实验二 UART 串口的设计与实现

一、 实验目的

1. 掌握基于 SystemVerilog HDL 的时序逻辑电路建模方法
2. 掌握串口的工作原理
2. 掌握计数器设计方法
3. 掌握移位寄存器设计方法
4. 掌握有限状态机的设计方法

二、 实验环境

1. 操作系统：Windows 11
2. 开发环境：Xilinx Vivado 2018.3
3. 硬件平台：远程 FPGA 云实验平台

三、 实验原理

1. 串口通信概述

串口，即串行通信接口，是采用串行通信方式的扩展接口。简言之，串行通信就是指外部设备和计算机之间使用一根数据线将数据逐位按顺序传输，**每次只发送 1 位，每一位数据占据一个固定的时间长度**。相比并行通信方式，串行通信方式的传输速度较慢，但其使用的数据线少（只要一对传输线就可以实现双向通信），在远距离传输中可以节约通信成本。此外，串行通信相比并行通信误码率低，因此得到了广泛应用。

通常，对于具有串行接口的外设而言，计算机与接口间通过总线按并行方式传输，而接口与外设之间按串行方式传输，因此串行通信接口的功能是：在发送时，把计算机送来的并行数据转换成串行数据，然后逐位的发送出去。在接收时，把外设发送过来的串行数据逐位接收，然后组装成并行数据，再送给计算机处理。实现上述功能的电路被称为**串口控制器**。

一般而言，串口通信主要有以下两种方式。

(1) 同步串行通信：是一种采用同步时钟，以串行方式与外设实现数据的交换和通信，也就是说发送方和接收方要严格同步。

(2) 异步串行接口：是一种采用异步方式，具有不规则数据段传输特性的串

行数据传输，发送方和接收方不要求同步。本实验介绍的通用异步收发器 Universal Asynchronous Receiver/Transmitter, UART）就属于这种接口。

2. 通用异步收发器（UART）

UART 是最常见的一种异步串行接口，广泛应用与各类计算机通信领域。一种常见的 UART 物理接口成为 DB9 接口（一种基于 RS232 电平标准的 9 针接口，分为公头和母头），如图 1 所示。

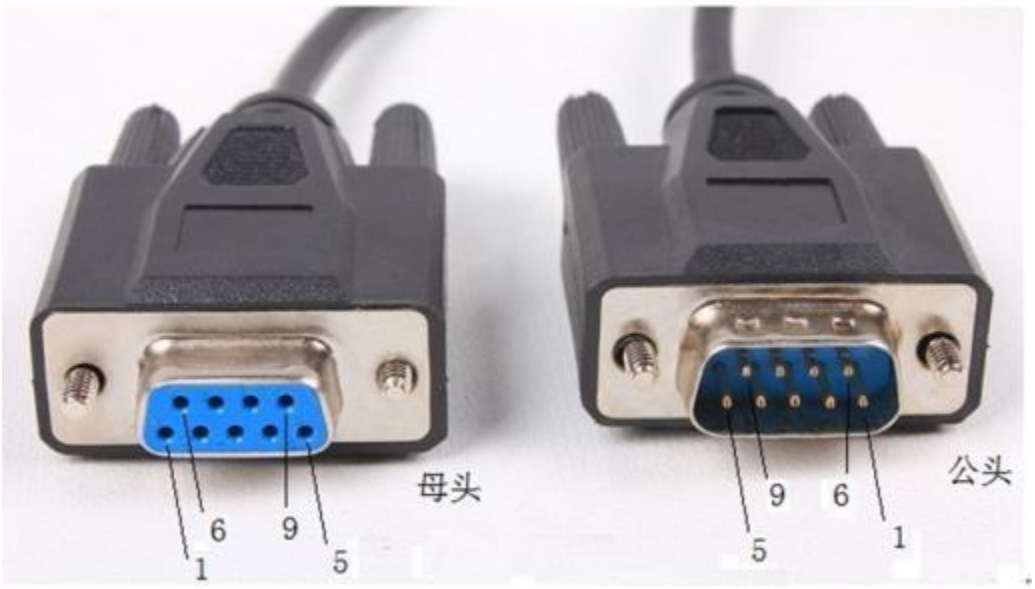


图 1 DB9 串行接口

随着系统的集成度越来越高，已经无法容纳体积“庞大”的串口集成于板卡和 PC 机之上，为了兼容串口信号，现多采用 USB 转 UART 方式实现串口通信。

无论哪种物理接口形式，UART 接口的信号十分简单，只需要两根信号线即可完成全双工的通信（即同时双向传输），分别是 **Tx** 和 **Rx**。其中，Tx 为输出信号，用于发送数据；Rx 为输入信号，用于接收数据，两个信号信均为 1 位宽。

3. UART 传输协议

使用 UART 接口进行通信时，以一个**数据帧**作为单位进行逐位传输。两个数据帧间的传输时间间隔是不固定的，然而在同一个数据帧中的两个相邻比特位间的传输时间间隔是固定的。

当 UART 接口处于空闲状态（非收发状态）时，**数据线始终处于高电平**；当 UART 接口处于收发状态时，一个数据帧通常由 4 部分组成，分别是**起始位**、

数据位、奇偶校验位和停止位，如图 2 所示。

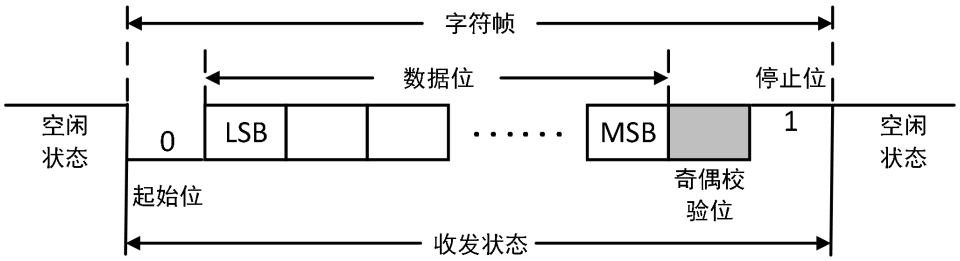


图 2 UART 接口收发数据帧的格式

- **起始位：**发送端发出一个逻辑“0”信号，即低电平，表示一个数据帧传输的开始。接收端接收到这个低电平，表示可以开始接收后续数据。
- **数据位：**紧位于起始位之后，通常由 8 个“0/1”比特位构成一个字符（1 个字节）。传输时，**从数据的最低有效位 LSB 开始传送**。
- **奇偶校验位：**该位是可选的，位于数据位之后，用于判断数据传输是否正确。如果是逻辑“1”的个数为奇数，则是奇校验；反之，则是偶校验。如果不需要校验，则可以不设置该位。
- **停止位：**一个数据帧的结束标志，可以是 1 位或 2 位的逻辑“1”（高电平）。停止位后，UART 接口将处于空闲状态，数据线一直处于高电平。

