Design and Simulation of a USB 2.0 Transceiver in SystemVerilog

Andrew Lu, Jiayi Liu, Leo Lesmes, Nhan Do
`https://github.com/leo-les/usb-transceiver`[1]

December 12, 2025

1 Technical

1.1 Overview

The USB Transceiver covers the physical (PHY) layer of USB communication, bridging the digital data used by a host controller (computers, smartphones, etc.) with the electrical signals sent through USB wires to a device controller (mouses, keyboards, flash drives, etc.). In other words, the transceiver acts as a bidirectional translator between digital logic (1s and 0s) and analog voltage levels on its two differential data wires, D+ and D-.

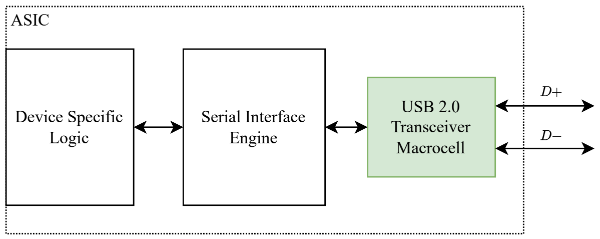

Figure 1: ASIC Functional Blocks[2]

Our project aimed to simulate the behavior of a USB 2.0 Low Speed Transceiver in SystemVerilog, which sends and receives data at 1.5 Mbps utilizing a 48 Mhz clock signal. To the left of the USB 2.0 Transceiver block is a Serial Interface Engine (SIE), which treats incoming data as packets and sends serial data to the transceiver. To the right of the block are D+/D- wires, connecting the transceiver to a device controller.

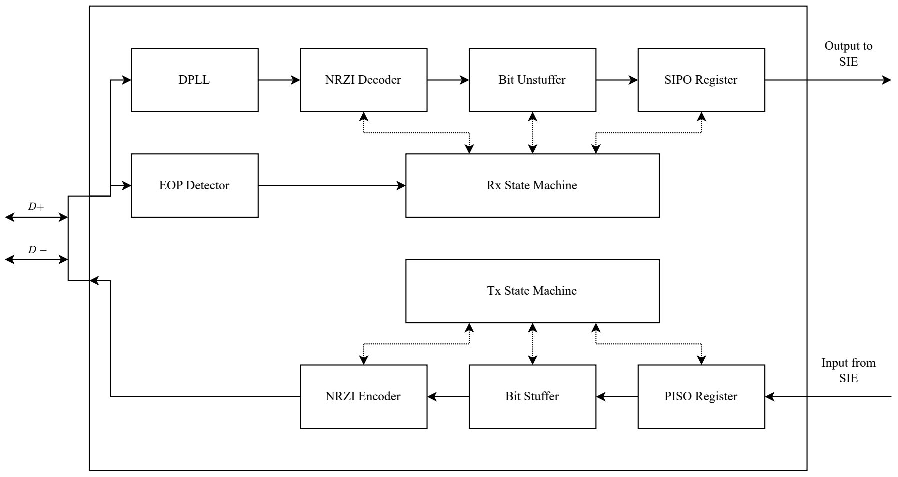

Figure 2: UTMI RTL Diagram

The Register-Transfer Level (RTL) design above maps out all the modules implemented in SystemVerilog making up our transceiver. The top and bottom halves of the diagram consists of all the modules receiving from and transmitting data to the D+/D- wires, respectively. The top module, represented by the box encompassing all the modules, initializes all the modules and connects their signals to allow for a layer of abstraction from the individual modules.

Hardware & Software

Numerous pieces of software were used to help develop this project, along with multiple coding languages. The names of the products used, along with their purpose in the project, are listed below.

Type	Name	Purpose
Software	Google Docs	Research/brainstorming, writing for proposal and check-ins
	Figma	Research/brainstorming, and drafting diagrams
	Draw.io	Used to create final diagrams and flowcharts
	Google Slides	Slideshow software that the Final Presentation was created in.
	Visual Studio Code	IDE used for coding the project and writing the final report.
	Github	Project was hosted and shared
	MyFPGA	In web-browser simulation tool for testing SystemVerilog code
Programming Language	SystemVerilog	Language that the UTMI was written in
	C++	
	Makefile	
	LaTeX	Final project report (this document) was written in LaTeX

Table 1: Products used to develop the project

Division of Labor

Numerous pieces of software were used to help develop this project, along with multiple coding languages. The names of the products used, along with their purpose in the project, are listed below.

