第30章　串口 RS232

30.1 章节导读

通用异步收发传输器（Universal Asynchronous Receiver/Transmitter），通常称作 UART。UART 是一种通用的数据通信协议，也是异步串行通信口（串口）的总称，它在发送数据时将并行数据转换成串行数据来传输，在接收数据时将接收到的串行数据转换成并行数据。它包括了 RS232、RS499、RS423、RS422 和 RS485 等接口标准规范和总线标准规范。

本章节我们会带领读者进行串口 RS232 相关知识的学习，通过理论与实践，最终设计并实现基于 RS232 的串口收、发功能模块，并完成串口数据回环实验。

30.2 理论学习

30.2.1 串口简介

串口作为常用的三大低速总线（UART、SPI、IIC）之一，在设计众多通信接口和调试时占有重要地位。但 UART 和 SPI、IIC 不同的是，它是异步通信接口，异步通信中的接收方并不知道数据什么时候会到达，所以双方收发端都要有各自的时钟，在数据传输过程中是不需要时钟的，发送方发送的时间间隔可以不均匀，接受方是在数据的起始位和停止位的帮助下实现信息同步的。而 SPI、IIC 是同步通信接口（后面的章节会做详细介绍），同步通信中双方使用频率一致的时钟，在数据传输过程中时钟伴随着数据一起传输，发送方和接收方使用的时钟都是由主机提供的。

UART 通信只有两根信号线，一根是发送数据端口线叫 tx（Transmitter），一根是接收数据端口线叫 rx（Receiver），如图 30-1 所示，对于 PC 来说它的 tx 要和对于 FPGA 来说的 rx 连接，同样 PC 的 rx 要和 FPGA 的 tx 连接，如果是两个 tx 或者两个 rx 连接那数据就不能正常被发送出去和接收到，所以不要弄混，记住 rx 和 tx 都是相对自身主体来讲的。UART 可以实现全双工，即可以同时进行发送数据和接收数据。

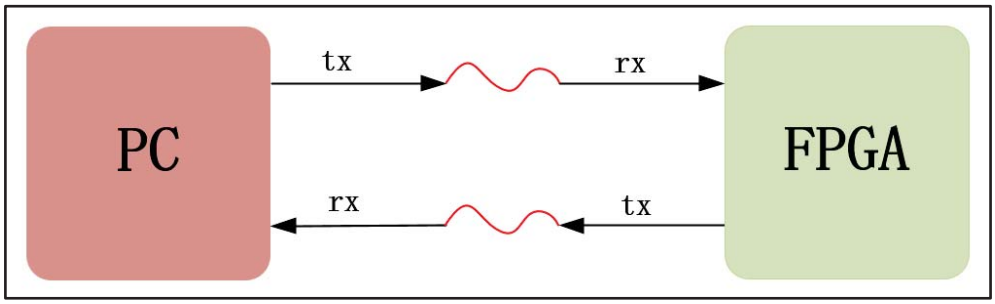

图 30-1　串口通信连接图

我们的任务是设计 FPGA 部分接收串口数据和发送串口数据的模块，最后我们把两个

模块拼接起来，其结构如图 30-2 所示，最后通过 loopback 测试（回环测试）来验证设计模块的正确性。所谓 loopback 测试就是发送端发送什么数据，接收端就接收什么数据，这也是非常常用的一种测试手段，如果 loopback 测试成功，则说明从数据发送端到数据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。

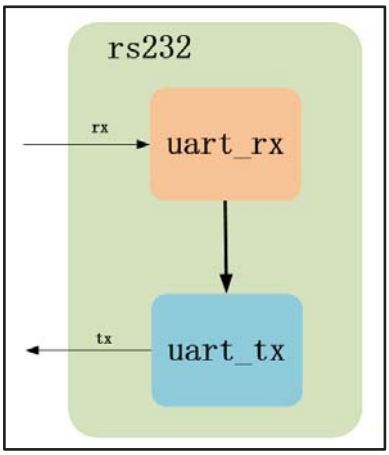

图 30-2 串口回环模块框图

串口 RS232 传输数据的距离虽然不远，传输速率也相对较慢，但是串口依然被广泛的用于电路系统的设计中，串口的好处主要表现在以下几个方面：

1、很多传感器芯片或 CPU 都带有串口功能，目的是在使用一些传感器或 CPU 时可以通过串口进行调试，十分方便；

2、在较为复杂的高速数据接口和数据链路集合的系统中往往联合调试比较困难，可以先使用串口将数据链路部分验证后，再把串口换成高速数据接口。如在做以太网相关的项目时，可以在调试时先使用串口把整个数据链路调通，然后再把串口换成以太网的接口；

3、串口的数据线一共就两根，也没有时钟线，节省了大量的管脚资源。

30.2.2 RS-232 信号线

在最初的应用中，RS-232 串口标准常用于计算机、路由与调制调解器(MODEN，俗称“猫”)之间的通讯，在这种通讯系统中，设备被分为数据终端设备 DTE(计算机、路由)和数据通讯设备 DCE(调制调解器)。我们以这种通讯模型讲解它们的信号线连接方式及各个信号线的作用。

在旧式的台式计算机中一般会有 RS-232 标准的 COM 口(也称 DB9 接口)，见图 30-3。

图 30-3 电脑主板上的 COM 口及串口线

其中接线口以针式引出信号线的称为公头，以孔式引出信号线的称为母头。在计算机中一般引出公头接口，而在调制调解器设备中引出的一般为母头，使用上图中的串口线即可把它与计算机连接起来。通讯时，串口线中传输的信号使用 RS-232 标准调制。在各种应用场合下， DB9 接口中的公头及母头的各个引脚的标准信号线接法见图 30-4。

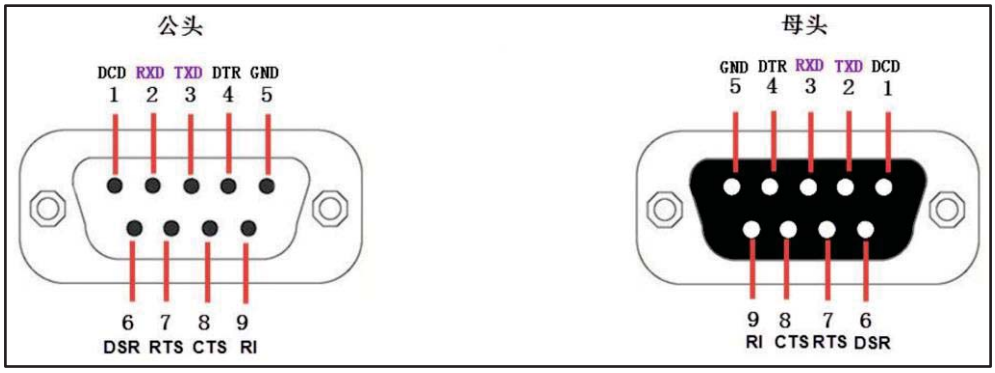

图 30-4 DB9 标准的公头及母头接法

序号	名称	符号	数据方向	说明
1	载波检测	DCD	DTE→DCE	Data Carrier Detect，数据载波检测，用于 DTE 告知对方，本机是否收到对方的载波信号
2	接收数据	RXD	DTE←DCE	Receive Data，数据接收信号，即输入 。
3	发送数据	TXD	DTE→DCE	Transmit Data，数据发送信号，即输出。两个设备之间的 TXD 与 RXD 应交叉相连
4	数据终端(DTE)就绪	DTR	DTE→DCE	Data Terminal Ready，数据终端就绪，用于 DTE 向对方告知本机是否已准备好
5	信号地	GND	-	地线，两个通讯设备之间的地电位可能不一样，这会影响收发双方的电平信号，所以两个串口设备之间必须要使用地线连接，即共地。
6	数据设备(DCE)就绪	DSR	DTE←DCE	Data Set Ready，数据发送就绪，用于 DCE 告知对方本机是否处于待命状态
7	请求发送	RTS	DTE→DCE	Request To Send，请求发送， DTE 请求 DCE 本设备向 DCE 端发送数据
8	允许发送	CTS	DTE←DCE	Clear To Send，允许发送，DCE 回应对方的 RTS 发送请求，告知对方是否可以发送数据
9	响铃指示	RI	DTE←DCE	Ring Indicator，响铃指示，表示 DCE 端与线路已接通

图 30-5 DB9 信号线说明

图 30-5 是计算机端的 DB9 公头标准接法，由于两个通讯设备之间的收发信号(RXD 与 TXD)应交叉相连，所以调制调解器端的 DB9 母头的收发信号接法一般与公头的相反，两个设备之间连接时，只要使用“直通型”的串口线连接起来即可，见图 30-6。

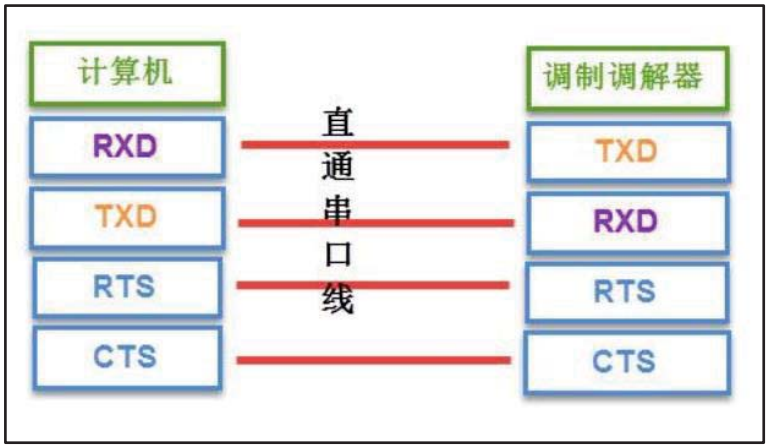

图 30-6 计算机与调制调解器的信号线连接

串口线中的 RTS、CTS、DSR、DTR 及 DCD 信号，使用逻辑 1 表示信号有效，逻辑 0 表示信号无效。例如，当计算机端控制 DTR 信号线表示为逻辑 1 时，它是为了告知远端的调制调解器，本机已准备好接收数据， 0 则表示还没准备就绪。

在目前的其它工业控制使用的串口通讯中，一般只使用 RXD、TXD 以及 GND 三条信号线，直接传输数据信号。而 RTS、CTS、DSR、DTR 及 DCD 信号都被裁剪掉了，如果您在前面被这些信号弄得晕头转向，那就直接忽略它们吧。

30.2.3 RS232 通信协议简介

1、RS232 是 UART 的一种，没有时钟线，只有两根数据线，分别是 rx 和 tx，这两根线都是 1bit 位宽的。其中 rx 是接收数据的线，tx 是发送数据的线。

2、rx 位宽为 1bit，PC 机通过串口调试助手往 FPGA 发 8bit 数据时，FPGA 通过串口线 rx 一位一位地接收，从最低位到最高位依次接收，最后在 FPGA 里面位拼接成 8 比特数据。

3、tx 位宽为 1bit，FPGA 通过串口往 PC 机发 8bit 数据时，FPGA 把 8bit 数据通过 tx 线一位一位的传给 PC 机，从最低位到最高位依次发送，最后上位机通过串口助手按照 RS232 协议把这一位一位的数据位拼接成 8bit 数据。

4、串口数据的发送与接收是基于帧结构的，即一帧一帧的发送与接收数据。每一帧除了中间包含 8bit 有效数据外，还在每一帧的开头都必须有一个起始位，且固定为 0；在每一帧的结束时也必须有一个停止位，且固定为 1，即最基本的帧结构（不包括校验等）有 10bit。在不发送或者不接收数据的情况下，rx 和 tx 处于空闲状态，此时 rx 和 tx 线都保持高电平，如果有数据帧传输时，首先会有一个起始位，然后是 8bit 的数据位，接着有 1bit 的停止位，然后 rx 和 tx 继续进入空闲状态，然后等待下一次的数据传输。如图 30-7 所示为一个最基本的 RS232 帧结构。

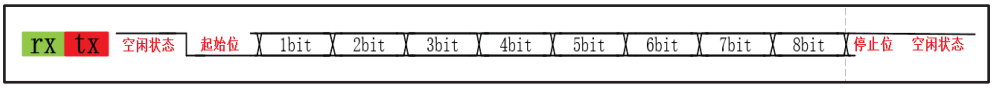

图 30-7 RS232 帧结构

5、波特率：在信息传输通道中，携带数据信息的信号单元叫码元（因为串口是 1bit 进行传输的，所以其码元就是代表一个二进制数），每秒钟通过信号传输的码元数称为码元的传输速率，简称波特率，常用符号“Baud”表示，其单位为“波特每秒（Bps）”。串口常见的波特率有 4800、9600、115200 等，我们选用 9600 的波特率进行串口章节的讲解。

6、比特率：每秒钟通信信道传输的信息量称为位传输速率，简称比特率，其单位为“每秒比特数（bps）”。比特率可由波特率计算得出，公式为：比特率=波特率 * 单个调制状态对应的二进制位数。如果使用的是 9600 的波特率，其串口的比特率为：9600Bps * 1bit= 9600bps。

7、由计算得串口发送或者接收 1bit 数据的时间为一个波特，即 1/9600 秒，如果用 50MHz（周期为 20ns）的系统时钟来计数，需要计数的个数为 cnt = (1s * 10^9)ns / 9600bit)ns / 20ns ≈ 5208 个系统时钟周期，即每个 bit 数据之间的间隔要在 50MHz 的时钟频率下计数 5208 次。

8、上位机通过串口发 8bit 数据时，会自动在发 8 位有效数据前发一个波特时间的起始位，也会自动在发完 8 位有效数据后发一个停止位。同理，串口助手接收上位机发送的数据前，必须检测到一个波特时间的起始位才能开始接收数据，接收完 8bit 的数据后，再接收一个波特时间的停止位。

30.3 实战演练

30.3.1 实验目标

设计并实现基于串口 RS232 的数据收、发模块，使用收、发模块，完成串口数据回环实验。

30.3.2 硬件资源

本次实验我们需使用到开发板上的 RS232 收发器芯片，RS232 收发器电路如图 30-8 所示。

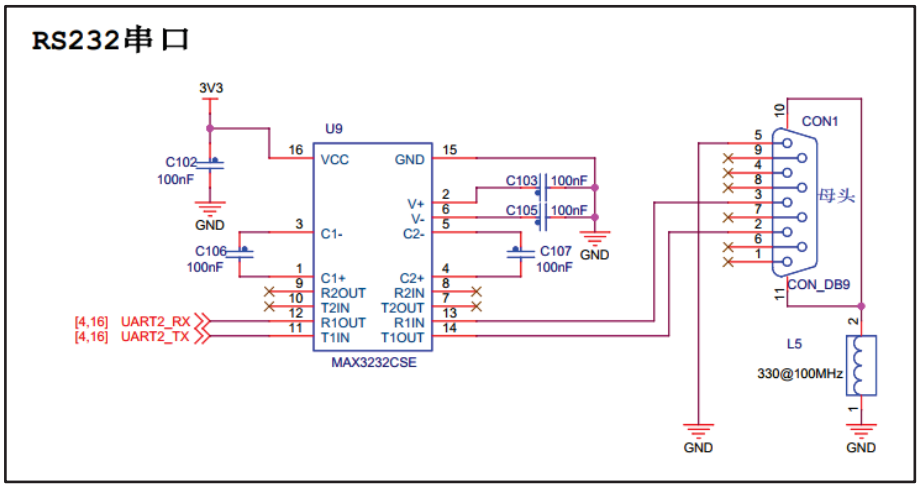

图 30-8 RS3232 收发器电路图

如图 30-8 所示，MAX3232 为 RS232 收发器芯片。由于 RS-232 电平标准的信号不能直接被控制器直接识别，所以这些信号会经过一个“电平转换芯片”转换成控制器能识别的“TTL”的电平信号，才能实现通讯。

根据通讯使用的电平标准不同，串口通讯可分为 TTL 标准及 RS-232 标准，见表格 30-1。

表格 30-1 TTL 电平标准与 RS232 电平标准

通信标准	电平标准（发送端）
TTL	逻辑 1：3.3V 逻辑 0：0V
RS-232	逻辑 1：-15V~-5V 逻辑 0：+5V~+15V

由于 FPGA 串口输入输出引脚为 TTL 电平， 用 3.3V 代表逻辑“1”， 0V 代表逻辑“0”；所以常常会使用 MA3232 芯片对 TTL 及 RS-232 电平的信号进行互相转换。同时开