综上，一个数据帧至少包含 10 位，1 位起始位、8 位数据位、1 位停止位。图 3 给出了传输数据“0x55”（即二进制 01010101）时，数据线上的波形。因为，传输数据从最低有效位开始传送，故实际数据发送顺序为 1-0-1-0-1-0-1-0。

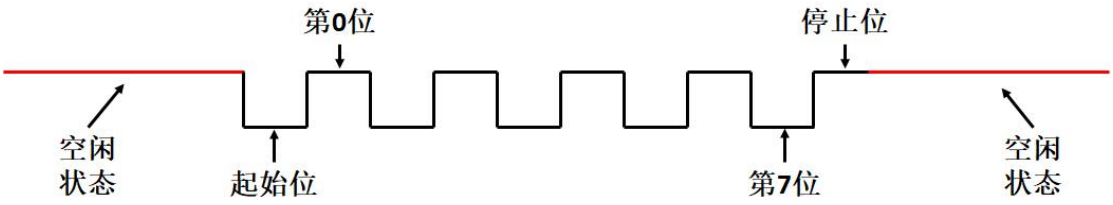


图 3 数据“0x55”的传输波形

采用 UART 进行通信时，由于收发双方不受统一时钟信号控制，因此两者必须提前约定一个传输速率，并按这个速率收发比特位，才能保证数据传输的可靠性。这个速率称为**波特率（Baud rate）**。波特率是对符号传输速率的一种度量。对于 UART 而言，一个符号对应 1 个比特位，故串口波特率就是指每秒所传输的比特数，也可称为**比特率（bits per second, bps）**。

UART 接口常用的波特率包括 2400bps、4800bps、9600bps、19200bps、38400bps、57600bps 等。由于波特率表示的是一种传输速率或传输频率，即单位时间的传输量，故也可用 Hz 表示。如前所述，由于没有共享时钟信号，使用 UART 进行数据传输之前，收发双方必须协商好一个波特率。也就是说，UART 接收端应该知道发送端发送数据的波特率（相应的发送端也需要知道接收段的波特率）。**大多数情况下，发送数据和接收数据的波特率是相同的**。对于波特率为 9600pbs 而言，发送每位数据需要花费（1/9600）= 104.2μs，故传输 8 位将花费 833.6μs。但一个数据帧至少包括 10 位数据，因此，实际花费时间为 1042μs。

四、 实验内容

1. 总体架构

本实验的任务是设计并实现 UART 串口数据的接收模块和发送模块，最后通过**回环测试（loopback 测试）**。所谓回环测试就是发送端发送什么数据，接收端就接收什么数据，这也是常用的一种测试手段。如果回环测试成功，则说明从发送端到接收端之间的数据链路是正常的，以此来验证数据链路的畅通。图 4 给出了回环测试的架构图。

图 4 中左侧为**串口调试助手**，是一个软件工具，它通过串行通信接口以标准通信协议与外部硬件设备进行数据交互，可以接受并显示外部硬件设备发送的数据，以及向外部硬件设备发送控制命令或数据。

图 4 中右侧为顶层模块，它使用 FPGA 提供的 100Mhz 时钟，通过数据线 rxd 和 txd 分别与串口调试助手的数据线 txd 和 rxd 相连。内部的子模块有缓冲区模块，异步接收模块和异步发送模块，其中缓冲区模块简单实现为一个 8 位的寄存器。关于各个模块的内部细节以及控制信号见下文。

整个回环测试流程为：

- 异步接收模块从串口调试助手中接收串行数据，组装成并行数据后输送到缓冲区模块。
- 缓冲区模块再将数据原封不动地输送到异步发送模块。
- 异步发送模块将来自缓冲区的并行数据拆分为串行数据后输送给串行调试助手。

本实验采用代码补全形式完成，仅需补全图 4 中红色矩形（**异步接收模块的状态机和移位寄存器、异步发送模块的状态机和计数器**）的代码，需要补全的位置详见框架代码中的“TODO”注释信息。

