数字逻辑与处理器基础实验 实验报告

> 实验三：串口收发器和存储器的使用
>
> 无 13 管思源 2021012702

实验目的

- 了解和掌握 UART 的工作原理
- 掌握仿真验证方法
- 为后续实验内容做准备

设计方案

实验已经提供了 UART 串口收发的完整代码（`UART_MEM`模块），要求我们设计仿真测试验证其功能。因此，我们创建一个`testbench`，它需要对`UART_MEM`模块的所有输入生成激励信号，并验证其输出是否符合预期。

`UART_MEM`模块的输入为

- `clk`：时钟信号，需要我们生成一个 100 MHz 的时钟
- `rst`：复位信号，一开始为高电平，过一段时间后设置为低电平
- `mem2uart`：发送控制信号，在接受期间为低电平，接收完成后设置为高电平
- `Rx_Serial`：串口接收信号，需要根据消息生成 UART 的激励信号

输出为

- `recv_done`：接收完成信号，需要验证其在接收完成后为高电平
- `send_done`：发送完成信号，需要验证其在发送完成后为高电平
- `Tx_Serial`：串口发送信号，需要验证其是否与消息的后半段一致

显然，设计的重点在于激励信号的生成与发送消息的验证，这相当于在`testbench`中手写一个 UART 串口收发器与待测模块进行通信。

我们首先使用`$readmemh`读取给定的十六进制消息

```
reg [7:0] msg [0:2*BYTE_LENGTH-1];
initial $readmemh("../../../../../hex.txt", msg);
```

然后遍历消息，逐字节按 UART 协议生成激励信号

```
for (i = 0; i < 2*BYTE_LENGTH; i = i + 1) begin
    // Start bit
    Rx_Serial = 0;
    #104167; // ns for 1 bit under 9600 baudrate
    // Data bits
    for (j = 0; j < 8; j = j + 1) begin
        Rx_Serial = msg[i][j];
        #104167; // ns for 1 bit under 9600 baudrate
    end
    // Stop bit
    Rx_Serial = 1;
    #156250; // ns for 1.5 bit under 9600 baudrate
end
```

这里每个字节的停止位为 1.5 位，与指导书要求一致。

在接收完成后，检查 `recv_done` 为高电平并将 `mem2uart` 设置为高电平（代码略）。此时监听 `Tx_Serial` 并采样得到模块发送的信号，将其与消息的后半段进行比较

```
for (i = 0; i < BYTE_LENGTH; i = i + 1) begin
    while (Tx_Serial == 1) #10;
    #156250; // ns for 1.5 bit under 9600 baudrate
    for (j = 0; j < 8; j = j + 1) begin
        msg_out[i][j] = Tx_Serial;
        #104167; // ns for 1 bit under 9600 baudrate
    end
    if (msg_out[i] != msg[i + BYTE_LENGTH]) begin
        pass = 0;
        $display("Error: data mismatch at %d", i);
    end
end
```

需要注意的是，这里我们在每个字节开始时等待 `Tx_Serial` 为低电平，来与发送的开始位对齐。同时，我们在检测到开始位后延时 1.5 位，从而在每个比特的中间采样。最后，我们在第 8 个消息位采样完成后仍延时了 1 位，来避免将消息位检测呈下一个字节的开始位。

在发送完成后，还需要检查 `send_done` 为高电平（代码略）。

对于所有需要验证正确性的输出信号，我设置了一个 `pass` 寄存器变量。当输出结果不符合预期时，`pass` 就会被设置为低电平。这样一来，我们可以方便地通过仿真结束后 `pass` 的值来判断测试是否全部通过。

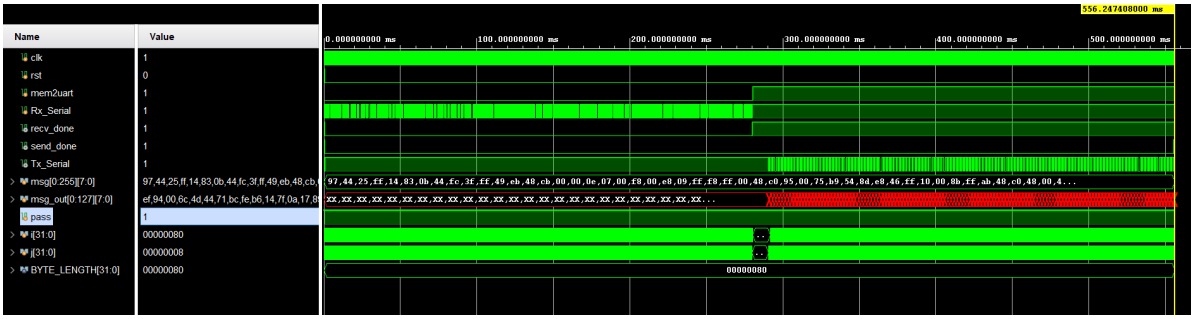
此外，值得一提的是，我在初始化、接收、发送三个阶段之间都加入了随机给定的延时，用于模拟真实的应用场景（详见代码）。

仿真结果与分析

对于上述设计的测试，我们在 Vivado 中进行行为级仿真。

一开始，仿真的进度非常慢，这主要是因为我们传输的消息较长（接收和发送总计 512 * 3 个 word、6 KB），且串口通信的波特率远低于时钟频率，导致测试运行结束本身需要的时间很长。