发板中还搭载了 USB 转串口的芯片 CH340，可方便大家使用 USB 线进行串口调试，如图 30-9 所示。

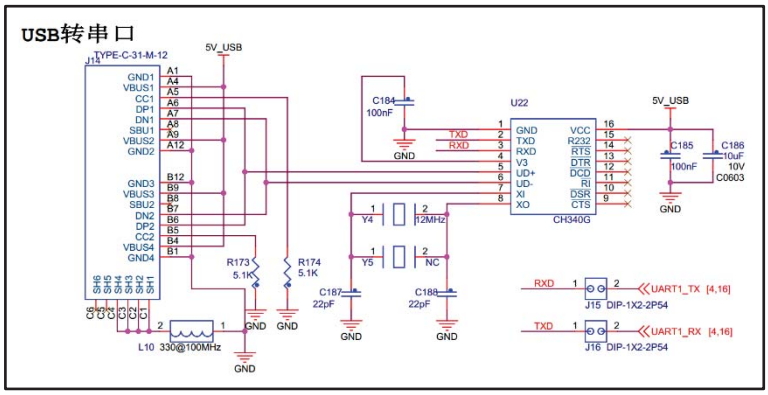

图 30-9 USB 转串口电路图

如图 30-9 所示。在使用时需将 J2、J3 口的 1、2 脚用跳帽连接起来才能正常使用，即开发板上的 J2、J3 中的 TXD 与 RX 短接、RXD 与 TX 短接

30.3.3 程序设计

1. 整体说明

实验工程整体框图，具体见图 30-10。

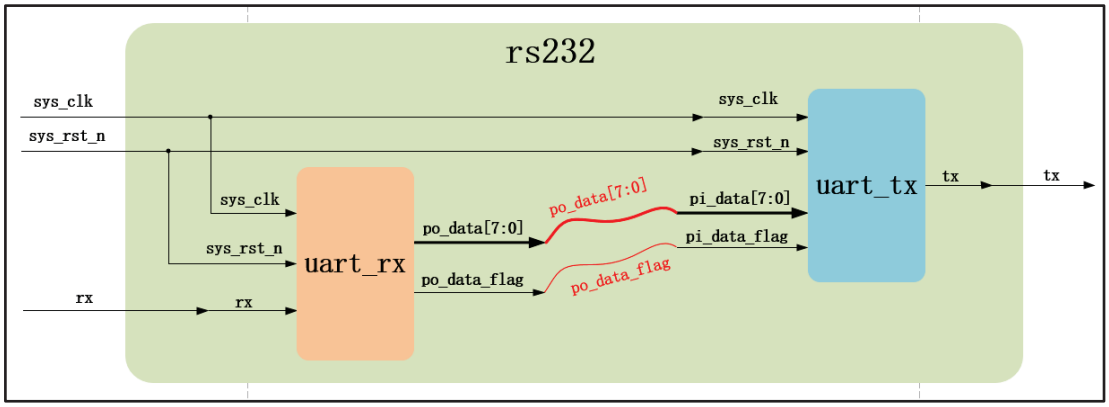

图 30-10 工程整体框图

通过上图可以看到，该工程共分 3 个模块。各模块简介见表格 30-2。

表格 30-2 rs232 工程模简介

模块名称	功能描述
uart_rx	串口数据接收模块
uart_tx	串口数据发送模块
rs232	顶层模块

下面分模块为大家讲解。

2. 串口数据接收模块

我们先设计串口接收模块，该模块的功能是接收通过 PC 机上的串口调试助手发送的固定波特率的数据，串口接收模块按照串口的协议准确接收串行数据，解析提取有用数据后需将其转化为并行数据，因为并行数据在 FPGA 内部传输的效率更高，转化为并行数据后同时产生一个数据有效信号标志信号伴随着并行的有效数据一同输出。

注：为什么还需要输出一个伴随并行数据有效的标志信号，这是因为后级模块或系统在使用该并行数据的时候可能无法知道该时刻采样的数据是不是稳定有效的，而数据有效标志信号的到来就说明数据才该时刻是稳定有效的，起到一个指示作用。当数据有效标志信号为高时，该并行数据就可以被后级模块或系统使用了。

模块框图

我们将串口接收模块取名为 uart_rx，根据功能简介我们对整个设计要求有了大致的了解，其中设计的关键点是如何将串行数据转化为并行数据，也就是如何正确接收串行数据的问题。PC 机通过串口调试助手发过来的信号没有时钟，所以 FPGA 在接收数据的时候要约定好一个固定的波特率，一个比特一个比特地接收数据，我们选择的波特率为 9600bps，也是 RS232 接口中相对较慢的一种速率。

整个模块肯定需要用到时序逻辑，所以先设计好时钟 sys_clk 和复位 sys_rst_n 两个输入信号，其次是相对于 FPGA 的 rx 端接收 PC 机通过串口调试助手发送过来的 1bit 输入信号。输出信号一个是 FPGA 的 rx 端接收到的数据转换成的 8bit 并行数据 po_data，另一个是 8bit 并行数据有效的标志信号 po_data_flag。

根据上面的分析设计出的 Visio 框图如图 30-11 所示。

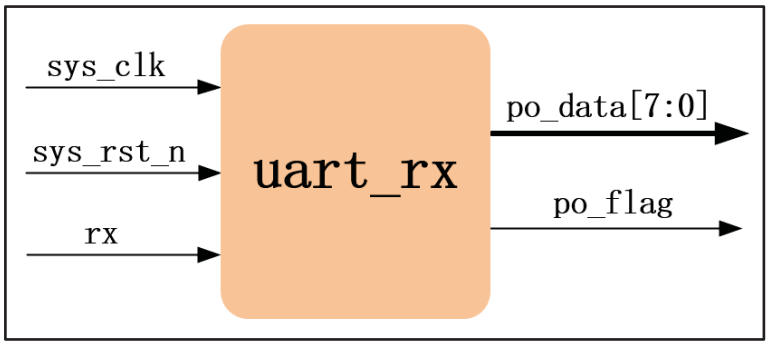

图 30-11 串口接收模块框图

端口列表与功能总结如表格 30-3 所示。

表格 30-3 串口接收模块输入输出信号描述

信号	位宽	类型	功能描述
sys_clk	1Bit	Input	工作时钟，频率 50MHz
sys_rst_n	1Bit	Input	复位信号，低电平有效
rx	1Bit	Input	串口接收信号
po_data	8Bit	Output	串口接收后转成的 8bit 数据
po_data_flag	1Bit	Output	串口接收后转成的 8bit 数据有效标志信号

波形设计

如图 30-12 所示，我们先把实现 uart_rx 功能整体的波形图列出，然后再详细介绍下面的波形是如何一步步设计实现的。

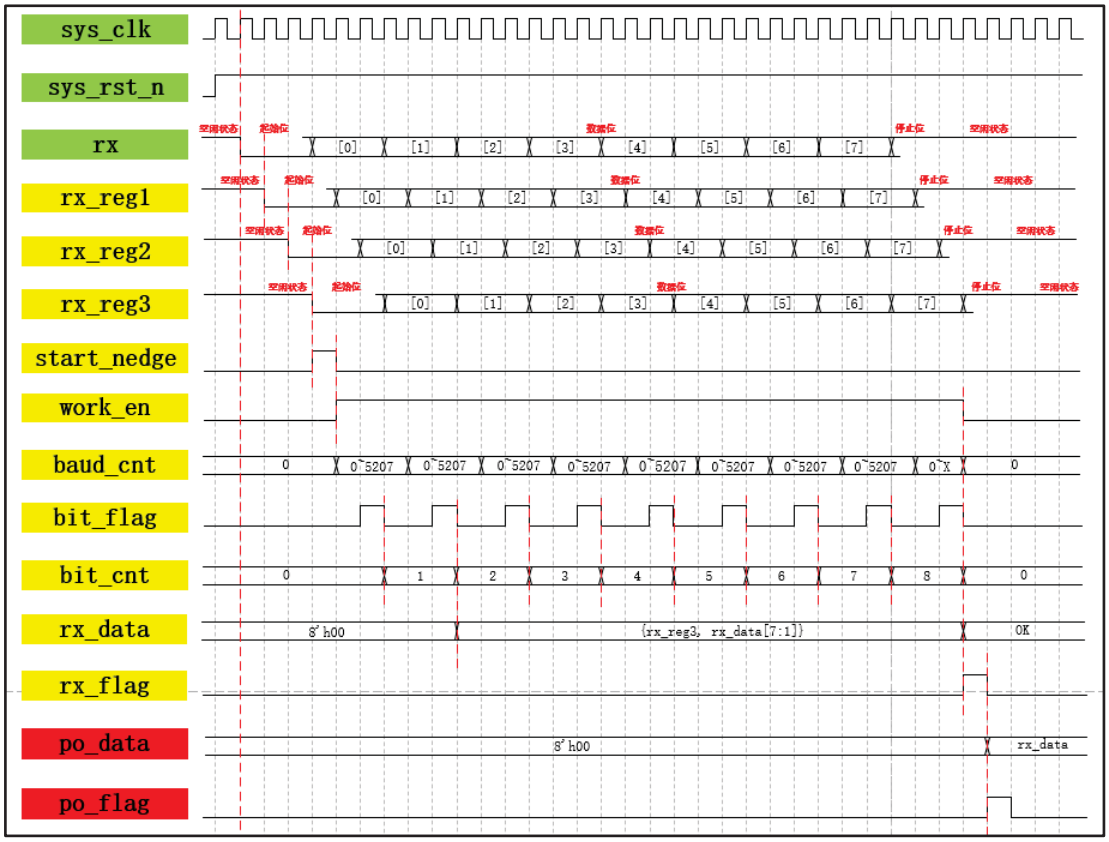

图 30-12 串口接收模块波形图

波形设计思路详细解析

第一部分：首先画出三个输入信号，必不可少的两个输入信号是时钟和复位，另一个是串行输入数据 rx，如图 30-13 所示，我们发现 rx 串行数据一开始直接打了两拍，就是经过了两级寄存器，理论上我们应该按照串口接收数据的时序要求找到 rx 的下降沿，然后开始接收起始位的数据，但为什么先将数据打了两拍呢？那就要先从跨时钟域会导致“亚稳态”的问题上说起。

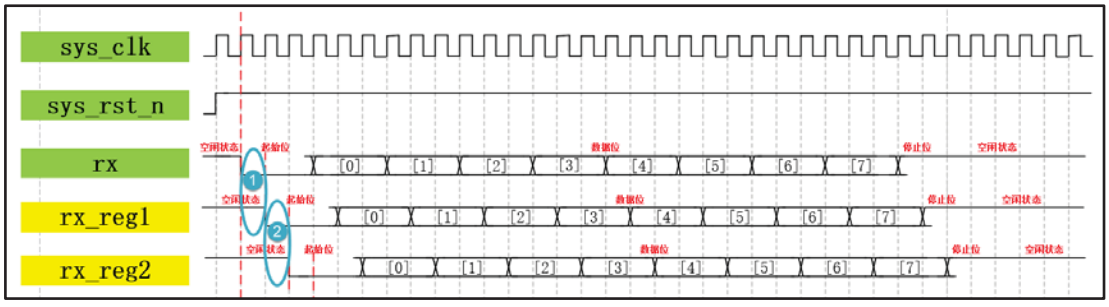

图 30-13 数据打拍波形图

大家都一定使用过示波器，当你使用示波器把一个矩形脉冲的上升沿或下降沿放大后会发现其上升沿和下降沿并不是瞬间被拉高或拉低的，而是有一个倾斜变化的过程，这在运放中被称为“压摆率”，如果 FPGA 的系统时钟刚好采集到 rx 信号上升沿或下降沿的中间位置附近（按照概率来讲，如果数据传输量足够大或传输速度足够快时一定会产生这种情况），即 FPGA 在接收 rx 数据时不满足内部寄存器的建立时间 Tsu（指触发器的时钟信号上升沿到来以前，数据稳定不变的最小时间）和保持时间 Th（指触发器的时钟信号上升沿到来以后，数据稳定不变的最小时间），此时 FPGA 的第一级寄存器的输出端在时钟沿到来之后比较长的一段时间内都处于不确定的状态，在 0 和 1 之间处于振荡状态，而不是等于串口输入的确定的 rx 值。

如图 30-14 所示为产生亚稳态的波形示意图，rx 信号经过 FPGA 中的第一级寄存器后输出的 rx_reg1 信号在时钟上升沿 Tco 时间后会有 Tmet（决断时间）的振荡时段，当第一个寄存器发生亚稳态后，经过 Tmet 的振荡稳定后，第二级寄存器就能采集到一个相对稳定的值。但由于振荡时间 Tmet 是受到很多因素影响的，所以 Tmet 时间有长有短。如图 30-15 所示，当 Tmet1 时间长到大于一个采样周期后，那第二级寄存器就会采集到亚稳态，但是从第二级寄存器输出的信号就是相对稳定的了。当然会人会问到第二级寄存器的 Tmet2 的持续时间会不会继续延长到大于一个采样周期？这种情况虽然会存在，但是其概率是极小的，寄存器本身就有减小 Tmet 时间让数据快速稳定的作用。

由于在 PC 机中波特率和 rx 信号是同步的，而 rx 信号和 FPGA 的系统时钟 sys_clk 是异步的关系，我们此时要做的是将慢速时钟域（PC 机中的波特率）系统中的 rx 信号同步到快速时钟域（FPGA 中的 sys_clk）系统中，所使用的方法叫电平同步，俗称“打两拍法”。所以 rx 信号进入 FPGA 后会首先经过一级寄存器，出现如图 30-14 所示的亚稳态现象，导致 rx_reg1 信号的状态不确定是 0 还是 1，就会受其影响使其他相关信号做出不同的判断，有的判断到“0”有的判断到“1”，有的也进入了亚稳态并产生连锁反应，导致后级相关逻辑电路混乱。为了避免这种情况，rx 信号进来后首先进行打一拍的处理，打一拍后产生 rx_reg1 信号。但 rx_reg1 可能还存在低概率的亚稳态现象，为了进一步降低出现亚稳态的概率，我们将从 rx_reg1 信号再打一拍后产生 rx_reg2 信号，使之能够较大概率保证 rx_reg2 信号是 0 或者 1 中的一种确定情况，这样 rx_reg2 所影响的后级电路就都是相对稳定的了。但是大家一定要注意：打两拍后虽然能让信号稳定到 0 或者 1 中确定的值，但究竟是 0 还是 1 却是随机的，与打拍之前输入信号的值没有必然的关系。

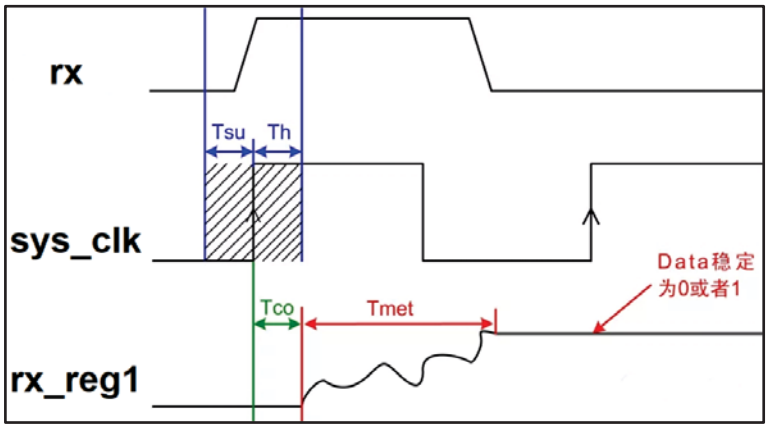

图 30-14 亚稳态产生波形图（一）

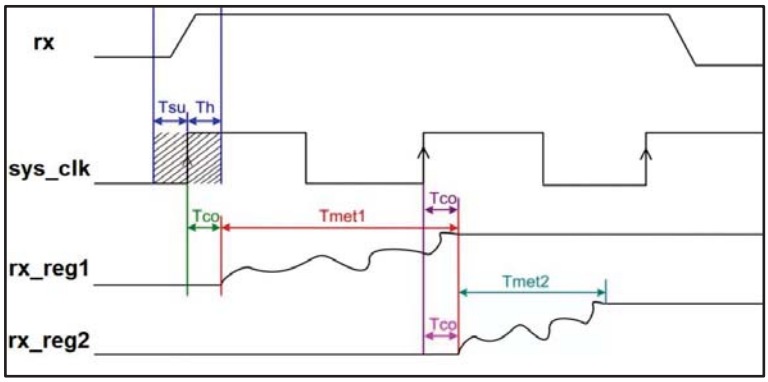

图 30-15 亚稳态产生波形图（二）

注：单比特信号从慢速时钟域同步到快速时钟域需要使用打两拍的方式消除亚稳态。第一级寄存器产生亚稳态并经过自身后可以稳定输出的概率为 70%~80%左右，第二级寄存器可以稳定输出的概率为 99%左右，后面再多加寄存器的级数改善效果就不明显了，所以数据进来后一般选择打两拍即可。

