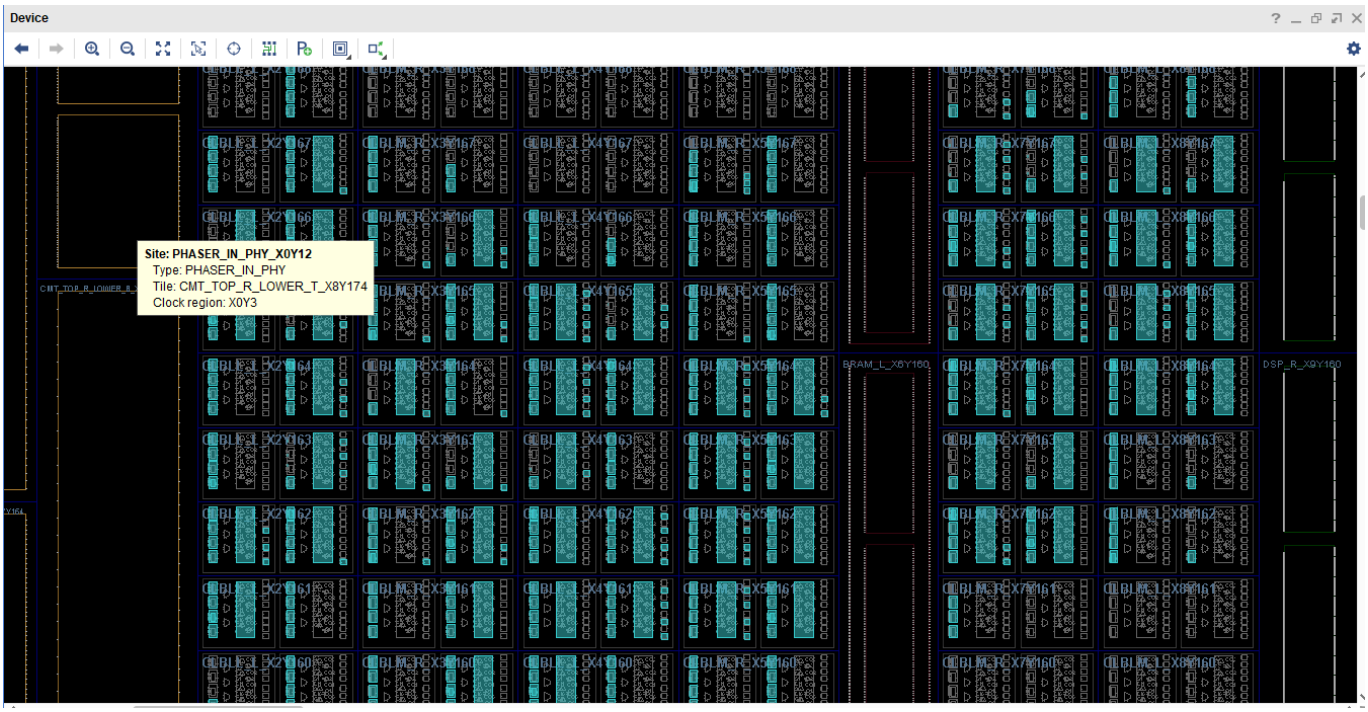
3. **Protocol Extensions**:

- Extend the design to support additional UART features, such as multi-drop communication or custom baud rates.
- Explore adding SPI modes (e.g., Mode 1 or Mode 3) to increase compatibility with different devices.

By addressing these challenges and exploring these optimizations, the UART-to-SPI interface can be made more robust and versatile, paving the way for more complex and high-performance designs.


Schematic
540 Cells
26 I/O Ports
736 Nets


Device


Device
Site: PHASER_IN_PHY_X0Y12
Type: PHASER_IN_PHY
Tile: CMT_TOP_R_LOWER_T_X8Y174
Clock region: X0Y3
BRAM_L_X6Y160
DSP_R_X9Y160