

数字集成电路设计实践报告

|  |  |
| --- | --- |
| 题目： | **SPI技术文档** |

二〇二四 年 一 月

目录

[1 前端设计 4](#_Toc155274075)

[1.1 整体结构图 4](#_Toc155274076)

[1.2 外部接口 4](#_Toc155274077)

[1.3 内部寄存器介绍 5](#_Toc155274078)

[1.3.1 控制寄存器1 5](#_Toc155274079)

[1.3.2 控制寄存器2 6](#_Toc155274080)

[1.3.3 分频寄存器 7](#_Toc155274081)

[1.4 代码解析 8](#_Toc155274082)

[1.4.1 spi\_master 8](#_Toc155274083)

[1.4.2 spi\_slave 13](#_Toc155274084)

[1.4.3 spi\_ms 17](#_Toc155274085)

[1.5 Testbench 22](#_Toc155274086)

[1.5.1 整体框架 22](#_Toc155274087)

[1.5.2 代码解析 22](#_Toc155274088)

[1.5.3 测试波形 28](#_Toc155274089)

[1.5.4 覆盖率测试 28](#_Toc155274090)

[2 DC综合 29](#_Toc155274091)

[2.1 约束参数 29](#_Toc155274092)

[2.2 时序报告 30](#_Toc155274093)

[3 后端 31](#_Toc155274094)

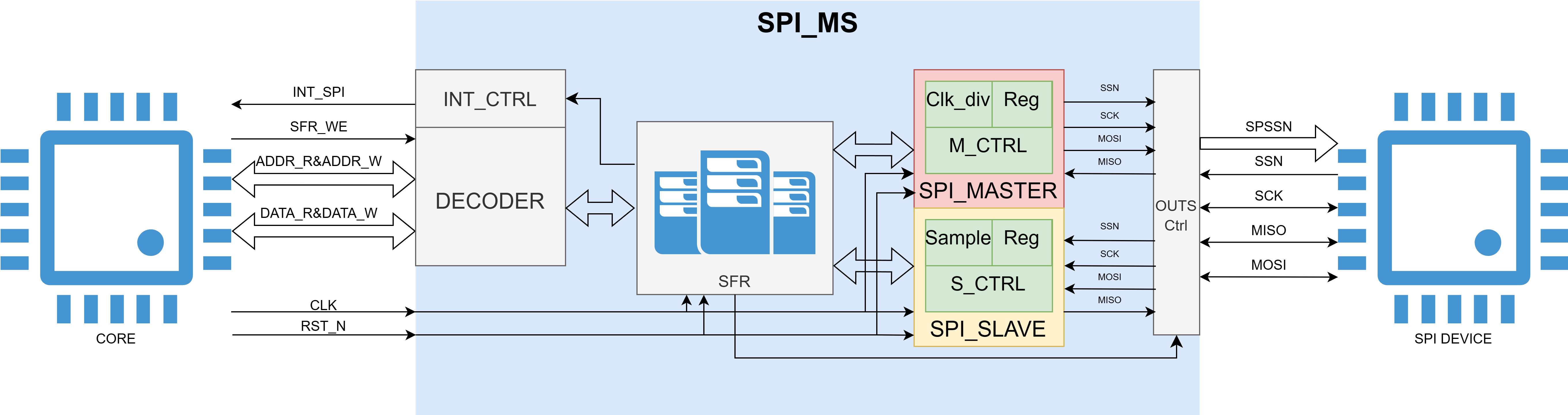
组内成员：孙海宁（组长），黄晓崇，欧泽宏，贾金宇，张承译，孙岩，王倩

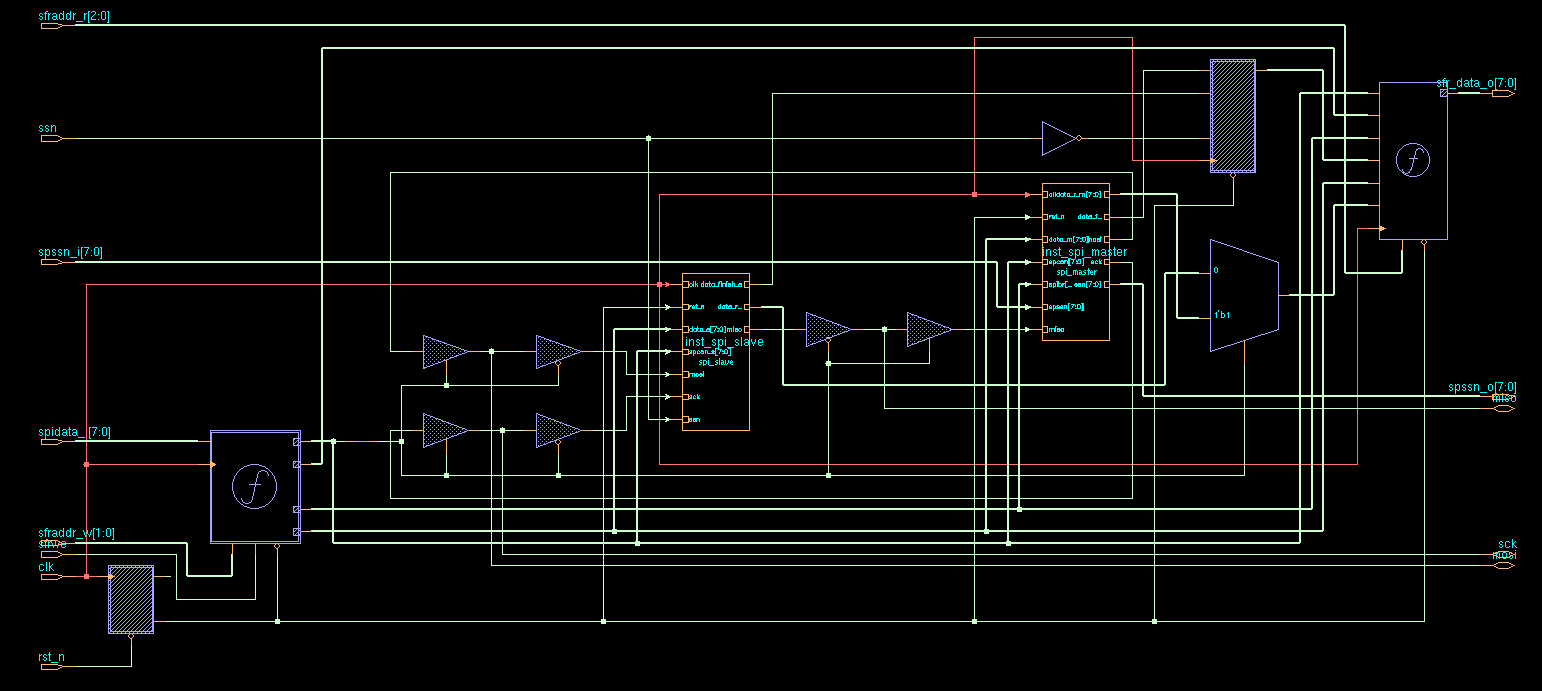
分工：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Coding | Pre-sim | DC | Layout | Post-sim | Summary(PPT,Word) |
| 孙海宁 | 孙海宁 | 孙海宁 | 孙海宁 | 黄晓崇 | 孙海宁 |
| 黄晓崇 | 黄晓崇 | 黄晓崇 | 孙岩 | 贾金宇 | 黄晓崇 |
| 张承译 | 张承译 | 王倩 | 王倩 |  |  |
| 欧泽宏 | 孙岩 | 欧泽宏 | 贾金宇 |  |  |

# 前端设计

## 整体结构图





简述一下数据的传输途径：首先通过外部总线来对内部的SFR进行读写，同时传入clk与rst\_n信号，写入的地址与数据通过内部译码器来对寄存器进行读写，而内部的master与slave模块有两个，通过读写SFR来进行工作模式的选择，两个模块分别连接三态门，最后与外部进行数据交换。

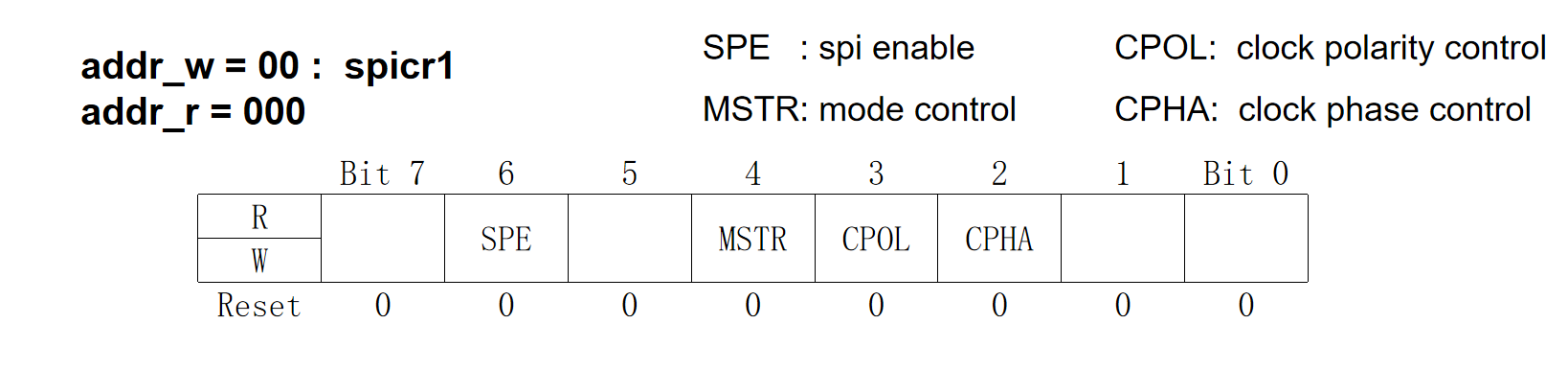
## 外部接口

|  |  |  |
| --- | --- | --- |
| **input** | **width** | **function** |
| clk | 1 | system clock |
| rst\_n | 1 | reset(active low) |
| sfraddr\_w | 2 | write addr |
| sfrwe | 1 | write enable |
| spidata\_i | 8 | write data |
| sfraddr\_r | 3 | read addr |
| spssn\_i | 8 | spi slave select |
| ssn | 1 | slave select |
| **output** | **width** | **function** |
| sfr\_data\_o | 8 | output register |
| spssn\_o | 8 | spi slave select |
| **inout** | **width** | **function** |
| mosi | 1 | master output |
| slave input |
| miso | 1 | master input |
| slave output |
| sck | 1 | spi clock |
|  |  |  |