另外单比特信号从快速时钟域同步到慢速时钟域还仅仅使用打两拍的方式会漏采数据，所以往往使用脉冲同步法或的握手信号法；而多比特信号跨时钟域需要进行格雷码编码（多比特顺序数才可以）后才能进行打两拍的处理，或者通过使用 FIFO、RAM 来处理数据与时钟同步的问题。

亚稳态振荡时间 Tmet 关系到后级寄存器的采集稳定问题，Tmet 影响因素包括：器件的生产工艺、温度、环境以及寄存器采集到亚稳态里稳定态的时刻等。甚至某些特定条件，如干扰、辐射等都会造成 Tmet 增长。

第二部分：由上面的分析，我们知道了为什么 rx 信号进入到 FPGA 后需要先打两拍的原因，打两拍后的 rx_reg2 信号就是我们可以在后级逻辑电路中使用的相对稳定的信号，只比 rx 信号延后两。下一步我们就可以根据串口接收数据的时序要求找到串口帧起始开始的标志——下降沿，然后按顺序接收数据。在触摸按键章节我们分析过如何产生上升沿和下降沿标志，这里我们可以直接使用。由第一部分的分析得 rx_reg1 信号可能是不稳定的，

而 rx_reg2 信号是相对稳定的，所以不能直接用 rx_reg1 信号和 rx_reg2 信号来产生下降沿标志信号，因为 rx_reg1 信号的不稳定性可能会导致由它产生的下降沿标志信号也不稳定。所以如图 30-16 所示，我们将 rx_reg2 信号再打一拍，得到 rx_reg3 信号，用 rx_reg2 信号和 rx_reg3 信号产生 staet_nedge 作为下降沿标志信号。

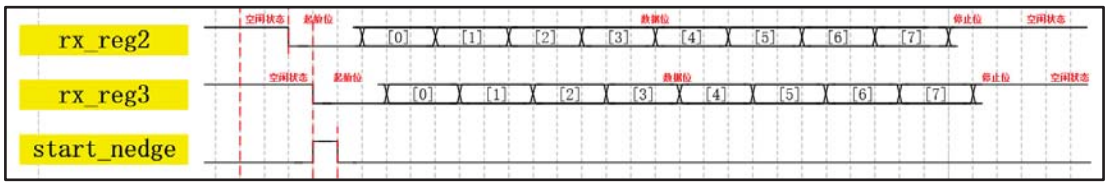

图 30-16 下降沿标志信号产生图

第三部分：我们检测到了第一个下降沿，后面的信号将以下降沿标志信号 start_nedge 为条件开始接收一帧 10bit 的数据。但新的问题又出现了，我们的 rx 信号本身就是 1bit 的，如果在判断第一个下降沿后，后面帧中的数据还可能会有下降沿出现，那我们会又产生一个 start_nedge 标志信号，这样就出现了误判断，那我们该如何避免这种情况呢?这是一个值得思考的问题，在不知道答案之前我们可以发挥自己的想象并尝试使用各种方法来解决这个问题。我们知道在 Verilog 代码中标志信号（flag）和使能信号（en）都是非常有用的，标志信号只有一拍，非常适合我们产生像下降沿标志这种信号，而使能信号就特别适合在此处使用，即对一段时间区域进行控制锁定。如图 30-17 所示，当下降沿标志信号 start_nedge 为高电平时拉高工作使能信号 work_en（什么时候拉低在后面讲解），在 work_en 信号为高的时间区域内虽然也会有下降沿 start_nedge 标志信号产生，但是我们可以根据 work_en 信号就可以判断出此时出现的 start_nedge 标志信号并不是我们想要的串口帧起始下降沿，从而将其过滤除掉。

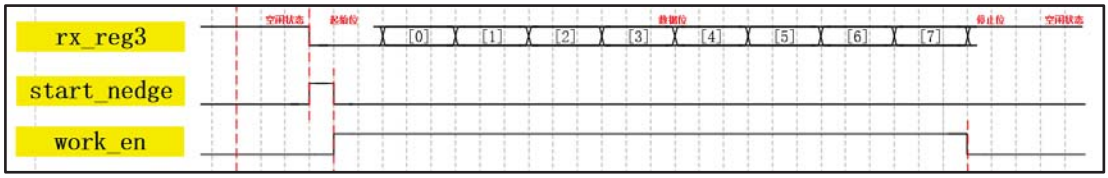

图 30-17 拉高工作使能信号波形图

解决了这个问题之后，我们正式开始接收一帧数据。我们使用的是 9600bps 的波特率和 PC 机进行串口通信，PC 机的串口调试助手要将发送数据波特率调整为 9600bps。而 FPGA 内部使用的系统时钟是 50MHz，前面也进行过计算，得出 1bit 需要的时间约为 5208 个（因为一帧只有 10bit，细微的近似计数差别不会产生数据错误，但是如果计数值差的过大，则会产生接收数据的错误）系统时钟周期，那么我们就需要产生一个能计 5208 个数的计数器来依次接收 10 个比特的数据，计数器每计 5208 个数就接收一个新比特的数据。如图 30-18 所示，计数器名为 baud_cnt，当 work_en 信号为高电平的时候就让计数器计数，当计数器计 5208 个数（从 0 到 5207）或 work_en 信号为低电平时计数器清零。

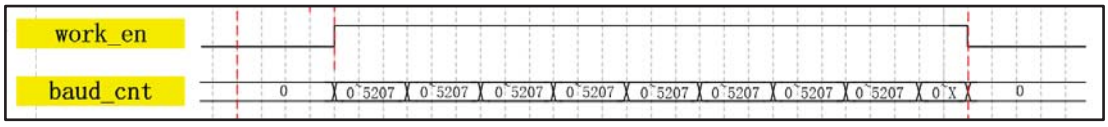

图 30-18 baud_cnt 计数器产生波形图

第四部分：现在我们可以根据波特率计数器一个一个接收数据了，我们发现 baud_cnt 计数器在计数值为 0 到 5207 期间都是数据有效的时刻，那我们该什么时候取数据呢？理论上讲，在数据变化的地方取数是不稳定的，所以我们选择当 baud_cnt 计数器计数到 2603，即中间位置时取数最稳定（其实只要 baud_cnt 计数器在计数值不是在 0 和 5207 这两个最不稳定的时刻取数都可以，更为准确的是多次取值取概率最大的情况）。所以如图 30-19 所示，在 baud_cnt 计数器计数到中点时产生一个时钟周期的 bit_flag 的取数标志信号，用于指示该时刻的数据可以被取走。

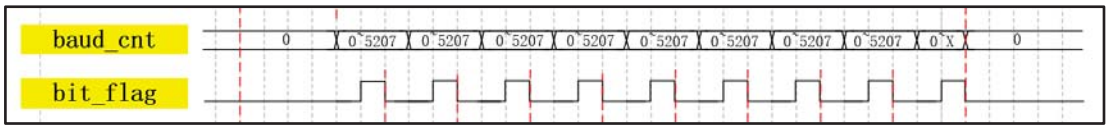

图 30-19 bit_flag 标志信号产生波形图

串口的数据是基于帧的，所以每接收完一帧数据 rx 信号都要被拉高，即恢复到空闲状态重新判断串口帧起始下降沿，以等待下一帧数据的接收，且一帧数据中还包括了起始位和停止位这种无用的数据，而对我们有价值的数据只是中间的 8bit 数据，也就是说我们需要准确的知道我们此时此刻接收的是第几比特，当接收够 10bit 数据后，我们就停止继续接收数据，等 rx 信号被拉高待恢复到空闲状态后再等待接收下一帧的数据。所以我们还需要产生一个用于计数该时刻接收的数据是第几个比特的 bit_cnt 计数器。如图 30-20 所示，刚好可以利用我们已经产生的 bit_flag 取数标志信号，对该信号进行计数既可以知道此时我们接收的数据是第几个比特了。这里我们只让 bit_cnt 计数器的计数值为 8 时再清零，虽然 bit_cnt 计数器的计数值从 0 计数到 8 只有 9 个 bit，但这 9 个 bit 中已经包含的我们所需要的 8bit 有用的数据，最后的 1bit 停止位没有用，可以不用再进行计数了，但如果非要将 bit_cnt 计数器的计数值计数到 9 后再清零也是可以的。

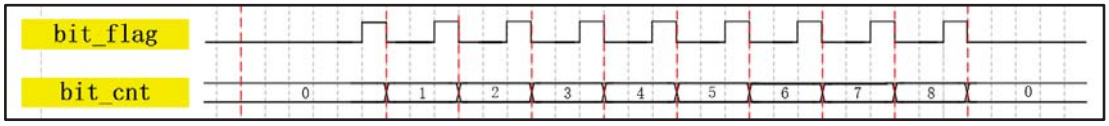

图 30-20 bit_cnt 计数器产生波形图

讲到这里我们不要忘记第三部分的遗留问题，那就是 work_en 信号何时拉低。如图 30-21 所示，当 bit_cnt 计数器计数到 8 且①处的 bit_flag 取数标志信号同时为高，说明我们已经接收到了所有的 8bit 有用数据，这两个条件必须同时满足时才能让 work_en 信号拉低。如果仅仅把 bit_cnt 计数器的计数值计数到 8 作为 work_en 信号拉低的条件，而掉①处的 bit_flag 取数标志信号为高这个条件，就会使 work_en 信号在绿色虚线位置处拉低，导致最后 1bit 数据丢失，致使后面接收的帧出错甚至接收不到数据。

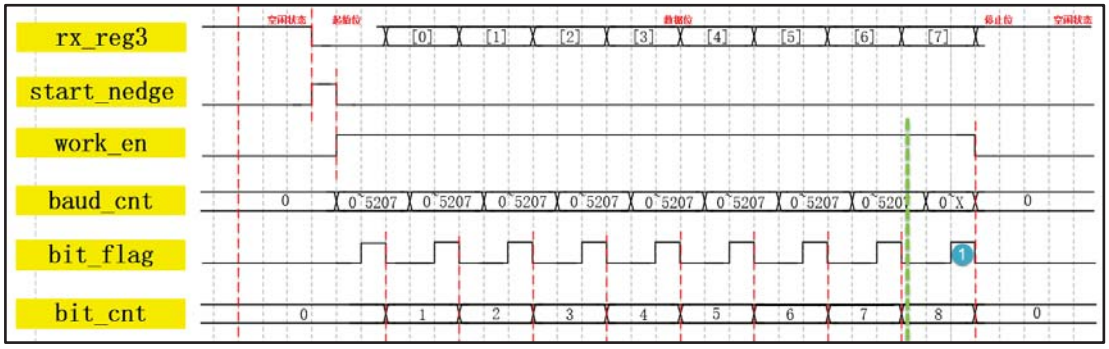

图 30-21 拉低 work_en 信号波形图

第五部分：我们接收到的 rx 信号是串行的，后面的系统要使用的是完整的 8bit 并行数据。也就是说我们还需要将 1bit 串行数据转换为 8bit 并行数据的串并转换的工作，这也是我们在接口设计中常遇到的一种操作。串并转换就需要做移位，我们要考虑清楚什么时候开始移位，不能提前也不能推后，否则会将无用的数据也移位进来，所以我们需要卡准时间。如图 30-22 所示 PC 机的串口调试助手发送的数据是先发送的低位后发送的高位，所以我们接收的 rx 信号也是先接收的低位后接收的高位，我们采用边接收边移位的操作。移位操作的方法我们已经在前面的流水灯章节中讲过，这里不再重复。接下来我们需要确定移位开始和结束的时间。如图 30-23 所示，当 bit_cnt 计数器的计数值为 1 时说明第一个有用数据已经接收到了，刚好剔除了起始位，就可以进行移位了。注意移位的条件，要在 bit_cnt 计数器的计数值为 1 到 8 区间内且 bit_flag 取数标志信号同时为高时才能移位，也就是移动 7 次即可，接收最后 1bit 有用数据时就不需要再进行移位了。当移位 7 次后 1bit 的串行数据已经变为 8bit 的并行数据了，此时产生一个移位完成标志信号 rx_flag。

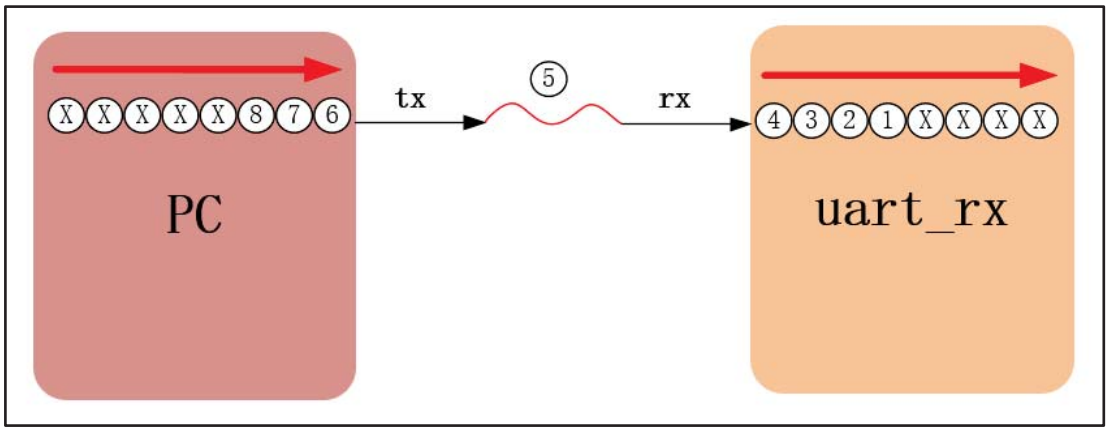

图 30-22 数据的接收图

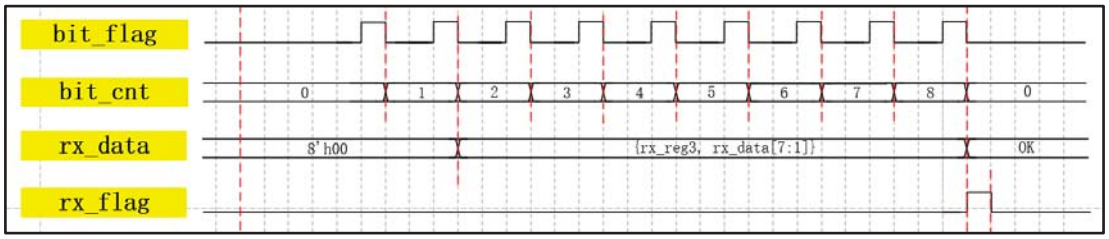

图 30-23 数据移位波形图

第六部分：此时有很多同学以为我们的串口接收模块就全部完成了，其实还差最后一点。rx_data 信号是参与移位的数据，在移位的过程中数据是变动的，不可以被后级模块所使用，而可以肯定的是在移位完成标志信号 rx_flag 为高时，rx_data 信号一定是移位完成的稳定的 8bit 有用数据。如图 30-24 所示，此时我们当移位完成标志信号 rx_flag 为高时让 rx_data 信号赋值给专门用于输出稳定 8bit 有用数据的 po_data 信号就可以了，但 rx_flag 信号又不能作为 po_data 信号有效的标志信号，所以需要将 rx_flag 信号再打一拍。最后输出的有用 8bit 数据为 po_data 信号和伴随 po_data 信号有效的标志信号 po_flag 信号。到此为止我们 uart_rx 模块的波形就全部设计好了。

