UART Implementation

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

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.

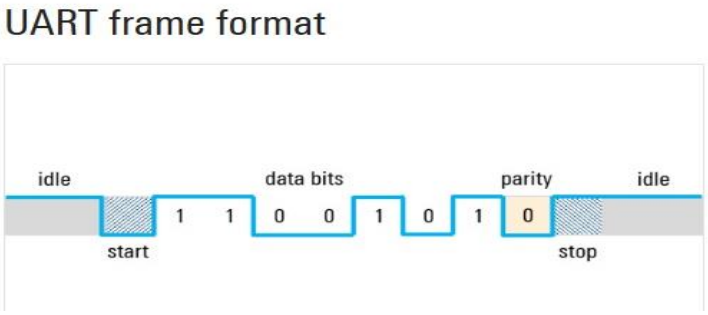
   Common options:

   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.

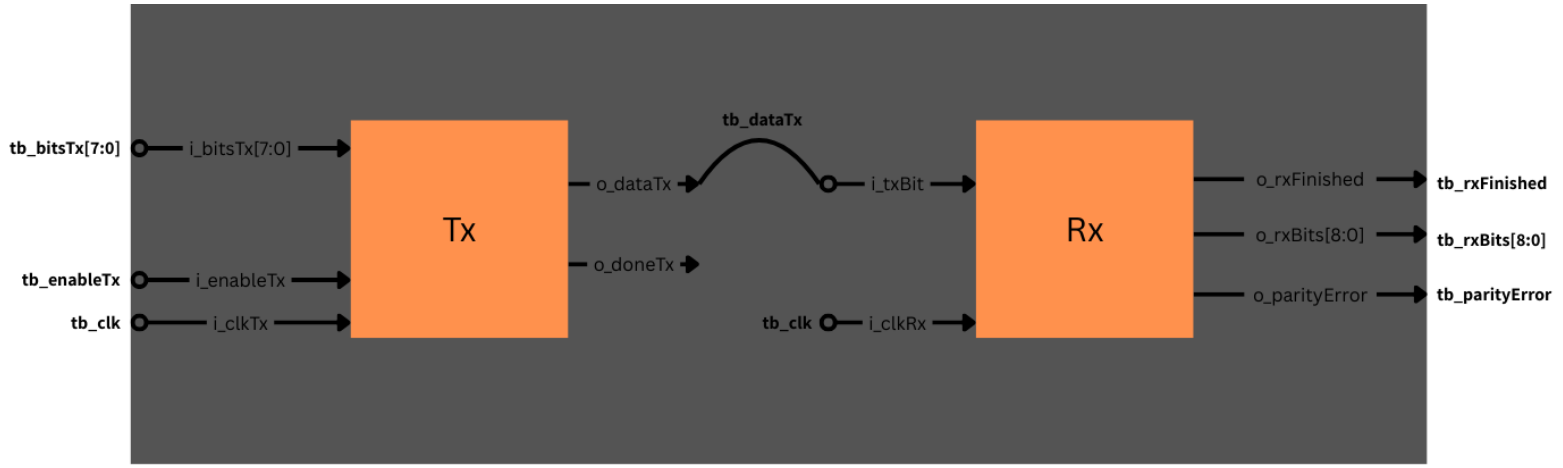
As mentioned previously, there is no clock that gets sent along with the data. In any interface that does not have a clock, the data must be sampled to recover it correctly. It needs to be sampled at least eight times faster than the rate of the data bits. This means that for an 115200 baud UART, the 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 another explanation of UART, we suggest watching this video: Understanding UART - YouTube

UART Module

Our UART transceiver is designed with the following parameters:

- Full Duplex UART, implemented with a half-duplex communication protocol
- Baud rate 115200 (10 MHz clock, 87 clocks per bit)
- 8 data bits + 1 parity bit (9 bits total, even parity logic)
- 1 start bit + 1 stop bit