这里需要注意的是inout接口实际是通过三态门与外界进行连接的。

## 内部寄存器介绍

### 控制寄存器1



SPE 为1时代表SPI开始工作

为0时代表不处于传输状态

MSTR 为1时代表处于slave模式

为0时代表处于master模式

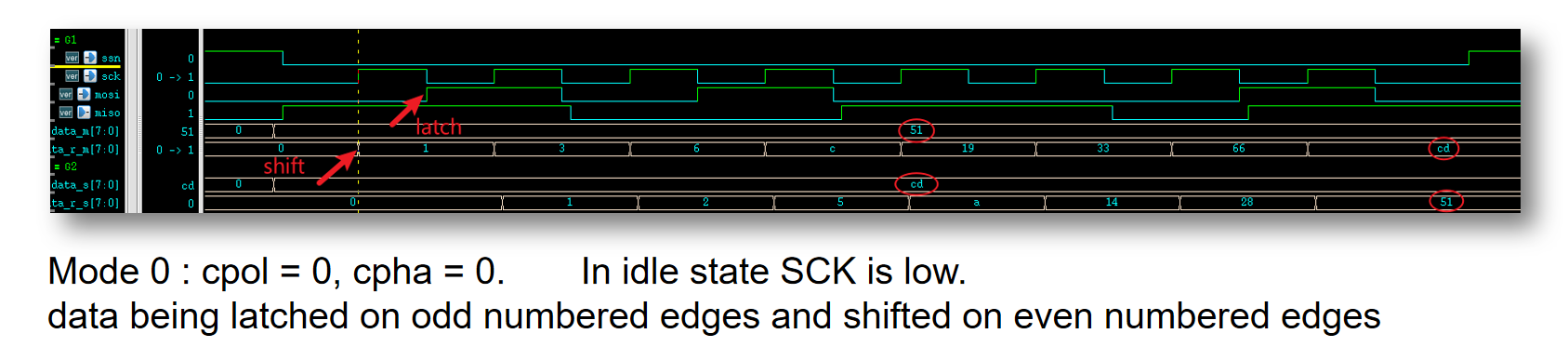
CPOL 为1时代表sck非工作状态为高电平

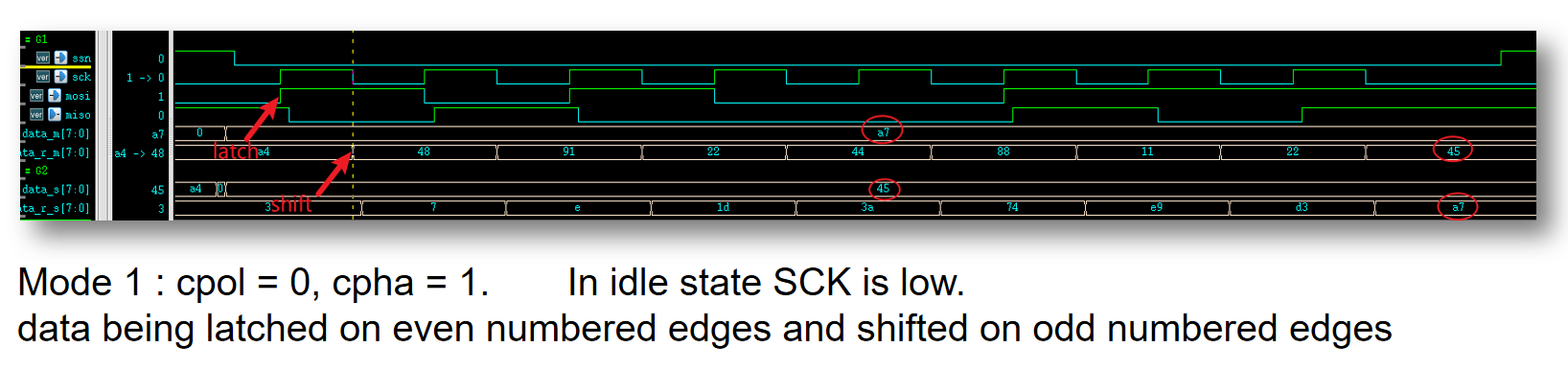
为0时代表sck非工作状态为低电平

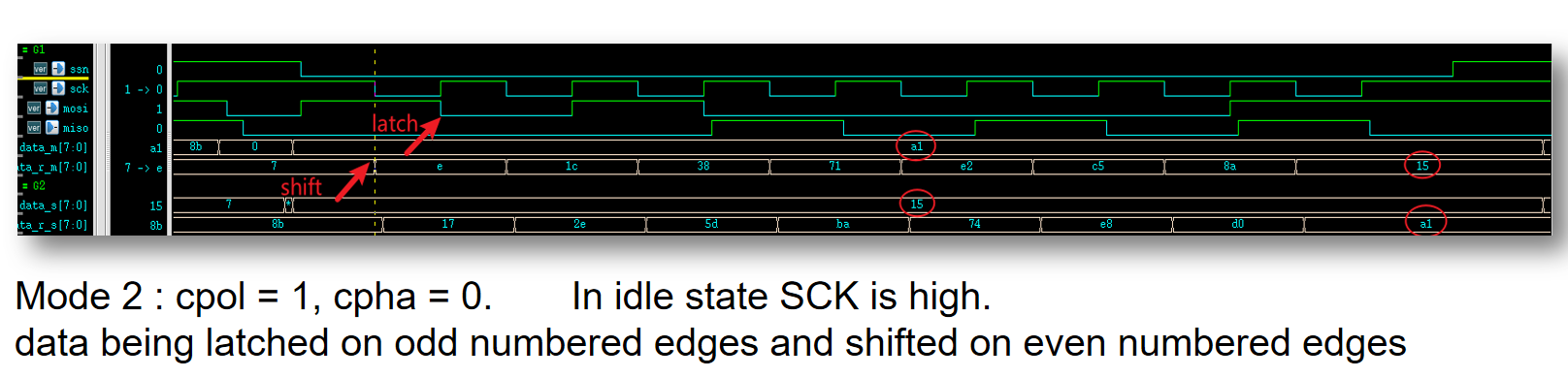
CPHA 为1时代表sck为上升沿锁存数据

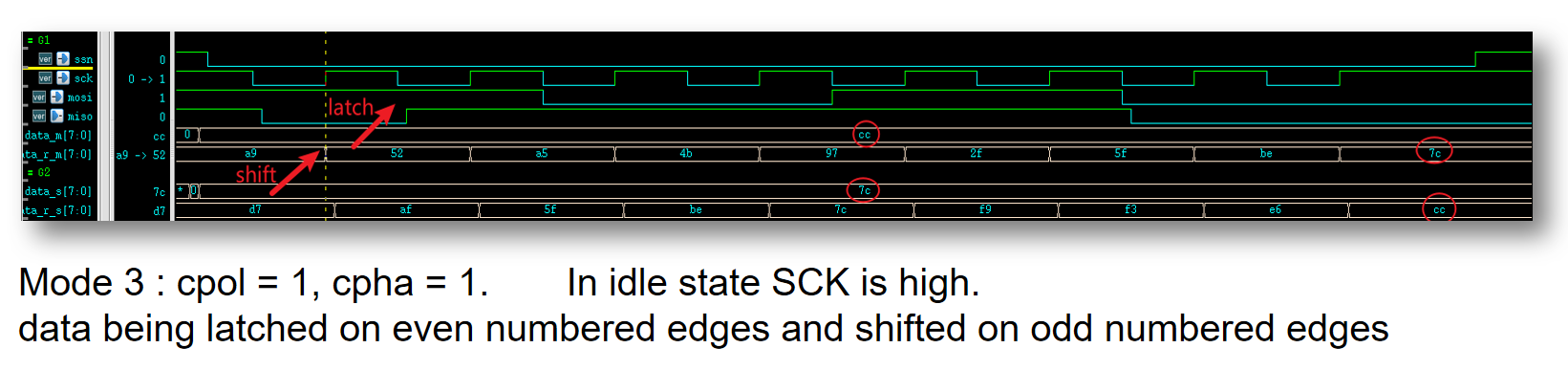
为0时代表sck为下降沿锁存数据

（实际情况如下图所示）

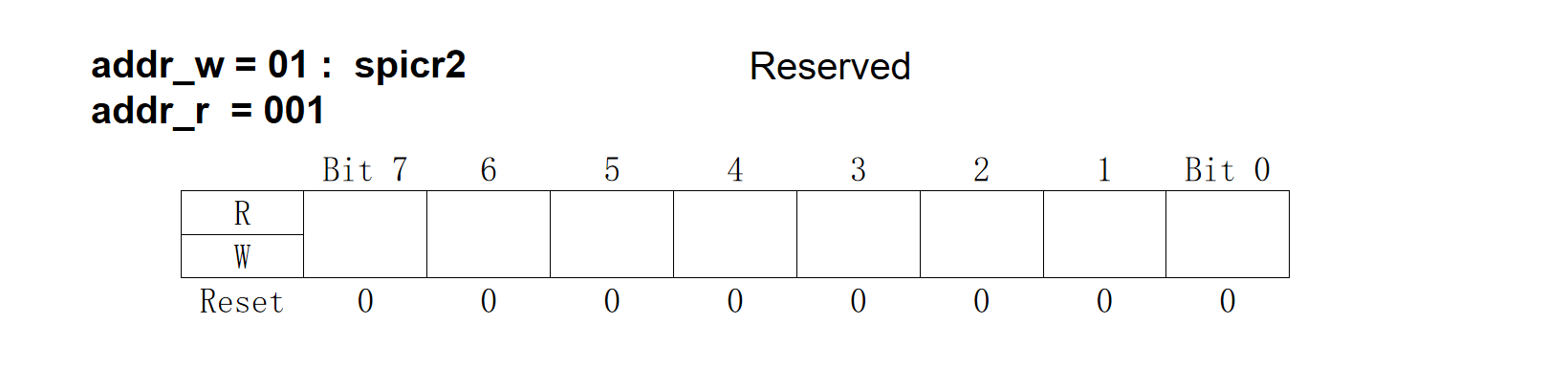






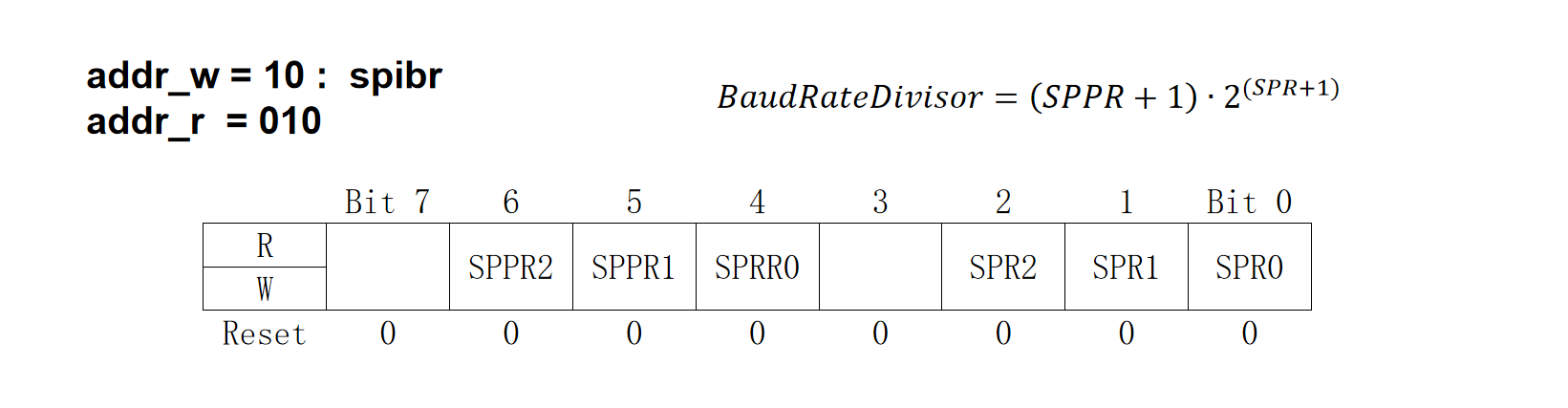


### 控制寄存器2



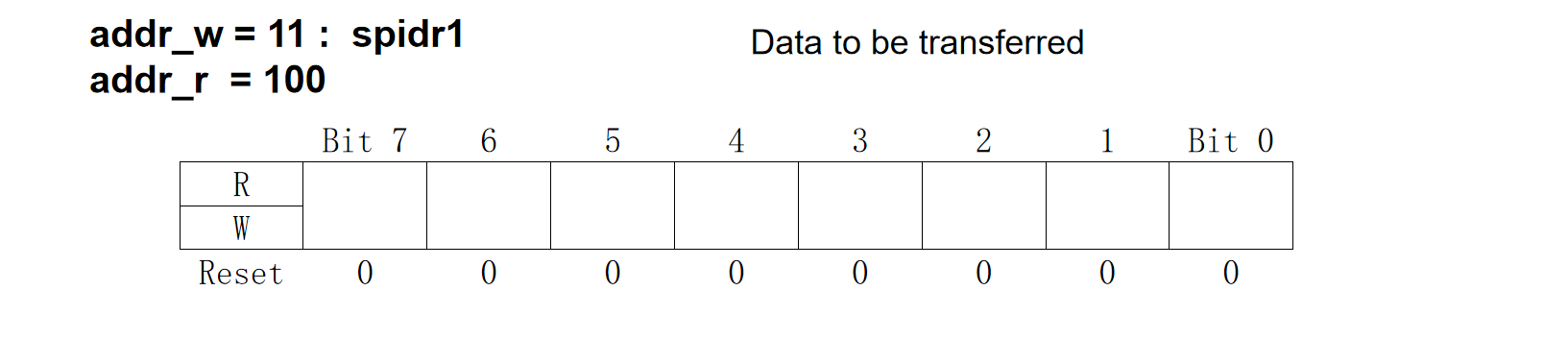
本寄存器可以用来存储状态等参数，可用于后续规划。

### 分频寄存器



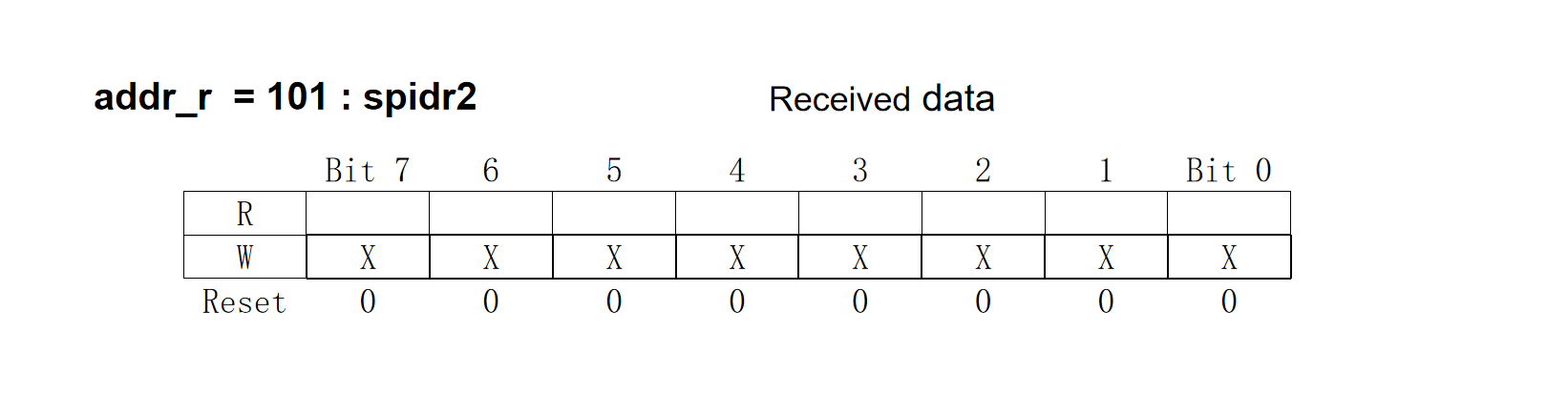
主要用于生成或接收sck时对于clk的分频选择。（计算公式如图所示，共支持64个频率模式）