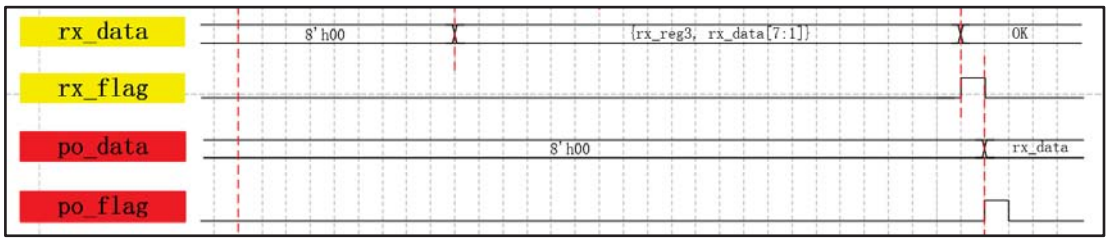

图 30-24 输出 po_data 波形图

代码编写

波形画出来了，再结合详细的波形分析，代码分分钟就可以搞定。写代码时还是和以前一样按照所画波形的顺序依次编写，这样在信号较多的情况下也不容易漏掉。为了增加模块的通用性，我们将波特率的计数值做成参数的形式，如果使用其他波特率进行通信，就可以将算好的计数值直接替换。模块参考代码详见代码清单 30-1。

代码清单 30-1 串口接收模块参考代码（uart_rx.v）

```
1  module  uart_rx
2  #(
3      parameter   UART_BPS    =   'd9600,          //串口波特率
4      parameter   CLK_FREQ    =   'd50_000_000     //时钟频率
5  )
6  (
7      input   wire            sys_clk     ,   //系统时钟 50MHz
8      input   wire            sys_rst_n   ,   //全局复位
9      input   wire            rx          ,   //串口接收数据
10
11     output  reg     [7:0]   po_data     ,   //串转并后的 8bit 数据
12     output  reg             po_flag         //串转并后的数据有效标志信号
13 );
14
15 //*************************************************************//
16 //***************** Parameter and Internal Signal *****************//
17 //*************************************************************//
18
19 //localparam    define
20 localparam  BAUD_CNT_MAX    =   CLK_FREQ/UART_BPS   ;
21
22 //reg   define
23 reg         rx_reg1     ;
24 reg         rx_reg2     ;
25 reg         rx_reg3     ;
26 reg         start_nedge ;
27 reg         work_en     ;
28 reg [12:0]  baud_cnt    ;
29 reg         bit_flag    ;
```

```
30 reg [3:0]   bit_cnt     ;
31 reg [7:0]   rx_data     ;
32 reg         rx_flag     ;
33
34 //*********************************************************************//
35 //***************************** Main Code *****************************//
36 //*********************************************************************//
37
38 //插入两级寄存器进行数据同步，用来消除亚稳态
39 //rx_reg1:第一级寄存器，寄存器空闲状态复位为 1
40 always@(posedge sys_clk or negedge sys_rst_n)
41     if(sys_rst_n == 1'b0)
42         rx_reg1 <= 1'b1;
43     else
44         rx_reg1 <= rx;
45
46 //rx_reg2:第二级寄存器，寄存器空闲状态复位为 1
47 always@(posedge sys_clk or negedge sys_rst_n)
48     if(sys_rst_n == 1'b0)
49         rx_reg2 <= 1'b1;
50     else
51         rx_reg2 <= rx_reg1;
52
53 //rx_reg3:第三级寄存器和第二级寄存器共同构成下降沿检测
54 always@(posedge sys_clk or negedge sys_rst_n)
55     if(sys_rst_n == 1'b0)
56         rx_reg3 <= 1'b1;
57     else
58         rx_reg3 <= rx_reg2;
59
60 //start_nedge:检测到下降沿时 start_nedge 产生一个时钟的高电平
61 always@(posedge sys_clk or negedge sys_rst_n)
62     if(sys_rst_n == 1'b0)
63         start_nedge <= 1'b0;
64     else    if((~rx_reg2) && (rx_reg3))
65         start_nedge <= 1'b1;
66     else
67         start_nedge <= 1'b0;
68
69 //work_en:接收数据工作使能信号
70 always@(posedge sys_clk or negedge sys_rst_n)
71     if(sys_rst_n == 1'b0)
72         work_en <= 1'b0;
73     else    if(start_nedge == 1'b1)
74         work_en <= 1'b1;
75     else    if((bit_cnt == 4'd8) && (bit_flag == 1'b1))
76         work_en <= 1'b0;
77
78 //baud_cnt:波特率计数器计数，从 0 计数到 5207
79 always@(posedge sys_clk or negedge sys_rst_n)
80     if(sys_rst_n == 1'b0)
81         baud_cnt <= 13'b0;
82     else    if((baud_cnt == BAUD_CNT_MAX - 1) || (work_en == 1'b0))
83         baud_cnt <= 13'b0;
84     else    if(work_en == 1'b1)
85         baud_cnt <= baud_cnt + 1'b1;
86
87 //bit_flag:当 baud_cnt 计数器计数到中间数时采样的数据最稳定，
88 //此时拉高一个标志信号表示数据可以被取走
89 always@(posedge sys_clk or negedge sys_rst_n)
90     if(sys_rst_n == 1'b0)
91         bit_flag <= 1'b0;
92     else    if(baud_cnt == BAUD_CNT_MAX/2 - 1)
93         bit_flag <= 1'b1;
94     else
```

```
95          bit_flag <= 1'b0;
96
97  //bit_cnt:有效数据个数计数器，当 8 个有效数据（不含起始位和停止位）
98  //都接收完成后计数器清零
99  always@(posedge sys_clk or negedge sys_rst_n)
100     if(sys_rst_n == 1'b0)
101         bit_cnt <= 4'b0;
102     else    if((bit_cnt == 4'd8) && (bit_flag == 1'b1))
103         bit_cnt <= 4'b0;
104     else    if(bit_flag ==1'b1)
105         bit_cnt <= bit_cnt + 1'b1;
106
107 //rx_data:输入数据进行移位
108 always@(posedge sys_clk or negedge sys_rst_n)
109     if(sys_rst_n == 1'b0)
110         rx_data <= 8'b0;
111     else    if((bit_cnt >= 4'd1)&&(bit_cnt <= 4'd8)&&(bit_flag == 1'b1))
112         rx_data <= {rx_reg3, rx_data[7:1]};
113
114 //rx_flag:输入数据移位完成时 rx_flag 拉高一个时钟的高电平
115 always@(posedge sys_clk or negedge sys_rst_n)
116     if(sys_rst_n == 1'b0)
117         rx_flag <= 1'b0;
118     else    if((bit_cnt == 4'd8) && (bit_flag == 1'b1))
119         rx_flag <= 1'b1;
120     else
121         rx_flag <= 1'b0;
122
123 //po_data:输出完整的 8 位有效数据
124 always@(posedge sys_clk or negedge sys_rst_n)
125     if(sys_rst_n == 1'b0)
126         po_data <= 8'b0;
127     else    if(rx_flag == 1'b1)
128         po_data <= rx_data;
129
130 //po_flag:输出数据有效标志（比 rx_flag 延后一个时钟周期，为了和 po_data 同步）
131 always@(posedge sys_clk or negedge sys_rst_n)
132     if(sys_rst_n == 1'b0)
133         po_flag <= 1'b0;
134     else
135         po_flag <= rx_flag;
136
137 endmodule
```

仿真文件编写

在编写仿真代码时，我们要模拟出 PC 机的串口调试助手发送串行数据帧的过程，我们首次使用 task 任务来实现数据一个一个发送的过程。模块仿真参考代码详见代码清单 30-2。

代码清单 30-2 串口接收模块仿真参考代码（tb_uart_rx.v）

```
1  module  tb_uart_rx();
2
3  //*****************************************************************//
4  //****************** Parameter and Internal Signal *******************//
5  //*****************************************************************//
6
7  //reg   define
8  reg             sys_clk;
9  reg             sys_rst_n;
10 reg             rx;
11
12 //wire  define
13 wire    [7:0]   po_data;
```

```
14 wire            po_flag;
15
16 //*****************************************************************//
17 //***************************** Main Code *************************//
18 //*****************************************************************//
19
20 //初始化系统时钟、全局复位和输入信号
21 initial begin
22         sys_clk    = 1'b1;
23         sys_rst_n <= 1'b0;
24         rx          <= 1'b1;
25         #20;
26         sys_rst_n <= 1'b1;
27 end
28
29 //模拟发送 8 次数据，分别为 0~7
30 initial begin
31         #200
32         rx_bit(8'd0);  //任务的调用，任务名+括号中要传递进任务的参数
33         rx_bit(8'd1);
34         rx_bit(8'd2);
35         rx_bit(8'd3);
36         rx_bit(8'd4);
37         rx_bit(8'd5);
38         rx_bit(8'd6);
39         rx_bit(8'd7);
40 end
41
42 //sys_clk:每 10ns 电平翻转一次，产生一个 50MHz 的时钟信号
43 always #10 sys_clk = ~sys_clk;
44
45 //定义一个名为 rx_bit 的任务，每次发送的数据有 10 位
46 //data 的值分别为 0~7 由 j 的值传递进来
47 //任务以 task 开头，后面紧跟着的是任务名，调用时使用
48 task rx_bit(
49     //传递到任务中的参数，调用任务的时候从外部传进来一个 8 位的值
50         input   [7:0]   data
51 );
52         integer i;       //定义一个常量
53 //用 for 循环产生一帧数据，for 括号中最后执行的内容只能写 i=i+1
54 //不可以写成 C 语言 i=i++的形式
55         for(i=0; i<10; i=i+1) begin
56             case(i)
57                 0: rx <= 1'b0;
58                 1: rx <= data[0];
59                 2: rx <= data[1];
60                 3: rx <= data[2];
61                 4: rx <= data[3];
62                 5: rx <= data[4];
63                 6: rx <= data[5];
64                 7: rx <= data[6];
65                 8: rx <= data[7];
66                 9: rx <= 1'b1;
67             endcase
68             #(5208*20); //每发送 1 位数据延时 5208 个时钟周期
69         end
70 endtask          //任务以 endtask 结束
71
72 //*****************************************************************//
73 //************************** Instantiation ************************//
74 //*****************************************************************//
75
76 //------------------------uart_rx_inst-----------------------
77 uart_rx uart_rx_inst(
```

```
78          .sys_clk    (sys_clk    ),  //input           sys_clk
79          .sys_rst_n  (sys_rst_n  ),  //input           sys_rst_n
80          .rx         (rx         ),  //input           rx
81
82          .po_data    (po_data    ),  //output  [7:0]   po_data
83          .po_flag    (po_flag    )   //output          po_flag
84  );
85
86  endmodule
```

仿真波形分析

打开 ModelSim 后先清空波形信号，重新添加要测试的模块，我们让波形跑了 10ms 即可完全显示所有波形，然后将波形窗口信号的排列顺序和所画的波形图顺序一致，这样可以更快速的发现我们设计中存在的问题。模拟 PC 机发送 8 次（数据值从 0 到 7）串行数据的波形如图 30-25 所示。红色圈①处为接收数据“1”的波形，将其放大详细观察。

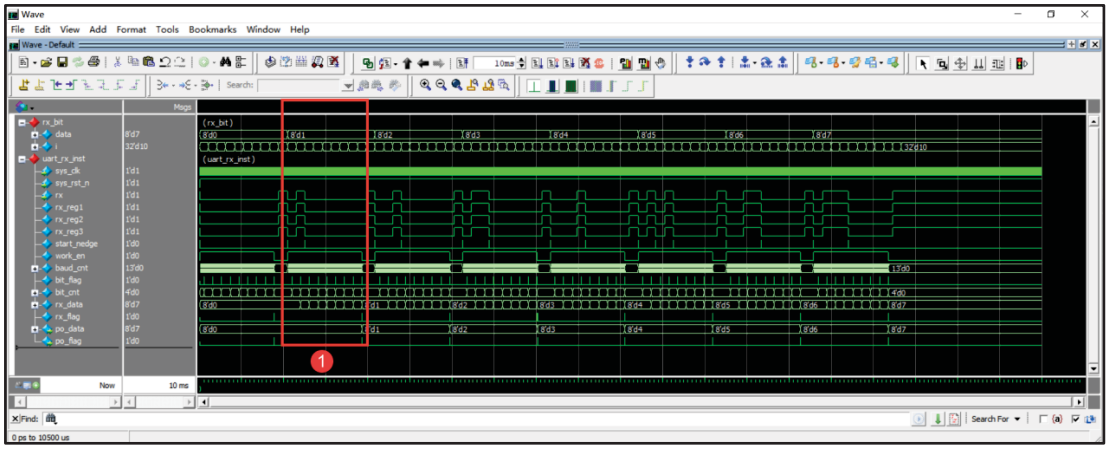

图 30-25 接收模块仿真波形图（一）

整体接收模块的波形如图 30-26 所示，我们可以看到数据是先接收的低位后接收的高位，一共是 10bit 数据。

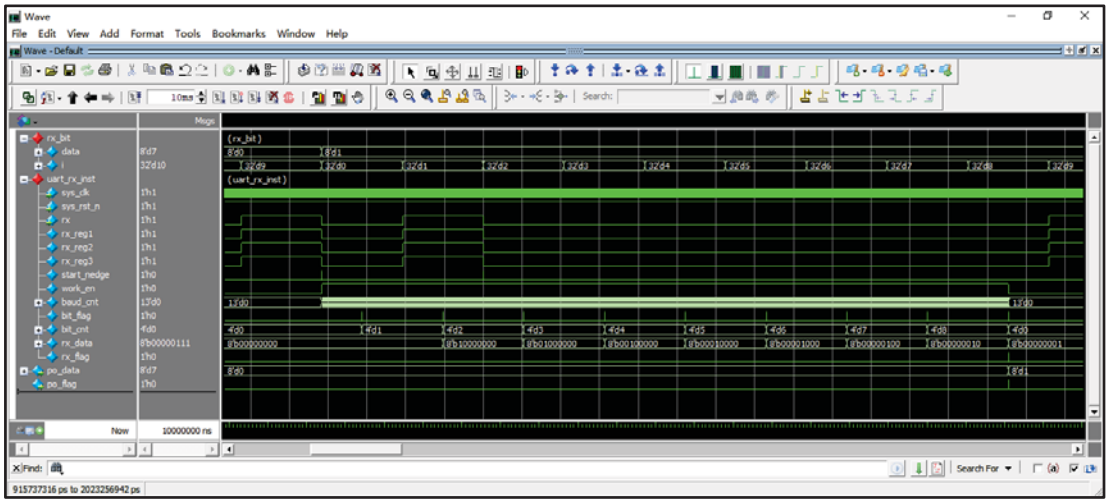

图 30-26 接收模块仿真波形图（二）

第一、第二、第三部分仿真波形如图 30-27 所示，我们可以清晰的看到将 rx 信号打三拍的操作，并产生了串口帧起始的下降沿标志信号，以及 work_en 信号在串口帧起始的下降沿标志信号为高时拉高，baud_cnt 计数器在 work_en 信号为高时开始计数。