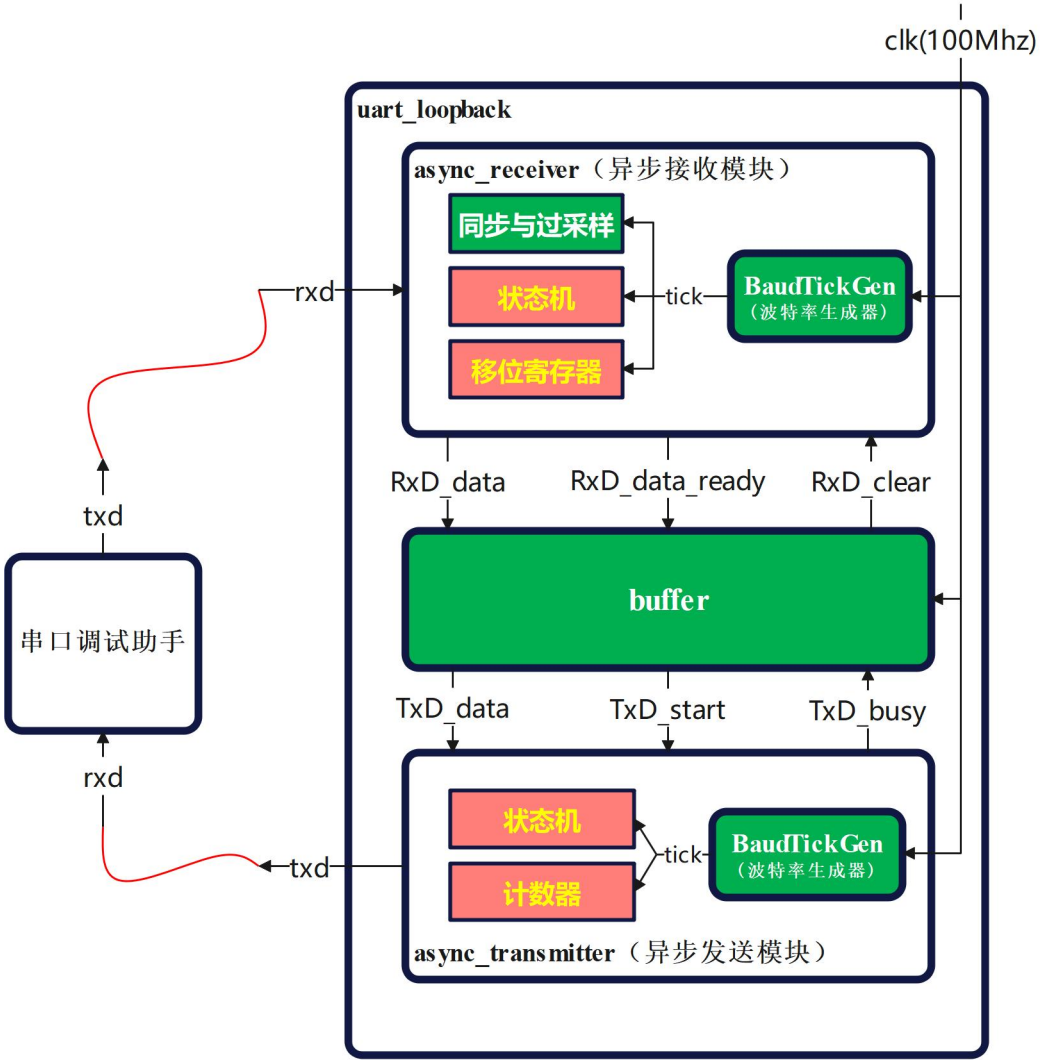


图 4 回环测试的总体架构

2. 波特率生成器模块

表 1 波特率生成器模块端口

端口	位宽	方向	功能
clk	1	input	系统时钟
tick	1	output	标记信号
enable	1	input	使能信号

图 4 中异步接收模块和异步发送模块内部都实例化了波特率生成器模块，简单来说，它就是一个为串行通信“打拍子”的节拍器，确保发送方和接收方以相同的速度来读写每一个数据位。

下面将分段解释代码含义，行号与文件 BaudTickGen.sv 一致

a. 波特率参数定义

```
5  // 参数定义，以下为默认值
6  parameter ClkFrequency = 100000000; // 时钟频率
7  parameter Baud = 9600; // 波特率
8  parameter Oversampling = 1; // 过采样倍数
```

波特率根据时钟频率（ClkFrequency）和指定的波特率（Baud）来生成“节拍”tick 信号，Oversampling 信号与过采样有关，将在接收模块展开阐述。

b. 创建计数器

波特率生成器是基于计数器（Acc）实现的，计数器具有两个属性：位宽（AccWidth）和递增量（Inc），满足等量关系$\frac{ClkFrequency}{Baud} \approx \frac{2^{AccWidth}}{Inc}$，ClkFrequency 和 Baud 已通过参数确定。

计算 AccWidth 考虑了精度的问题：

```
10 // 定义 log2(x) 函数，结果向上取整
11 function integer log2(input integer v); begin log2=0;
   while(v>>log2) log2=log2+1; end endfunction
12
13 // 计算波特率发生器计数器 (Acc) 位宽 (AccWidth)
14 // 基础位宽：log2(ClkFrequency/Baud) 确保能覆盖完整的波特率周期
15 // 额外 8 位：提高时序精度
16 localparam AccWidth = log2(ClkFrequency/Baud)+8;
17 logic [AccWidth:0] Acc = 0; // 定义计数器并初始化为 0
```

计算 Inc 时，运用了防溢出的技巧，并且对结果进行了四舍五入：

```
19 // 计算计数器递增量 (Inc)
20 // 计算公式为 Inc = ((Baud * Oversampling) * 2^AccWidth) / ClkFrequency
21 // 为防止 ((Baud * Oversampling) << AccWidth) 溢出，首先计算一个安全的右移位数
22 // 原理：通过预先右移 (31 - AccWidth) 位，确保 (Baud * Oversampling) 在放大前被缩小，
23 //       从而为后续左移 (AccWidth - ShiftLimiter) 位留出足够的位宽空间。
24 localparam ShiftLimiter = log2(Baud*Oversampling >> (31-AccWidth));
25 // 采用缩放法避免除法溢出：将分子分母同比例缩放后再进行整数除法
26 // "+(ClkFrequency>>(ShiftLimiter+1))"目的是实现四舍五入以减少误差
27 localparam Inc = ((Baud*Oversampling << (AccWidth-ShiftLimiter))
   +(ClkFrequency>>(ShiftLimiter+1)))/(ClkFrequency>>ShiftLimiter);
```

c. 更新计数器

计数器实际有效位宽为[AccWidth - 1:0]，最高位是进位标记位，当产生进位时，最高位为 1，tick 信号有效。

```
// 计数器，enable 信号有效时才更新计数器
// tick 赋值为计数器的最高位
always_ff @(posedge clk) if(enable) Acc <= Acc[AccWidth-1:0] + Inc[AccWidth-1:0];
else Acc <= Inc[AccWidth-1:0];
assign tick = Acc[AccWidth];
```

波特率生成器模块代码已全部提供，但是上述实现过程考虑了很多因素，非常值得学习。

3. 发送模块

表 2 发送模块端口

接口	位宽	方向	功能
clk	1	input	系统时钟
TxD_start	1	input	发送端开始工作标志位
TxD_data	8	input	待发送的数据
TxD	1	output	串行输出
TxD_busy	1	output	发送端忙碌标志位

发送模块的核心功能在于完成数据帧的封装与串行化输出：将待发送数据（TxD_data）按照通信协议组装为包含起始位与停止位的完整数据帧，并依据预设波特率将其转换为串行数据流（TxD），实现逐位顺序发送。

下面将分段解释代码含义，行号与文件 async_transmitter.sv 一致

a. 实例化波特率生成器模块

这里用到了 ifdef 条件编译，在文件第 8 行取消注释后，宏定义 SIMULATION 生效，波特率时钟将与系统时钟相等，每个系统时钟周期将发送 1bit 数据。如不取消注释，则波特率时钟由波特率生成器模块生成，实例化模块时，可以传入参数 ClkFrequency 和 Baud。这样设计是方便在仿真环境和实际场景间快速切换，在仿真环境下，仿真速度快，波形图短，便于调试。

