UART Implementation

In this document I will be explaining the basics of UART and how to implement it using Verilog, Tang Nano 20k FPGA and TeraTerm.

What is UART?

UART (AKA Serial Port, RS-232, COM Port, RS-485) is an acronym for Universal Asynchronous Receiver Transmitter. A UART is one of the simplest methods of communicating with an FPGA. It can be used to send commands from a computer to an FPGA and vice versa.

A UART is an interface that sends out data, usually one byte at a time over a single wire. It does not forward a clock with the data, making it asynchronous.

There are 2 ways a UART can operate:

1. **Half-Duplex:** Two transmitters sharing a line.
   For example, imagine a single-lane road that supports traffic in both directions, but only one car can travel at a time. Cars from one direction must wait if a car from the opposite direction is already on the road. Similarly, in half-duplex UART communication, devices share a single line for transmitting and receiving data, but only one device can transmit at a time.

2. **Full-Duplex:** Two transmitters each with their own lines.
   Using the half-duplex analogy, this time imagine a two-lane road where cars can travel in both directions simultaneously, each direction having its own dedicated lane. In full-duplex UART communication, there are separate lines for transmitting and receiving data, allowing both devices to send and receive simultaneously.

UARTS has several parameters that can be set by the user.

1. **Baud Rate:** The baud rate specifies the speed of data transmission in bits per second (bps).

   For example:

   9600: 9600 bits transmitted per second.

   19200: 19200 bits transmitted per second.

   115200: 115200 bits transmitted per second.

   The baud rate must be the same on both the transmitter and receiver sides for communication to work. Faster baud rates (like 115200) are ideal for high-speed data transfer but may be more susceptible to errors in noisy environments, while lower baud rates (like 9600) are slower but more reliable.

2. **Number of Data Bits:** This defines the number of bits used to represent each character in the transmitted data.
   Common options include:
   a. 7 bits: Typically used in older systems (like ASCII).
   b. 8 bits: Standard for modern communication, allowing for more data (e.g., extended ASCII).

   The transmitter and receiver must agree on the number of data bits for successful communication.

3. **Parity Bit:** The parity bit is an optional error-detection mechanism.
   It can be:
   a. On: A parity bit is added to the data. The parity ensures the total number of 1s in the transmitted data is either even (even parity) or odd (odd parity). This helps detect errors in the data.
   b. Off: No parity bit is added, meaning no error detection is performed.

   Example:

   Capital “S” (1 0 1 0 0 1 1) contains a total of three zeros and 4 ones. If using even parity, the parity bit is zero because there already is an even number of ones. If using odd parity, then the parity bit must be one in order to make the frame have an odd number of 1s.
   The parity bit can only detect a single flipped bit. If more than one bit is flipped, there’s no way to reliably detect these using a single parity bit.

4. **<u>Start and Stop Bits:</u>** Start bits indicate the beginning of a data frame and stop bits indicate the end of a data frame. They help the receiver recognize the boundary between consecutive data frames.

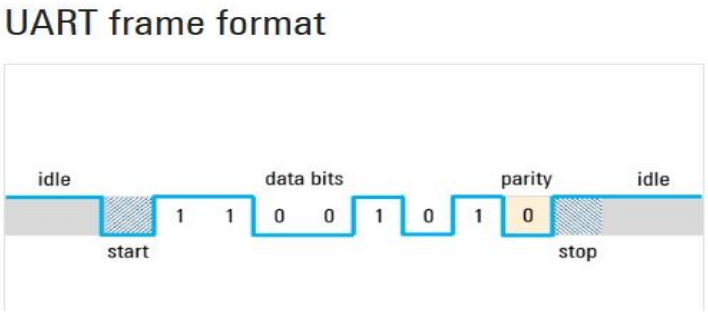
   Common options:

   0 Stop Bits: No stop bit.
   1 Stop Bit: A single stop bit (commonly used).
   2 Stop Bits: Two stop bits for more reliable communication (but slightly slower).

   Start bit is always 1 bit and is indicated by low (logic 0).