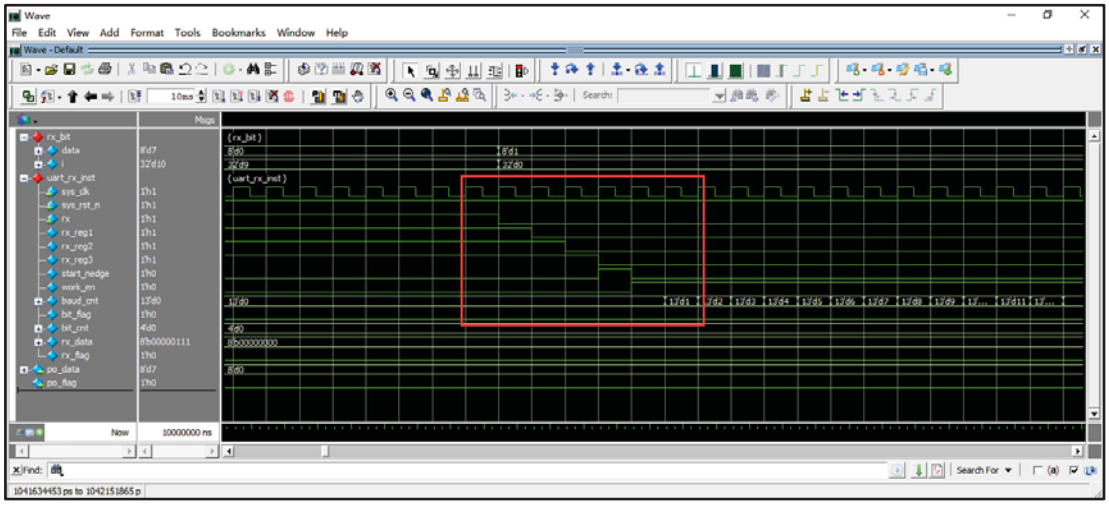

图 30-27 接收模块仿真波形图（三）

第四部分仿真波形如图 30-28 所示，取数标志信号 bit_flag 在 baud_cnt 计数器计数到 2603 时产生一个时钟周期的脉冲。

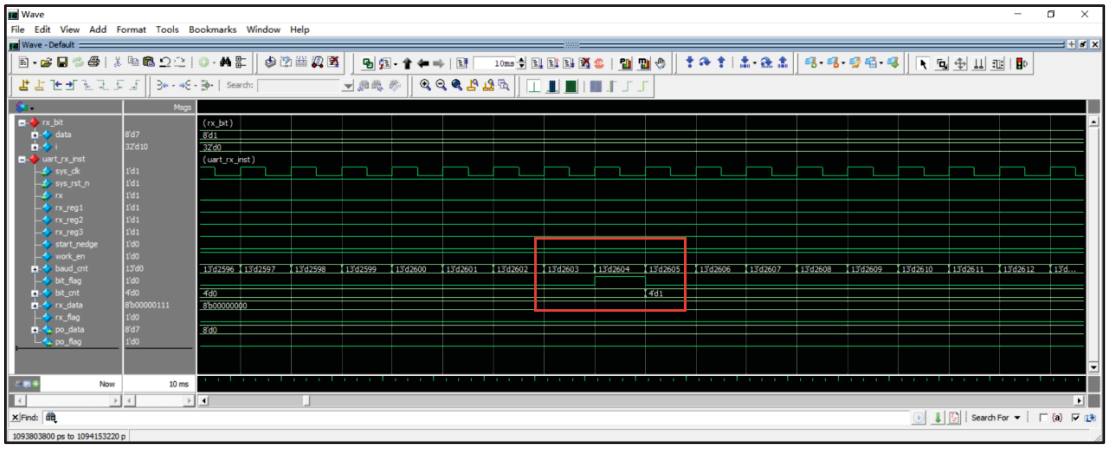

图 30-28 接收模块仿真波形图（四）

第五部分仿真波形如图 30-29 所示，可以看到 rx_data 信号在 bit_cnt 计数器的计数值为 1 到 8 区间内且 bit_flag 取数标志信号同时为高时移位的过程。

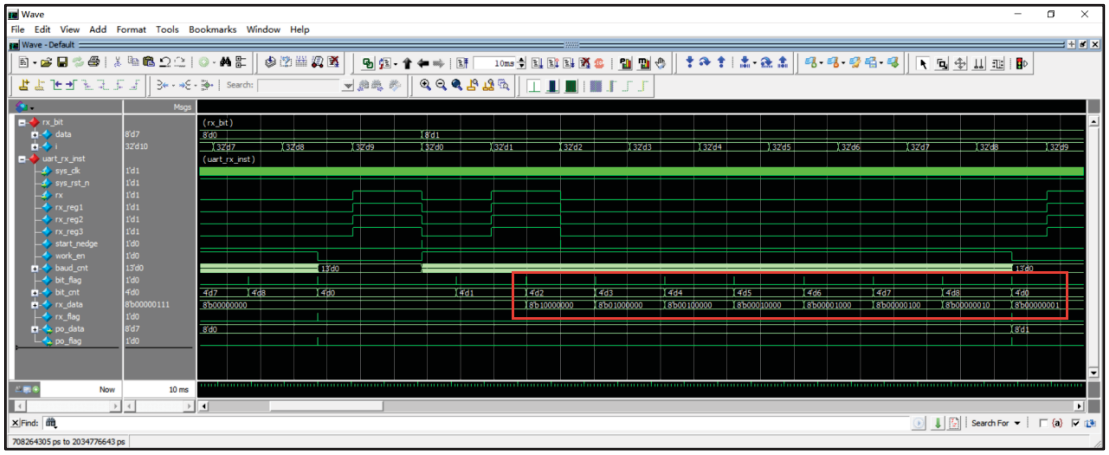

图 30-29 接收模块仿真波形图（五）

第六部分仿真波形如图 30-30 所示，我们可以看到 work_en 信号拉低的时间，以及产生供后级模块使用的 8bit 有用数据 po_data 和数据有效标志信号 po_flag。

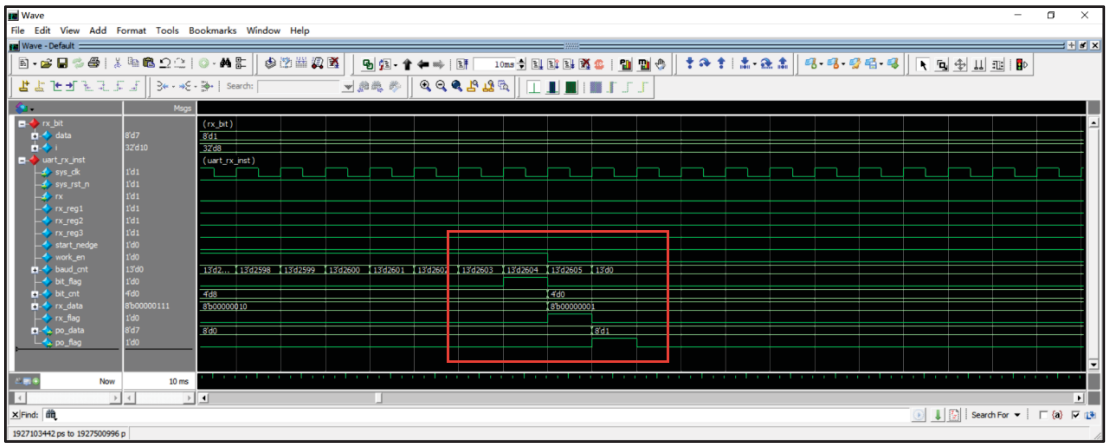

图 30-30 接收模块仿真波形图（六）

3. 串口数据发送模块

接下来我们继续进行串口发送模块的设计，该模块的功能是将 FPGA 中的数据以固定的波特率发送到 PC 机的串口调试助手并打印出来，串口发送模块按照串口的协议组装成帧，然后按照顺序一个比特一个比特将数据发送至 PC 机，而 FPGA 内部的数据往往都是并行的，需将其转化为串行数据发送。

模块设计

我们将串口接收模块取名为 uart_tx，根据功能简介我们对整个设计要求有了大致的了解，其中设计的关键点是如何将串并行数据转化为串行数据并发送出去，也就是按照顺序将并行数据发送至 PC 机上。FPGA 发送的串行数据同样没有时钟，所以要和 PC 机接约定好使用相同的波特率，一个一个地发送比特，为了后面做串口的回环测试我们仍选择使用 9600bps 的波特率。

整个模块也必须用到时序逻辑，所以先设计好时钟 sys_clk 和复位 sys_rst_n 两个输入信号，其次是 FPGA 要发送的 8bits 有用数据 pi_data 和伴随数据有效的标志信号 pi_flag。其次是相对于 FPGA 的 tx 端发送至 PC 机中的 1bit 输出信号。

根据上面的分析设计出的 Visio 框图如图 30-31 所示。

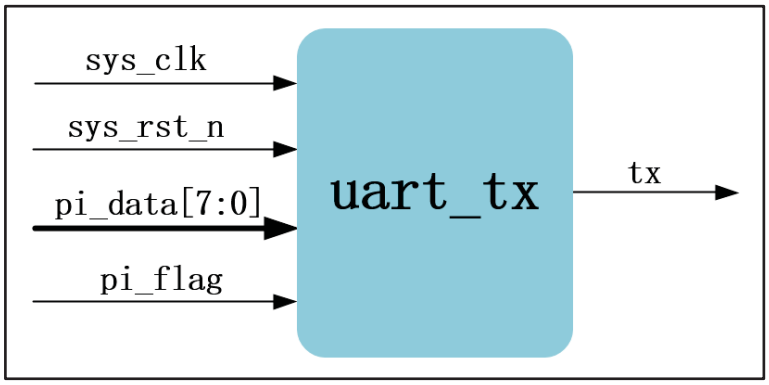

图 30-31 串口发送模块框图

端口列表与功能总结如表格 30-4 所示。

表格 30-4 串口发送模块输入输出信号描述

信号	位宽	类型	功能描述
sys_clk	1Bit	Input	工作时钟，频率 50MHz
pi_data	8Bit	Input	发要送的 8bit 并行数据
pi_data_flag	1Bit	Input	要发送的 8bit 并行数据有效标志信号
tx	1Bit	Output	串口发送信号

波形设计

如图 30-32 所示，我们先把实现 uart_tx 功能整体的波形图列出，然后再详细介绍下面的波形是如何一步步设计实现的。

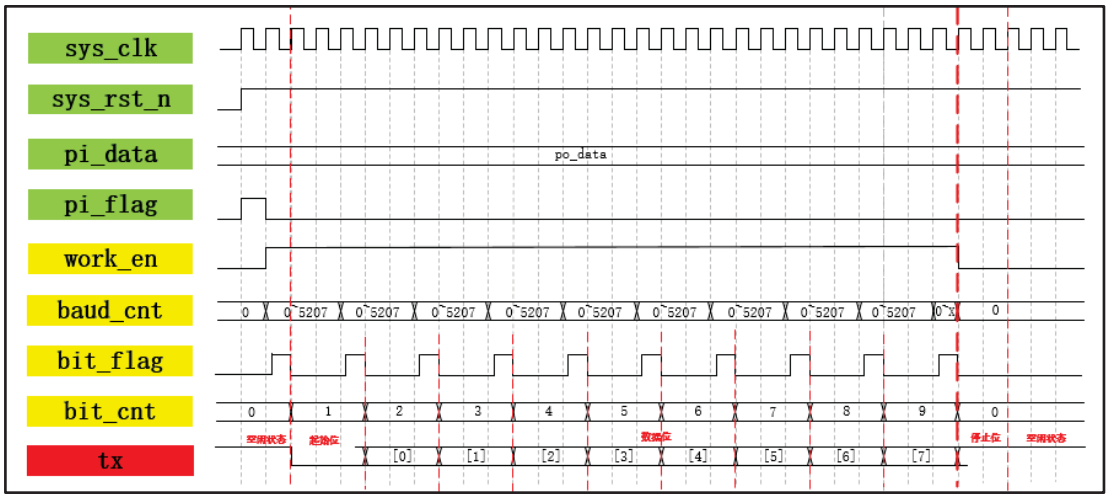

图 30-32 串口发送模块波形图

波形设计思路详细解析

第一部分：首先把四个输入信号画出，分别是时钟、复位、8bit 有用数据和数据有效标志信号。8bit 有用数据 pi_data 和数据有效标志信号 pi_flag 是上一级系统发送过来的，我们设

计的模块只需要负责接收即可。当数据有效标志信号 pi_flag 为高时表示数据已经是稳定的可以被使用的，这时就可以把 8bit 数据接收过来了，然后再将这个 8bit 数据按照顺序一个一个串行发送出去。我们已经和 PC 机约定好了使用 9600bps 的波特率，所以发送 1bit 数据需要的时间也约为 5208 个系统时钟周期，这就需要产生一个和接收数据时一样的波特率计数器，我们取名为 baud_cnt，该计数器每计 5208 个数就发送一个新比特的数据，一共发送 10 个比特。但是仔细一想问题就出现了，波特率计数器 baud_cnt 计数的条件是什么呢？当检测到数据有效标志信号 pi_flag 为高时就开始计数吗？这是不行的，因为 pi_flag 信号只维持一个时钟周期的高电平，并不能让 pi_flag 信号为高作为波特率计数器 baud_cnt 计数的条件，所以我们需要一个控制波特率计数器 baud_cnt 何时计数的使能信号。如图 30-33 所示，我们产生一个名为 work_en 的工作使能信号，当检测到数据有效标志信号 pi_flag 为高电平时拉高工作使能信号 work_en（什么时候拉低在后面讲解），因为 work_en 信号是持续的高电平，所以当 work_en 信号为高电平时波特率计数器 baud_cnt 进行计数。

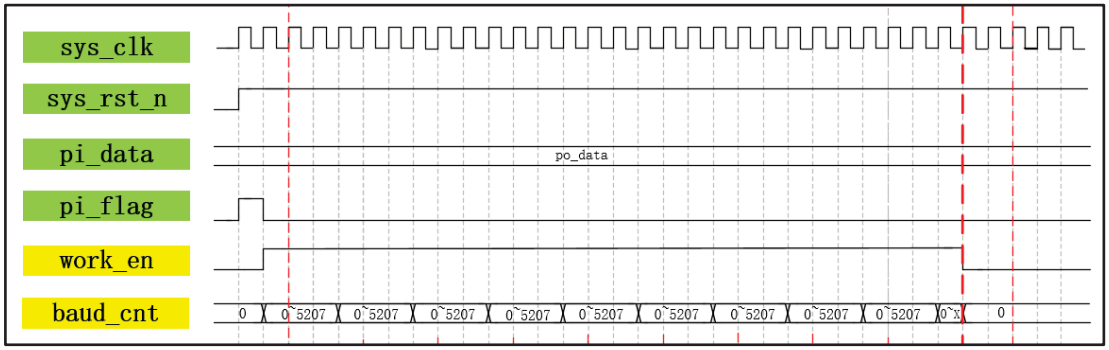

图 30-33 串口发送模块波形图部分（一）

第二部分：下面我们就可以按照 5208 个系统时钟周期的波特率间隔来发送 1bit 数据了。那应该在什么位置开发送呢？我们要先在 5208 个系统时钟周期内确定好一个发送的点，后面再发送的数据间隔都是 5208 个时钟周期即可。理论上我们在第一个 5208 系统时钟周期内的任意一个位置发送数据都可以，这和接收数据时要在中间位置不同，所以我们直接让当 baud_cnt 计数器的计数值为 1（选择其他的值也可以，但是尽量不要选择 baud_cnt 计数器的计数值为 0 或 5207 这种端点，因为容易出问题）的时候作为发送数据的点，而下一个 baud_cnt 计数器的计数值为 1 的时候和上一个正好相差 5208 个系统时钟周期，是完全可以满足要求的。

那此时就可以发送数据了吗？我们再来思考一下，发送数据时要发送一帧，也就是需要发送固定 1bit 为 0 的起始位、8bit 的有用数据和固定 1bit 为 1 的停止位，每当 baud_cnt 计数器的计数值为 1 的时候就发送 1bit 的数据，发送第一个起始位的时候没有问题，发送 8 个有用数据位置的时候也没有问题，发送最后一个停止位的时候仍是正确的，但是我们只需要发送 10 个 bit 的数据就结束了，后面就不需要再发送数据了，此时如果 work_en 信号还持续为高那么 baud_cnt 计数器也就会一直计数，那么发送完 10bit 数据后还会继续发送，这是我们不需要的。所以当发送完一帧数据后我们要将 work_en 信号拉低，从而使 baud_cnt 计数器停止，才能够不继续发送数据。那什么时候停止呢？一定要在 10 个比特的

数据都发送完才能停止，那么我们就需要有一个用于计数当前发送了多少个数据的计数器，我们取名为 bit_cnt。bit_cnt 计数器在 baud_cnt 计数器的计数值每次为 1 的时候加 1 即可。如图 30-34 所示，为了更加直观的表达，我们再多加一个信号，每当 baud_cnt 计数器的计数值为 1 的时候产生一个时钟周期的名为 bit_flag 的允许发送数据标志信号，bit_cnt 计数器当标志信号为高时加 1（计数到多少后面在后面讲解）。

