ModelSim课程实践报告

SPI 参数寄存器配置与反馈项目报告

1. 项目概述与要求

本项目旨在通过 Verilog HDL 实现一个 SPI 从机模块，该模块能够接收 SPI 主机发送的命令，实现对内部 8 个参数寄存器（`regs[0]` 到 `regs[7]`）的配置，并支持将这些参数寄存器中的数据反馈给主机。

自定义输入数据格式:

- **同步头**: 固定为 `0x7E7E`（2 字节）
- **参数标记 (TAG)**: 1 字节（范围 `0x00` ~ `0x07` 用于参数配置， `0x88` 用于参数反馈）
- **参数信息 (DATA)**: 1 字节

功能要求:

1. **参数配置**: 当接收到的参数标记 (TAG) 在 `0x00` 到 `0x07` 范围内时，从机将接收到的参数信息 (DATA) 写入对应地址的参数寄存器中.
2. **参数反馈**: 当接收到的参数标记 (TAG) 为 `0x88` 时，从机将内部 `regs[0]` 到 `regs[7]` 中的所有 8 个字节的数据依次通过 SPI 接口反馈给主机.

2. SPI 协议基础信息

SPI (Serial Peripheral Interface) 是一种高速、全双工、同步的串行通信总线协议，广泛应用于微控制器与外设之间的数据交换。

SPI 的主要特点:

- **四线制**: 通常包括四根信号线：
  - **SCLK (Serial Clock)**: 串行时钟，由主机提供，用于同步数据传输。
  - **MOSI (Master Out Slave In)**: 主机输出，从机输入的数据线。
  - **MISO (Master In Slave Out)**: 主机输入，从机输出的数据线。
  - **CS_n (Chip Select Not)**: 片选信号，低电平有效，用于选择目标从机设备。
- **全双工**: 主机和从机可以同时进行数据发送和接收。
- **同步**: 数据传输由时钟信号同步。
- **MSB First**: 通常数据传输按最高有效位 (MSB) 优先的顺序进行。
- **CPOL/CPHA**: SPI 协议有四种模式，由时钟极性 (CPOL) 和时钟相位 (CPHA) 决定。本项目中， `CPOL=0, CPHA=0` ，表示时钟空闲时为低电平，在时钟的第一个跳变沿（上升沿）采样数据，在时钟的第二个跳变沿（下降沿）改变输出数据.

3. 代码实现方式和逻辑

本项目包含两个 Verilog 模块：`spi_slave_param` (从机) 和 `tb_spi_master` (测试主机)。

3.1 `spi_slave_param` (从机模块)

从机模块 `spi_slave_param` 负责根据 SPI 协议接收来自主机的数据，解析命令，并执行参数配置或参数反馈操作。

模块接口:

- `sclk`: SPI 时钟输入
- `cs_n`: 片选信号输入 (低有效)
- `mosi`: 主机输出，从机输入的数据线
- `miso`: 从机输出，主机输入的数据线

内部状态机:

从机模块采用一个 5 状态的状态机来管理数据接收和命令处理流程:

- `WAIT1 (3'd0)`: 等待接收第一个同步头 `0x7E`.
- `WAIT2 (3'd1)`: 已经收到一个 `0x7E`，等待接收第二个同步头 `0x7E`.
- `READ_TAG (3'd2)`: 收到两个 `0x7E` 同步头后，进入此状态接收参数标记 (TAG) 字节.
- `READ_DAT (3'd3)`: 接收参数信息 (DATA) 字节. 根据 `tag_byte` 的值，决定是写入寄存器还是进入反馈模式.
- `FEEDBACK (3'd4)`: 当 `tag_byte` 为 `0x88` 时进入此状态，将内部 `regs[0]` 到 `regs[7]` 的数据依次回送给主机.