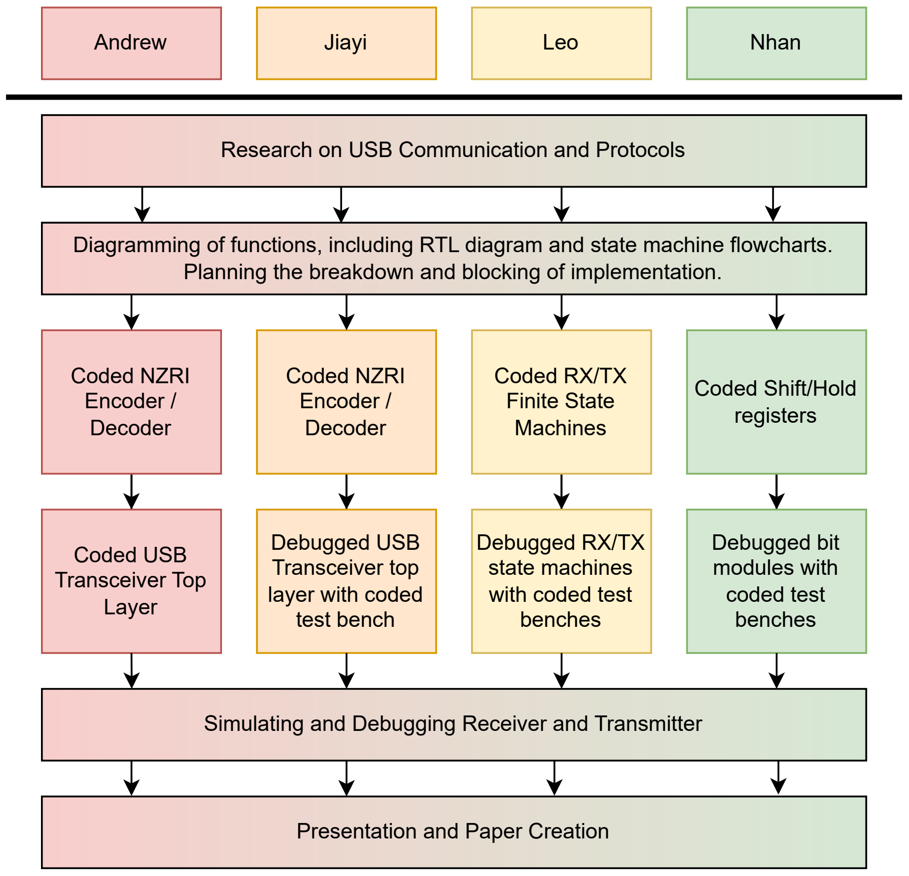
Andrew
Jiayi
Leo
Nhan
Research on USB Communication and Protocols
Diagramming of functions, including RTL diagram and state machine flowcharts. Planning the breakdown and blocking of implementation.
Coded NZRI Encoder / Decoder
Coded NZRI Encoder / Decoder
Coded RX/TX Finite State Machines
Coded Shift/Hold registers
Coded USB Transceiver Top Layer
Debugged USB Transceiver top layer with coded test bench
Debugged RX/TX state machines with coded test benches
Debugged bit modules with coded test benches
Simulating and Debugging Receiver and Transmitter
Presentation and Paper Creation

Figure 3: Division of Labor

Timeline

We kept our project to a pretty strict timeline

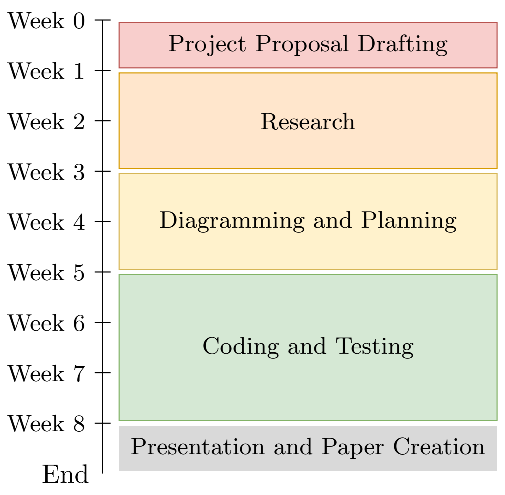

Figure 4: Timeline

1.2 Design

RX State Machine: The RX State Machine monitors the reception of data from the differential data wires, primarily checking for the

TX State Machine: The TX State Machine controls the transmitting process. It organizes the states in which the data is loaded and sent. It controls when the data is loaded into the shift register and waits until a 8 bits (1 byte) is present and then serially shifts out the bits. The state machine ensures data is smoothly transferred from an SIE to the differential data wires and protects against cases in which data might be corrupted or imprecisely loaded.