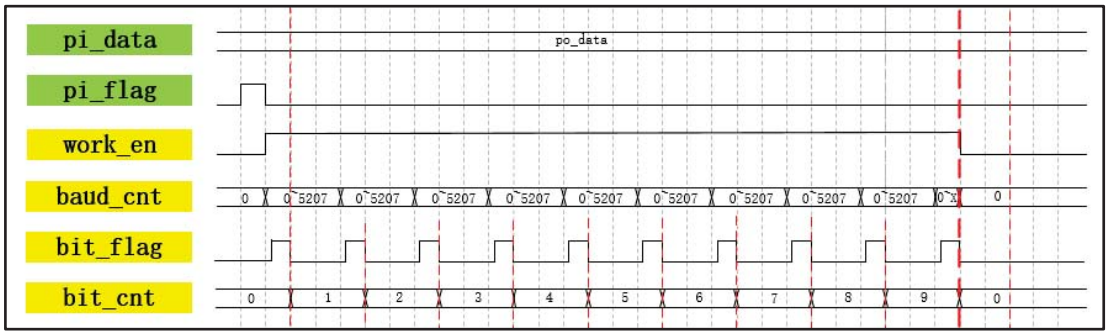

图 30-34 串口发送模块波形图部分（二）

第三部分：最后就是数据按顺序一个一个发出去，因为接收的时候是先接收的低位后接收的高位，所以如图 30-35 所示，发送的时候也是先发送低位，后发送高位。

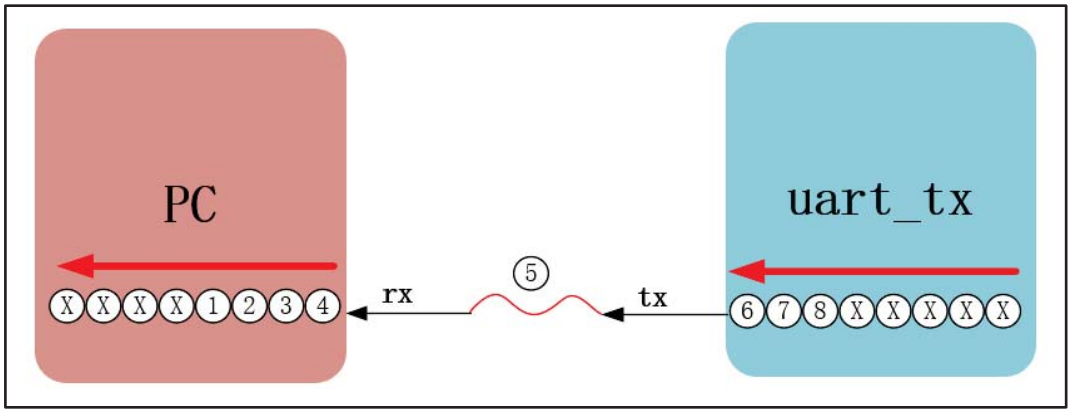

图 30-35 数据的发送图

不要忘记前面还有两个遗留问题没做，一个是 work_en 信号什么时候拉低，另一个是 bit_cnt 计数器计数到多少清零，也是最后的收尾工作了。先说 bit_cnt 计数器什么时候清零的问题，有的同学说要计数到 9，有的同学说要计数到 10，其实我们不妨都尝试一下。如所图 30-36 示，假如我们让 bit_cnt 计数器计数到 9，可以发现最后一个停止位没有对应的计数了，这会有问题吗？我们仔细分析就可以知道，停止位和空闲情况下都为高电平，所以最有一个停止位就没有必要再单独计数了，所以 bit_cnt 计数器计数到 9 清零是完全可以的，当然让 bit_cnt 计数器计数到 10 更是可以的。

最后再来说说 work_en 信号拉低的条件，work_en 存在的原因就是为了方便 baud_cnt 计数器计数的，当我们不需要 baud_cnt 计数器计数的时候也就可以让 work_en 信号拉低了。当 bit_cnt 计数器计数到 9 且 bit_flag 信号有效时停止位就可以被发送出去了，此时就不再需要 baud_cnt 计数器计数了，就可以把 work_en 信号拉低了，但同时还要将 baud_cnt

计数器清零，等待下一次发送数据时再从 0 开始计数。到此为止我们 uart_tx 模块的波形也全部设计好了。

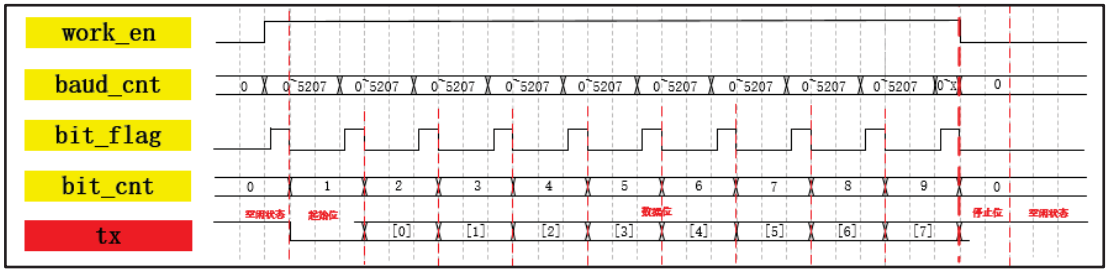

图 30-36 串口发送模块波形图部分（三）

代码编写

波形画出来了，再结合详细的波形分析，就可以进行代码的编写了，写代码时继续按照所画波形的顺序依次编写。为了增加模块的通用性，这里也将波特率的计数值做成参数的形式，如果使用其他速率的波特率进行通信，就可以将算好的计数值直接替换。模块参考代码详见代码清单 30-3。

代码清单 30-3 串口发送模块参考代码（uart_tx.v）

```
module  uart_tx
#(
    parameter   UART_BPS    =   'd9600,         //串口波特率
    parameter   CLK_FREQ    =   'd50_000_000    //时钟频率
)
(
     input   wire            sys_clk     ,   //系统时钟 50MHz
     input   wire            sys_rst_n   ,   //全局复位
     input   wire    [7:0]   pi_data     ,   //模块输入的 8bit 数据
     input   wire            pi_flag     ,   //并行数据有效标志信号

     output  reg             tx              //串转并后的 1bit 数据
);

//*****************************************************************//
//****************** Parameter and Internal Signal *****************//
//*****************************************************************//

//localparam    define
localparam  BAUD_CNT_MAX    =   CLK_FREQ/UART_BPS   ;

//reg   define
reg [12:0]  baud_cnt;
reg         bit_flag;
reg [3:0]   bit_cnt ;
reg         work_en ;

//*****************************************************************//
//***************************** Main Code *************************//
//*****************************************************************//

//work_en:接收数据工作使能信号
always@(posedge sys_clk or negedge sys_rst_n)
        if(sys_rst_n == 1'b0)
            work_en <= 1'b0;
        else    if(pi_flag == 1'b1)
            work_en <= 1'b1;
```

```
38          else    if((bit_flag == 1'b1) && (bit_cnt == 4'd9))
39              work_en <= 1'b0;
40
41  //baud_cnt:波特率计数器计数，从 0 计数到 5207
42  always@(posedge sys_clk or negedge sys_rst_n)
43          if(sys_rst_n == 1'b0)
44              baud_cnt <= 13'b0;
45          else    if((baud_cnt == BAUD_CNT_MAX - 1) || (work_en == 1'b0))
46              baud_cnt <= 13'b0;
47          else    if(work_en == 1'b1)
48              baud_cnt <= baud_cnt + 1'b1;
49
50  //bit_flag:当 baud_cnt 计数器计数到 1 时让 bit_flag 拉高一个时钟的高电平
51  always@(posedge sys_clk or negedge sys_rst_n)
52          if(sys_rst_n == 1'b0)
53              bit_flag <= 1'b0;
54          else    if(baud_cnt == 13'd1)
55              bit_flag <= 1'b1;
56          else
57              bit_flag <= 1'b0;
58
59  //bit_cnt:数据位数个数计数，10 个有效数据（含起始位和停止位）到来后计数器清零
60  always@(posedge sys_clk or negedge sys_rst_n)
61      if(sys_rst_n == 1'b0)
62          bit_cnt <= 4'b0;
63      else    if((bit_flag == 1'b1) && (bit_cnt == 4'd9))
64          bit_cnt <= 4'b0;
65      else    if((bit_flag == 1'b1) && (work_en == 1'b1))
66          bit_cnt <= bit_cnt + 1'b1;
67
68  //tx:输出数据在满足 rs232 协议（起始位为 0，停止位为 1）的情况下一位一位输出
69  always@(posedge sys_clk or negedge sys_rst_n)
70          if(sys_rst_n == 1'b0)
71              tx <= 1'b1; //空闲状态时为高电平
72          else    if(bit_flag == 1'b1)
73              case(bit_cnt)
74                  0       : tx <= 1'b0;
75                  1       : tx <= pi_data[0];
76                  2       : tx <= pi_data[1];
77                  3       : tx <= pi_data[2];
78                  4       : tx <= pi_data[3];
79                  5       : tx <= pi_data[4];
80                  6       : tx <= pi_data[5];
81                  7       : tx <= pi_data[6];
82                  8       : tx <= pi_data[7];
83                  9       : tx <= 1'b1;
84                  default : tx <= 1'b1;
85              endcase
86
87  endmodule
```

仿真文件编写

在编写仿真代码时，我们要模拟出 PC 机的串口调试助手发送串行数据帧的过程，我们和接收时一样，发送 8 个并行数据从 0 到 7，同时每个数据要有一个伴随数据有效的标志信号。这次我们不使用 task，可以看到我们的仿真代码会很长。模块仿真参考代码详见代码清单 30-4。

代码清单 30-4 串口发送模块仿真参考代码（tb_uart_tx.v）

```
1  module  tb_uart_tx();
2
3  //*****************************************************************//
4  //****************** Parameter and Internal Signal ******************//
```

```
 5 //*********************************************************************//
 6
 7 //reg   define
 8 reg         sys_clk;
 9 reg         sys_rst_n;
10 reg [7:0]   pi_data;
11 reg         pi_flag;
12
13 //wire  define
14 wire        tx;
15
16 //*********************************************************************//
17 //***************************** Main Code *****************************//
18 //*********************************************************************//
19
20 //初始化系统时钟、全局复位
21 initial begin
22         sys_clk    = 1'b1;
23         sys_rst_n <= 1'b0;
24         #20;
25         sys_rst_n <= 1'b1;
26 end
27
28 //模拟发送 7 次数据，分别为 0~7
29 initial begin
30         pi_data <= 8'b0;
31         pi_flag <= 1'b0;
32         #200
33         //发送数据 0
34         pi_data <= 8'd0;
35         pi_flag <= 1'b1;
36         #20
37         pi_flag <= 1'b0;
38 //每发送 1bit 数据需要 5208 个时钟周期，一帧数据为 10bit
39 //所以需要数据延时(5208*20*10)后再产生下一个数据
40         #(5208*20*10);
41         //发送数据 1
42         pi_data <= 8'd1;
43         pi_flag <= 1'b1;
44         #20
45         pi_flag <= 1'b0;
46         #(5208*20*10);
47         //发送数据 2
48         pi_data <= 8'd2;
49         pi_flag <= 1'b1;
50         #20
51         pi_flag <= 1'b0;
52         #(5208*20*10);
53         //发送数据 3
54         pi_data <= 8'd3;
55         pi_flag <= 1'b1;
56         #20
57         pi_flag <= 1'b0;
58         #(5208*20*10);
59         //发送数据 4
60         pi_data <= 8'd4;
61         pi_flag <= 1'b1;
62         #20
63         pi_flag <= 1'b0;
64         #(5208*20*10);
65         //发送数据 5
66         pi_data <= 8'd5;
67         pi_flag <= 1'b1;
68         #20
69         pi_flag <= 1'b0;
```

```
70          #(5208*20*10);
71          //发送数据 6
72          pi_data <= 8'd6;
73          pi_flag <= 1'b1;
74          #20
75          pi_flag <= 1'b0;
76          #(5208*20*10);
77          //发送数据 7
78          pi_data <= 8'd7;
79          pi_flag <= 1'b1;
80          #20
81          pi_flag <= 1'b0;
82  end
83
84  //sys_clk:每 10ns 电平翻转一次，产生一个 50MHz 的时钟信号
85  always #10 sys_clk = ~sys_clk;
86
87  //*****************************************************************//
88  //************************** Instantiation ************************//
89  //*****************************************************************//
90
91  //-----------------------uart_rx_inst-----------------------
92  uart_tx uart_tx_inst(
93          .sys_clk    (sys_clk    ),  //input           sys_clk
94          .sys_rst_n  (sys_rst_n  ),  //input           sys_rst_n
95          .pi_data    (pi_data    ),  //output  [7:0]   pi_data
96          .pi_flag    (pi_flag    ),  //output          pi_flag
97
98          .tx         (tx         )   //input           tx
99  );
100
101 endmodule
```

仿真波形分析

打开 ModelSim 后先清空波形信号，重新添加要测试的模块，我们让波形跑了 10ms 即可完全显示所有波形，先将波形窗口信号的排列顺序和所画的波形图顺序一致再进行观察。模拟上级系统发送 8 次（数据值从 0 到 7）数据和数据有效标志信号，其整体波形如图 30-37 所示。红色圈①处为发送数据“1”的波形，将其放大详细观察。

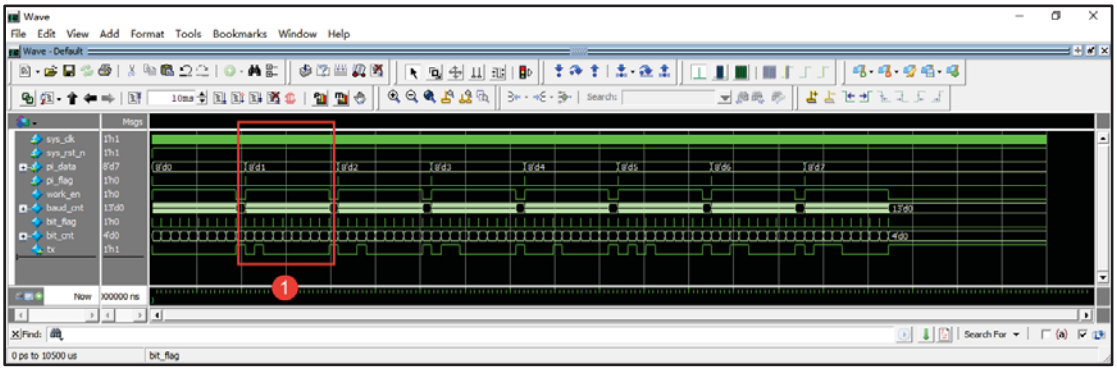

图 30-37 发送模块仿真波形图（一）

整体发送模块的波形如图 30-38 所示，我们可以看到数据是先发送的低位后发送的高位，一共是 10bit 数据。

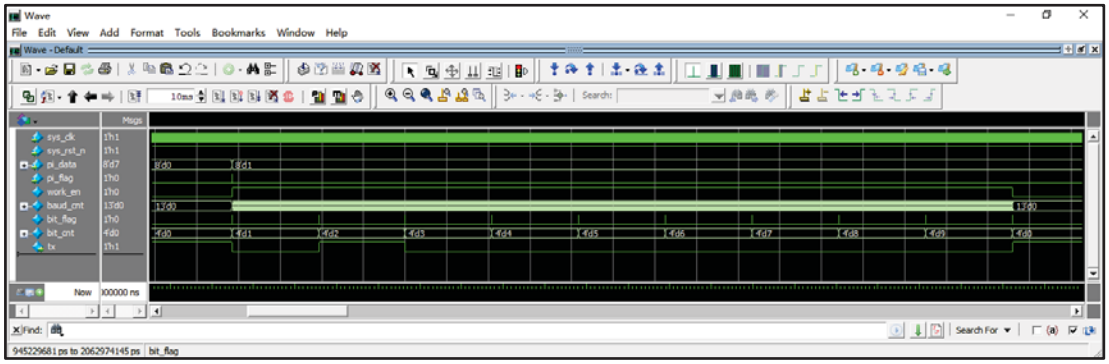

图 30-38 发送模块仿真波形图（二）

第一、第二部分仿真波形如图 30-39 所示，可以看到要发送的 8bit 并行数据为 1，同时伴随着一个数据有效标志信号，当数据有效标志信号为高时间 work_en 信号拉高，当 work_en 信号为高期间 baud_cnt 计数器进行计数。baud_cnt 计数器计数值为 1 时 bit_flag 信号为高，当检测到 bit_flag 信号为高时 tx 就发送一个数据，同时 bit_cnt 计数器加 1。