数据寄存器1：



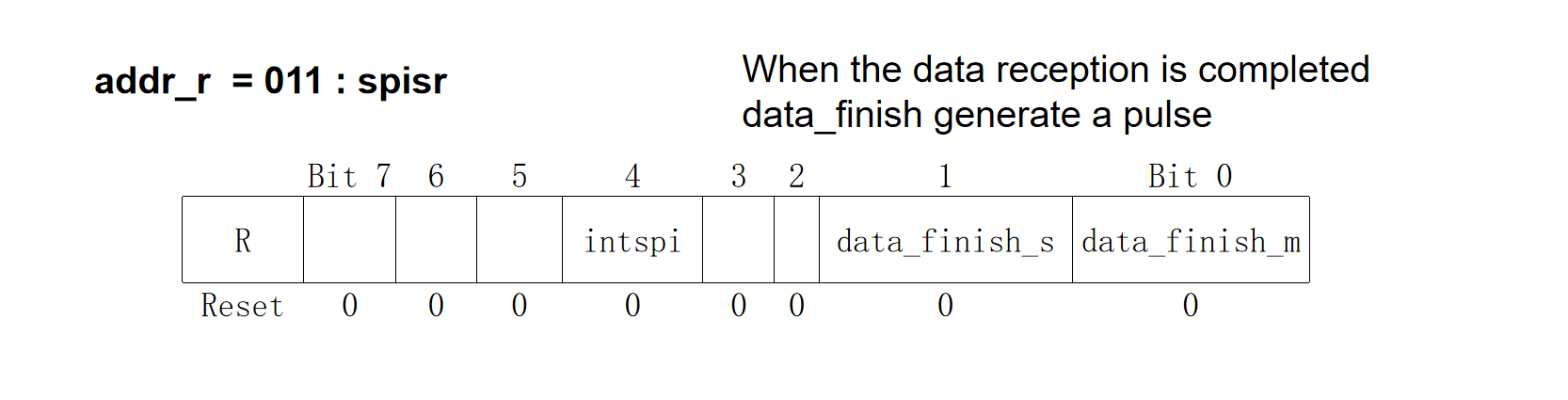
作为SPI的数据传输寄存器。

数据寄存器2：



作为SPI的数据接收寄存器。

状态寄存器：



用于存储中断信号与主从模式的数据传输完成信号。

## 代码解析

### spi\_master

#### input变量声明

|  |  |
| --- | --- |
| clk | 系统时钟 |
| rst\_n | 系统复位，低电平有效 |
| data\_m [7:0] | 主机需要发送的数据 |
| spcon [7:0] | spi 控制寄存器1输入 |
| spibr [7:0] | spi 分频寄存器输入 |
| spssn [7:0] | spi 片选信号输入 |
| miso | 主机输入，从机输出，接收从机数据 |

#### output变量声明

|  |  |
| --- | --- |
| data\_r\_m [7:0] (reg) | 主机接收到的数据 |
| data\_finish\_m (reg) | 主机接收8bit数据完成信号 |
| mosi (reg) | 主机输出，从机输入，发送主机数据 |
| sck (reg) | spi 时钟，分频产生 |
| ssn [7:0] (wire) | spi 片选信号输出 |

#### 内部变量声明

wire型变量声明

|  |  |
| --- | --- |
| tr\_en | 传输使能，为1时开始传输数据 |
| cpol | spi 极性控制信号 |
| cpha | spi 相位控制信号 |
| clk\_div [7:0] | spi 分频数，由分频寄存器计算得出 |
| sppr\_add1 [3:0] | 计算分频数的中间信号 |

reg型变量声明

|  |  |
| --- | --- |
| clk\_cnt [7:0] | spi 分频计数器 |
| sck\_edge\_cnt [4:0] | 记录sck边沿数 |
| sck\_edge\_level | sck边沿电平，sck发生跳变时为1 |
| tr\_done | 传输结束信号 |
| tr\_done\_dly1 | 传输技术信号延时1拍 |
| bit\_count [2:0] | 用于发送数据时计数 |

#### 各模块简介

* 变量声明及wire型变量赋值

wire tr\_en ;   
 assign tr\_en = ~(&spssn) && spcon[6] ; // tx or rx enable   
 assign ssn = spssn ;   
   
 // cpol = 1, Active-low clocks selected. In idle state SCK is high.   
 // cpol = 0, Active-high clocks selected. In idle state SCK is low.   
 // cpha = 1, data being latched on even numbered edges and shifted on odd numbered edges   
 // cpha = 0, data being latched on odd numbered edges and shifted on even numbered edges   
 wire cpol, cpha ;   
 assign {cpol, cpha} = spcon[2:1] ;   
   
 reg [7:0] clk\_cnt ;   
 wire [7:0] clk\_div ; // div the clk to generate the spi clk   
 wire [3:0] sppr\_add1 ;   
   
 assign sppr\_add1 = spibr[6:4] + 3'b001 ;   
 assign clk\_div = sppr\_add1 << spibr[2:0] ;   
   
 reg [4:0] sck\_edge\_cnt ; // trace the sck edge   
 reg sck\_edge\_level ; // trace the sck level   
   
 reg tr\_done ; // when tx or rx done , set it   
 reg tr\_done\_dly1 ;   
   
 reg [2:0] bit\_count ; // bit count to transfer data

* 时钟计数模块，用于分频

// clk count for div   
 always @(posedge clk or negedge rst\_n) begin   
 if(~rst\_n) begin   
 clk\_cnt <= 8'd1;   
 end   
 else begin   
 if (tr\_en) begin   
 if (clk\_cnt == clk\_div) begin   
 clk\_cnt <= 8'd1 ;   
 end   
 else begin   
 clk\_cnt <= clk\_cnt + 1'b1 ;   
 end   
 end   
 else begin   
 clk\_cnt <= clk\_cnt ;   
 end   
 end   
 end

* 分频产生sck，记录sck边沿电平及边沿数

// sck\_edge\_level = 1 to generate sck edge and data transfer   
 // sck\_edge\_cnt counts the number of sck edge   
 always @(posedge clk or negedge rst\_n) begin   
 if(~rst\_n) begin   
 sck\_edge\_level <= 1'b0;   
 sck\_edge\_cnt <= 5'd0 ;   
 end   
 else begin   
 if (tr\_en) begin   
 if (clk\_cnt == clk\_div) begin   
 if (sck\_edge\_cnt == 5'd16) begin   
 sck\_edge\_level <= 1'b0 ;   
 sck\_edge\_cnt <= 5'd0 ;   
 end   
 else begin   
 sck\_edge\_level <= 1'b1 ;   
 sck\_edge\_cnt <= sck\_edge\_cnt + 1'b1;   
 end   
 end   
 else begin   
 sck\_edge\_level <= 1'b0 ;   
 end   
 end   
 else begin   
 sck\_edge\_level <= 1'b0 ;   
 sck\_edge\_cnt <= 5'd0 ;   
 end   
 end   
 end

* 数据发送与接收模块

always @(posedge clk or negedge rst\_n) begin   
 if(~rst\_n) begin   
 sck <= cpol ;   
 data\_r\_m <= 8'd0 ;   
 bit\_count <= 3'b111 ;   
 mosi <= 1'b0 ;   
 end else begin   
 if (tr\_en) begin   
 if (sck\_edge\_level) begin   
 case (sck\_edge\_cnt)   
 1, 3, 5, 7, 9, 11, 13, 15:begin   
 sck <= ~sck ;   
 if (cpha) begin   
 mosi <= data\_m [bit\_count] ;   
 bit\_count <= bit\_count - 1'b1;   
 end   
 else begin   
 data\_r\_m <= {data\_r\_m[6:0], miso} ;   
 end   
 end   
 2, 4, 6, 8, 10, 12, 14, 16:begin   
 sck <= ~sck ;   
 if (cpha) begin   
 data\_r\_m <= {data\_r\_m[6:0], miso} ;   
 end   
 else begin   
 mosi <= data\_m [bit\_count] ;   
 bit\_count <= bit\_count - 1'b1 ;   
 end   
 end   
 endcase   
 end   
 end   
   
 // idle state   
 else begin   
 sck <= cpol ;   
 if (cpha) begin   
 bit\_count <= 4'd7 ;   
 end   
 else begin   
 mosi <= data\_m[7] ;   
 bit\_count <= 4'd6 ;   
 end   
 end   
 end   
 end

* 数据传输完成信号产生模块

always @(posedge clk or negedge rst\_n) begin   
 if (rst\_n == 1'b0) begin   
 tr\_done\_dly1 <= 1'b0 ;   
 end   
 else begin   
 tr\_done\_dly1 <= tr\_done ;   
 end   
 end   
   
   
 // if tr\_done = 1, 8 bits data\_m is being transfer   
 always @(posedge clk or negedge rst\_n) begin   
 if(~rst\_n) begin   
 tr\_done <= 1'b0;   
 end else begin   
 if (tr\_en && sck\_edge\_cnt == 5'd16) begin   
 tr\_done <= 1'b1 ;   
 end else begin   
 tr\_done <= 1'b0 ;   
 end   
 end   
 end   
   
 always @(\*) begin   
 data\_finish\_m = tr\_done && ~tr\_done\_dly1;   
 end

### spi\_slave

#### input变量声明

|  |  |
| --- | --- |
| clk | 系统时钟 |
| rst\_n | 系统复位，低电平有效 |
| data\_s [7:0] | 从机需要发送的数据 |
| spcon\_s [7:0] | spi 控制寄存器1输入 |
| mosi | 主机输出，从机输入，接收主机数据 |
| sck | spi 时钟，主机产生，从机接收 |
| ssn | spi 片选信号 |

#### output变量声明

|  |  |
| --- | --- |
| data\_r\_s [7:0] (reg) | 从机接收到的数据 |
| data\_finish\_s (reg) | 从机接收8bit数据完成信号 |
| miso (reg) | 主机输入，从机输出，发送从机数据 |

#### 内部变量声明

wire型变量声明

|  |  |
| --- | --- |
| tr\_en | 传输使能，为1时开始传输数据 |
| cpol | spi 极性控制信号 |
| cpha | spi 相位控制信号 |
| sck\_edge\_level | sck边沿检测，sck发生跳变时为1 |

reg型变量声明

|  |  |
| --- | --- |
| sck\_edge\_cnt [4:0] | 记录sck边沿数 |
| sck\_dly1 | sck延时一拍，用于边沿检测 |
| sck\_dly2 | sck延时两拍 |
| bit\_count [2:0] | 用于发送数据时计数 |

#### 各模块简介

* 变量声明及wire型变量赋值

// cpol = 1, Active-low clocks selected. In idle state SCK is high.   
 // cpol = 0, Active-high clocks selected. In idle state SCK is low.   
 // cpha = 1, data being latched on even numbered edges and shifted on odd numbered edges   
 // cpha = 0, data being latched on odd numbered edges and shifted on even numbered edges   
 wire cpol, cpha ;   
 assign {cpol, cpha} = spcon\_s[2:1] ;   
   
 reg [4:0] sck\_edge\_cnt ; // trace the sck edge   
 wire sck\_edge\_level ; // trace the sck level   
 reg sck\_dly1 ;   
 reg sck\_dly2 ;   
   
 wire tr\_en ;   
 assign tr\_en = ~ssn ; // tx or rx enable   
   
 reg [2:0] bit\_count ; // bit count to transfer data

* sck边沿检测并计数

always @(posedge clk or negedge rst\_n) begin   
 if(~rst\_n) begin   
 sck\_dly1 <= 1'b0 ;   
 sck\_dly2 <= 1'b0 ;   
 end else begin   
 sck\_dly1 <= sck ;   
 sck\_dly2 <= sck\_dly1 ;   
 end   
 end   
   
