Spi\_clgen.V

`include "spi\_defines.v" `include "timescale.v"

module spi\_clgen (clk\_in, rst, go, enable, last\_clk, divider, clk\_out, pos\_edge, neg\_edge); parameter Tp = 1;

input clk\_in; // input clock (system clock)

input rst; // reset

input enable; // clock enable input go; // start transfer input last\_clk; // last clock

input [`SPI\_DIVIDER\_LEN-1:0] divider; // clock divider (output clock is divided by

this value)

|  |  |
| --- | --- |
| output | clk\_out; // output clock |
| output | pos\_edge; // pulse marking positive edge of clk\_out |
| output | neg\_edge; // pulse marking negative edge of clk\_out |
| reg | clk\_out; |
| reg | pos\_edge; |
| reg | neg\_edge; |

reg [`SPI\_DIVIDER\_LEN-1:0] cnt; // clock counter wire cnt\_zero; // conter is equal to zero wire cnt\_one; // conter is equal to one

assign cnt\_zero = cnt == {`SPI\_DIVIDER\_LEN{1'b0}}; assign cnt\_one = cnt == {{`SPI\_DIVIDER\_LEN-1{1'b0}}, 1'b1};

// Counter counts half period always @(posedge clk\_in or posedge rst) begin

if(rst)

cnt <= #Tp {`SPI\_DIVIDER\_LEN{1'b1}};

else

begin

if(!enable || cnt\_zero)

cnt <= #Tp divider;

else cnt <= #Tp cnt - {{`SPI\_DIVIDER\_LEN-1{1'b0}}, 1'b1}; end end

// clk\_out is asserted every other half period always @(posedge clk\_in or posedge rst) begin

if(rst)

clk\_out <= #Tp 1'b0;

else clk\_out <= #Tp (enable && cnt\_zero && (!last\_clk || clk\_out)) ? ~clk\_out : clk\_out;

end

// Pos and neg edge signals always @(posedge clk\_in or posedge rst) begin

if(rst)

begin

pos\_edge <= #Tp 1'b0; neg\_edge <= #Tp 1'b0;

end else

begin

pos\_edge <= #Tp (enable && !clk\_out && cnt\_one) || (!(|divider) && clk\_out) ||

(!(|divider) && go && !enable); neg\_edge <= #Tp (enable && clk\_out && cnt\_one) || (!(|divider) && !clk\_out &&

enable); end

end endmodule

Spi\_defines.v

// Number of bits used for devider register. If used in system with // low frequency of system clock this can be reduced.

// Use SPI\_DIVIDER\_LEN for fine tuning theexact number.

//

//`define SPI\_DIVIDER\_LEN\_8 `define SPI\_DIVIDER\_LEN\_16

//`define SPI\_DIVIDER\_LEN\_24

//`define SPI\_DIVIDER\_LEN\_32

|  |  |  |
| --- | --- | --- |
| `ifdef SPI\_DIVIDER\_LEN\_8 |  |  |
| `define SPI\_DIVIDER\_LEN `endif  `ifdef SPI\_DIVIDER\_LEN\_16 | 8 | // Can be set from 1 to 8 |
| `define SPI\_DIVIDER\_LEN `endif  `ifdef SPI\_DIVIDER\_LEN\_24 | 16 | // Can be set from 9 to 16 |
| `define SPI\_DIVIDER\_LEN  `endif | 24 | // Can be set from 17 to 24 |

`ifdef SPI\_DIVIDER\_LEN\_32

`define SPI\_DIVIDER\_LEN 32 // Can be set from 25 to 32

# `endif

//

// Maximum nuber of bits that can be send/received at once.

// Use SPI\_MAX\_CHAR for fine tuning the exact number, when using

// SPI\_MAX\_CHAR\_32, SPI\_MAX\_CHAR\_24, SPI\_MAX\_CHAR\_16, SPI\_MAX\_CHAR\_8.