Port Explanation

Tx

```
i_bitsTx[7:0] – 8 bits (1 byte) of data that enters the transmitter
i_enableTx    - turn transmitter on/off
i_clkTx       - transmitter clock

o_dataTx      - 1 bit output (9 bits in total – 8 data + 1 parity)
o_doneTx      - indicates if the transmitter finished transmitting full frame
```

Rx

```
i_txBit           – 1 bit input from transmitter (9 bits in total – 8 data + 1 parity)
i_clkRx           – receiver clock

o_rxFinished      – indicates if the receiver finished receiving full frame
o_rxBits[8:0][1]  – full frame received
o_parityError     – indicates if the data is received correctly
```

[1] ***Note:*** *Initially, the UART Rx output* (o_rxBits[8:0]) *consisted of 9 bits (8 data bits + 1 parity bit). However, it was later modified to output only the 8 data bits, since the receiving side does not require the parity bit for operation. The parity bit is solely used for error checking during data reception. Also, The UART modules were updated to include a dedicated i_reset input to ensure reliable initialization and improved system control.*

Simulation and Results[2]

We have built 3 testbenches to thoroughly check our code.

One standalone testbench for each component and one “top level” testbench for the whole UART transceiver.

The total number of bits in the frame is 11, meaning the total frame duration is 95.7 us. To ensure the capture of the full frame, set in the simulation program you are using to run for at least 2 times the frame duration.

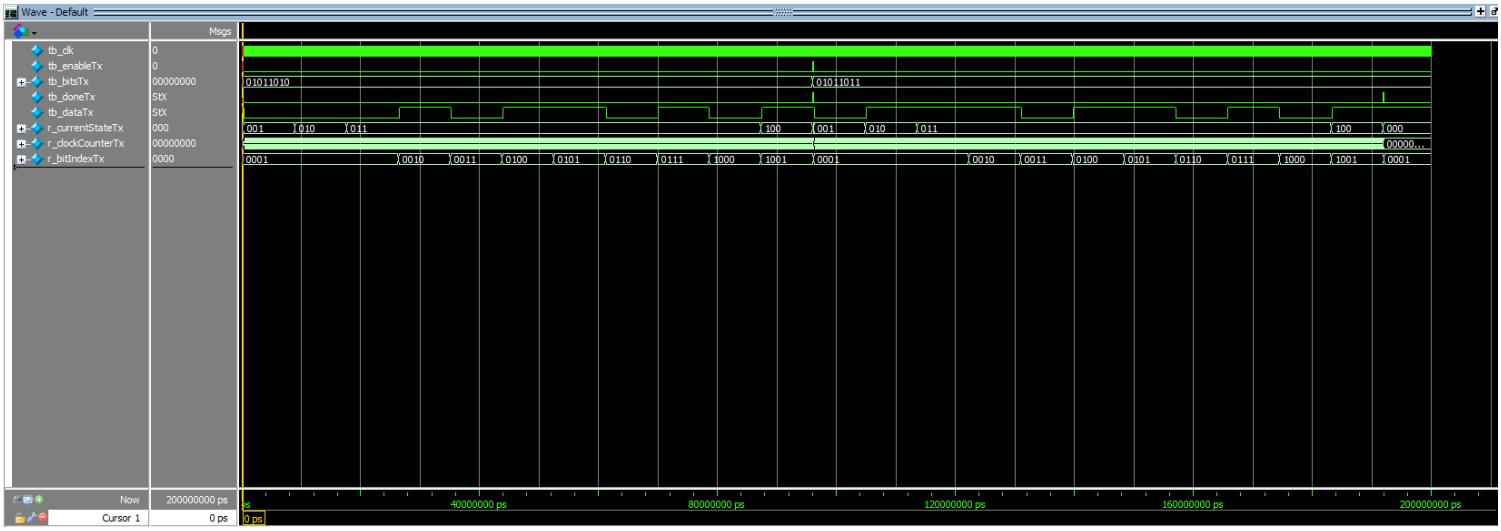
According to our calculations, the actual measured baud rate is approximately ~114,631 bps, about 0.5% error.

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

Tx Testbench

The testbench is simple.