主要逻辑:

- **下降沿推出下一位到 `miso`**:
  - 当 `cs_n` 为高电平时，`miso` 处于高阻态 `1'bz`.
  - 当 `cs_n` 为低电平且状态为 `FEEDBACK` 时，`miso` 会在每个时钟周期的下降沿按 MSB first 的顺序输出 `regs[fb_ptr]` 中的当前位. `fb_ptr` 用于指向当前正在反馈的寄存器.
- **上升沿采样 `mosi`，累积到 `shift_reg`**:
  - 在 `cs_n` 为高电平（非片选）时，模块复位，状态机回到 `WAIT1` 状态，并清零位计数器 `bit_cnt`、移位寄存器 `shift_reg` 等.
  - 在 `cs_n` 为低电平时，每个时钟的上升沿都会采样 `mosi` 的当前值，并将其移入 `shift_reg` 的最低位.
  - 当 `bit_cnt` 达到 `7` (即接收到一个完整字节) 时:
    - `byte_rcv` 锁存完整的 8 位数据.
    - 根据当前 `state` 决定状态转换：

        - 在 `WAIT1` 和 `WAIT2` 状态下，检查接收到的字节是否为 `0x7E` 以完成同步头匹配.
        - 在 `READ_TAG` 状态下，将接收到的字节存入 `tag_byte`，并切换到 `READ_DAT` 状态.
        - 在 `READ_DAT` 状态下，将接收到的字节存入 `data_byte`. 如果 `tag_byte` 是 `0x88`，则进入 `FEEDBACK` 状态并初始化 `fb_ptr`；否则（`tag_byte` 在 `0x00` 到 `0x07` 范围内），将 `data_byte` 写入 `regs[tag_byte[2:0]]`，然后返回 `WAIT1` 状态.
        - 在 `FEEDBACK` 状态下，每次发送完一个字节后 `fb_ptr` 递增，直到所有 8 个字节发送完毕后回到 `WAIT1` 状态.
    - `bit_cnt` 清零，准备接收下一个字节.
- 如果 `bit_cnt` 未达到 `7`，则 `bit_cnt` 递增.

3.2 `tb_spi_master` (测试主机模块)

测试主机模块 `tb_spi_master` 用于模拟 SPI 主机，向 `spi_slave_param` 发送命令和数据，并验证从机的响应。

模块接口:

- `clk`, `cs_n`, `mosi`: 作为输入连接到 `spi_slave_param` 的相应端口.
- `miso`: 作为输出连接到 `spi_slave_param` 的相应端口.

任务定义:

- `spi_byte(input [7:0] din, output [7:0] dout)`:
    - 此任务模拟 SPI 单字节传输。它在时钟下降沿设置 `mosi` 的值，并在时钟上升沿采样 `miso` 的值，从而实现 MSB First 的数据传输.
- `spi_cmd(input [7:0] tag, input [7:0] data)`:
    - 此任务模拟发送一个完整的 SPI 命令包，包括拉低 `cs_n`，发送两个 `0x7E` 同步头，然后发送 `tag` 字节和 `data` 字节，最后拉高 `cs_n` 结束命令.

仿真流程:

1. **初始化**: 在仿真开始时，将从机内部的 `regs` 寄存器清零，避免出现未知值 `X`.
2. **参数配置**: 循环 `8` 次，依次向 `regs[0]` 到 `regs[7]` 写入数据。例如，`regs[0]` 写入 `0x11`，`regs[1]` 写入 `0x22`，以此类推. 每次写入都通过调用 `spi_cmd` 任务完成。
    - 通过一个 `for (i = 0; i < 8; i++) spi_cmd(i, (8'h11 << i));` 的循环，依次将 8 个参数寄存器写成:

```
// 依次将 regs[0]~regs[7] 写入如下值:
// regs[0] = 0x11   // 0x11 << 0
// regs[1] = 0x22   // 0x11 << 1
// regs[2] = 0x44   // 0x11 << 2
```

```
// regs[3] = 0x88   // 0x11 << 3
// regs[4] = 0x10   // 0x11 << 4 = 0x110，但实际只取低 8 位 = 0x10
// regs[5] = 0x20   // 0x11 << 5 = 0x220，但实际只取低 8 位 = 0x20
// regs[6] = 0x40   // 0x11 << 6 = 0x440，但低 8 位 = 0x40
// regs[7] = 0x80   // 0x11 << 7 = 0x880，但低 8 位 = 0x80
```