仅当发送模块正在发送数据时，波特率生成器才工作，并且确保每次发送数据帧时，波特率时钟都从初始状态开始，从而避免帧间干扰，因此 enable 端口连接 TxD_busy 信号。

```
parameter ClkFrequency = 100000000; // 100MHz
parameter Baud = 9600;

/* -------------- 波特率时钟生成控制 -------------- */
`ifdef SIMULATION
// 仿真环境下：为加速仿真，每个时钟周期输出一个位
logic BitTick = 1'b1;  // output one bit per clock cycle
`else
// 实际硬件：使用精确的波特率发生器，按目标波特率生成标记信号
logic BitTick;
BaudTickGen #(ClkFrequency, Baud)
tickgen(.clk(clk), .enable(TxD_busy), .tick(BitTick));
`endif
/* ---------------------------------------------- */
```

b. 状态机（需要补充）

通过维护状态机，确定当前发送的位是起始位、停止位还是数据位（如果是数据位，还需要确定是第几个数据位），至少需要 11 个状态：1 个空闲状态（IDLE）和 10 个非空闲状态（1 个起始状态、1 个停止状态和 8 个数据状态），请思考状态转换图并补全代码实现。 **仅在 bit_tick 有效的时钟周期，状态机才会发生状态转移，确保波特率同步。**

```
/* -------------- 发送端状态机控制逻辑 -------------- */
// 功能描述：
//    - TxD_state: 发送状态机寄存器，定义串行传输的各个阶段
//    - TxD_ready: 状态机就绪标志，指示可接收新数据传输请求
//    - TxD_busy:  发送忙标志，指示当前正处于数据传输状态
// 状态机特性：
//    - 空闲状态(IDLE): 等待 TxD_start 启动信号，准备接收新数据，编码为 4'b0000
//    - 起始状态(START): 空闲状态下 TxD_start 有效时更新，编码为 4'b0100;
//    - 状态转移条件: 在 BitTick 有效时推进传输状态，确保波特率同步
//    - 同步设计: TxD_start 为同步信号，在时钟上升沿采样有效
logic [3:0] TxD_state = 4'b0;  // 发送状态机状态寄存器
logic TxD_ready;               // 发送器就绪标志
assign TxD_ready = (TxD_state == 4'b0);  // 状态 0 为空闲就绪状态
assign TxD_busy  = ~TxD_ready;           // 非空闲状态均为忙状态
always_ff @(posedge clk)
```

```
begin
    case(TxD_state)
        4'b0000: if(TxD_start) TxD_state <= 4'b0100; // 空闲状态 -> 起始状态
        // TODO: 补充状态机代码
        //       已经给出了空闲状态到起始状态的转换
        //       状态寄存器的位宽以及状态编码允许自由修改
        default: if(BitTick) TxD_state <= 4'b0000;
    endcase
end
/* ------------------------------------------------ */
```

c. 计数器（需要补充）

维护计数器来遍历数据帧的各个数据位，即用计数器的值来索引数据位。维护的难点在于更新计数器的条件，它与状态机当前状态密切相关。最终需要根据计数器的值和状态机当前状态确定 TxD 的值。

```
/* -------------- 发送数据帧缓存与位计数器 -------------- */
// 功能描述：

//   - bit_cnt: 位索引计数器，用于遍历数据帧的各个数据位
// 设计要点：
//   - 在 TxD_ready 和 TxD_start 同时有效时重置计数值
//   - 位计数器随状态机推进，为串行数据输出提供位选择索引
// TODO: 补充计数器代码
// TODO: TxD 除了发送数据位，也要发送起始位和停止位
//  assign TxD = ...;
/* ---------------------------------------- */
```

4. 接收模块

表 3 接收模块端口

接口	位宽	方向	功能
clk	1	input	系统时钟
RxD	1	input	串行输入
RxD_data_ready	1	output	数据有效标志
RxD_clear	1	input	数据清空标志
RxD_data	8	output	最终接收的并行数据

接收模块的核心功能是按规定的波特率速度接收来自串口调试助手的串行数据（RxD），并按照通信协议对数据进行解析，将其转换为并行数据（RxD_data）。但接收模块比发送模块更复杂，主要体现在**过采样、同步和滤波**操作。

下面将分段解释代码含义，行号与文件 async_receiver.sv 一致。

a. 过采样

在接收模块中，需要进行过采样操作。下面解释过采样的原理：这里先引入一个概念“位周期”，即一个数据位在总线上的持续时间。图 5 为常规采样

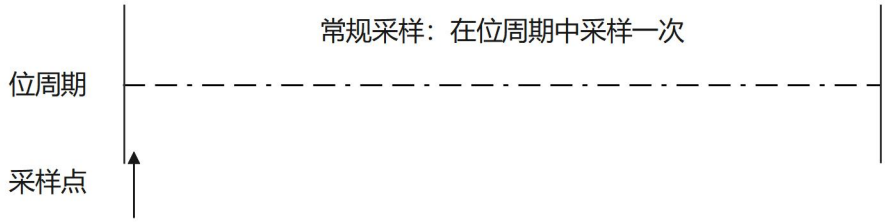


图 5 常规采样示意图

而过采样顾名思义就是采样频率比波特率更高，这体现在代码中的过采样倍数 Oversampling，本实验设置为 8，在一个位周期内采样了 8 次，见图 6.

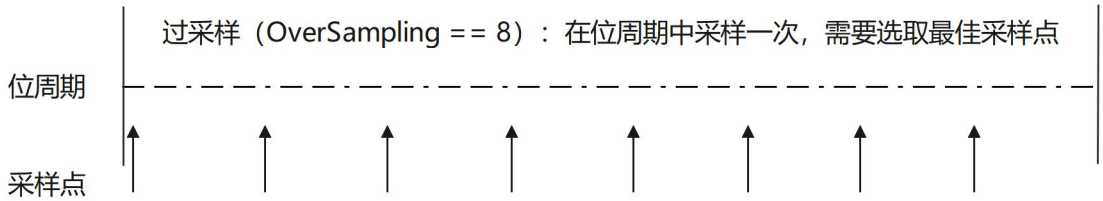


图 6 过采样示意图

最佳的采样时刻处于中间位置：第 4 个或者第 5 个采样点。在提供的代码框架中，基于计数器选取第 4 个采样点作为最佳采样点。

过采样在工程中的作用是**抗干扰**：

- 一方面，起始位的检测是接收过程的起点，其准确性至关重要，传输线路上的毛刺或噪声可能产生一个短暂的负脉冲，若只采样一次容易被误判为起始位。
- 另一方面，在数据位中心进行采样是可靠的，这是因为一个位的电平在位的中心区域是最稳定的，边沿区域则因信号跳变而充满不确定性。