We tested 2 cases: one frame with parity bit = 0, the other frame with parity bit = 1.



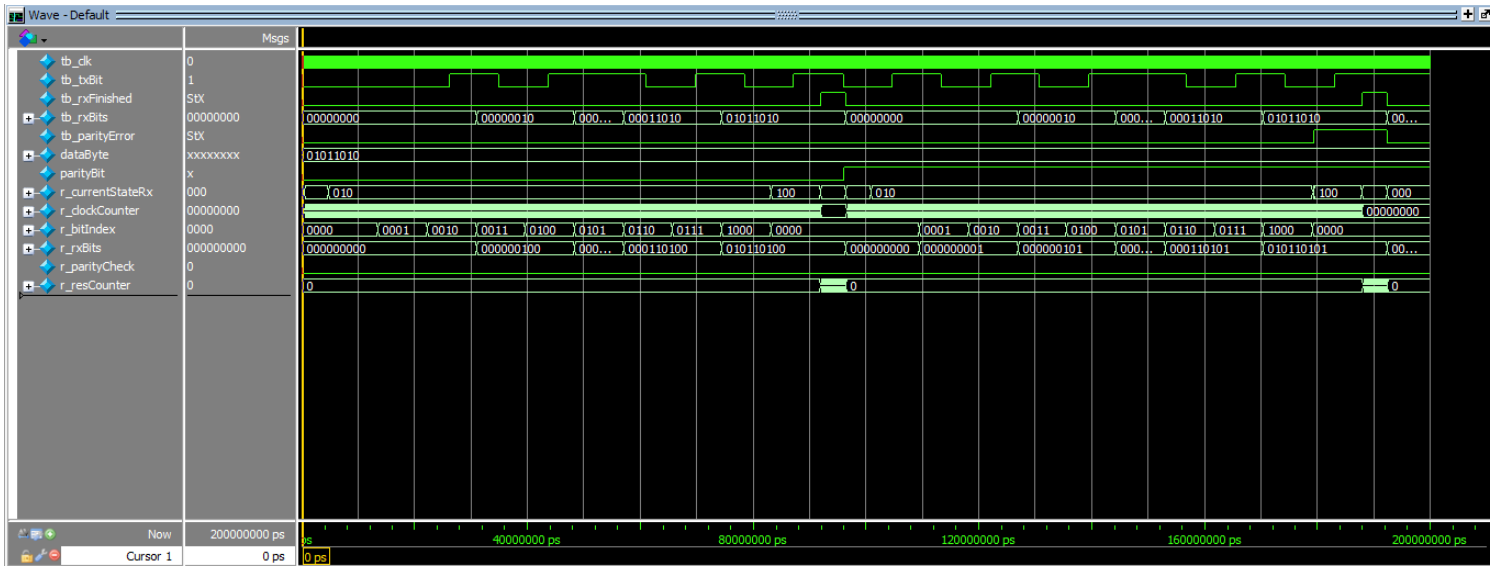

```
# Injecting 0x5A to Tx
# Transmission 1 finished
# Injecting 0x5B to Tx
# Transmission 2 finished
```

[2] ***Note:*** *After future planning, we identified that our original TX module incorrectly sent the parity bit before the data, which does not comply with the standard UART format. We corrected the bit order to follow the standard UART sequence (Start → Data → Parity → Stop), separated the parity bit from the data for clarity, and added a reset input for clean initialization. These changes make the modules more robust, easier to maintain, and fully compatible with PC serial communication. In RX, we improved reliability by adding stop bit validation and clarifying parity handling to accurately detect framing errors.*