`8'h11 << i` 左移操作先算出一个较宽的值（如 `0x11 << 4 = 0x110`），然后截取低 8 位送给 DUT。

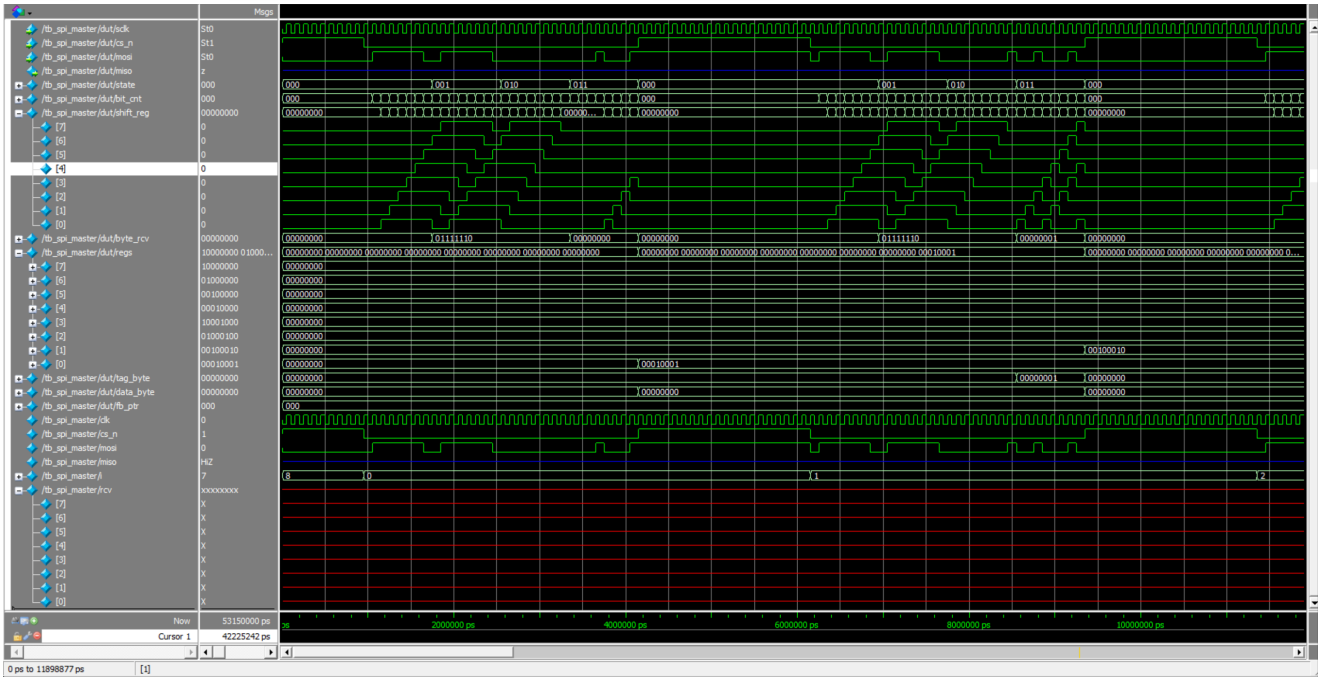
3. **参数反馈**:
   - 拉低 `cs_n`.
   - 发送 `0x7E7E` 同步头.
   - 发送 `0x88` 作为 `tag` 字节 (表示进入反馈模式).
   - 发送一个任意的 `0x00` 作为 `data` 字节 (在反馈模式下 `data` 字节无意义).
   - 随后，连续调用 `spi_byte` 任务 8 次，每次发送 `0x00` (作为主机输出的占位符)，并接收从机反馈的 `regs` 数据到 `rcv` 变量中.
   - 最后拉高 `cs_n` 结束反馈命令.
4. **仿真结束**: `#1000 $stop;` 停止仿真.