为了方便切换仿真环境，加快仿真速度，这部分代码也采用了 ifdef 条件编译。　在仿真环境下，不需要进行过采样和后续的同步和滤波操作。

```
// 串口接收过采样倍数配置
parameter Oversampling = 8;  // 必须设置为 2 的幂次方
// 过采样原理：以固定倍率在一个位周期内对 RxD 信号进行多次采样，并确保在最佳时刻捕获每个数据位
// 默认 8 倍过采样，可设置为 16 倍以获得更高的接收质量（抗噪能力更强）

/* -------------- 波特率时钟生成控制 & 输入处理 -------------- */
`ifdef SIMULATION
// 仿真环境下：为加速仿真，每个时钟周期接收一个位，忽略同步滤波和过采样
logic RxD_bit; assign RxD_bit = RxD;     // 直接使用 RxD 输入，无同步滤波
logic sampleNow = 1'b1; // 持续采样，全速运行
`else

// 实际硬件：使用精确的波特率发生器，包含同步、滤波和过采样
// ---------- 第一部分：波特率时钟生成 ----------
logic OversamplingTick;
BaudTickGen #(ClkFrequency, Baud, Oversampling) tickgen(.clk(clk), .enable(1'b1), .tick(OversamplingTick));
```

选取第四个采样点作为最佳采样点，选取过程基于计数器实现，为了避免接收每个数据帧之间的干扰，在空闲时刻复位过采样计数器

```
// ---------- 第二部分：过采样时序控制 ----------
logic [3:0] RxD_state = 0; // 接收端现态，初始化为 0
// 计算过采样计数器位宽
function integer log2(input integer v); begin log2=0; while(v>>log2) log2=log2+1; end endfunction
localparam l2o = log2(Oversampling);

// 过采样计数器：在每个位周期内计数过采样点数
logic [l2o-2:0] OversamplingCnt = 0;
always_ff @(posedge clk) if(OversamplingTick) OversamplingCnt <= (RxD_state==0) ? 1'd0 : OversamplingCnt + 1'd1;

// 采样时刻判断：在位周期的中间时刻采样（最佳采样点）
logic sampleNow;logic sampleNow = OversamplingTick && (OversamplingCnt==Oversampling/2-1);
```

b. 同步

在串口调试助手中，输入的 RxD 信号与接收端时钟 clk 处于异步时钟域关系。为实现 RxD 信号从慢速时钟域（串口调试助手）到快速时钟域（接收端）的可靠传输，需**通过两位移位寄存器进行同步处理，通常称为“打两拍”**。如图 5

所示，若 adata 为源自慢速时钟域 aclk 的异步信号，在进入快速时钟域 bclk 前，需连续经过两个 bclk 控制的寄存器进行采样，从而获得稳定的同步信号 bdata。若仅采用单级同步，则因亚稳态风险无法保证信号的稳定性和系统可靠性。

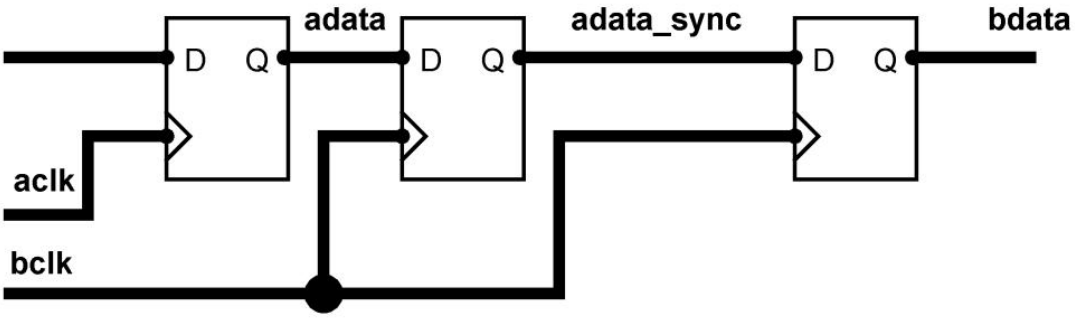


图 5 电平同步电路

用一个位宽为 2 的寄存器来实现“打两拍”

```
// ---------- 第三部分：输入同步 ----------
// 同步 RxD 到系统时钟域，防止亚稳态
logic [1:0] RxD_sync = 2'b11;
always_ff @(posedge clk) if(OversamplingTick) RxD_sync <= {RxD_sync[0], RxD};
```

c. 滤波

滤波是指在位周期内，对过采样的多个采样值进行分析，确定最终采样值。注意区分“采样点”和“采样值”，前者强调时刻，后者强调结果。虽然进行了过采样，但仍不能直接选取某一个采样点的值来作为最终采样值，这是因为信号在传输过程中可能受到噪声干扰，导致个别采样点出现错误。即使直接选择中点采样，一旦这个点恰好受到噪声干扰，就会导致整个数据位采样错误。

为了抵抗这种瞬时噪声，提高系统的鲁棒性，我们通过以下机制进行滤波：

- 根据过采样中各个采样点的信号来更新滤波计数器（Filter_cnt），更新逻辑见图 6（更新逻辑类似状态机）

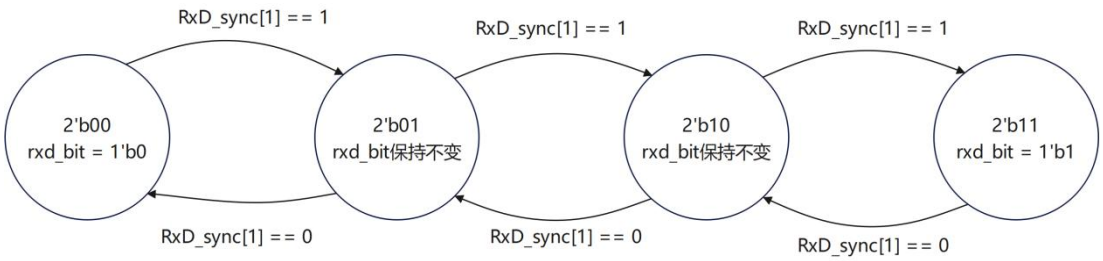


图 6 滤波计数器更新逻辑

- 根据滤波计数器的值来决定最终采样值（类似根据状态机的输出逻辑）
  - 若 Filter_cnt == 2'b11，RxD_bit <= 1'b1
  - 若 Filter_cnt == 2'b00，RxD_bit <= 1'b0
  - 若 Filter_cnt == 2'b01 或 2'b10，RxD_bit 不变