Rx Testbench

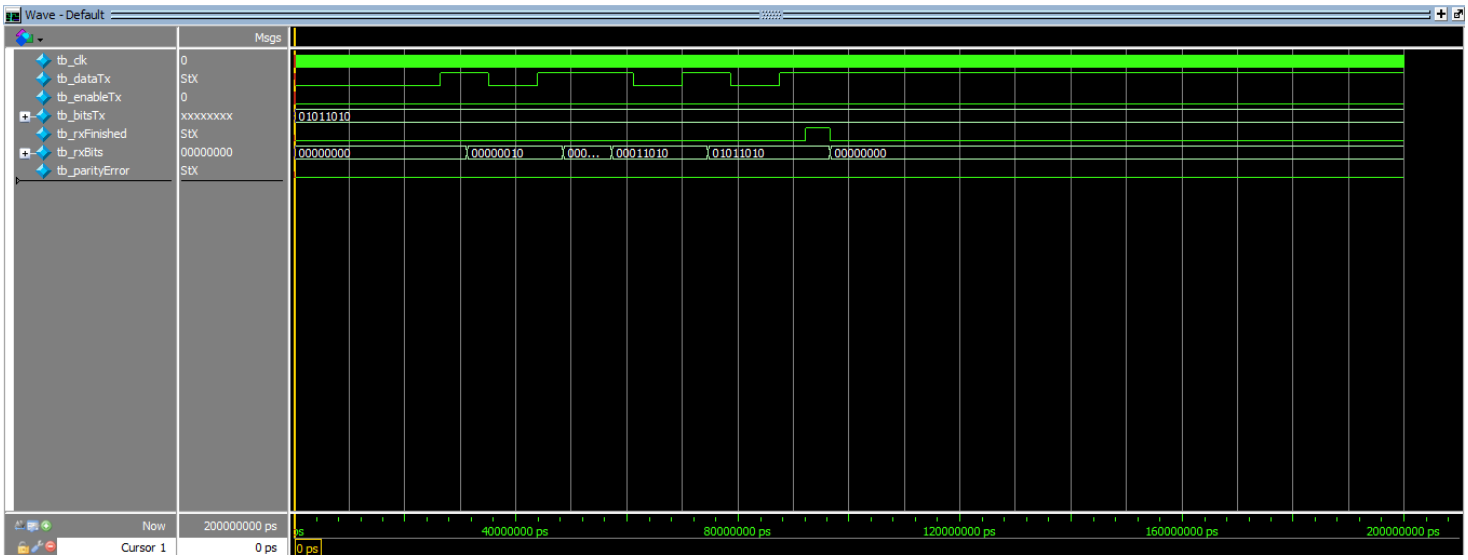
For this testbench, we wrote a task to simulate Tx.

We tested 2 cases: one frame without parity error, the other frame with parity error.




```
# Sending valid UART frame (0x5A)
# Transmit task started at 150000
# Start bit at 150000
# Parity Bit: 0 at time: 8850000
# Bit 0: 0 at time: 17550000
# Bit 1: 1 at time: 26250000
# Bit 2: 0 at time: 34950000
# Bit 3: 1 at time: 43650000
# Bit 4: 1 at time: 52350000
# Bit 5: 0 at time: 61050000
# Bit 6: 1 at time: 69750000
# Bit 7: 0 at time: 78450000
# Stop Bit at 87150000
# Transmit Task Finished at 95850000
# PASS: Recieved 0x5a, Parity Error = 0
# Sending UART frame (0x5A) with parity error
# Transmit task started at 95950000
# Start bit at 95950000
# Parity Bit: 1 at time: 104650000
# Bit 0: 0 at time: 113350000
# Bit 1: 1 at time: 122050000
# Bit 2: 0 at time: 130750000
# Bit 3: 1 at time: 139450000
# Bit 4: 1 at time: 148150000
# Bit 5: 0 at time: 156850000
# Bit 6: 1 at time: 165550000
# Bit 7: 0 at time: 174250000
# Stop Bit at 182950000
# Transmit Task Finished at 191650000
# PASS: Parity error detected
# Test finished
```