   Example:
   For an 8-bit UART frame with no parity and 1 stop bit:

   Start Bit (1 bit): LOW (logic 0).
   Data Bits (8 bits): Varying levels (HIGH or LOW, depending on the data).
   Stop Bit (1 bit): HIGH (logic 1).
   Line returns to the idle state (logic HIGH).
   The receiver "knows" it is a stop bit because it occurs at the expected location after the data and parity bit(s), and it validates that the line is HIGH for the specified duration.



5. **Flow Control:** Flow control manages the rate of data transmission between devices to prevent buffer overflows.

   Options include:

   None: No flow control is used, meaning the sender transmits at will, and the receiver must keep up.
   On: Software-based flow control (e.g., XON/XOFF).
   Hardware: Uses hardware signals (e.g., RTS/CTS) to control the transmission speed, ensuring the receiver is ready before data is sent.
   These settings must match on both sides for UART communication to function correctly.

As mentioned previously, there is no clock that gets sent along with the data. In any interface that does not have a clock, the data must be sampled to recover it correctly. It needs to be sampled at least eight times faster than the rate of the data bits. This means that for an 115200 baud UART, the data needs to be sampled at least 921.6 KHz (115200 baud * 8). A faster sampling clock can be used.

If you would like to see more explanations, here are some resources:

Understanding UART - YouTube

UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter | Analog Devices

UART Module

My UART transceiver is designed with the following parameters:

- Full Duplex UART, implemented with a half-duplex communication protocol
- Baud rate 115200 (27 MHz clock, ~234 clocks per bit)
- 8 data bits
- 1 start bit + 1 stop bit

Simulation and Results

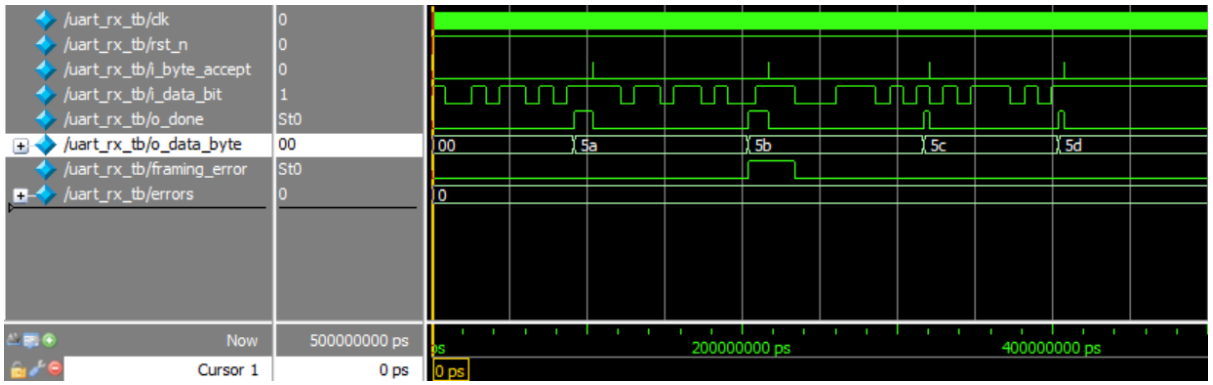
I built 2 testbenches to thoroughly check the code – uart_rx_tb and uart_tx_tb.

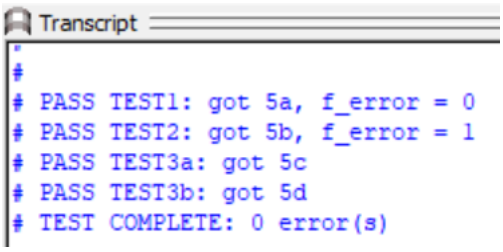
The total number of bits in each frame is 10, meaning the total frame duration is about 120 us. To ensure the capture of the all the tests, I set the runtime to be 500 us.

According to our calculations, the actual measured baud rate is approximately ~114,631 bps, about 0.5% error.