仅在 Filter_cnt == 2’ b11 时更新 RxD_bit 为 1；在 Filter_cnt == 2’ b00 时更新 RxD_bit 为 0；其余情况下，RxD_bit 保持原来的值。

```
// ---------- 第四部分：数字滤波（抗噪声） ----------
// 采用递增/递减计数器实现数字滤波，消除 RxD 线上的毛刺和噪声
// 原理：通过多次采样饱和机制，只有连续多个相同电平才被确认为有效信号
logic [1:0] Filter_cnt = 2'b11; // 2 位滤波计数器，初始值为 2'b11
logic RxD_bit = 1'b1;   // 滤波后的最终 RxD 位输出
always_ff @(posedge clk)
if(OversamplingTick)    // 仅在过采样时钟节拍处进行滤波处理
begin
    // ===== 滤波计数器更新逻辑 =====
    // 情况 1：当前同步后的 RxD 为高电平(1)，且计数器未达到上限(3)
    // 说明检测到高电平，逐步增加计数器（向高电平方向积累）
    if(RxD_sync[1]==1'b1 && Filter_cnt!=2'b11)
       Filter_cnt <= Filter_cnt + 1'd1;
    // 情况 2：当前同步后的 RxD 为低电平(0)，且计数器未达到下限(0)
    // 说明检测到低电平，逐步减少计数器（向低电平方向积累）
    else if(RxD_sync[1]==1'b0 && Filter_cnt!=2'b00)
       Filter_cnt <= Filter_cnt - 1'd1;
    // 当计数器积累到最大值(3)时，确认为稳定的高电平
    if(Filter_cnt==2'b11) RxD_bit <= 1'b1;
    // 当计数器减少到最小值(0)时，确认为稳定的低电平
    else if(Filter_cnt==2'b00) RxD_bit <= 1'b0;
    // 注意：当计数器值为 1 或 2 时，保持之前的 RxD_bit 值不变，从而过滤掉不稳定的噪声
end
```

d. 状态机（需要补充）

通过维护状态机，确定当前发送的位是起始位、停止位还是数据位（如果是数据位，还需要确定是第几个数据位）。状态寄存器已经在过采样中定义并初始化了（40 行），在实现状态机过程中，**必须使用同步、滤波后的 RxD_bit 信号和过采样得到的最佳采样点 sampleNow 来更新状态机**。

状态机一共有 11 个状态：1 个空闲状态和 10 个非空闲状态（1 个起始状态、8 个数据状态和 1 个停止状态），请思考状态转换图并通过代码实现状态机。

```
/* ----------------- 状态机 ----------------- */
// 功能描述：
//       - RxD_bit: 经过同步和过滤之后的数据
//       - sampleNow: 最佳采样点
//       - RxD_state: 状态寄存器
// 设计要点：
//       - 仿真环境下：检测到起始位的同时，也接收起始位
//       - 现实环境下：区分检测起始位与接收起始位，仅在最佳采样点更新状态
    always_ff @(posedge clk)
    case(RxD_state)
          4'b0000: if(~RxD_bit) RxD_state <= `ifdef SIMULATION 4'b1000 `else 4'b0001 `endif;  // 检测起始位
          4'b0001: if(sampleNow) RxD_state  <= 4'b1000;    // 接收起始位
          // TODO: 补充状态机代码
          //         已给出检测起始位和接收起始位的代码
          //         状态寄存器的位宽以及状态编码允许自由修改
          default: if(sampleNow) RxD_state  <= 4'b0000;
    endcase
/* ------------------------------------------ */
```

e. 移位寄存器（需要补充）

接收模块的核心功能之一是将串行数据组装为并行数据，这可以通过移位寄存器实现，根据当前状态更新移位寄存器，请思考移位方向并通过代码实现移位寄存器。

```
/* ----------------- 移位寄存器 ----------------- */
// TODO: 补充移位寄存器代码（数据并行化）
//       RxD_bit 信号经过了同步和过滤，请使用 RxD_bit 信号而不是 RxD 信号
//       此后，在时钟上升沿，sampleNow 信号有效时才根据状态更新移位寄存器
/* ------------------------------------------- */

// TODO: output logic
//       RxD_data 为移位寄存器的值
```

f. 数据有效标志信号（需要修改）

并行数据输出 RxD_data 在帧组装过程中处于中间状态，不可使用。仅当完整数据帧接收完毕（即成功采样到停止位，**4'b0010 为停止状态的编码，需要根据自定义的编码进行修改！！！**），数据才被视为有效。此时， RxD_data_ready 信号置 1，指示 RxD_data 可供读取。该有效状态将一直保持，直至外部电路发

出确认信号 RxD_clear 将其清除，准备下一次传输。

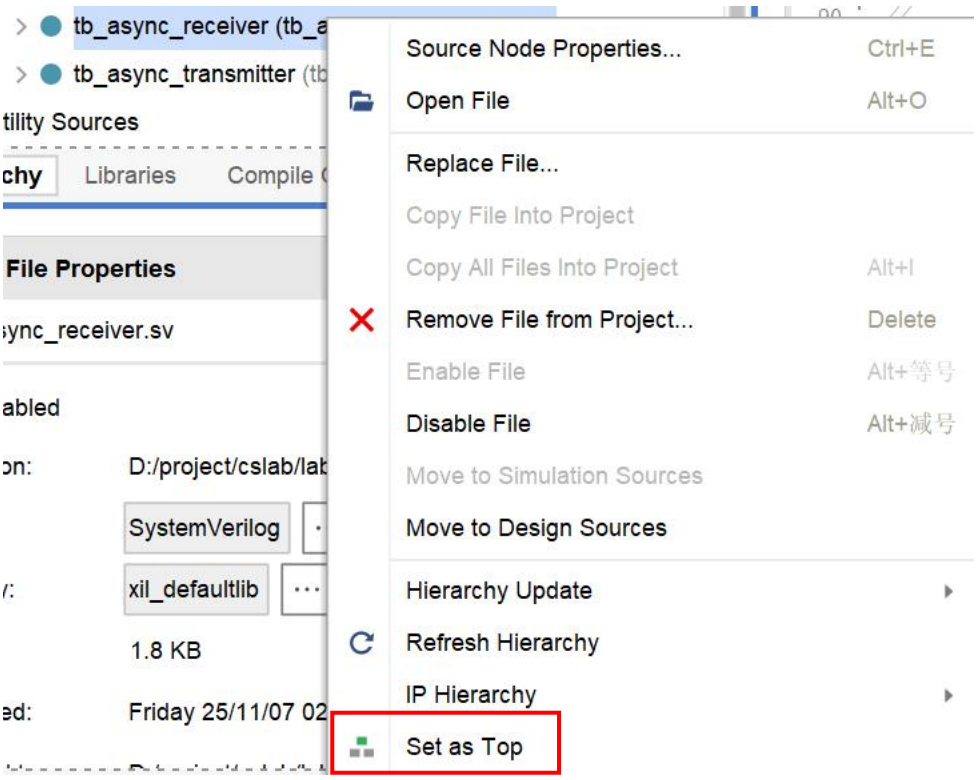
```
always_ff @(posedge clk)
begin
    if(RxD_clear)
        RxD_data_ready <= 0;
    else begin
        // 确保暂停位被接收，4'b0010 为停止状态的编码，需要根据自定义的编码进行修改！！！
        RxD_data_ready <= RxD_data_ready | (sampleNow && RxD_state==4'b0010 && RxD_bit);
    end
end
```