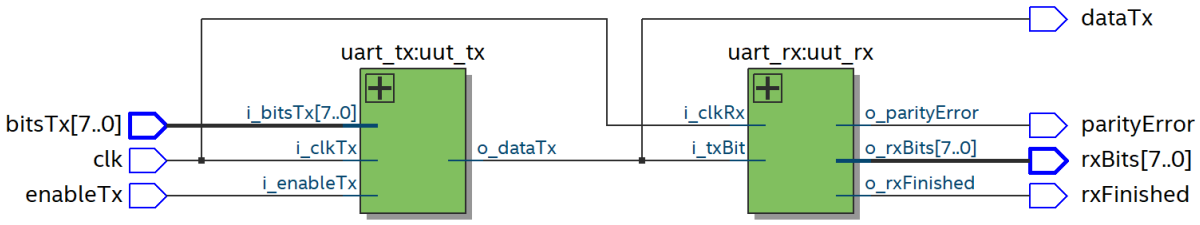
Tx + Rx Testbench

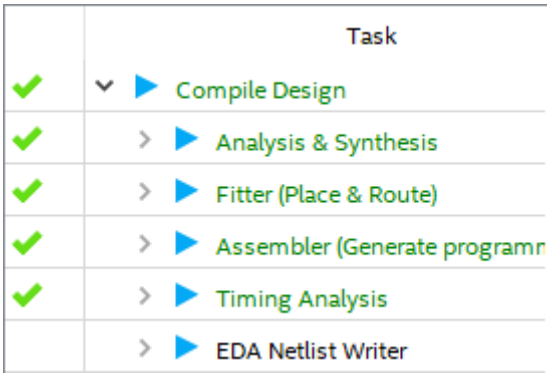



```
# Injecting 0x5A to Tx
# PASS: Recieved 0x5a, Parity Error = 0
# Test finished
```

After seeing that all the simulations are working as intended, we wanted to make sure that our project is synthesized correctly.

For that, we used Quartus Prime and we have also created a top level module (uart_top).





As you can see from the images above, the project was synthesized correctly.

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

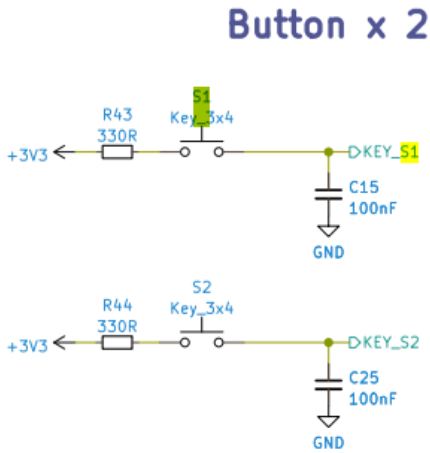
We will be using the Sipeed Tang Nano 20k.

FPGA – Sipeed Tang Nano 20k

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

We implemented UART communication using a full-duplex hardware interface, but with a half-duplex protocol. The PC sends a command to the FPGA via UART RX, and the FPGA responds via UART TX based on the received command. A simple state machine ensures that only one side transmits at a time. 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 enable button and S2 to be the reset button.
Because we are using physical buttons, we need debounce S1, as it has control logic (a single press may generate multiple enable pulses, causing the UART to send the same byte multiple times).
On the other hand, S2 does not need to be debounced (if the button bounces, it just re-triggers reset a few extra times, and our logic stays reset regardless).

It is important to note that the buttons are connected with pull-up resistors, meaning the button is active-low, as shown in the schematics:



So, we need to be careful and not forget to invert the signals.

Currently, our clksPerBit is set to 87. The Tang Nano 20k uses an onboard 27MHz clock. That means that we need to change the clksPerBit to 234 in order to have 115200 baud. If we wanted to keep the 10MHz clock, we would need a different clock source, which is not really necessary for us.

Now we need to change our top module in order to interface between the PC and the FPGA. In order to preserve the original uart_top module (for the synthesis in Quartus Prime), we will create uart_top_fpga, to discern between them.

We will remove some external control and outputs which were used in the testbench, because in our new command-response version, the logic is now fully self-contained inside the FPGA. That means that we must add the omitted ports as internal wires.
We are also modifying dataRx and dataTx to a single bit – UART is a serial protocol and only 1 bit is sent and received.