4. 仿真波形

以下是 ModelSim 仿真生成的波形截图，展示了不同阶段的 SPI 通信情况。

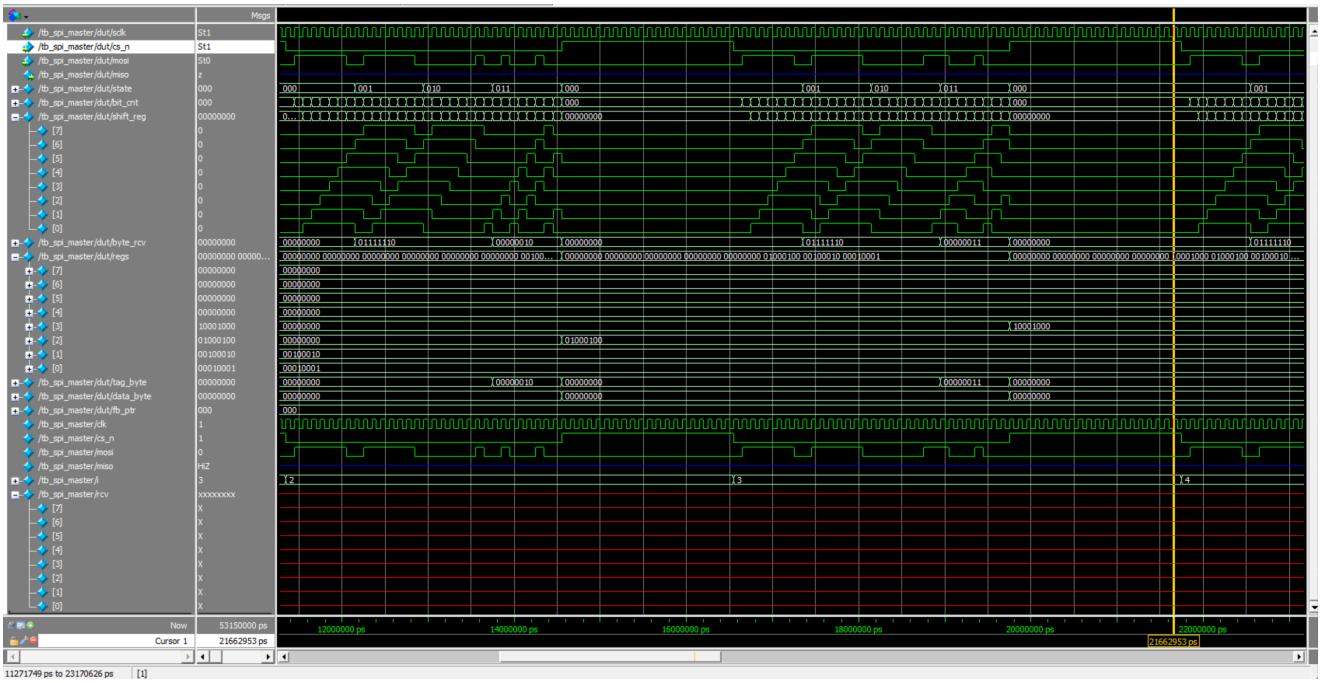
4.1 `byte1-byte2.png` (第一个与第二个参数写入)

此波形展示了主机发送第一个同步头 `0x7E` 和第二个同步头 `0x7E`，随后发送 TAG `0x00` 和 DATA `0x11` 到从机，将 `0x11` 写入 `regs[0]` 的过程。紧接着发送同步头 `0x7E7E` TAG `0x01` 和 DATA `0x22` 到从机，将 `0x22` 写入 `regs[1]` 的过程。



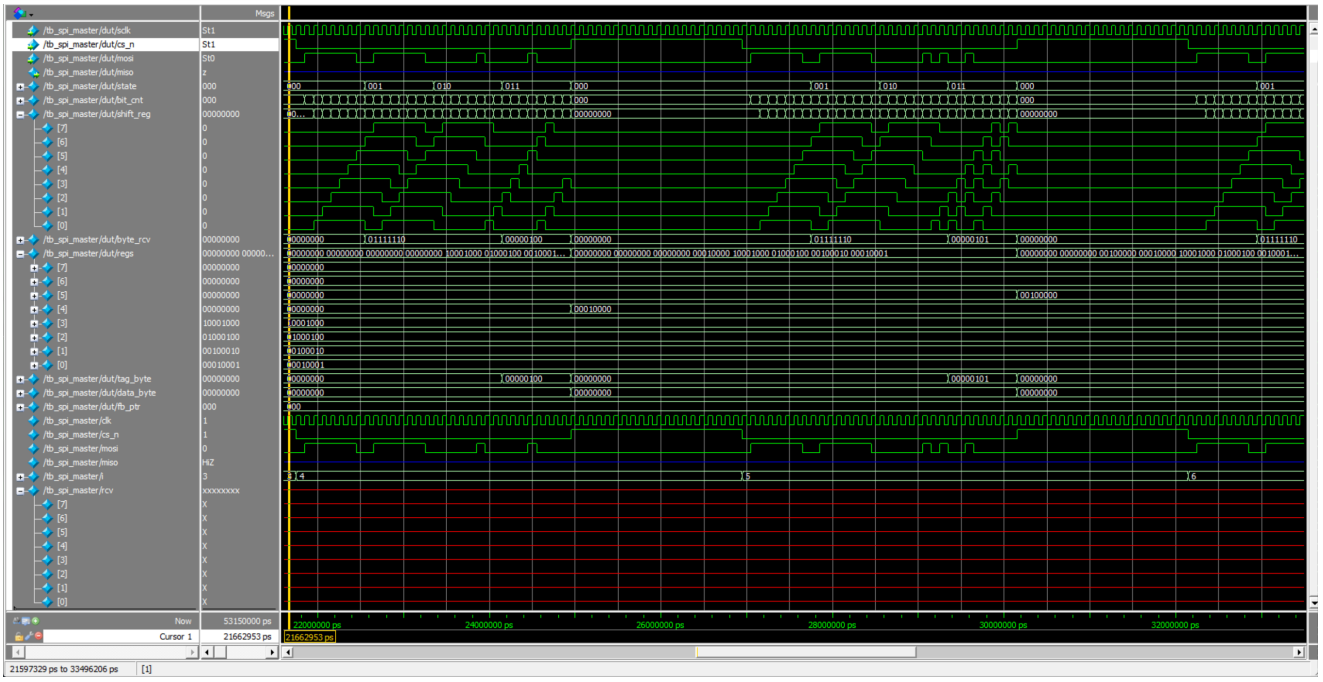
4.2 `byte3-byte4.png` (第三个和第四个参数写入)