DPLL: The Digital Phase Locked Loop (DPLL) module generates a pulse at the middle of each incoming clock signal when the transceiver is receiving signals from D+/D-, ensuring clock synchronization of all the modules with the incoming signals. The pulse is propagated to the rest of the receiving modules.

EOP Detector: The EOP detector is used to detect the End Of Packet signal, or EOP, on the receiving side of the transceiver. When the receiver detects an EOP, it knows that there is no more incoming data. EOP is signalled by two clock cycles of the SE0 state, where both data lines are pulled low, followed by the low-speed idle state, where D+ is 1 and D- is 0.

NRZI Decoder/Encoder: The USB 2.0 protocol transmits and receives data through the NRZI encoding schematic. Instead of encoding data through 0s and 1s on the data line, data is encoded through the presence or absence of edges on the data line, with both the receiver and transmitter operating at the same clock frequency. A zero is encoded in this new line of data through a signal transition, either from 0 to 1, or 1 to 0, and a 1 is encoded through a constant signal level. When decoding an encoded signal, we know that edges on the incoming data line encode a 0, and no change in the data line encodes a 1.

Bit Stuffing/Unstuffing: Though the NRZI protocol can be efficient, it also comes with downsides. Because a 1 is encoded with an unchanging signal, the high frequency transmitter and receiver are at risk of desynchronization when too many 1s are transmitted in a row. Thus, the concept of bit stuffing was invented. Whenever six 1s are transmitted in a row on a data line, a 0 is "stuffed" into the data sequence, following the six 1s, which manifests as a level transition in the data. This stuffed bit acts as a re-synchronization signal for the receiver and transmitter, ensuring a stable communication connection. On the receiving end, the stuffed bits must be removed to recover the original data, which is accomplished through the use of a bit unstuffer. An unstuffer implements the reverse of the stuffing protocol, ignoring the stuffed bit that always follows six consecutive 1s. The unstuffer can also be used to detect transmission errors if it receives data that has not been bit stuffed.

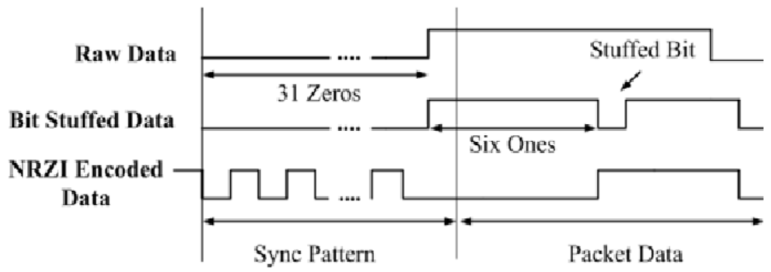

Figure 5: Bit Stuffing and NRZI Encoding

SIPO Register: SIPO (Serial-in-parallel-out): SIPO shift register is a sequential logic device that can store and shift data bits. "Serial-in" means that the data is entered into the register one bit at a time. "Parallel out" means that data is accessed simultaneously.

PISO Register: PISO (Parallel-in-serial-out): PISO shift register accepts parallel data and output serial data. The data is loaded into the register simultaneously through multiple input lines. The output comes out as serial data (bit by bit).

1.3 Simulation

Testing & Benchmarks

- USB transceiver transmits and receives data as stated in our proposal.
- Individual modules were tested and verified with testbenches. SystemVerilog testbenches were created and run using Makefile. For the final simulation, we used the ecelabs.io website to get our design simulated and collect data from the generated waveforms.
- Predetermined bytes were used as test cases. After analyzing the waveforms, we confirmed that it correctly decodes and unstuffs the data for receiving as well as stuffs and encodes the data for transmitting.