After synthesizing our code in Gowin IDE, we get this error:

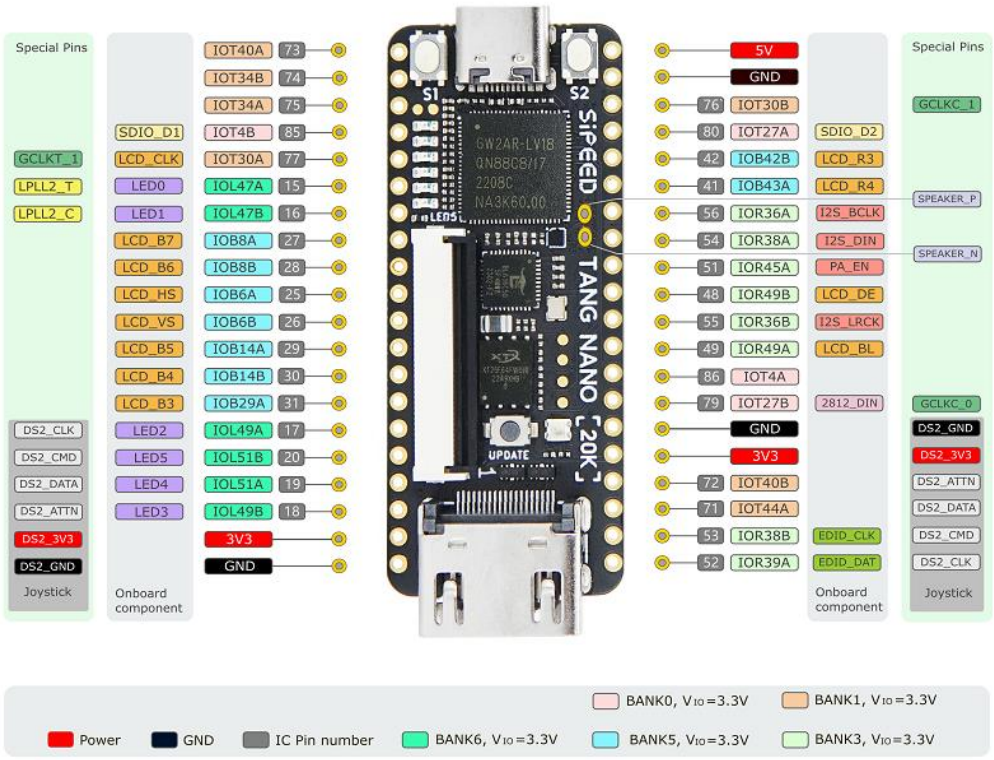
```
ERROR (EX3183): Parameter initial value cannot be omitted in this mode of Verilog
```

It is a common issue caused by trying to assign initial values to reg variables in-line, which is apparently not supported in the Verilog synthesis mode that Gowin uses by default.

In that case, we are going to create a copy of our Rx and Tx files and change them to match Gowin standards.