五．实验步骤（建议）

1. 解压缩 UART.zip，打开提供的工程文件 UART.xpr，目前工程中已经提供了 uart_loopback.sv、async_receiver.sv、async_transmitter.sv、BaudTickGen.sv 和顶层文件 top.v。模块的端口定义不要进行任何修改，只需根据 TODO 注释对模块内部进行补充，即可完成 UART 串口设计。

2. 仿真测试文件已经提供，一共有两个以 tb 为前缀的文件分别用于测试接收模块和发送模块（tb_async_receiver.sv、tb_async_transmitter.sv）。

完成设计后，若要测试接收模块，则将 tb_async_receiver.sv 设置为顶层仿真模块（右键点击 tb_async_receiver.sv，选择 Set as Top，等待其 Update 完毕）



若要测试发送模块，则将 tb_async_transmitter.sv 设置为顶层仿真模块（右键点击 tb_async_transmitter.sv，选择 Set as Top，等待其 Update 完毕）

为了缩短仿真时间，简化波形图，在仿真前取消 async_receiver.sv、async_transmitter.sv、tb_async_receiver.sv、tb_async_transmitter 文件中的 SIMULATION 宏定义的注释，让宏定义生效

```
// The RS-232 settings are fixed
// TX: 8-bit data, 2 stop, no-parity
// RX: 8-bit data, 1 stop, no-parity (the receiver can accept more stop bits of course)

`define SIMULATION    // in this mode, TX outputs one bit per clock cycle
                      // and RX receives one bit per clock cycle (for fast simulations)
```

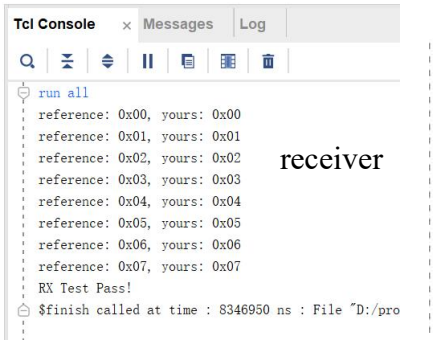
```
1  `timescale 1ns / 1ps
2  `define SIMULATION    // in this mode, TX outputs one bit per clock cycle
3                        // and RX receives one bit per clock cycle (for fast simulations)
```

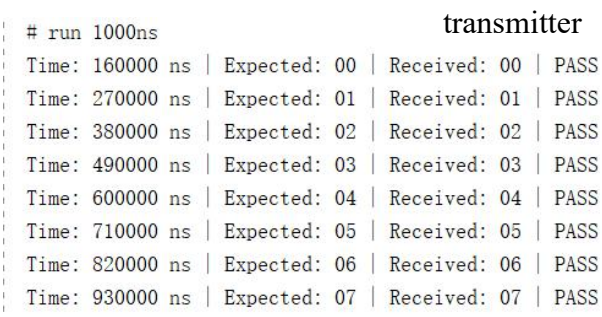
工程中提供的两个仿真代码分别对发送模块和接收模块进行了仿真测试，对于发送端，模拟发送数字 0x00 - 0x07