 // detect the edge signal of sck and count it   
   
 assign sck\_edge\_level = sck\_dly1 ^ sck ;   
   
 always @(posedge clk or negedge rst\_n) begin   
 if(~rst\_n) begin   
 sck\_edge\_cnt <= 0;   
 end else begin   
 if (tr\_en) begin   
 if (sck\_edge\_cnt == 5'd16) begin   
 sck\_edge\_cnt <= 0 ;   
 end else begin   
 sck\_edge\_cnt <= sck\_edge\_level ? sck\_edge\_cnt + 1 : sck\_edge\_cnt ;   
 end   
 end else begin   
 sck\_edge\_cnt <= 0 ;   
 end   
 end   
 end

* 数据发送与接收模块

// data transfer between master and slave   
 always @(posedge clk or negedge rst\_n) begin   
 if(~rst\_n) begin   
 data\_r\_s <= 8'd0 ;   
 bit\_count <= 3'b111 ;   
 miso <= 1'b0 ;   
 end else begin   
 if (tr\_en) begin   
 if (sck\_edge\_level) begin   
 case (sck\_edge\_cnt)   
 1, 3, 5, 7, 9, 11, 13, 15:begin   
 if (cpha) begin   
 data\_r\_s <= {data\_r\_s[6:0], mosi} ;   
 end else begin   
 miso <= data\_s[bit\_count] ;   
 bit\_count <= bit\_count - 1'b1 ;   
 end   
 end   
   
 0, 2, 4, 6, 8, 10, 12, 14:begin   
 if (cpha) begin   
 miso <= data\_s[bit\_count] ;   
 bit\_count <= bit\_count - 1'b1 ;   
 end else begin   
 data\_r\_s <= {data\_r\_s[6:0], mosi} ;   
 end   
 end   
 // default : data\_r\_s <= data\_r\_s ;   
 endcase   
 end   
 end   
 else begin   
 if (cpha) begin   
 bit\_count <= 3'b111 ;   
 end else begin   
 miso <= data\_s[7] ;   
 bit\_count <= 3'b110 ;   
 end   
   
 end   
 end   
 end

* 数据传输完成信号产生模块

// grnerate tr\_finish signal   
 always @(posedge clk or negedge rst\_n) begin   
 if(~rst\_n) begin   
 data\_finish\_s <= 1'b0;   
 end else begin   
 if (tr\_en) begin   
 if (sck\_edge\_cnt == 5'd16) begin   
 data\_finish\_s <= 1'b1 ;   
 end else begin   
 data\_finish\_s <= 1'b0 ;   
 end   
 end else begin   
 data\_finish\_s <= 1'b0 ;   
 end   
   
 end   
 end

### spi\_ms

#### input变量声明

|  |  |
| --- | --- |
| clk | 系统时钟 |
| rst\_n | 系统复位，低电平有效 |
| sfraddr\_w [1:0] | sfr写地址 |
| sfrwe | sfr 写使能 |
| spidata\_i [7:0] | 写数据输入 |
| sfraddr\_r [2:0] | sfr 读地址 |
| spssn\_i [7:0] | 作为主机时ssn输入 |
| ssn | 作为从机时ssn输入 |

#### output变量声明

|  |  |
| --- | --- |
| sfr\_data\_o [7:0] (reg) | sfr读数据输出 |
| spssn\_o [7:0] (wire) | 作为主机时ssn输出 |

#### inout变量声明

|  |  |
| --- | --- |
| mosi | 作为主机时为输出，从机时为输入 |
| miso | 作为主机时为输入，从机时为输出 |
| sck | 作为主机时为输出，从机时为输入 |

#### 内部变量声明

* wire型变量声明

|  |  |
| --- | --- |
| spidr2 [7:0] | spi 数据寄存器2 |
| spidr2\_m [7:0] | 主机数据寄存器2 |
| spidr2\_s [7:0] | 从机数据寄存器2 |
| data\_finish\_m | 主机传输完成信号 |
| data\_finish\_s | 从机传输完成信号 |
| miso\_m | 主机miso |
| miso\_s | 从机miso |
| mosi\_m | 主机mosi |
| mosi\_s | 从机mosi |
| sck\_m | 主机sck |
| sck\_s | 从机sck |
| intspi | 中断信号 |
| mstr | 主机从机选择信号，为1选择主机 |

* reg型变量声明

|  |  |
| --- | --- |
| spicr1 | spi 控制寄存器1 |
| spicr2 | spi 控制寄存器2 |
| spibr | spi 分频寄存器 |
| spisr | spi 状态寄存器 |
| spidr1 | spi 数据寄存器1 |
| rst\_n\_sync\_pre | 异步复位，同步释放前信号 |
| rst\_n\_sync | 异步复位，同步释放信号 |

#### 各模块简介

* 三态门缓冲模块

wire mstr = spicr1[4] ; // mstr = 0 means slave model, mstr = 1 means master model   
   
 assign sck = mstr ? sck\_m : 1'bz ;   
 assign sck\_s = mstr ? 1'bz : sck ;   
   
 assign mosi = mstr ? mosi\_m: 1'bz ;   
 assign mosi\_s = mstr ? 1'bz : mosi ;   
   
 assign miso\_m = mstr ? miso : 1'bz ;   
 assign miso = mstr ? 1'bz : miso\_s;

* 同步复位，异步释放模块

reg rst\_n\_sync\_pre ;   
 reg rst\_n\_sync ;   
   
 always @ (posedge clk or negedge rst\_n) begin   
 if (rst\_n == 1'b0) begin   
 rst\_n\_sync\_pre <= 1'b0 ;   
 rst\_n\_sync <= 1'b0 ;   
 end else begin   
 rst\_n\_sync\_pre <= 1'b1 ;   
 rst\_n\_sync <= rst\_n\_sync\_pre ;   
 end   
 end

* 例化主机和从机

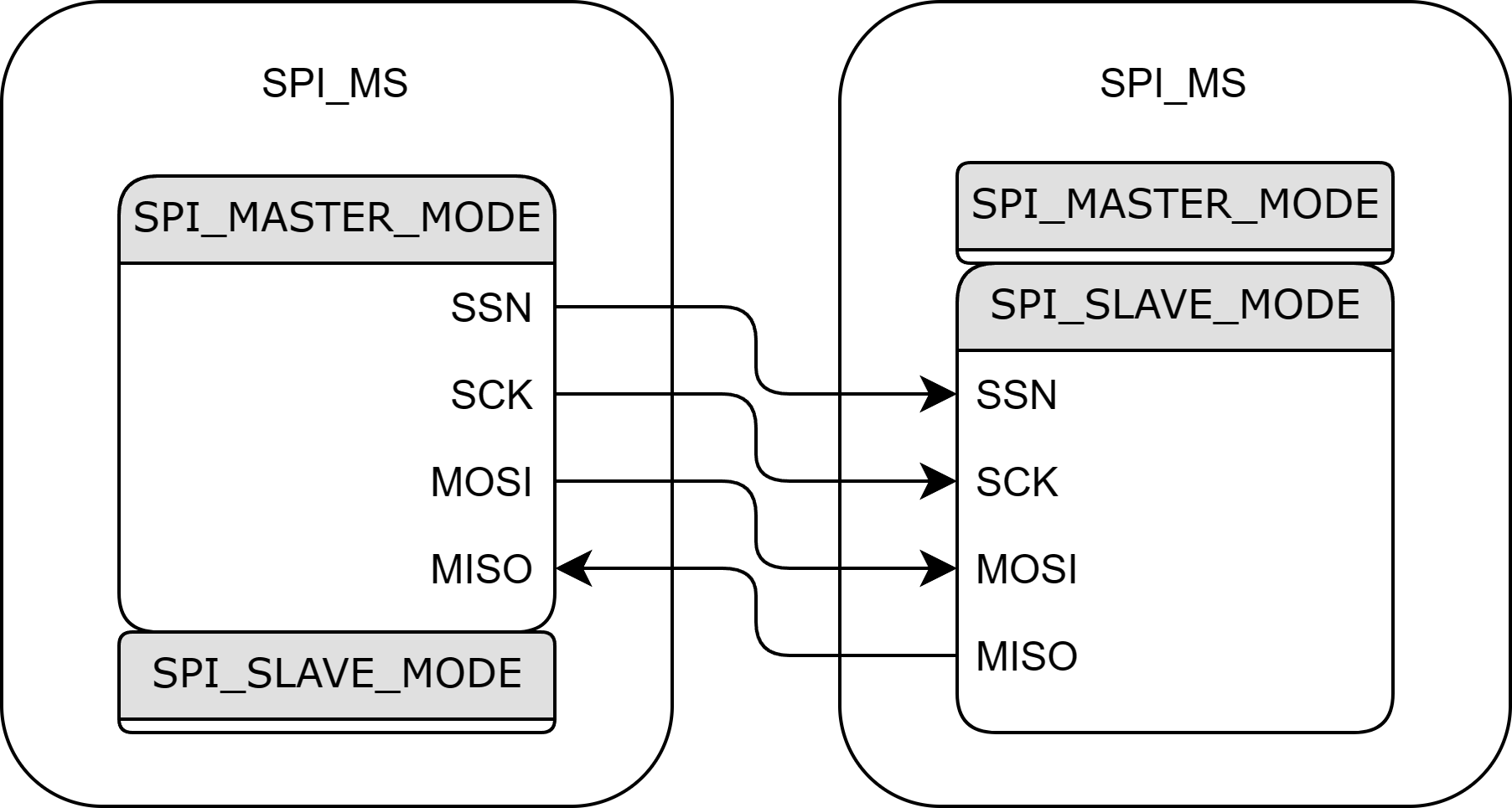
spi\_master inst\_spi\_master   
 (   
 .clk (clk ),   
 .rst\_n (rst\_n\_sync ),   
 .data\_m (spidr1 ),   
 .spcon (spicr1 ),   
 .spibr (spibr ),   
 .spssn (spssn\_i ),   
 .data\_r\_m (spidr2\_m ),   
 .data\_finish\_m (data\_finish\_m),   
 .miso (miso\_m ),   
 .mosi (mosi\_m ),   
 .sck (sck\_m ),   
 .ssn (spssn\_o )   
 );   
   
 spi\_slave inst\_spi\_slave   
 (   
 .clk (clk ),   
 .rst\_n (rst\_n\_sync ),   
 .data\_s (spidr1 ),   
 .spcon\_s (spicr1 ),   
 .data\_finish\_s (data\_finish\_s),   
 .data\_r\_s (spidr2\_s ),   
 .mosi (mosi\_s ),   
 .miso (miso\_s ),   
 .sck (sck\_s ),   
 .ssn (ssn )   
 );