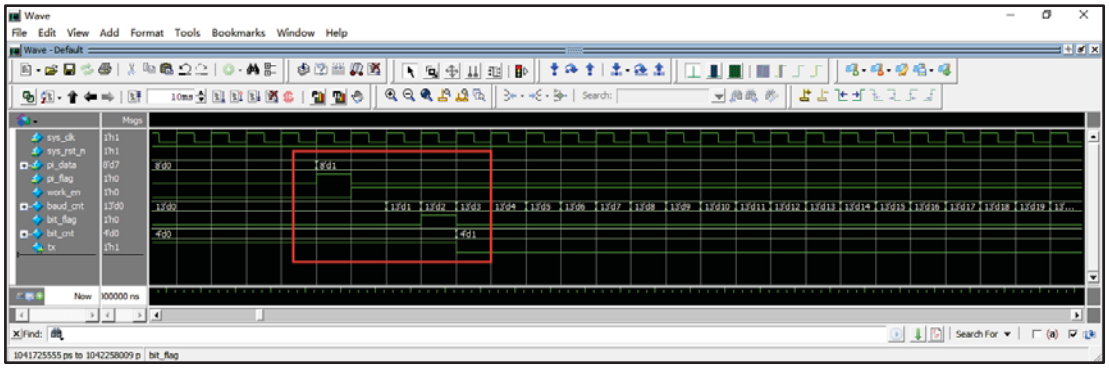

图 30-39 发送模块仿真波形图（三）

第三部分仿真波形如图 30-40 所示，我们可以清晰地看到最后一个 bit_flag 信号为高的时刻，且 bit_cnt 计数器也计数到 9，将停止位发送出去，同时 work_en 信号拉低，baud_cnt 计数器检测到 work_en 信号为低电平后立刻清零并停止计数，等待下一次发送数据时再工作。

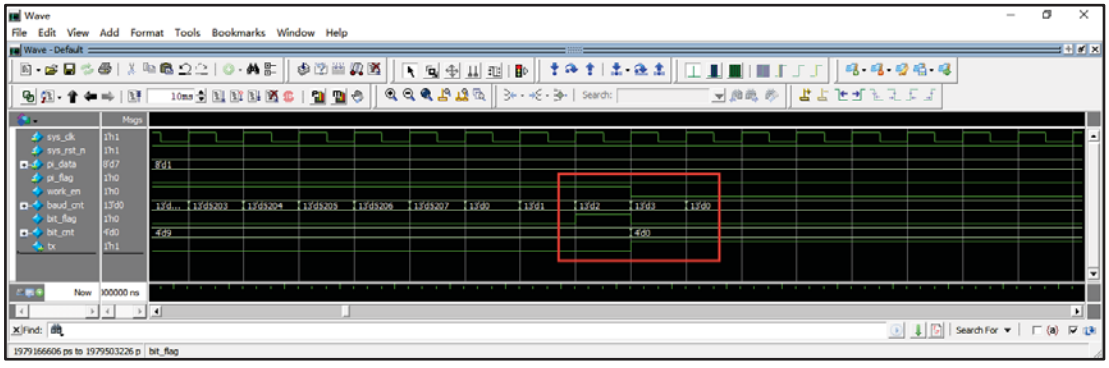

图 30-40 发送模块仿真波形图（四）

4. 顶层模块

串口的接收模块 uart_rx 和发送模块 uart_tx 我们都设计好了。在本章的最开始我们也讲过串口可以作为很好用的调试工具使用，比如和其他系统一起做回环测试，我们也称之为 loopback 的测试。串口发送模块和串口接收模块因为波特率相同，功能又互补，所以他们自身就可以直接连接到一起进行工作来实现最简单的 loopback 测试。大致流程为 PC 机的串口调试助手发送一串数据，经过 FPGA 后再传回到 PC 机的串口调试助手中打印显示。

模块设计

FPGA 对外可以看成一个整体的模块，如图 30-41 所示，输入需要时钟、复位，同时还有 PC 机发送过来的串行数据 rx 信号，输出为发送给 PC 机串行数据 tx 信号。我们之前学习过层次化的设计，这次也用到了，我们先设计好的 uart_rx 模块和 uart_tx 模块，再组装成系统，可以被看做是自底向上（Bottom-Up）的设计，需要将一个顶层模块来实例化 uart_rx 模块和 uart_tx 模块，我们将这个顶层模块取名为 rs232。

如图图 30-42 所示，我们将 uart_rx 模块的输出信号 po_data 和 po_flag 分别连接到 uart_tx 模块的输入信号 pi_data 和 pi_flag，中间的连线我们仍按照 uart_rx 模块的叫法来。loopback 的数据传输的详细过程为：PC 机的串口调试助手发送一帧串行数据，给 rs232 模块的 rx 端，rs232 的 rx 端接收到数据后传给 uart_rx 模块的 rx 端，uart_rx 模块负责解析出一帧数据中的有用数据，并将其转化为 8bit 并行数据 po_data 和数据有效标志信号 po_flag。8bit 并行数据 po_data 和数据有效标志信号 po_flag 通过 FPGA 的内部连线直接传输给 uart_rx 模块的 8bit 数据输入端 pi_data 和数据有效标志信号输入端 pi_flag，将接收到的并行数据重新封装成帧后串行发送到 tx 端，uart_rx 模块的 tx 端再把数据传给 rs232 的 tx 端，rs232 的 tx 端再将数据传回到 PC 机的串口调试助手中打印显示。实现了发送什么就接收什么，如果发送和接收的数据不一致，那就说明整个链路存在错误。

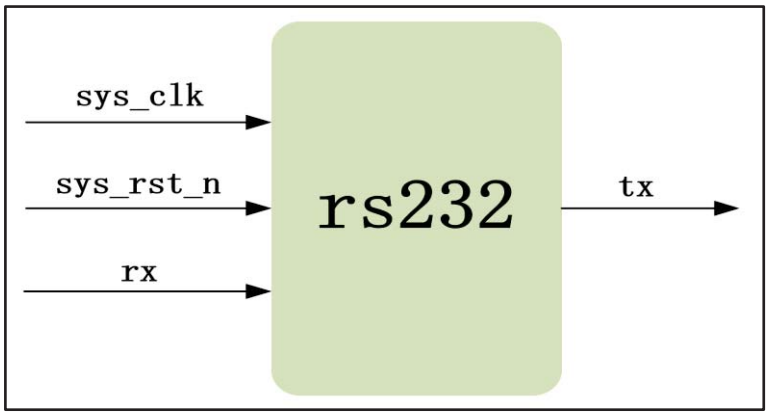

图 30-41 顶层模块框图

根据上面的分析设计出的 Visio 框图如图 30-42 所示。

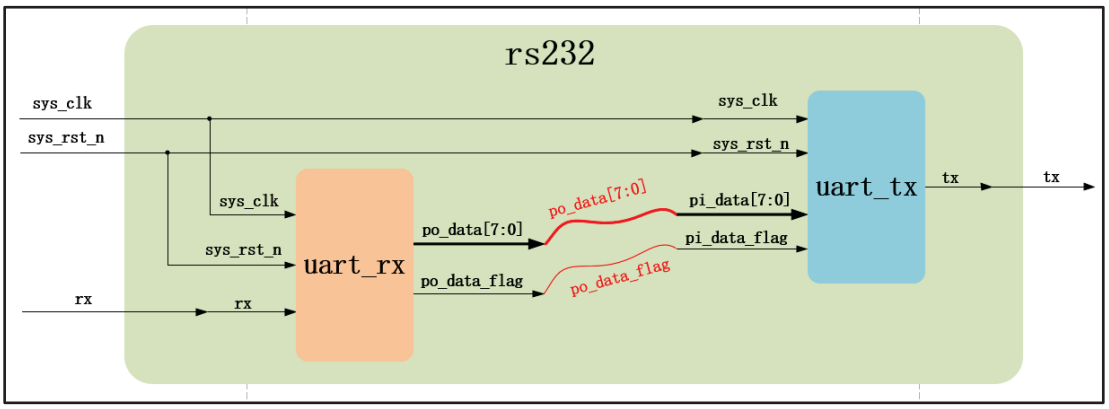

图 30-42 模块整体框图

端口列表与功能总结如表格 30-5 所示。

表格 30-5 顶层模块输入输出信号描述

信号	位宽	类型	功能描述
sys_clk	1Bit	Input	工作时钟，频率 50MHz
sys_rst_n	1Bit	Input	复位信号，低电平有效
rx	1Bit	Input	串口接收信号
tx	1Bit	Output	串口发送信号

代码编写

结构清晰了，底层模块也有了，剩下的工作就是在顶层实例化子模块，然后进行连线即可，连线的时候多比特数据需要特别注意位宽匹配要准确。顶层模块参考代码详见代码清单 30-5。

代码清单 30-5 顶层模块参考代码（rs232.v）

```
1  module  rs232(
2      input   wire    sys_clk     ,   //系统时钟 50MHz
3      input   wire    sys_rst_n   ,   //全局复位
4      input   wire    rx          ,   //串口接收数据
5
6      output  wire    tx              //串口发送数据
7  );
8
9  //*****************************************************************//
10 //****************** Parameter and Internal Signal *******************//
11 //*****************************************************************//
12
13 //parameter define
14 parameter   UART_BPS    =   14'd9600;       //比特率
15 parameter   CLK_FREQ    =   26'd50_000_000; //时钟频率
16
17 //wire  define
18 wire    [7:0]   po_data;
19 wire            po_flag
20
21 //*****************************************************************//
22 //************************** Instantiation ************************//
23 //*****************************************************************//
24
25 //-----------------------uart_rx_inst-----------------------
26 uart_rx
27 #(
```

```
28      .UART_BPS    (UART_BPS),          //串口波特率
29      .CLK_FREQ    (CLK_FREQ)           //时钟频率
30  )
31  uart_rx_inst
32  (
33      .sys_clk     (sys_clk     ),  //input             sys_clk
34      .sys_rst_n   (sys_rst_n   ),  //input             sys_rst_n
35      .rx          (rx          ),  //input             rx
36
37      .po_data     (po_data     ),  //output    [7:0]   po_data
38      .po_flag     (po_flag     )   //output            po_flag
39  );
40
41  //------------------------uart_tx_inst------------------------
42  uart_tx
43  #(
44      .UART_BPS    (UART_BPS),          //串口波特率
45      .CLK_FREQ    (CLK_FREQ)           //时钟频率
46  )
47  uart_tx_inst
48  (
49      .sys_clk     (sys_clk     ),  //input             sys_clk
50      .sys_rst_n   (sys_rst_n   ),  //input             sys_rst_n
51      .pi_data     (po_data     ),  //input     [7:0]   pi_data
52      .pi_flag     (po_flag     ),  //input             pi_flag
53
54      .tx          (tx          )   //output            tx
55  );
56
57  endmodule
```

RTL 原理图

顶层模块介绍完毕，使用 ISE 软件对实验工程进行编译，工程通过编译后查看实验工程 RTL 原理图。工程 RTL 原理图，具体见图 30-43。由图可知，实验工程的 RTL 原理图与实验整体框图相同，各信号线均已正确连接。

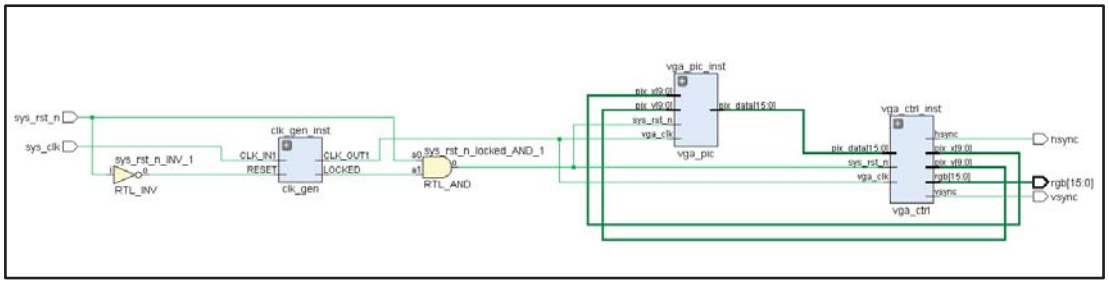

图 30-43 RTL 原理图

仿真文件编写

在编写仿真代码时，我们仍要模拟出 PC 机的串口调试助手发送串行数据帧的过程，如果直接使用 uart_rx 的仿真代码也是可以的，但我们为了让大家更熟练的使用 task，我们又增加了一个 task，在一个 task 中调用另一个 task，使得仿真代码更加简洁高效。顶层仿真参考代码详见代码清单 30-6。

代码清单 30-6 顶层仿真参考代码（tb_rs232.v）

```
1  module  tb_rs232();
2
```

```
 3 //////////////////////////////////////////////////////////////////////
 4 //****************** Parameter and Internal Signal *******************//
 5 //////////////////////////////////////////////////////////////////////
 6
 7 //reg   define
 8 reg     sys_clk;
 9 reg     sys_rst_n;
10 reg     rx;
11
12 //wire  define
13 wire    tx;
14
15 //////////////////////////////////////////////////////////////////////
16 //***************************** Main Code ****************************//
17 //////////////////////////////////////////////////////////////////////
18
19 //初始化系统时钟、全局复位和输入信号
20 initial begin
21     sys_clk    = 1'b1;
22     sys_rst_n <= 1'b0;
23     rx        <= 1'b1;
24     #20;
25     sys_rst_n <= 1'b1;
26 end
27
28 //调用任务 rx_byte
29 initial begin
30     #200
31     rx_byte();
32 end
33
34 //sys_clk:每 10ns 电平翻转一次，产生一个 50MHz 的时钟信号
35 always #10 sys_clk = ~sys_clk;
36
37 //创建任务 rx_byte，本次任务调用 rx_bit 任务，发送 8 次数据，分别为 0~7
38 task    rx_byte();  //因为不需要外部传递参数，所以括号中没有输入
39     integer j;
40     for(j=0; j<8; j=j+1)    //调用 8 次 rx_bit 任务，每次发送的值从 0 变化 7
41         rx_bit(j);
42 endtask
43
44 //创建任务 rx_bit，每次发送的数据有 10 位，data 的值分别为 0 到 7 由 j 的值传递进来
45 task    rx_bit(
46     input   [7:0]   data
47 );
48     integer i;
49     for(i=0; i<10; i=i+1)   begin
50         case(i)
51             0: rx <= 1'b0;
52             1: rx <= data[0];
53             2: rx <= data[1];
54             3: rx <= data[2];
55             4: rx <= data[3];
56             5: rx <= data[4];
57             6: rx <= data[5];
58             7: rx <= data[6];
59             8: rx <= data[7];
60             9: rx <= 1'b1;
61         endcase
62         #(5208*20); //每发送 1 位数据延时 5208 个时钟周期
63     end
64 endtask
65
66 //////////////////////////////////////////////////////////////////////
67 //************************** Instantiation ***************************//
```

```
68 //*****************************************************************//
69
70 //------------------------rs232_inst------------------------
71 rs232   rs232_inst(
72     .sys_clk    (sys_clk    ),  //input          sys_clk
73     .sys_rst_n  (sys_rst_n  ),  //input          sys_rst_n
74     .rx         (rx         ),  //input          rx
75
76     .tx         (tx         )   //output         tx
77 );
78
79 endmodule
```

仿真波形分析

打开 ModelSim 后先清空波形信号，重新添加要测试的模块，把 uart_rx 模块和 uart_tx 模块的波形都添加进来并分组，我们仍让波形跑 10ms，先将波形窗口信号的排列顺序和所画的波形图顺序一致再进行观察。模拟 PC 机发送 8 次（数据值从 0 到 7）串行数据的波形如图 30-44 所示。我们放大红色圈①处的波形进行详细观察。