此波形展示了主机发送同步头 `0x7E7E`，然后发送 TAG `0x02` 和 DATA `0x44` 到从机，将 `0x44` 写入 `regs[2]`，紧接着发送同步头 `0x7E7E` TAG `0x03` 和 DATA `0x88` 到从机，将 `0x88` 写入 `regs[3]` 的过程。



4.3 `byte5-byte6.png` (第五个和第六个参数写入)

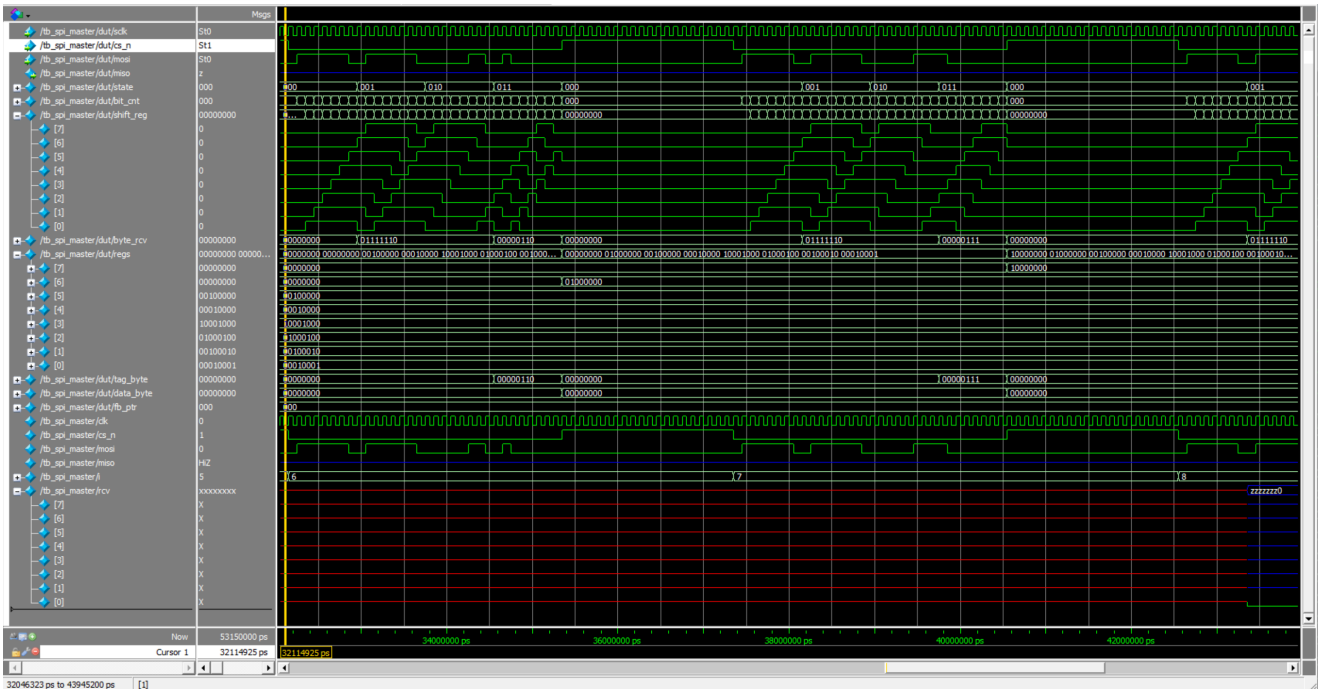
此波形展示了主机发送同步头 `0x7E7E`，然后发送 TAG `0x04` 和 DATA `0x10` 到从机，将 `0x10` 写入 `regs[4]`，紧接着发送同步头 `0x7E7E` TAG `0x05` 和 DATA `0x20` 到从机，将 `0x20` 写入 `regs[5]` 的过程。