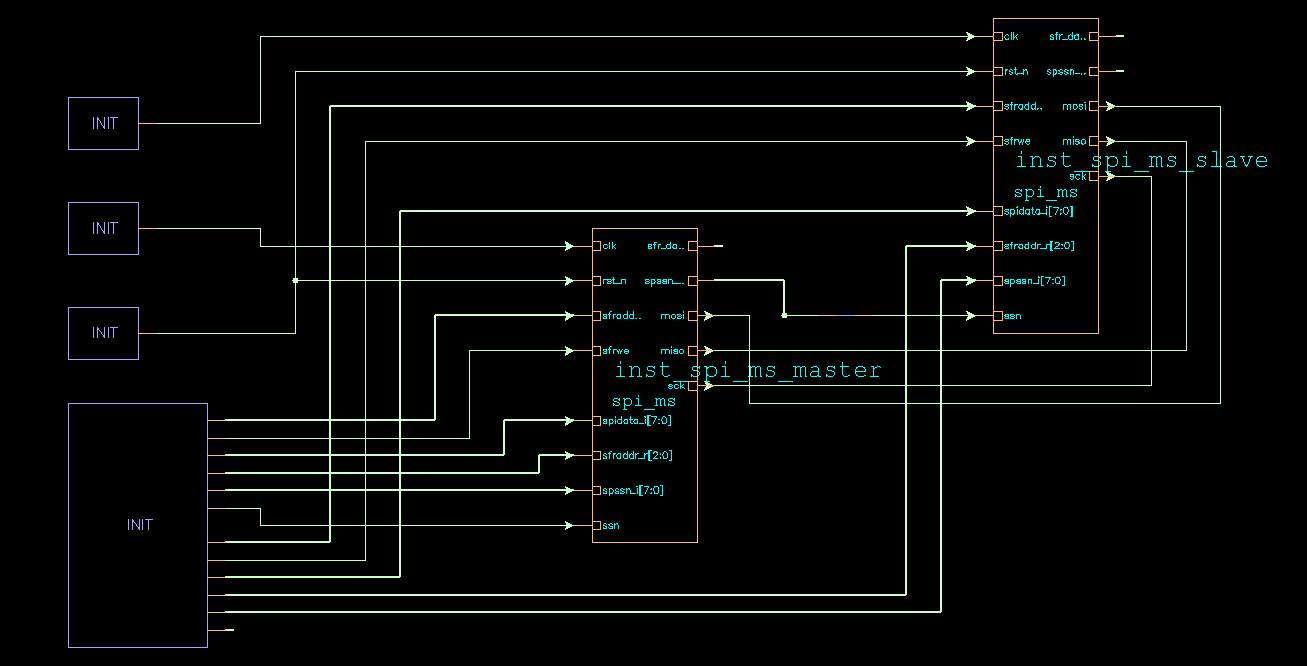
* 寄存器读写模块

assign spidr2 = mstr ? spidr2\_m : spidr2\_s ; // mstr = 1, master model   
   
 always @(posedge clk or negedge rst\_n\_sync) begin   
 if (rst\_n\_sync == 1'b0) begin   
 spisr <= 8'b0 ;   
 end   
 else begin   
 spisr[0] <= data\_finish\_m ;   
 spisr[1] <= data\_finish\_s ;   
 spisr[4] <= intspi ;   
 end   
 end   
   
 always @(posedge clk or negedge rst\_n\_sync) begin   
 if (rst\_n\_sync == 0) begin   
 spicr1 <= 8'b0;   
 spicr2 <= 8'b0;   
 spibr <= 8'b0;   
 spidr1 <= 8'b0;   
 end else if (sfrwe) begin   
 case (sfraddr\_w)   
 2'b00: spicr1 <= spidata\_i;   
 2'b01: spicr2 <= spidata\_i;   
 2'b10: spibr <= spidata\_i;   
 2'b11: spidr1 <= spidata\_i;   
 default:;   
 endcase   
 end   
 end   
   
   
/\*-----------------------------------------------\   
 -- sfr\_data\_o : sfr output data --   
\-----------------------------------------------\*/   
   
 always @(posedge clk or negedge rst\_n\_sync)begin   
 if (rst\_n\_sync == 1'b0) begin   
 sfr\_data\_o <= 8'b0000\_0000 ;   
 end else begin   
 case(sfraddr\_r)   
 3'b000 : sfr\_data\_o <= spicr1 ;   
 3'b001 : sfr\_data\_o <= spicr2 ;   
 3'b010 : sfr\_data\_o <= spibr ;   
 3'b011 : sfr\_data\_o <= spisr ;   
 3'b100 : sfr\_data\_o <= spidr1 ;   
 3'b101 : sfr\_data\_o <= spidr2 ;   
 default: sfr\_data\_o <= 8'b0000\_0000 ;   
 endcase   
 end   
 end

## Testbench

### 整体框架





### 代码解析

* 时钟及复位

// clock   
 logic m\_clk;   
 initial begin   
 m\_clk = '0;   
 forever #(20) m\_clk = ~m\_clk;   
 end   
   
 logic s\_clk;   
 initial begin   
 s\_clk = '0;   
 #2   
 s\_clk = '1;   
 forever #(20) s\_clk = ~s\_clk;   
 end   
   
 // asynchronous reset   
 logic rst\_n;   
 initial begin   
 rst\_n <= '0;   
 #80   
 rst\_n <= '1;   
 end

* 例化模块并连接主机和从机

logic [1:0] m\_sfraddr\_w;   
 logic m\_sfrwe;   
 logic [7:0] m\_spidata\_i;   
 logic [2:0] m\_sfraddr\_r;   
 logic [7:0] m\_sfr\_data\_o;   
 logic [7:0] m\_spssn\_i;   
 logic [7:0] m\_spssn\_o;   
 logic m\_ssn;   
   
 wire mosi;   
 wire miso;   
 wire sck;   
   
 logic [1:0] s\_sfraddr\_w;   
 logic s\_sfrwe;   
 logic [7:0] s\_spidata\_i;   
 logic [2:0] s\_sfraddr\_r;   
 logic [7:0] s\_sfr\_data\_o;   
 logic [7:0] s\_spssn\_i;   
 logic [7:0] s\_spssn\_o;   
   
 logic s\_ssn;   
   
 spi\_ms inst\_spi\_ms\_master   
 (   
 .clk (m\_clk),   
 .rst\_n (rst\_n),   
 .sfraddr\_w (m\_sfraddr\_w),   
 .sfrwe (m\_sfrwe),   
 .spidata\_i (m\_spidata\_i),   
 .sfraddr\_r (m\_sfraddr\_r),   
 .sfr\_data\_o (m\_sfr\_data\_o),   
 .spssn\_i (m\_spssn\_i),   
 .spssn\_o (m\_spssn\_o),   
 .mosi (mosi),   
 .miso (miso),   
 .sck (sck),   
 .ssn (m\_ssn)   
 );   
   
 assign s\_ssn = m\_spssn\_o[0] ;   
   
 spi\_ms inst\_spi\_ms\_slave   
 (   
 .clk (s\_clk),   
 .rst\_n (rst\_n),   
 .sfraddr\_w (s\_sfraddr\_w),   
 .sfrwe (s\_sfrwe),   
 .spidata\_i (s\_spidata\_i),   
 .sfraddr\_r (s\_sfraddr\_r),   
 .sfr\_data\_o (s\_sfr\_data\_o),   
 .spssn\_i (s\_spssn\_i),   
 .spssn\_o (s\_spssn\_o),   
 .mosi (mosi),   
 .miso (miso),   
 .sck (sck),   
 .ssn (s\_ssn)   
 );

* 初始化及寄存器测试

task init();   
 m\_sfraddr\_w <= '0;   
 m\_sfrwe <= '0;   
 m\_spidata\_i <= '0;   
 m\_sfraddr\_r <= '0;   
 m\_spssn\_i <= 8'hff;   
 m\_ssn <= '1;   
 s\_sfraddr\_w <= '0;   
 s\_sfrwe <= '0;   
 s\_spidata\_i <= '0;   
 s\_sfraddr\_r <= '0;   
 s\_spssn\_i <= 8'hff;   
 endtask   
   
 task spssn\_test();   
 for (int i = 0; i < 256; i++) begin   
 m\_spssn\_i <= i ;   
 s\_spssn\_i <= i ;   
 @(posedge m\_clk) ;   
 end   
 endtask   
   
 task sfr\_data\_o\_test();   
 for (int i = 0; i < 8; i++) begin   
 m\_sfraddr\_r <= i ;   
 s\_sfraddr\_r <= i ;   
 end   
 endtask

* 配置主机和从机

task set\_mode\_m(int mode\_sel\_m, int m\_spibr);   
 m\_sfrwe <= '1;   
 @(posedge m\_clk);   
 m\_sfraddr\_w <= 2'b00;   
 m\_sfraddr\_r <= 3'b000;   
 // spidata\_i <= 8'b0101\_0000; // spi enable, master mode, mode 00   
 m\_spidata\_i <= mode\_sel\_m; // [6]:SPE, [4]:MSTR, mode 00 [3]:cpol, [2]: cpha   
 repeat(2)@(posedge m\_clk);   
 m\_sfraddr\_w <= 2'b01;   
 m\_sfraddr\_r <= 3'b001;   
 m\_spidata\_i <= 8'b0000\_0001;   
 repeat(2)@(posedge m\_clk);   
 m\_sfraddr\_w <= 2'b10;   
 m\_sfraddr\_r <= 3'b010;   
 m\_spidata\_i <= {1'b0, m\_spibr[5:3], 1'b0, m\_spibr[2:0]}; // m\_clk / 16   
 repeat(2)@(posedge m\_clk);   
 m\_sfraddr\_w <= 2'b11;   
 m\_sfraddr\_r <= 3'b011;   
 m\_spidata\_i <= 8'b0000\_0000; // ready to transfer data   
 @(posedge m\_clk);   
 read\_sfr();   
 endtask   
   
 task set\_mode\_s(int mode\_sel\_s);   
 s\_sfrwe <= '1;   
 @(posedge s\_clk);   
 s\_sfraddr\_w <= 2'b00;   
 s\_sfraddr\_r <= 3'b000;   
 // spidata\_i <= 8'b0101\_0000; // spi enable, master mode, mode 00   
 s\_spidata\_i <= mode\_sel\_s; // [6]:SPE, [4]:MSTR, mode 00 [3]:cpol, [2]: cpha   
 repeat(2)@(posedge s\_clk);   
 s\_sfraddr\_w <= 2'b01;   
 s\_sfraddr\_r <= 3'b001;   
 s\_spidata\_i <= 8'b0000\_0001;   
 repeat(2)@(posedge s\_clk);   
 s\_sfraddr\_w <= 2'b10;   
 s\_sfraddr\_r <= 3'b010;   
 s\_spidata\_i <= 8'b0000\_0011; // clk / 16   
 repeat(2)@(posedge s\_clk);   
 s\_sfraddr\_w <= 2'b11;   
 s\_sfraddr\_r <= 3'b011;   
 s\_spidata\_i <= 8'b0000\_0000; // ready to transfer data   
 @(posedge s\_clk);   
 read\_sfr();   
 endtask   
   
 task read\_sfr();   
 m\_sfraddr\_r <= 3'b101 ;   
 s\_sfraddr\_r <= 3'b101 ;   
 repeat(2)@(posedge m\_clk);   
 endtask

* 主从间传输数据

int m\_spibr\_post2 ;   
 task transfer\_data\_master\_slave(int iter\_ms, int m\_spibr);   
 for(int it = 0; it < iter\_ms; it++) begin   
 m\_spidata\_i <= $urandom\_range(0,255);   
 s\_spidata\_i <= $urandom\_range(0,255);   
 repeat(2)@(posedge m\_clk);   
 m\_spssn\_i <= 8'hfe;   
 m\_spibr\_post2 = m\_spibr[2:0] ;   
 repeat((m\_spibr[5:3]+1) \* $pow(2,m\_spibr\_post2+1) \* 8 + 3)@(posedge m\_clk);   
 m\_spssn\_i <= 8'hff;   
 repeat(10)@(posedge m\_clk);   
 end   
 endtask

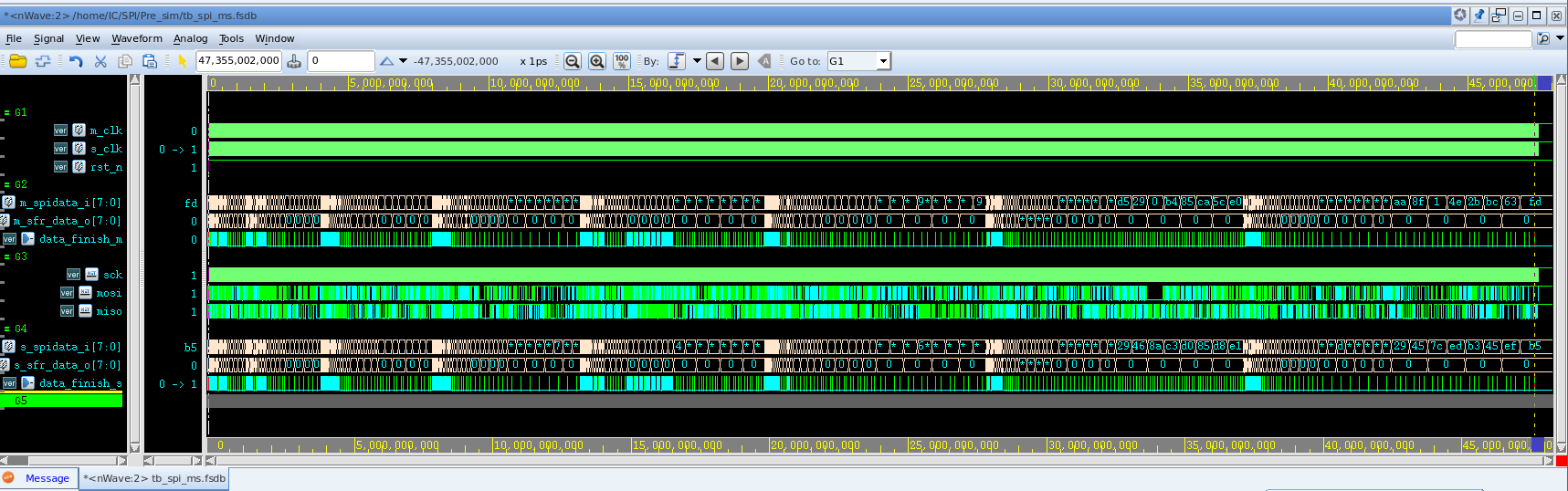
* 任务调用

init();   
 repeat(10)@(posedge m\_clk);   
 for (int m\_spibr = 0; m\_spibr < 64; m\_spibr++) begin   
   
 set\_mode\_m(8'b0101\_0000, m\_spibr);   
 set\_mode\_s(8'b0100\_0000);   
 transfer\_data\_master\_slave(2, m\_spibr);   
   
 set\_mode\_m(8'b0101\_0010, m\_spibr);   
 set\_mode\_s(8'b0100\_0010);   
 transfer\_data\_master\_slave(2, m\_spibr);   
   
 set\_mode\_m(8'b0101\_0100, m\_spibr);   
 set\_mode\_s(8'b0100\_0100);   
 transfer\_data\_master\_slave(2, m\_spibr);   
   
 set\_mode\_m(8'b0101\_0110, m\_spibr);   
 set\_mode\_s(8'b0100\_0110);   
 transfer\_data\_master\_slave(2, m\_spibr);   
 end   
   
 repeat(10)@(posedge m\_clk);   
 spssn\_test();   
 sfr\_data\_o\_test();   
 repeat(10)@(posedge m\_clk);   
 $finish;