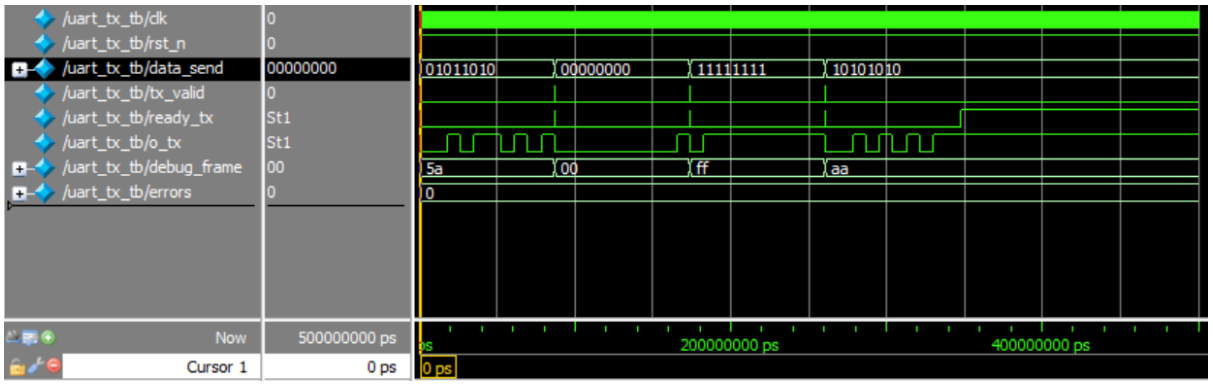
In most UART systems, the tolerance is up to 2% baud rate mismatch before errors occur.

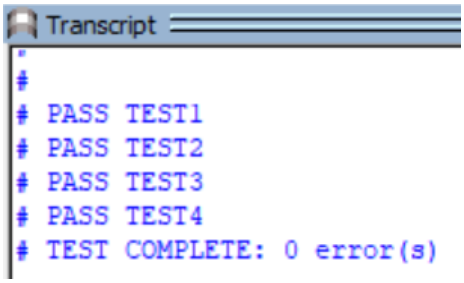
Rx Testbench





Tx Testbench





From all the simulations, I can see that our UART design works as intended. In the next chapter, I will demonstrate the use of our design on a real FPGA development board kit.

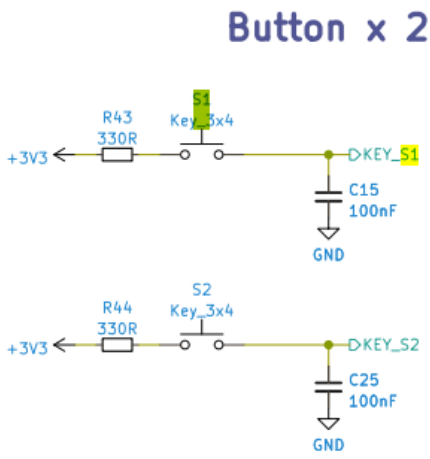
I will be using the Sipeed Tang Nano 20k.

FPGA – Sipeed Tang Nano 20k

First, we need to establish our goal with this implementation.

I implemented a full-duplex UART. To test it out on hardware, I created a simple loopback test. The user should type anything they want and see what they typed on the Tera Term console. Additionally, onboard LEDs indicate the last command received from the PC for debugging and user feedback.
On the board there are two white physical buttons: S1 and S2.
We will configure S1 to be the reset button.
Because we are using physical buttons, I wrote uart_debounce to make sure that the button acts appropriately.