Waveform Results

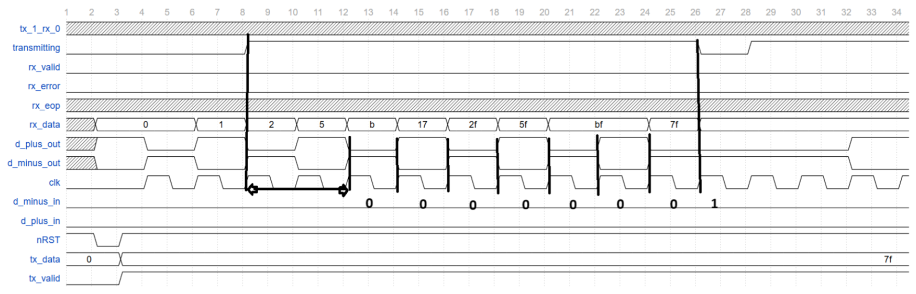

Figure 6: Transmitting 8'b00000001 (SYNC) Byte Waveform

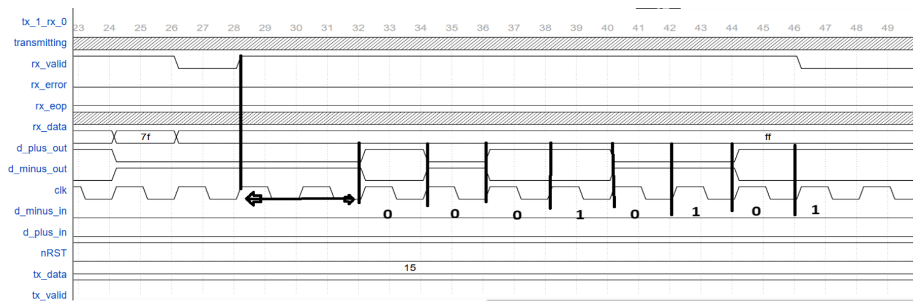

Figure 7: Transmitting 8'b00010101 Byte Waveform

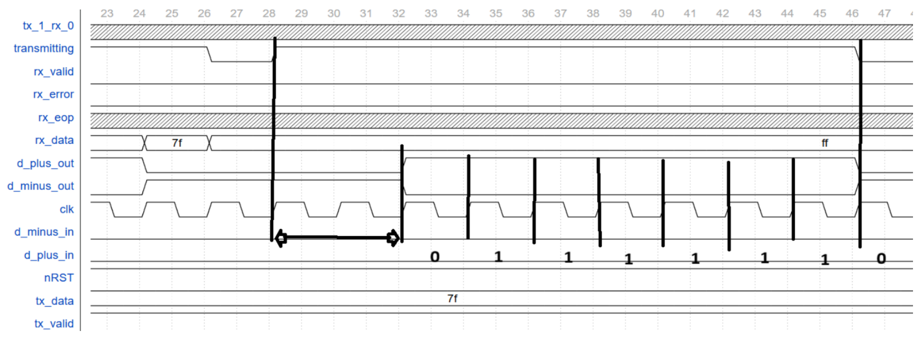

Figure 8: Transmitting 8'01111111 Byte Waveform

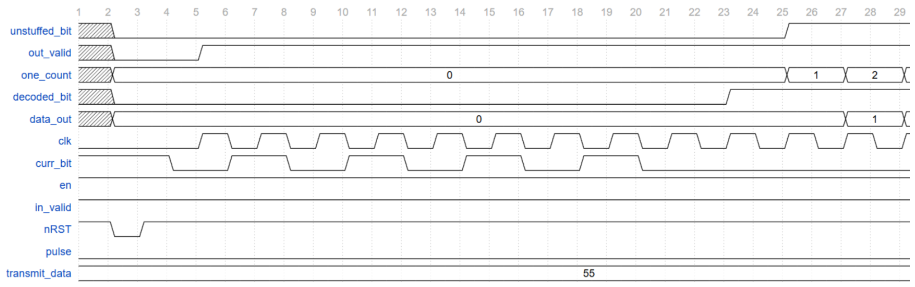

Figure 9: Receiving 8'b01010101 Byte Waveform

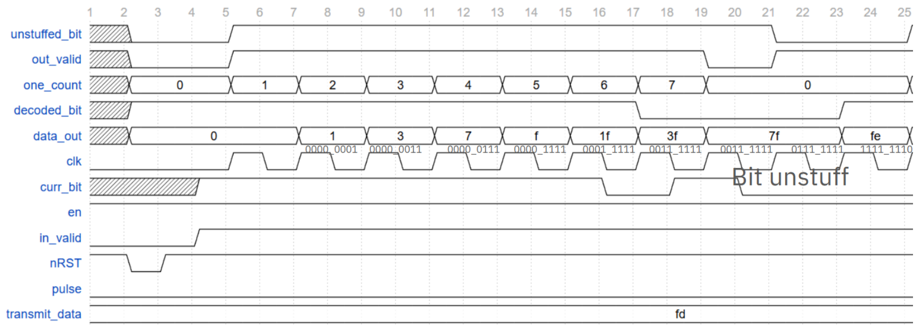

Figure 10: Receiving 8'b11111101 Byte Waveform

Operation Documentation

The USB transceiver is packaged into a top-level usb-transceiver.sv SystemVerilog file. At a high level, it is similar to a Python or C++ class, with abstracted inputs and outputs that the user can interface with to control the whole module, and a hardware analogue would be a physical USB transceiver with input and output pins. To send data to the transceiver, simply pulse the input D+and D- with whatever signal you want to send the transceiver, with D- being the logical NOT of D+. The received data will appear in the data-in shift register after it is received. To transmit data using the transceiver, use the transmit enable and transmit valid pins to load in data to the transmission shift register, which will first send a SYNC signal, confirming the start of transmission and then shift out each bit, stuffed and encoded, to the output D+ and D- wires, where D- is the inverse of D+.

Expansion/Next Steps

For our next steps, we were inspired by a post-presentation question to implement our transceiver on an actual USB device. We would have to implement the actual serial interface logic, which manages packet transfers, handshakes, and interfaces with the device hardware. Following this, the device itself would be implemented in hardware with SystemVerilog, which could be a mouse, keyboard, or any other kind of USB peripheral. Finally, we would flash this onto an FPGA and test our USB device by connecting through PMOD or wires to a USB port, allowing us to put our device and transceiver to use.

2 Reflection

Learning Outcomes

Building this from the ground up forced us to apply our digital design skills. Besides that, it also required us to find reliable sources and read technical documentations. These are crucial skill sets which can be applied to our future ECE courses and career paths. We also significantly improved our testbench writing by figuring out how to generate the Makefile, create the simulations, and analyze the results. Ultimately, we understand the technical concepts of a USB transceiver design. As for a potential continuing of this project in Intro II, we could implement peripheral interfaces like the SIE and/or test a hardware implementation of the USB transceiver.

Preferences

Andrew: I enjoyed implementing all of the logic behind the transceiver in SystemVerilog and producing the waveform outputs of our working modules, though I disliked the tedious process of debugging the modules with testbenches.