4.4 `byte7-byte8.png` (第七个和第八个参数写入)

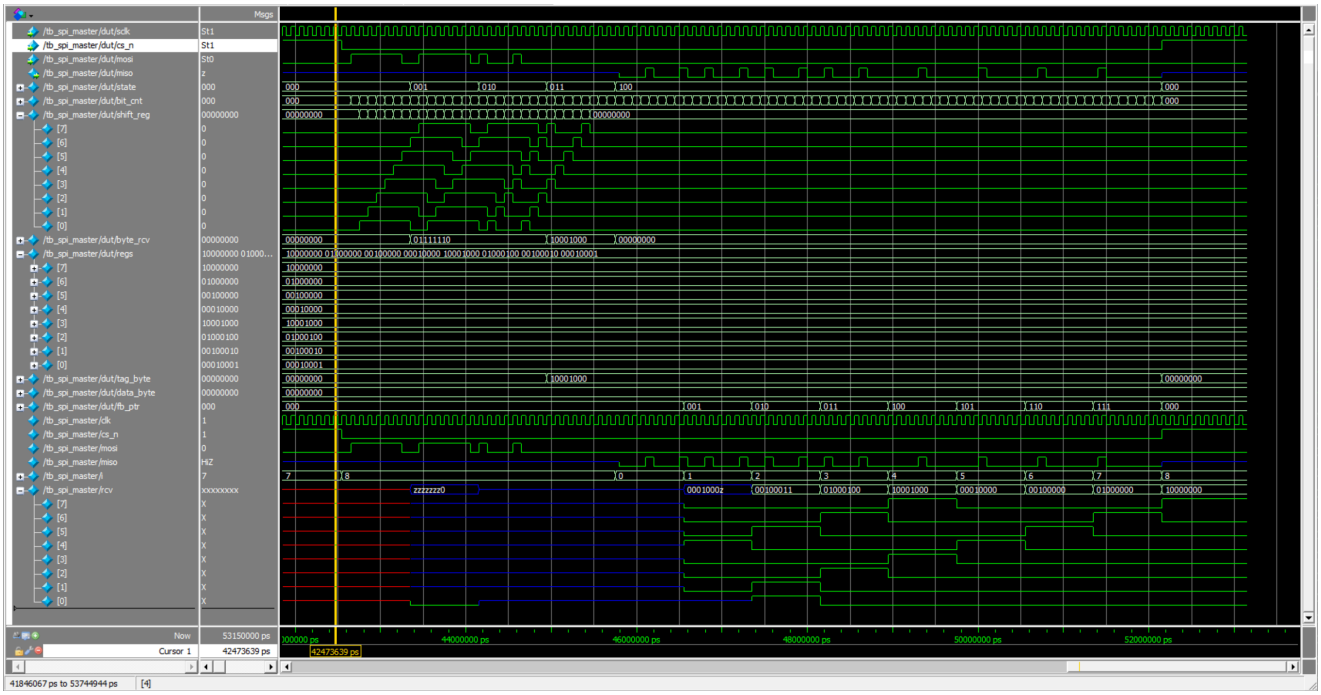
此波形展示了主机发送同步头 `0x7E7E`，然后发送 TAG `0x06` 和 DATA `0x40` 到从机，将 `0x40` 写入 `regs[6]`，紧接着发送同步头 `0x7E7E` TAG `0x07` 和 DATA `0x80` 到从机，将

`0x80` 写入 `regs[7]` 的过程。



4.5 `feedback.png` (参数反馈)

此波形展示了主机发送同步头 `0x7E7E` 和 TAG `0x88` (表示反馈模式)，以及一个任意 DATA 字节 `0x00`。随后，从机将 `regs[0]` 到 `regs[7]` 中存储的数据（即 `0x11, 0x22, 0x44, 0x88, 0x10, 0x20, 0x40, 0x80`）依次通过 `miso` 线反馈给主机。`Data` 反馈在 `tb_spi_master/rcv` 中。