* dump 波形

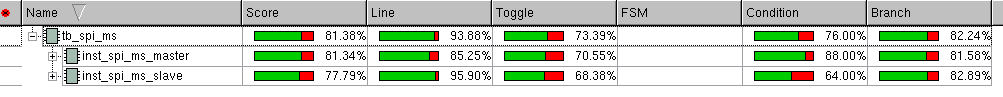
initial begin   
 $display("random seed : %0d", $unsigned($get\_initial\_random\_seed()));   
 if ( $test$plusargs("fsdb") ) begin   
 $fsdbDumpfile("tb\_spi\_ms.fsdb");   
 $fsdbDumpvars(0, "tb\_spi\_ms");   
 end   
 end

### 测试波形



### 覆盖率测试

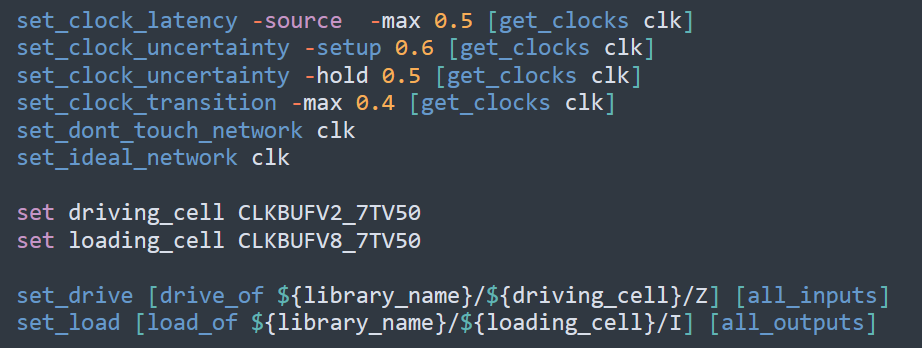
对于翻转，分支等覆盖率进行测试：



# DC综合

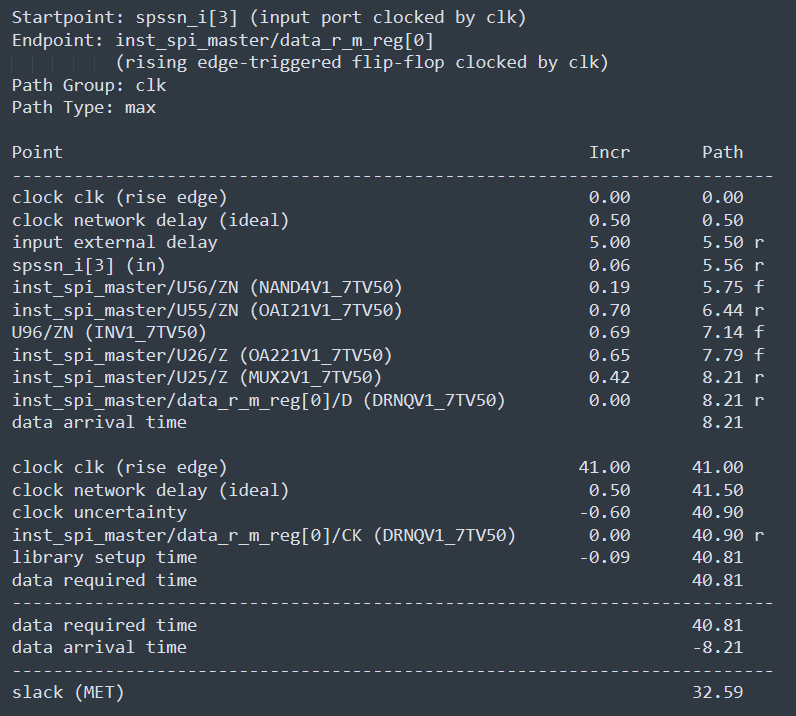
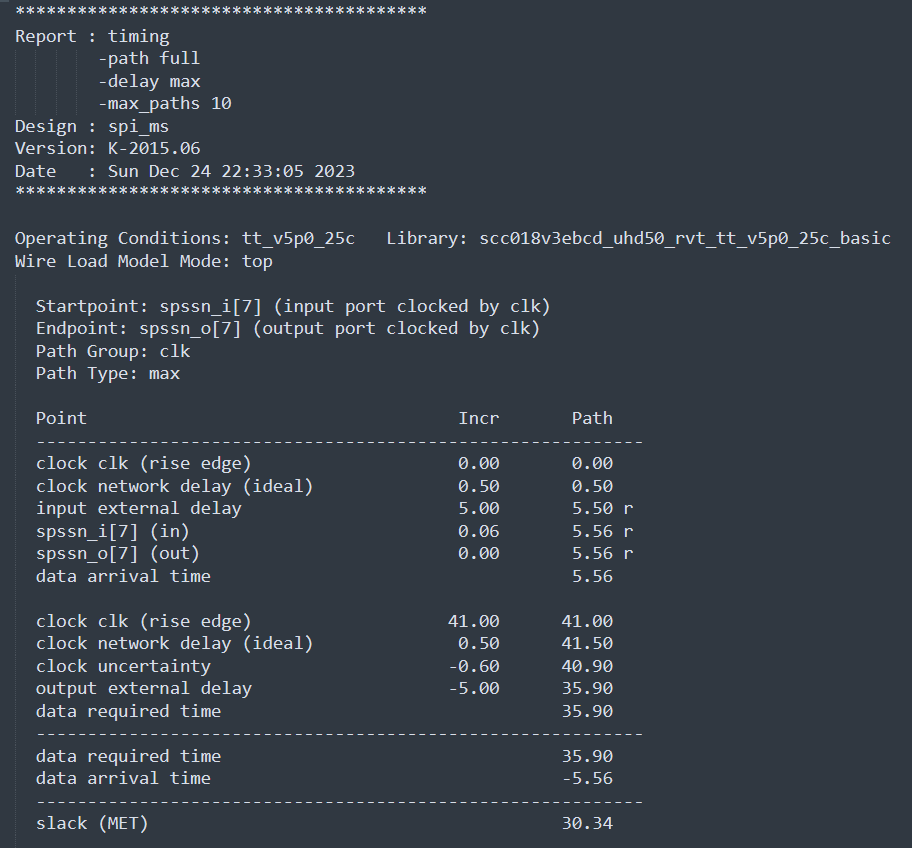
## 约束参数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SPI |  |  |  |  |
|  | MIN | TYP | MAX | UNIT |
|  |  |  |  |  |
| f\_clk | 24 |  |  | MHz |
| source latency |  |  | 0.5 | ns |
| uncertainty(setup) |  |  | 0.6 | ns |
| uncertainty(hold) |  |  | 0.5 | ns |
| clock\_transition |  |  | 0.4 | ns |
|  |  |  |  |  |
| input\_delay | 0 |  | 500 | ns |
| output\_delay | 0 |  | 500 | ns |
|  |  |  |  |  |
| driving cell | CLKBUFV2\_7TV50 | | |  |
| loading cell | CLKBUFV8\_7TV50 | | |  |