对于接收模块则是模拟接收数字 0x00 - 0x07，并且仿真时会在控制台打印接收到的数字和预期数字进行对比，仿真正确后，会打印提示词。对于发送模块，

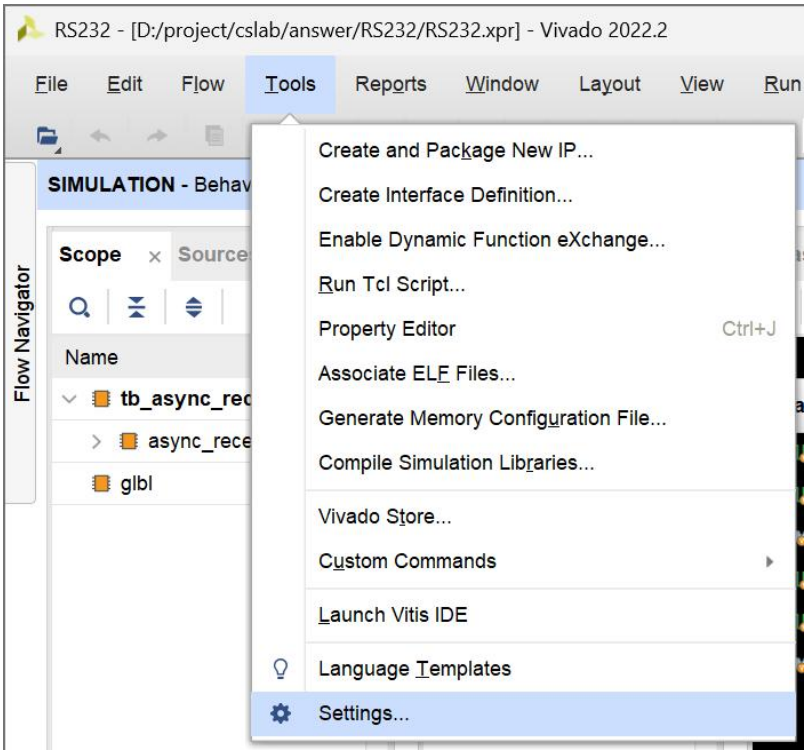
控制台会打印发送的信号和实际接收的信号，如果一致，则发送模块实现正确。

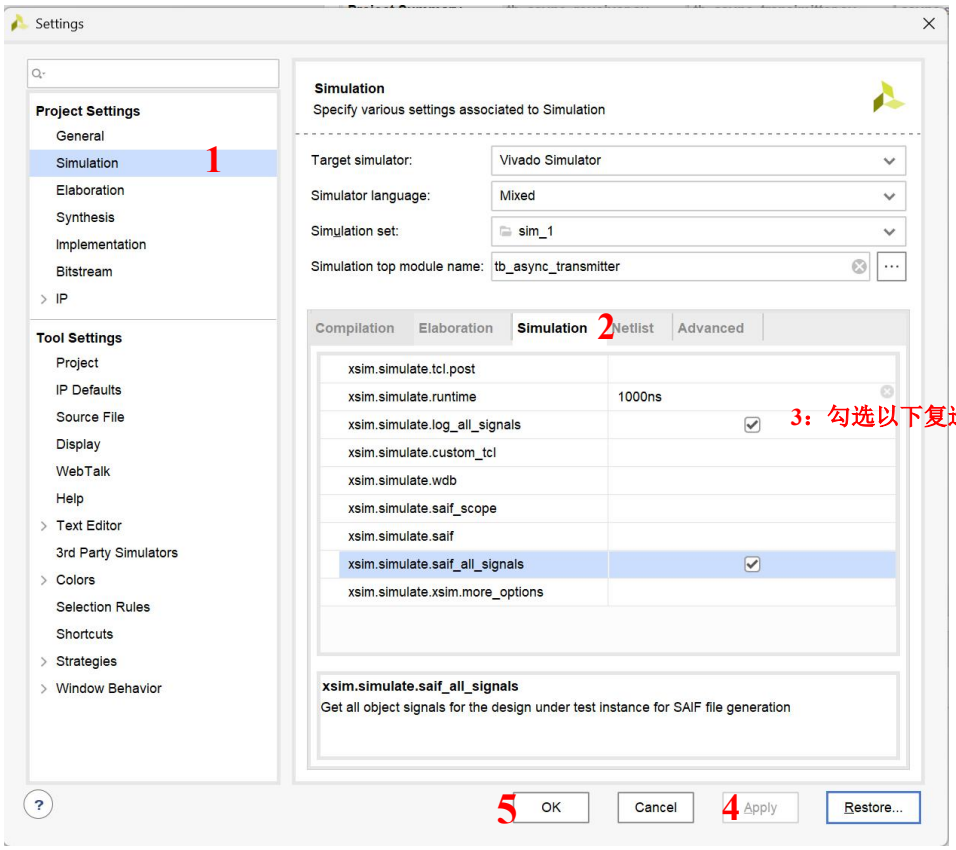
仿真波形图见**《Lab2-WAVE.pdf》**





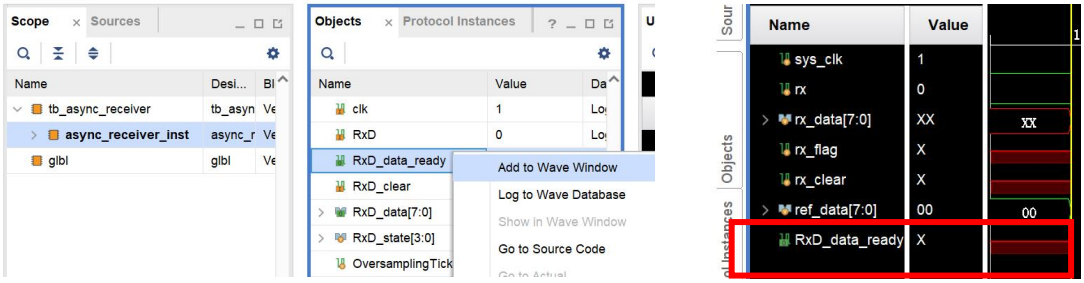
这里有一个很有用的小 Tips：先进入设置



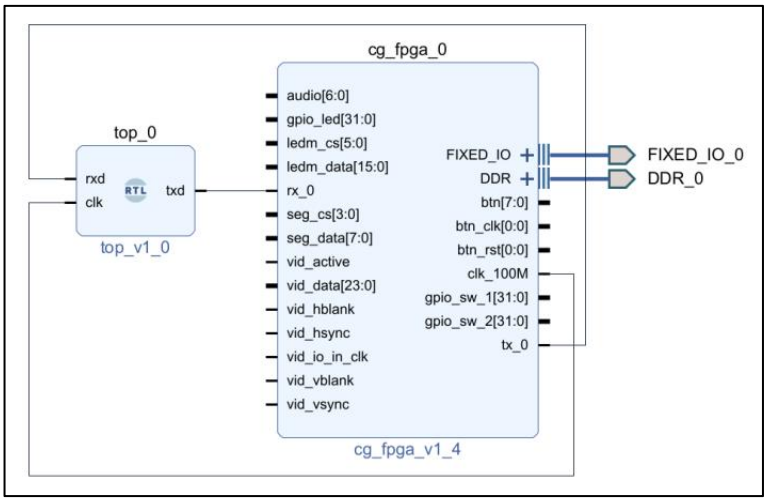


之后仿真时可以拖动任一信号观察波形

例如，添加 async_receiver_inst 模块的 RxD_data_ready 信号，只需选中后，右键 Add to Wave Window，即可在波形图中看到，而无需重新仿真。



3. 仿真无误后，进行 Block Design



4. 对工程进行综合、实现并烧写 bit 流。单击 PROGRAM AND DEBUG->Generate Bitstream，在弹出的窗口中选择 OK，此过程需等待数分钟。

5. 登录希冀平台，进入 RS-232 实验。连接远程开发版并选择 bit 流。bit 流文件位于工程文件目录下的 RS232.runs/impl_1/design_1_wrapper.bit。烧写完成后，单击串口通信设置中的 Open 摁钮打开串口调试窗，选中消息窗口设置中的 ASCII Mode，在发送消息窗口处输入英文数字和符号的组合，点击发送即可在串口通信消息窗口中看到同样的消息。

六．实验方式

1. 每位同学独立上机编程实验，实验指导教师指导。

七． 参考内容

1. 教材内容和课件

八． 实验报告

1. 画出发送模块和接受模块中的状态转换图。
2. 发送模块和接收模块中补充的 Verilog 代码。
3. 远程 FPGA 平台验证的截图。

九. 附加题

1. 在接收模块中，为什么 RxD 信号要同步和过滤，为什么要进行过采样？

2. 在发送模块中，输出 TxD 信号是基于计数器实现的，你可以改为用移位寄存器来实现吗；在接收模块中，输出 RxD_data 信号是基于移位寄存器实现的，你可以改为用计数器来实现吗？