5. 总结

本项目成功地实现了基于 SPI 协议的参数寄存器配置与反馈功能。通过 Verilog HDL 设计的 `spi_slave_param` 模块能够正确解析自定义的 SPI 命令格式，实现对内部 8 个参数寄存器的读写操作。`tb_spi_master` 测试主机模块验证了从机在参数配置和参数反馈两种工作模式下

的正确性。ModelSim 仿真波形清晰地展示了 SPI 通信的时序和数据传输过程，证明了设计的有效性。

6. Verilog 代码

6.1 `spi_slave_param.v` (从机代码)

```
`timescale 1ns/1ps
module spi_slave_param
(
    input  wire sclk,    // SPI 时钟，CPOL=0, CPHA=0
    input  wire cs_n,    // 片选，低有效
    input  wire mosi,    // 主机发 从机收
    output reg  miso     // 从机发 主机收，高阻或有效
);

    // ------------------------------------------------------
    // 状态机定义（5 个状态）
    //   0 = WAIT1   : 等待第一个 0x7E
    //   1 = WAIT2   : 已经收到一个 0x7E，等待第二个 0x7E
    //   2 = READ_TAG：收到两个 0x7E 后，这里接收 TAG 字节
    //   3 = READ_DAT：接收 DATA 字节（或进入反馈模式）
    //   4 = FEEDBACK：TAG=0x88 时进入，从 regs[0] … regs[7] 回送
    // ------------------------------------------------------
    localparam WAIT1    = 3'd0;
    localparam WAIT2    = 3'd1;
    localparam READ_TAG = 3'd2;
    localparam READ_DAT = 3'd3;
    localparam FEEDBACK = 3'd4;

    reg [2:0]  state;        // 当前状态
    reg [2:0]  bit_cnt;      // 接收位计数 0~7
    reg [7:0]  shift_reg;    // 接收移位寄存器（7:0 用来拼 byte）
    reg [7:0]  byte_rcv;     // 一字节接收完成后锁存：{shift_reg[6:0], mosi}
    reg [7:0]  tag_byte;     // 存 TAG 字节
    reg [7:0]  data_byte;    // 存 DATA 字节（若 TAG!=0x88，则写入 regs）
    reg [7:0]  regs [7:0];   // 8 个参数寄存器
    reg [2:0]  fb_ptr;       // 反馈指针 0~7

    // ------------------------------------------------------
    // 下降沿推出下一位到 miso
    //   - cs_n = 1 时：高阻
    //   - state=FEEDBACK 时：逐位输出 regs[fb_ptr]
    // ------------------------------------------------------
    always @(negedge sclk or posedge cs_n) begin
        if (cs_n) begin
            miso   <= 1'bz;
            fb_ptr <= 3'd0;
        end else if (state == FEEDBACK) begin
```

```
            // 每个时钟周期 bit_cnt=0..7，按 MSB first 从 regs[fb_ptr] 推出
            miso <= regs[fb_ptr][7 - bit_cnt];
        end
    end

    // ------------------------------------------------------
    // 上升沿采样 MOSI，累积到 shift_reg；bit_cnt 0~7 循环
    // 收满 1 字节 (bit_cnt==7) 时：
    //   - 拼 byte_rcv
    //   - 根据当前 state 做状态转换
    //   - 收完后 清 bit_cnt
    // ------------------------------------------------------
    always @(posedge sclk or posedge cs_n) begin
        if (cs_n) begin
            state    <= WAIT1;
            bit_cnt  <= 3'd0;
            shift_reg<= 8'd0;
            byte_rcv <= 8'd0;
            tag_byte <= 8'd0;
            data_byte<= 8'd0;
            // regs 永远保持之前写入的值（上电后会有 X，可以在 testbench 里先清零或依次写入）
        end
        else begin
            // 先把新 bit 推到 shift_reg
            shift_reg <= { shift_reg[6:0], mosi };

            if (bit_cnt == 3'd7) begin
                // 拼接出一个完整字节
                byte_rcv <= { shift_reg[6:0], mosi };

                // 一字节到来，基于原 state 做转移
                case (state)
                    WAIT1: begin
                        if ({ shift_reg[6:0], mosi } == 8'h7E)
                            state <= WAIT2;
                        else
                            state <= WAIT1;
                    end

                    WAIT2: begin
                        if ({ shift_reg[6:0], mosi } == 8'h7E)
                            state <= READ_TAG;
                        else begin
                            // 如果不是第二个 0x7E，但自身又是 0x7E，可以保持 WAIT2
                            if ({ shift_reg[6:0], mosi } == 8'h7E)
                                state <= WAIT2;
                            else
                                state <= WAIT1;
```

```
                    end
                end

                READ_TAG: begin
                    tag_byte <= { shift_reg[6:0], mosi };
                    state    <= READ_DAT;
                end

                READ_DAT: begin
                    data_byte <= { shift_reg[6:0], mosi };
                    if (tag_byte == 8'h88) begin
                        // 进入反馈模式
                        fb_ptr <= 3'd0;
                        state  <= FEEDBACK;
                    end else begin
                        // 写回寄存器（仅当 tag_byte 在 0x00~0x07 范围内）
                        if (tag_byte < 8'h08) begin
                            regs[tag_byte[2:0]] <= { shift_reg[6:0],
mosi };
                        end
                        state <= WAIT1;  // 写完一个包，回去重新找同步头
                    end
                end

                FEEDBACK: begin
                    if (fb_ptr == 3'd7) begin
                        state <= WAIT1;   // 8 字节都发完了，回到最初等待下
一个同步头
                    end
                    fb_ptr <= fb_ptr + 1'b1;
                end

                default: state <= WAIT1;
            endcase

            bit_cnt <= 3'd0;  // 收满一字节后清 0
        end
        else begin
            bit_cnt <= bit_cnt + 1'b1;
        end
    end
end
endmodule
```

