### spi\_master

#### 1.1.1.1 input变量声明

| clk |
| --- |
| rst\_n |
| data\_m [7:0] |
| spcon [7:0] |
| spibr [7:0] |
| spssn [7:0] |
| miso |

#### 1.1.1.2 output变量声明

| data\_r\_m [7:0] (reg) |
| --- |
| data\_finish\_m (reg) |
| mosi (reg) |
| sck (reg) |
| ssn [7:0] (wire) |

#### 1.1.1.3 内部变量声明

wire型变量声明

| tr\_en |
| --- |
| cpol |
| cpha |
| clk\_div [7:0] |
| sppr\_add1 [3:0] |

reg型变量声明

| clk\_cnt [7:0] |
| --- |
| sck\_edge\_cnt [4:0] |
| sck\_edge\_level |
| tr\_done |
| tr\_done\_dly1 |
| bit\_count [2:0] |

#### 1.1.1.4 各模块介绍

* 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

* 数据传输接收信号产生模块

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 |
| --- |
| rst\_n |
| data\_s [7:0] |
| spcon\_s [7:0] |
| mosi |
| sck |
| ssn |

#### output变量声明

| data\_r\_s [7:0] (reg) |
| --- |
| data\_finish\_s (reg) |
| miso (reg) |

#### 内部变量声明

wire型变量声明

| tr\_en |
| --- |
| cpol |
| cpha |
| sck\_edge\_level |

reg型变量声明

| sck\_edge\_cnt [4:0] |
| --- |
| sck\_dly1 |
| sck\_dly2 |
| bit\_count [2:0] |

#### 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 |
| --- |
| rst\_n |
| sfraddr\_w [1:0] |
| sfrwe |
| spidata\_i [7:0] |
| sfraddr\_r [2:0] |
| spssn\_i [7:0] |
| ssn |

#### output变量声明

| sfr\_data\_o [7:0] (reg) |
| --- |
| spssn\_o [7:0] (wire) |

#### inout变量声明

| mosi |
| --- |
| miso |
| sck |

#### 内部变量声明

l wire型变量声明

| spidr2 [7:0] |
| --- |
| spidr2\_m [7:0] |
| spidr2\_s [7:0] |
| data\_finish\_m |
| data\_finish\_s |
| miso\_m |
| miso\_s |
| mosi\_m |
| mosi\_s |
| sck\_m |
| sck\_s |
| intspi |
| mstr |

l reg型变量声明

| spicr1 |
| --- |
| spicr2 |
| spibr |
| spisr |
| spidr1 |
| rst\_n\_sync\_pre |
| rst\_n\_sync |

#### 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

* 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