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

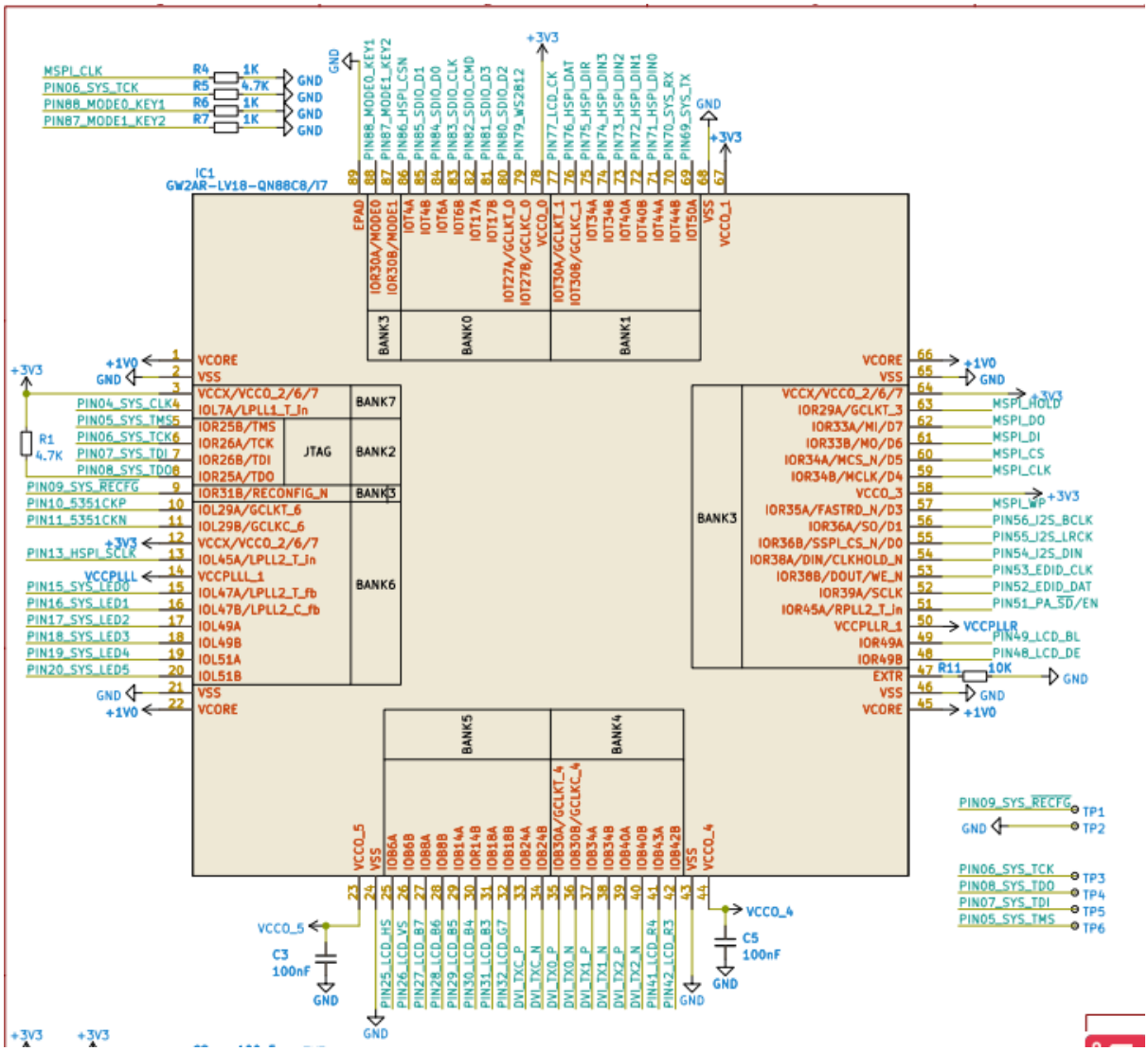
Here is the PIN diagram of the Tang Nano 20k:



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, we came across the schematic of FPGA located here:

下载站 - Sipeed

Downloading the schematic, you can clearly see all the pin numbers and the layout of the board.



Looking at the system clock here, we can see that it is located on pin 4.

Now, using the I/O constraints in the Floor Planner tab in the Gowin IDE, we can 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 constraining, 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 USB-to-JTAG bride.

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

Communication Using Tera Term

Connecting to the FPGA is super simple. Just open up the program and choose Serial, and then the correct port.
Next, go to Setup -> Serial port and choose the correct settings.

Let's go back a bit and remember what our receiver expects to get. If the FPGA receives:

- 'A', it responds with '1' and lights LED 1.
- 'B', it responds with '2' and lights LED 2.
- Anything else, it responds with 'X' and lights LED 3.

So, let's test out all of those responses.

A quick note – In order to send raw hex bytes (like we wanted), you need to create a file that contains one byte (for example, 0x01). In order to create that, you need to use a hex editor (like Notepad++ with Hex Plugin). Because of this "hassle", we will change the UART receiver to expect 'A' and 'B' instead of raw 0x01 and 0x02, as it is much easier for testing, and the responses as well.