6.2 `tb_spi_master.v` (测试主机代码)

```
`timescale 1ns/1ps
module tb_spi_master;
```

```
reg clk, cs_n, mosi;
wire miso;
spi_slave_param dut(
    .sclk(clk),
    .cs_n(cs_n),
    .mosi(mosi),
    .miso(miso)
);

// 10 MHz SPI 时钟：时隙=100ns
initial clk = 0;
always #50 clk = ~clk;

// SPI 读/写一个字节：MSB first
// 行为：每个时钟下降沿把 mosi 设好，紧接着上升沿采 miso
task spi_byte(input  [7:0] din, output [7:0] dout);
    integer i;
    begin
        dout = 8'h00;
        for (i = 7; i >= 0; i = i - 1) begin
            mosi <= din[i];
            @(negedge clk);        // 先把 mosi 推到 slave
            @(posedge clk); dout[i] <= miso;  // 上升沿采样 miso
        end
    end
endtask

// 发送一次 “同步头(0x7E,0x7E) + TAG + DATA”
task spi_cmd(input [7:0] tag, input [7:0] data);
    reg [7:0] dummy;
    begin
        cs_n <= 1'b0;                // 拉低片选
        spi_byte(8'h7E, dummy);
        spi_byte(8'h7E, dummy);
        spi_byte(tag , dummy);
        spi_byte(data, dummy);
        cs_n <= 1'b1;                // 结束该命令
        repeat (20) @(posedge clk); // 空闲一段时间
    end
endtask

integer i;
reg [7:0] rcv;
initial begin
    // 先把寄存器清 0（避免 X）
    for (i = 0; i < 8; i = i + 1)
        dut.regs[i] = 8'h00;

    // 等待形稳
    cs_n = 1'b1; mosi = 1'b0;
```

```
        repeat (10) @(posedge clk);

        // 依次写 regs[0]~regs[7]
        for (i = 0; i < 8; i = i + 1) begin
            spi_cmd(i[7:0], (8'h11 << i));
            // 比如写 regs[0]=0x11, regs[1]=0x22, …
        end

        // 发起反馈命令（TAG=0x88, DATA 随意填）
        cs_n <= 1'b0;
        spi_byte(8'h7E, rcv);
        spi_byte(8'h7E, rcv);
        spi_byte(8'h88, rcv);
        spi_byte(8'h00, rcv);
        // 然后连续 8 字节读出 regs[0]~regs[7]
        for (i = 0; i < 8; i = i + 1)
            spi_byte(8'h00, rcv);

        cs_n <= 1'b1;
        #1000 $stop;   // 使用 $stop 使 ModelSim 不会自动退出
    end
endmodule
```