后来，我将传输消息长度降为 32 * 3 个 word，对应上文 `BYTE_LENGTH = 128`，仿真在约 3 分钟内结束，以下是仿真结果：



可以看到，`pass` 寄存器（左侧高亮的一行）在测试结束后为高电平，说明所有输出信号都符合预期。

进一步还可以观察到，前半段（接收阶段）`Rx_Serial` 在高低电平之间不断转换，的确在接收消息，而 `Tx_Serial` 始终为高电平，说明此时模块没有发送消息。在后半段（发送阶段）`Tx_Serial` 在高低电平之间不断转换，的确在发送消息，而 `Rx_Serial` 始终为高电平，说明此时模块没有接收消息。在 `Rx_Serial` 停止变动时，`recv_done` 变为高电平，说明接收完成。在 `Tx_Serial` 停止变动后，`send_done` 变为高电平，说明发送完成。

另外，我们注意到，在接收和发送阶段之间还相隔了相当长的一段时间，这段时间里 `mem2uart` 信号为高电平，所以这不是测试中加入的延时导致的。按理说，`mem2uart` 信号置为高电平后，模块应该立即开始发送消息，为什么还等待了那么长时间呢？

阅读代码后我发现，问题主要出在 `mem` 模块中 `memData` 的定义上（如下）

```
reg [31:0] memData [MEM_SIZE:1];
```

由于`memData`跳过了地址 0，而`UART_MEM`模块中使用它的地址是从 0 开始计数，因此在发送时做了特殊处理，需要在`addr1 == 16'd0`时禁用发送模块并等待 20 * 4 个比特周期。加上一开始`cntByteTime`累加到`CLKS_PER_BIT*20`的 20 个比特周期，从`mem2uart`信号置为高电平到发送开始之间就相隔了 100 个比特周期，也就是 100 / 9600 = 10.4 ms，这与仿真波形相符。

> 鉴于这是实验提供的测试代码，我并不打算修改掉这个问题。但这个发现恰能说明仿真使我们能够更直观地观察模块的行为、发现代码的漏洞，因此这里单独报告出来

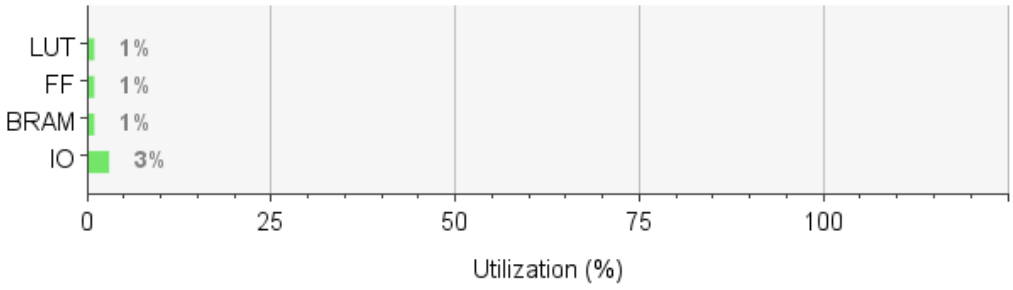
综合结果与分析

下面综合的结果仍基于`MEM_SIZE = 512`，未降低传输消息长度。

硬件资源使用情况

Summary

Resource	Utilization	Available	Utilization %
LUT	121	20800	0.58
FF	193	41600	0.46
BRAM	0.50	50	1.00
IO	7	250	2.80



Hierarchy

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	Block RAM Tile (50)	Bonded IOB (250)	BUFGCTRL (32)
N UART_MEM	121	193	63	121	0.5	7	1
mem1 (mem)	9	3	3	9	0.5	0	0
uart_rx_inst (uart_rx)	73	33	31	73	0	0	0
uart_tx_inst (uart_tx)	28	29	14	28	0	0	0

如图所示，系统使用了 121 个 LUT 和 193 个寄存器。

等等，模块里不是存储了 512 * 2 个 word 吗？怎么只用了 193 个寄存器？经过探究，我发现问题出在这个 BRAM 上，这是一种高速的片上随机存储模块，是 FPGA 的硬件资源之一。当模块代码符合 BRAM 规范（大量随机数据的同步读写）时，Vivado 会自动将其综合为 BRAM，而不是寄存器。

在这块开发板上，一块 BRAM（RAMB36E1）可以存储 1024 个 word（4 KB），它还可以拆分成两个储存 512 个 word 的 BRAM（RAMB18E1），也就是上面我们看到的 0.5 BRAM。

此外，我们还注意到，硬件资源中只显示了用于储存后半段数据的`mem1`模块，而没有显示用于储存前半段指令`mem0`模块。这是因为我们在接收完前半段指令后并没有对其进行任何操作，也与模块输出无关，因此 Vivado 在综合时将其优化掉了。

静态时序分析结果

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.719 ns	Worst Hold Slack (WHS):	0.077 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	438	Total Number of Endpoints:	438	Total Number of Endpoints:	196

All user specified timing constraints are met.

结果显示，系统已满足时钟约束，其中建立时间有 5.719 ns 的裕量，非常充足。

关键代码及文件清单

代码文件（按模块例化结构）

- `testbench.v`：仿真测试文件
  - `UART_MEM.v`：UART 收发模块的顶层模块，用于控制收发逻辑
    - `uart_rx.v`：UART 接收模块
    - `uart_tx.v`：UART 发送模块
    - `mem.v`：存储器模块

其他文件

- `hex.txt`：十六进制消息文件，用于仿真时读取，与实验提供的相同
- `UART_MEM.xdc`：硬件约束文件
- `UART_MEM.bit`：生成的比特流文件，可直接烧写到开发板
- `img/*.png`：仿真和综合结果截图