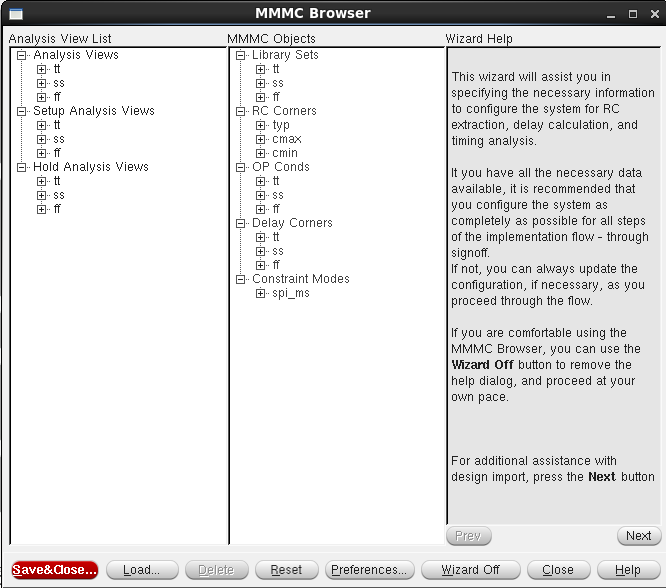
## 时序报告

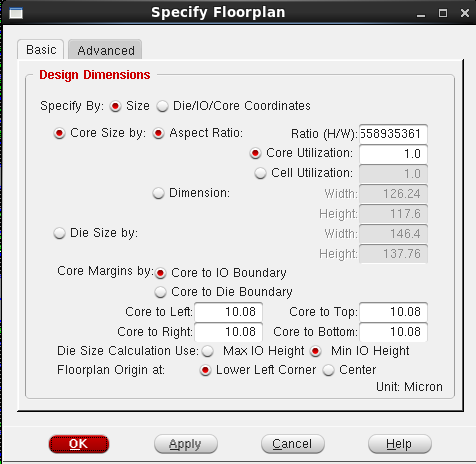
时间裕量满足条件：



# 后端

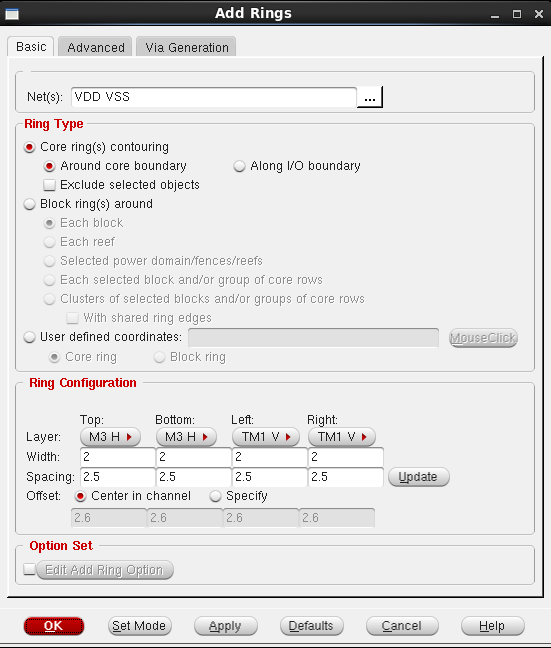
对应三种条件的部分进行仿真：分别为ss，tt，ff，分别添加三种库文件。



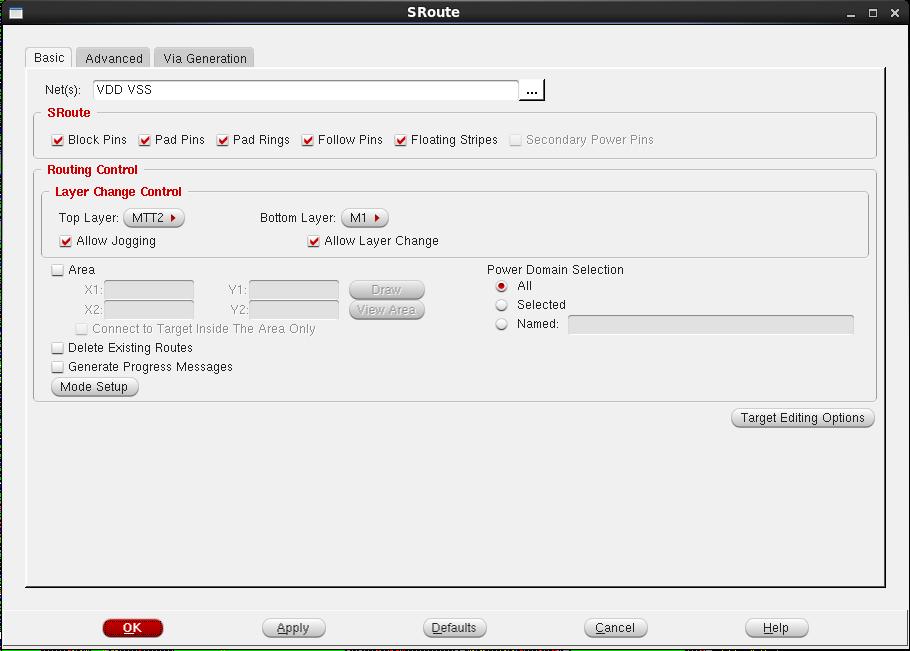


利用完之后芯片利用率：为1.0，设定间距为10

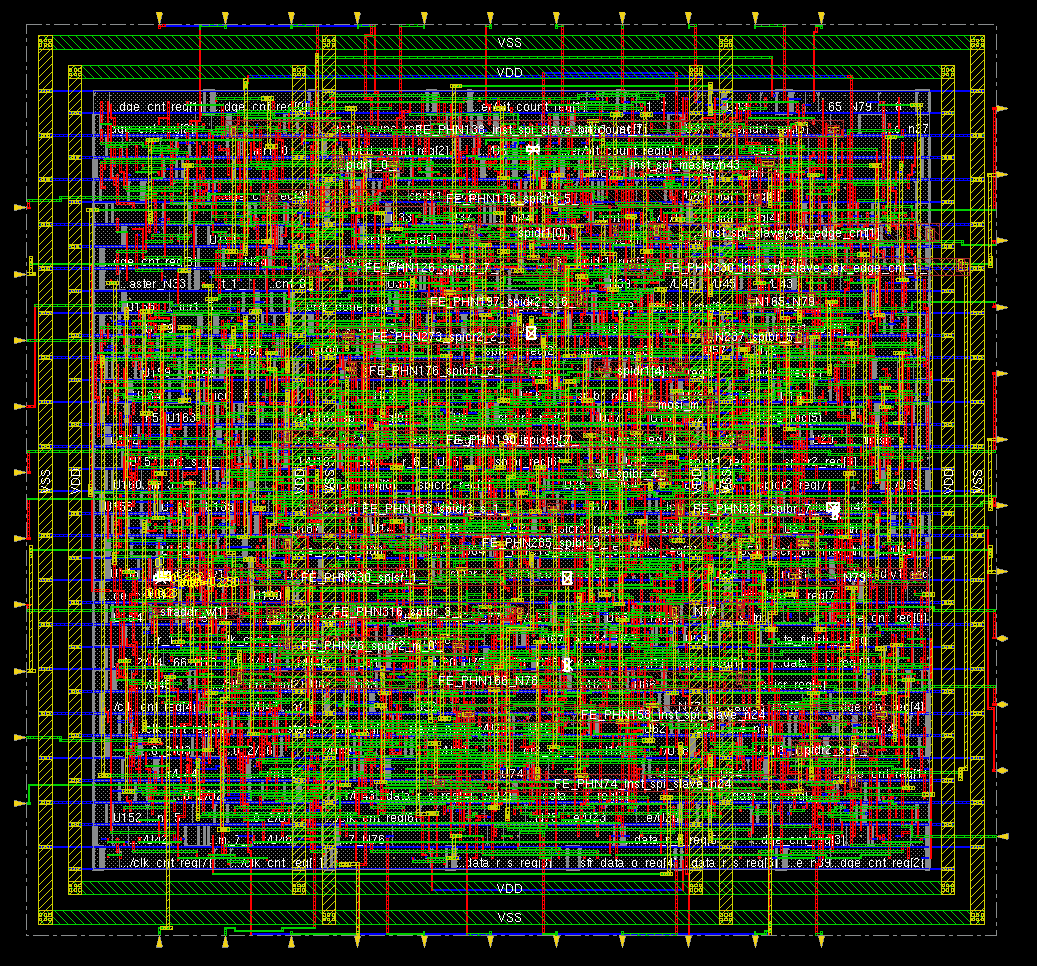
金属层分配等问题：



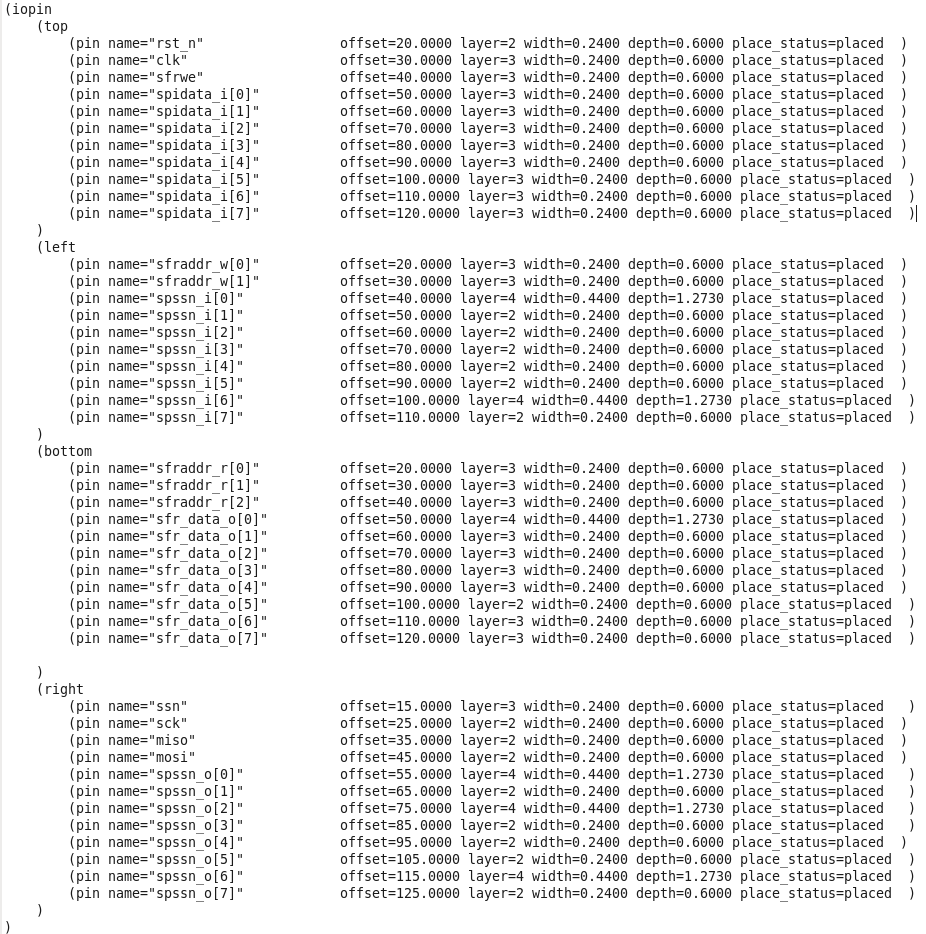




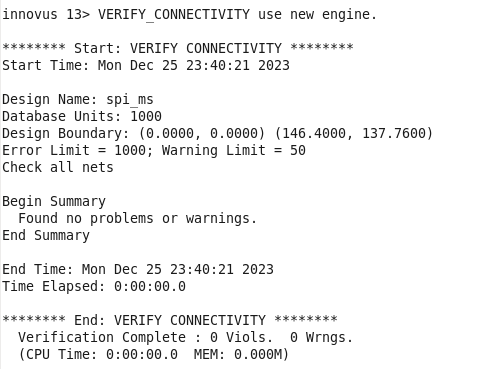
布好线之后的后端版图：



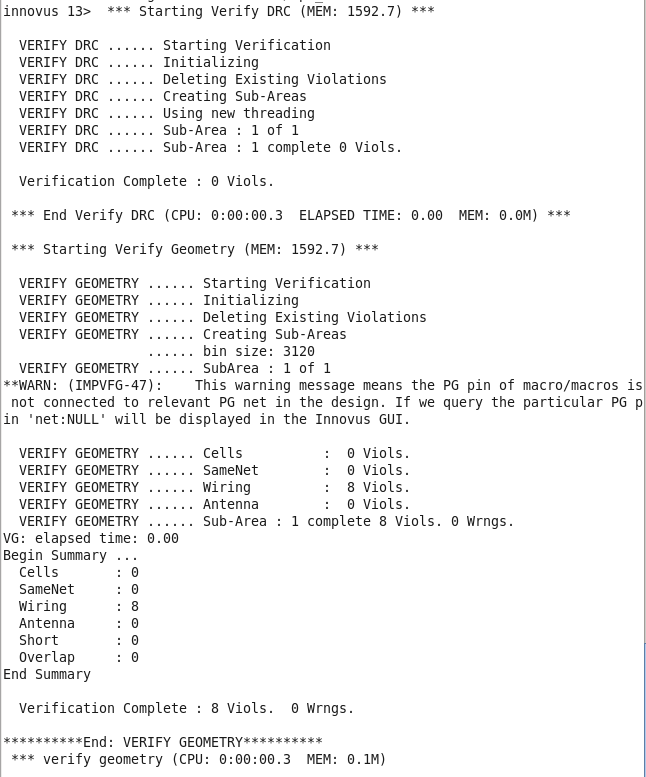
IO文件部分