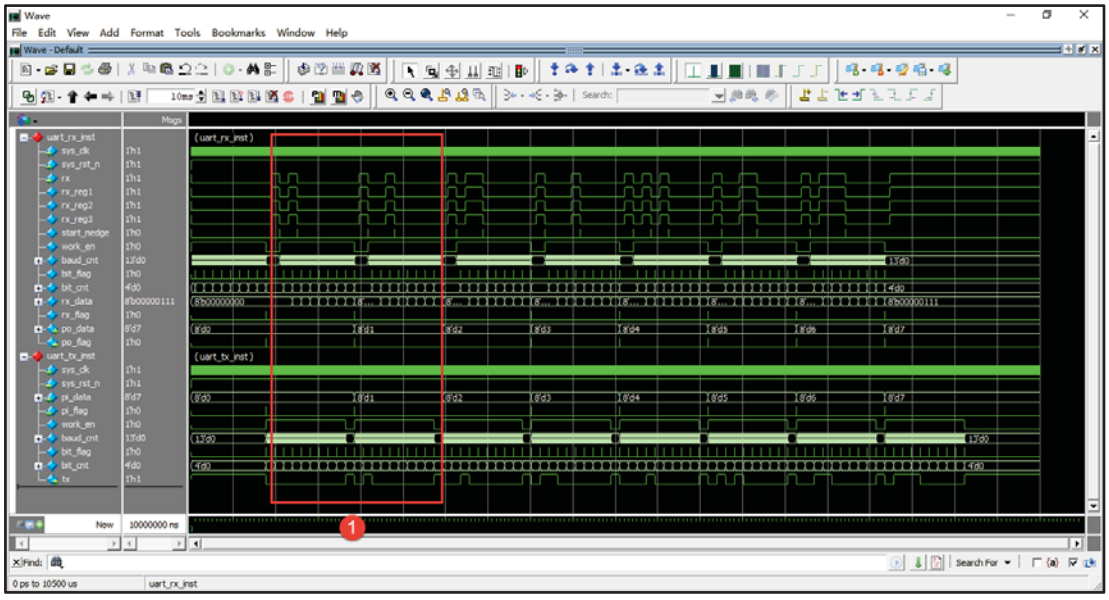

图 30-44 顶层仿真波形图（一）

如图 30-45 所示，我们可以看到发送的串行数据和对应接收的串行数据是一样的，也就说明我们的 rx 端口和 tx 端口的数据相同，同时也再一次验证了我们设计的 uart_rx 模块和 uart_tx 模块都是正确的。其实当我们验证过了 uart_rx 模块后完全可以不用单独再设计 uart_tx 模块的仿真代码，而继续使用 uart_rx 模块的仿真代码然后通过 loopback 测试来验证 uart_tx 模块设计的是否正确。

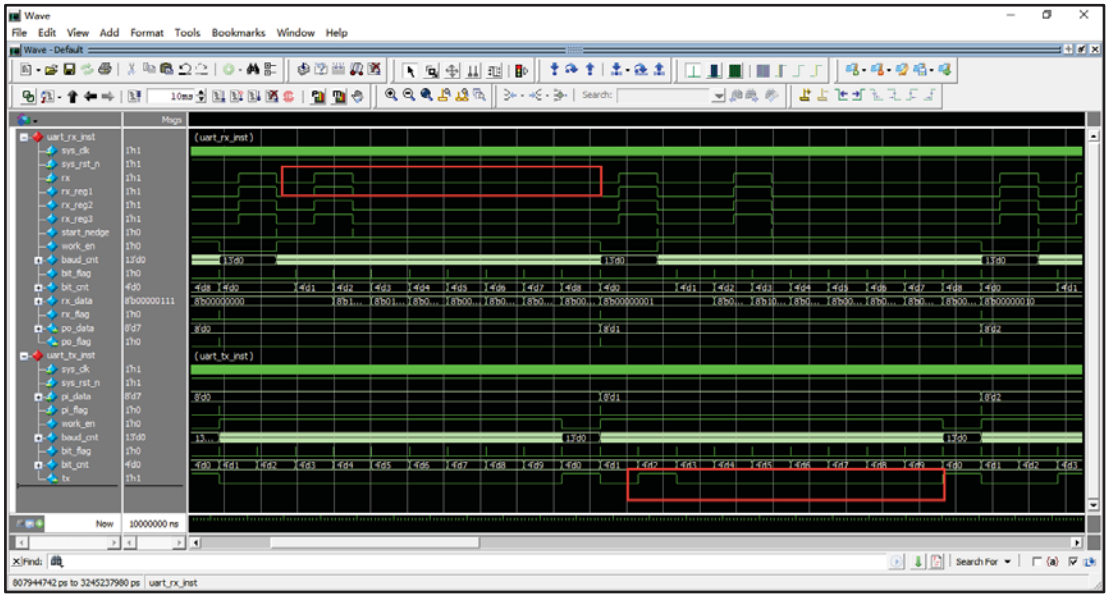

图 30-45 顶层仿真波形图（二）

30.3.4 上板验证测试

1. 引脚约束

仿真验证通过后，准备上板验证，上板验证时我们可选择用 RS232 串口线或 USB 线（踏浪 pro 开发上的 USB 接口为 Tape-C 型）进行验证，他们唯一的不同就是引脚的配置和跳帽的连接。使用 USB 线的引脚配置如表格 30-6 所示，使用串口线的引脚配置如表格 30-7 所示。

表格 30-6 USB 转串口引脚分配表

信号名	信号类型	对应引脚	备注
sys_clk	Iinput	V10	时钟
sys_rst_n	Input	R7	复位
rx	Input	F10	串口接收数据
tx	Output	G11	串口发送数据

表格 30-7 RS232 串口引脚分配表

信号名	信号类型	对应引脚	备注
sys_clk	Input	V10	时钟
sys_rst_n	Input	R7	复位
rx	Input	A12	串口接收数据
tx	Output	B12	串口发送数据

USB 转串口引脚约束代码如代码清单 30-7 所示：

代码清单 30-7 USB 转串口引脚约束参考代码（rs232.ucf）

```
1 #时钟复位信号
2 NET "sys_clk"   LOC = V10   |   IOSTANDARD = LVCMOS33;
3 NET "sys_rst_n" LOC = R7    |   IOSTANDARD = LVCMOS33;
4
5 #RS232 接口信号（USB 转串口）
6 NET "rx"        LOC = F10   |   IOSTANDARD = LVCMOS33;
7 NET "tx"        LOC = G11   |   IOSTANDARD = LVCMOS33;
```

RS232 串口引脚约束代码如代码清单 30-8 所示：

代码清单 30-8 RS232 串口引脚约束参考代码（rs232.ucf）

```
1 #时钟复位信号
2 NET "sys_clk"   LOC = V10   |   IOSTANDARD = LVCMOS33;
3 NET "sys_rst_n" LOC = R7    |   IOSTANDARD = LVCMOS33;
4
5 #RS232 接口信号
6 NET "rx"        LOC = A12   |   IOSTANDARD = LVCMOS33;
7 NET "tx"        LOC = B12   |   IOSTANDARD = LVCMOS33;
```

2. 结果验证

若要进行 USB 转串口的验证，按照如图 30-46 所示连接 12V 电源、下载器、USB 数据线以及短路帽；若要使用串口线进行验证，则按照如图 30-47 所示连接 12V 电源、下载器、串口线。

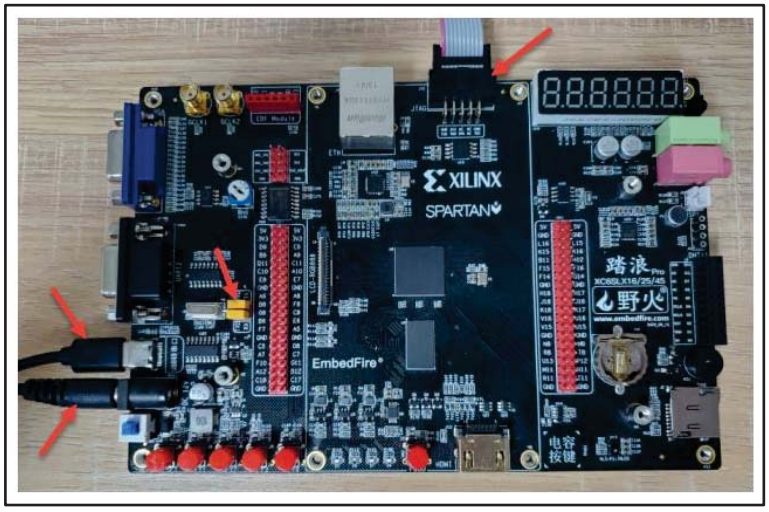

图 30-46 程序下载连接线（一）

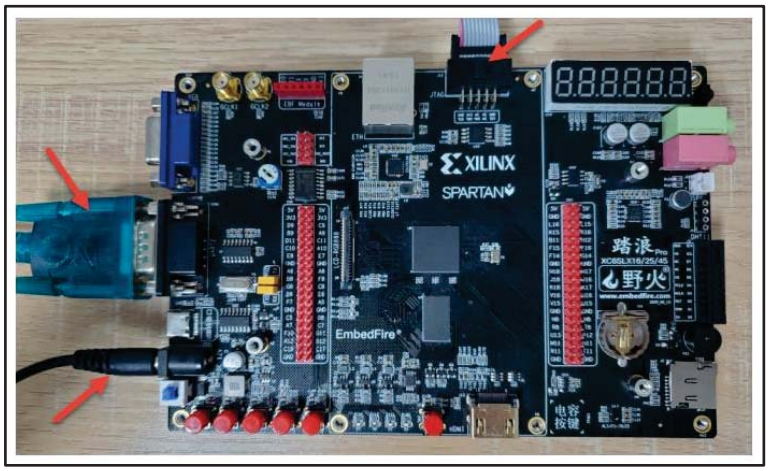

图 30-47 程序下载连接线（二）

如图 30-48 所示，使用“ISE iMPACT”为开发板下载程序。

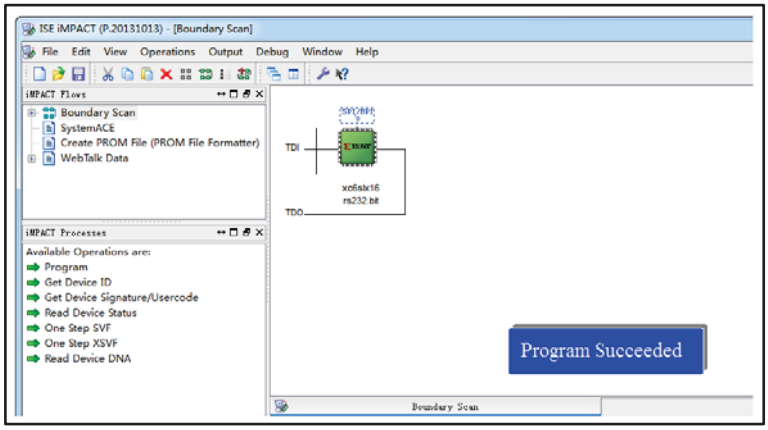

图 30-48 下载成功界面

下载成功后即可以开始验证了。按照实验目标描述进行操作，若显示结果与实验目标描述相同，则说明验证成功。

若我们使用的是绑定 USB 转串口的引脚进行编译下载的工程，我们就使用 USB 连接线（Tape-C 型）进行 PC 机与开发板连接进行 lookback 的测试；若我们使用的是绑定 RS232 串口的引脚进行编译下载的工程，我们就使用 RS232 串口线进行 PC 即与开发板连接进行 lookback 的测试，大家可根据自己已有的连接线进行选择。

连接下载完毕之后我们即可开始测试。打开串口助手，从串口中我们可以看到检测到有接口，若没有检测到接口，请先检查接线是否连接正确。同时我们将波特率设置为我们代码中的波特率 9600，具体的设置如图 30-49 所示。

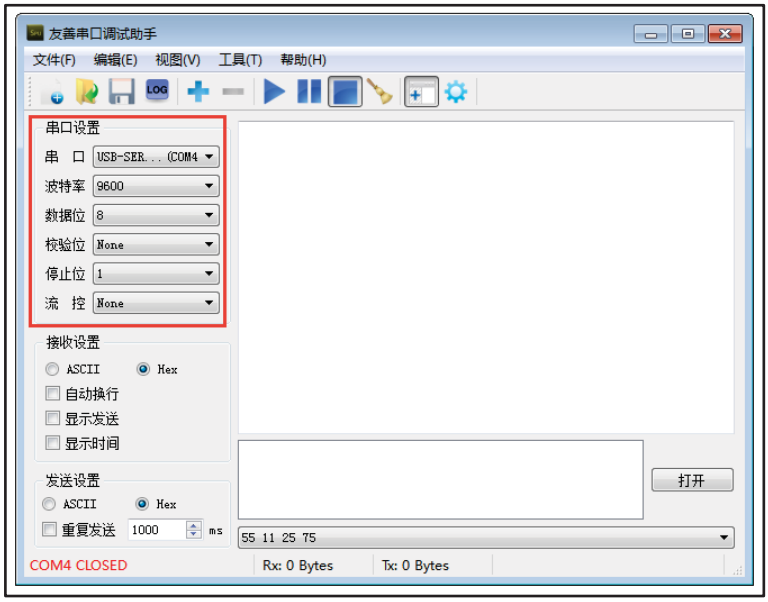

图 30-49 串口回环测试图（一）

设置完之后我们就可以进行发送数据了，我们发送任意字节数据，若接收的数据与发送的数据一致，则说明验证成功，如图 30-50 所示。

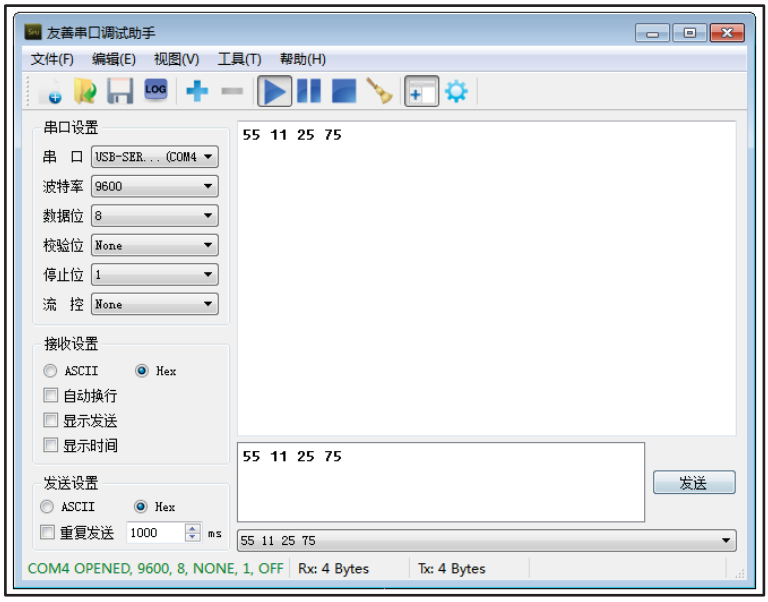

图 30-50 串口回环测试图（二）

30.4 章末总结

在本章的 Testbench 的设计中我们第一次使用到了 task 任务以及 for 循环语句，这两个语法都在仿真中使用的较多，虽然都是可以综合的但还是推荐初学者尽量不要在 RTL 代码中使用，尤其是对它们理解不深刻的情况下。而我们在 Testbench 中使用就不用担心这么多，且可以大大简化我们的代码，提高效率，是十分好用的，也推荐大家以后再 Testbench 中多尝试使用。

我们还学习到一个很好用的调试方法，就是 loopbang 测试。本章中我们只是做了最简单的串口回环，以后我们还可以在 uart_rx 模块和 uart_tx 模块中间加入我们设计的更加复杂的其他模块来进行验证。

本章可以说是我们学习 FPGA 以来的最有代表性的一个小项目了，无论是波形设计还是代码编写都比之前要复杂，所以说这个小项目非常有意义，也很重要，希望大家能够再次深刻体会我们系统的设计方法和流程，并能够自己完全实现。对于后面的学习还会有更加复杂、系统的实战项目，需要大家在学习的过程中多思考、多练习、多总结，最终做到完全掌握应用自如。

新语法总结

重点掌握

1. task（可以互相调用）
2. for（虽不多见，但是在 Testbench 中很高效）

知识点总结

1. 理解亚稳态产生的原理，掌握单比特数据从慢速时钟域到快速时钟域处理亚稳态的方法。
2. 学会使用边沿检测，并记住代码的格式，理解原理。
3. 串并转换是接口中很常用的一种方法，用到了移位，要熟练掌握。
4. 掌握 loopback 测试的方法，以后用于我们模块中代码的调试。

30.5 拓展训练

1. 通过对亚稳态现象的分析，我们可以进一步思考，系统复位往往是连接到外部的按键上，那按键对与 FPGA 系统的复位输入是否也存在“亚稳态”的问题呢？如果存在我们该如何解决？
2. 能否将我们前面的按键消抖也用边沿检测的方法来实现呢？
3. 尝试更改更高频率的波特率进行 loopback 的测试。