Jiayi: Through this project, I learned more about what digital design looks like at a professional level. It requires precise timing and meeting hardware and protocol specifications, which can be tedious at times. However, the system-level design is interesting and fun. I learned that I enjoy designing and conceptually building digital systems, but I do not like the low level details of actually implementing it.

Leo: I also gained a lot more interest in exploring Analog Design from this project.

Nhan: After working on this project, I found my interest in working with digital designs. I hope to have more opportunities to get involved in digital projects in the future. I would also like to gain a deeper understanding of digital peripherals and communication.

Challenges

The main challenge of the project was finding and understanding all the technical specifications for what we were trying to design. We had to spend a significant amount of time on doing research on the Internet, reading articles, and finding other reliable sources. If challenging topics were encountered, our team discussed them together and tried to understand them together.

Another challenge that we encountered was establishing a large project with a newly learned language. To overcome the difficulties, we had to read numerous articles and follow many guidelines to get the right direction. Although debugging costed us a considerable amount of time, we successfully got out design simulated and collected the results.

When it came to scheduling, it was difficult to manage the timeline over a long period as we had other to-dos to finish. However, our team usually met whenever a member ran into issues and tried to solve the problems. When everybody had finished with their modules, we worked together on the top module and implemented our parts into the final design. After the design was created, results analysis jumped in. Eventually, all the important milestones that we set in the beginning were completed as the team effectively cooperated with each other.

When it came to actually coding the project, the hardest part was integrating all of the modules together and passing all of the test cases. As each teammate self-design their own modules, it took us some time to merge all the parts together and make them work effectively. At the end of the design process, our team worked together on the FSMs and the top module. Everyone understood their parts and explained them to everyone, which greatly accelerated our progress.

References

[1] Andrew Lu, Jiayi Liu, Leo Lesmes, and Nhan Do. *usb-transceiver*. URL: `https://github.com/leo-les/usb-transceiver`.

[2] Intel Corporation. *USB 2.0 Transceiver Macrocell Interface (UTMI) Specification*. Technical Specification. Version 1.05. Intel Corporation, Mar. 2001. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/usb2-transceiver-macrocell-interface-specification.pdf`.

[3] Robert Murphy. *USB 101: An Introduction to Universal Serial Bus 2.0*. Application Note AN57294. Document No. 001-57294 Rev. *H. Cypress Semiconductor Corporation (now Infineon Technologies), 2014. URL: `https://www.infineon.com/assets/row/public/documents/cross-divisions/42/infineon-an57294-usb-101-an-introduction-to-universal-serial-bus-2.0-applicationnotes-en.pdf`.