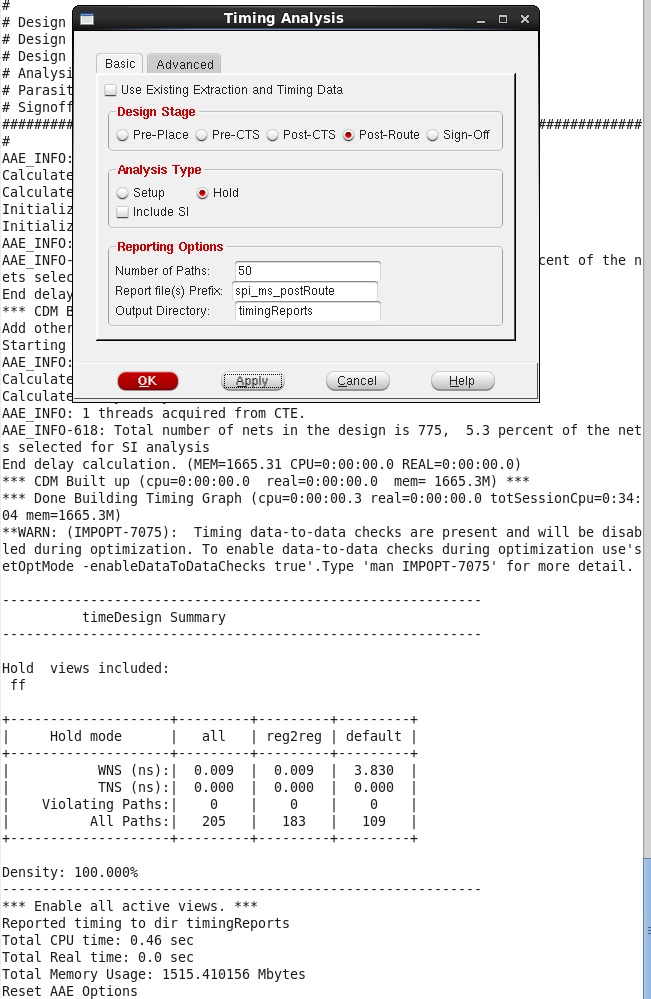
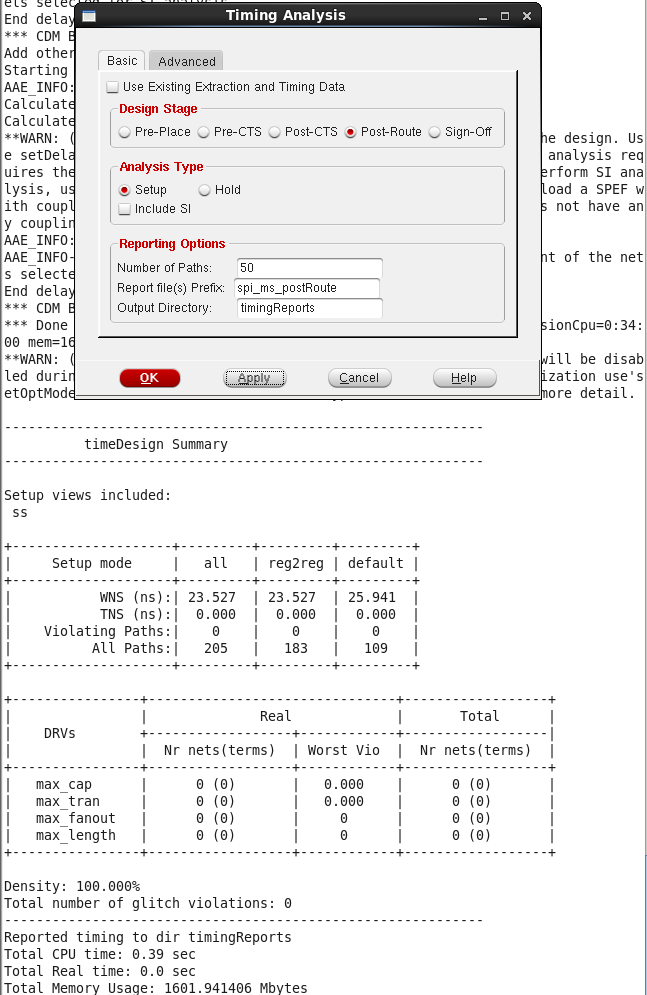
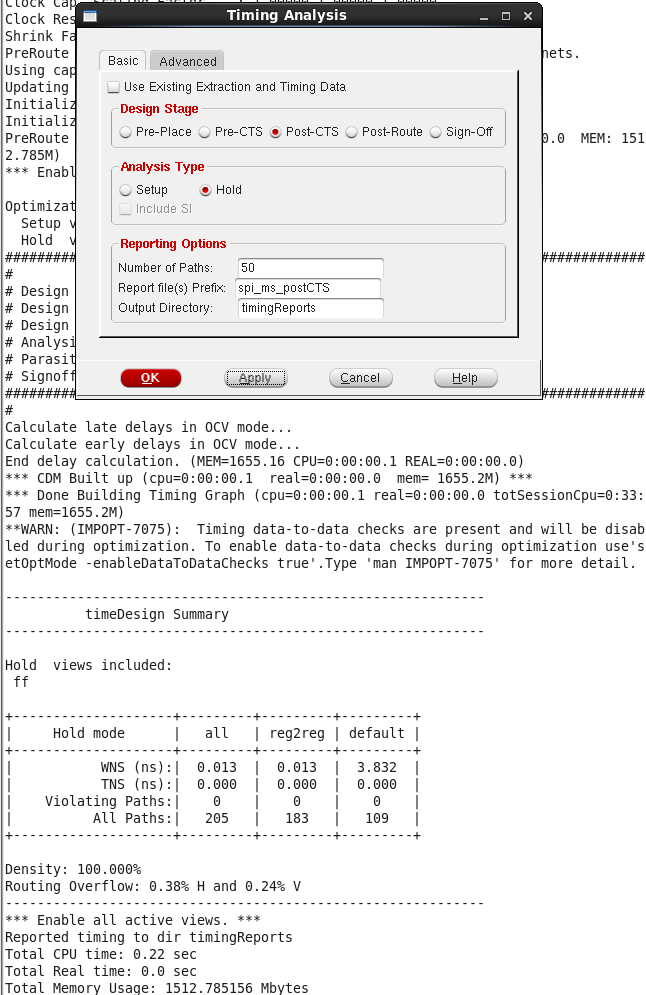
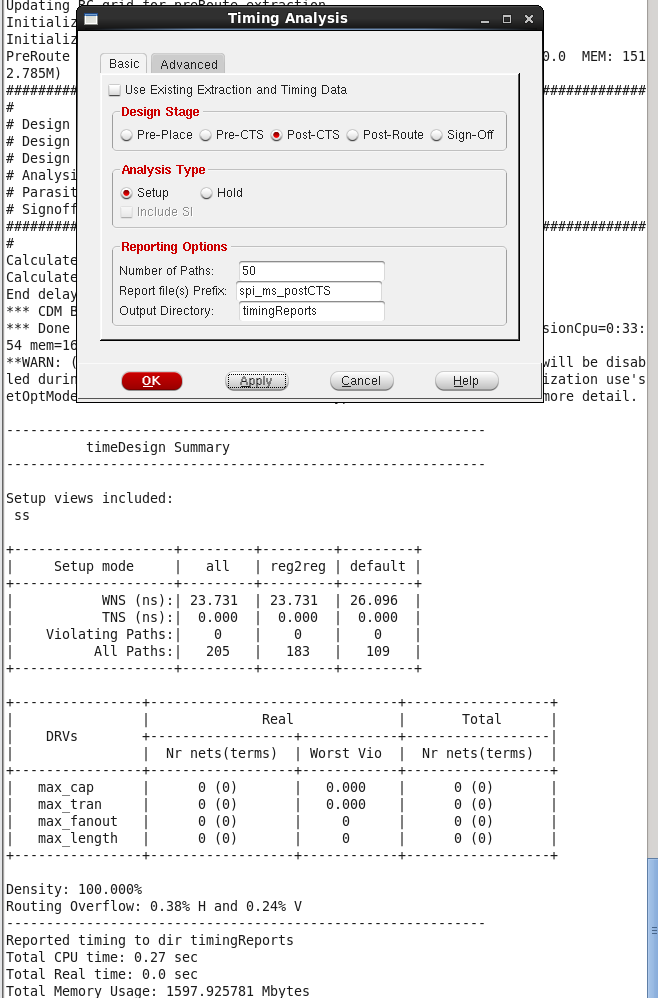
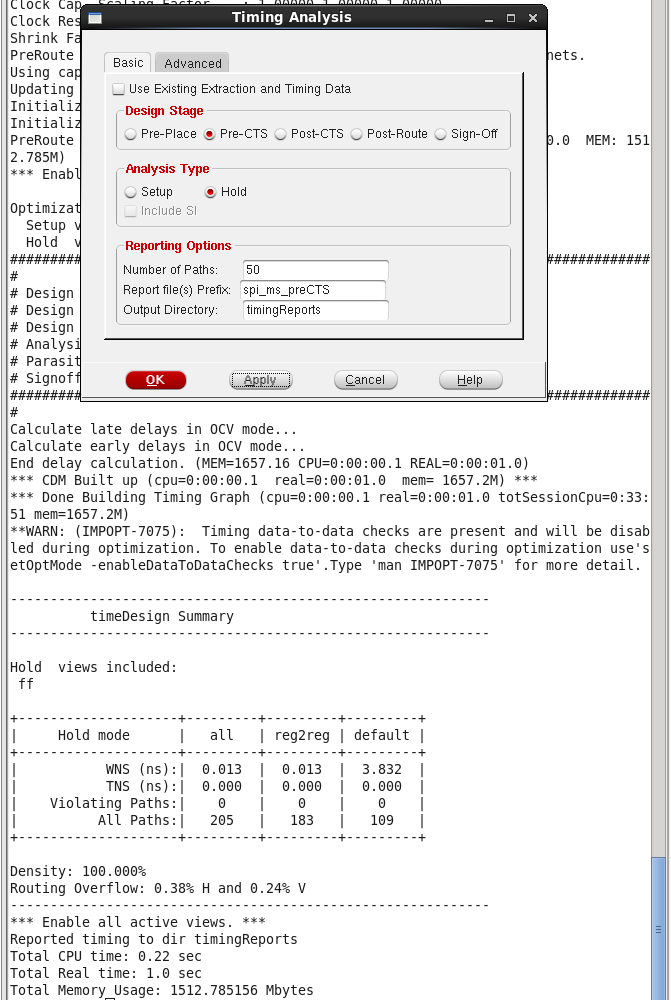
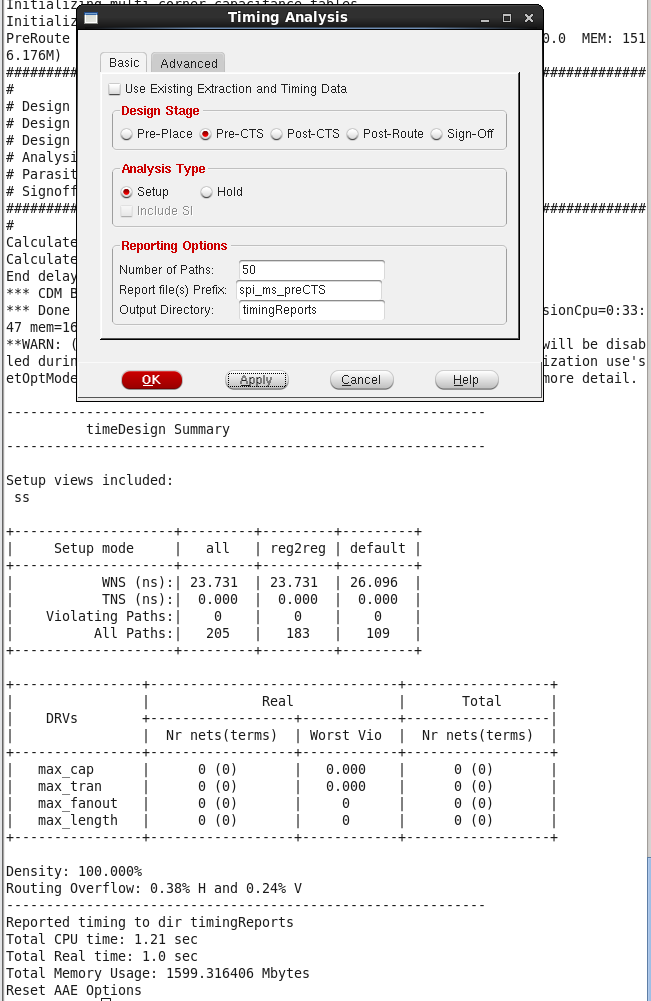
进行连接检查：



进行DRC与几何检查：这里几何检查有违例出现主要是因为通孔间距过近，这个问题可以主动修改。

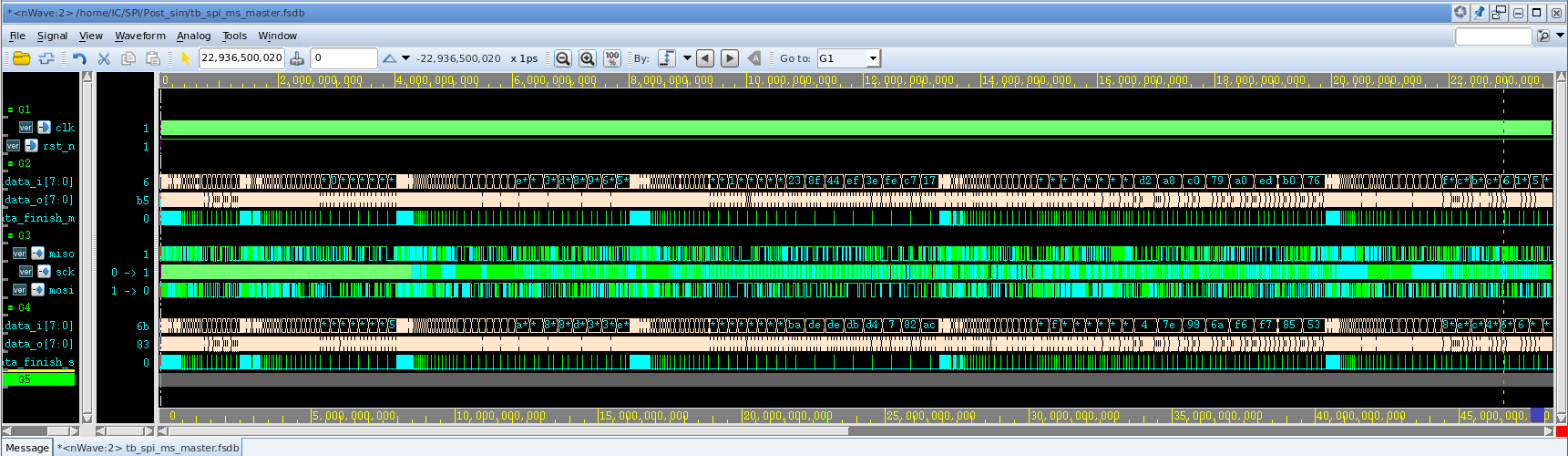


对于保持时间与建立时间进行检查：



后仿部分：

功能实现与前仿一致，至此，设计完成，部分需进行后续修改微调。



优化方面：

1. clk与sck的采样问题需要进行修改。
2. 对于位宽已经其他参数进行参数化书写，可以进行修改配置。