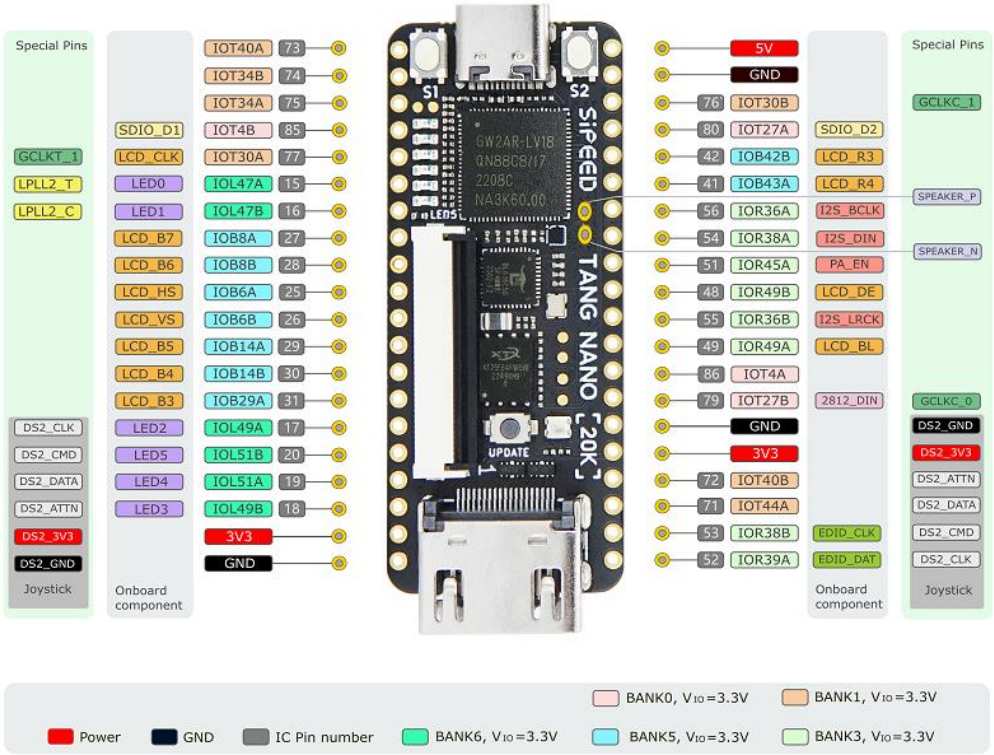
It is important to note that the buttons are active-high, as shown in the schematics:



Since the whole reset logic that I have implemented is low-active, in uart_top I inverted the logic.

After synthesizing the code, we need to set constraints to bind the ports defined in our code to the desired FPGA pins.

Here is the PIN diagram of the Tang Nano 20k:

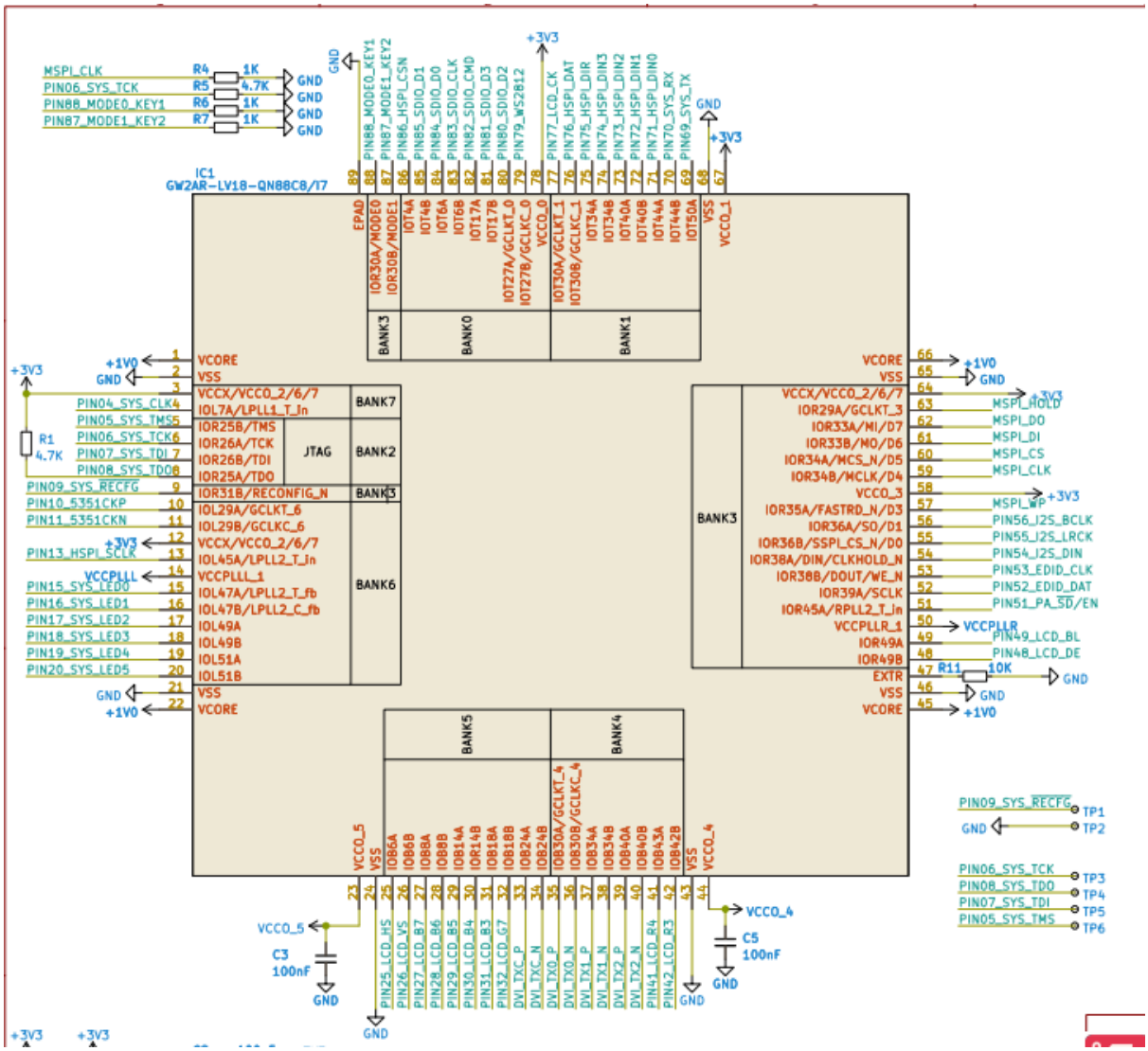


You may have noticed that there are “missing” pin numbers. For example, where is the system clock pin?
After a bit of digging on the wiki, I came across the schematic of FPGA located here:

下载站 - Sipeed

It is also needed to update the BL616 chip firmware.
Follow this guide:
Update debugger - Sipeed Wiki

Downloading the schematic, you can clearly see all the pin numbers and the layout of the IC1 GW2AR-LV18-QN88C8/I7 chip.



Now, using the I/O constraints in the Floor Planner tab in the Gowin IDE, I set each port to its location.

Important note – when constraining, select IO Type: LVCMOS33. The floor planner sometimes defaults to LVCMOS18 for I/O standard and BACK_VCCIO=1.8. It means that in the selected pins, the voltage to be delivered is 1.8V. That is incorrect as all the banks are wired to 3.3V in Tang Nano 20K. If you do not select the right IO Type, you will get an error message.

After finishing the physical constraints, it is needed to create a timing constraints file using the Timing Constraints editor, to specify the clock speed (27MHz).

Next, run Place & Route. The purpose is to synthesize the generated netlist and our defined constraints to calculate the optimal solution through IDE, then allocate resources reasonably on the FPGA chip.

Next, we will burn the program into the FPGA. For this, we will use the Programmer in the IDE. After opening the program, we were prompted with a cable setting screen. Make sure that the settings were configured properly (use Query/Detect cable to confirm that) and for the frequency to match as well.
In the operation setting we have 2 choices: SRAM Program / SPI Flash Program.
The first operation writes the bitstream into volatile memory (will be lost on power off – used for testing). The second operation is to be used if you want to keep the design on a permanent storage (for writing to an onboard flash). Decide what is best for you, we decided to use SPI Flash Program, so that every time we plug in the board, it will boot with the UART design automatically.

To do so: Configure device -> Access Mode: External Flash Mode -> Operation: exFlash Erase, Program, Verify through GAO-Bridge.

This operation ensures a clean flash, loads the .fs file, confirms successful write and uses onboard BL616 chip.

The next step is to communicate with the FPGA. To do so, I will use Tera Term.

Communication Using Tera Term

Connecting to the FPGA is super simple. Just open the program ,choose Serial, and then the correct port (should be something like COM11/COM12, try each one).
Next, go to Setup -> Serial port and choose the correct settings.

And that's it!