//

`define SPI\_MAX\_CHAR\_128 //`define SPI\_MAX\_CHAR\_64

//`define SPI\_MAX\_CHAR\_32

//`define SPI\_MAX\_CHAR\_24

//`define SPI\_MAX\_CHAR\_16

//`define SPI\_MAX\_CHAR\_8

`ifdef SPI\_MAX\_CHAR\_128

`define SPI\_MAX\_CHAR 128 // Can only be set to 128

## `define SPI\_CHAR\_LEN\_BITS 7

## `endif

# `ifdef SPI\_MAX\_CHAR\_64

`define SPI\_MAX\_CHAR 64 // Can only be set to 64

`define SPI\_CHAR\_LEN\_BITS 6

`endif

`ifdef SPI\_MAX\_CHAR\_32

`define SPI\_MAX\_CHAR 32 // Can be set from 25 to 32

`define SPI\_CHAR\_LEN\_BITS 5

`endif

`ifdef SPI\_MAX\_CHAR\_24

`define SPI\_MAX\_CHAR 24 // Can be set from 17 to 24

`define SPI\_CHAR\_LEN\_BITS 5

`endif

`ifdef SPI\_MAX\_CHAR\_16

`define SPI\_MAX\_CHAR 16 // Can be set from 9 to 16

`define SPI\_CHAR\_LEN\_BITS 4

`endif

`ifdef SPI\_MAX\_CHAR\_8

`define SPI\_MAX\_CHAR 8 // Can be set from 1 to 8

`define SPI\_CHAR\_LEN\_BITS 3

`endif

//

// Number of device select signals. Use SPI\_SS\_NB for fine tuning the // exact number.

//

//`define SPI\_SS\_NB\_8

//`define SPI\_SS\_NB\_16

//`define SPI\_SS\_NB\_24

`define SPI\_SS\_NB\_32

`ifdef SPI\_SS\_NB\_8

`define SPI\_SS\_NB 8 // Can be set from 1 to 8

`endif

`ifdef SPI\_SS\_NB\_16

`define SPI\_SS\_NB 16 // Can be set from 9 to 16

`endif

`ifdef SPI\_SS\_NB\_24

`define SPI\_SS\_NB 24 // Can be set from 17 to 24

`endif

`ifdef SPI\_SS\_NB\_32

`define SPI\_SS\_NB 32 // Can be set from 25 to 32

`endif

//

// Bits of WISHBONE address used for partial decoding of SPI registers.

//

`define SPI\_OFS\_BITS 4:2

//

// Register offset

//

`define SPI\_RX\_0 0

`define SPI\_RX\_1 1

`define SPI\_RX\_2 2

`define SPI\_RX\_3 3

`define SPI\_TX\_0 0

`define SPI\_TX\_1 1

`define SPI\_TX\_2 2

`define SPI\_TX\_3 3

`define SPI\_CTRL 4

`define SPI\_DEVIDE 5

`define SPI\_SS 6

//

// Number of bits in ctrl register

//

`define SPI\_CTRL\_BIT\_NB 14

//

// Control register bit position

//

`define SPI\_CTRL\_ASS 13

`define SPI\_CTRL\_IE 12

`define SPI\_CTRL\_LSB 11

`define SPI\_CTRL\_TX\_NEGEDGE 10

`define SPI\_CTRL\_RX\_NEGEDGE 9

`define SPI\_CTRL\_GO 8

`define SPI\_CTRL\_RES\_1 7

`define SPI\_CTRL\_CHAR\_LEN 6:0

Spi\_shift.v

`include "spi\_defines.v" `include "timescale.v"

module spi\_shift (clk, rst, latch, byte\_sel, len, lsb, go,

pos\_edge, neg\_edge, rx\_negedge, tx\_negedge, tip, last,

p\_in, p\_out, s\_clk, s\_in, s\_out); parameter Tp = 1;

|  |  |
| --- | --- |
| input | clk; // system clock |
| input | rst; // reset |
| input | [3:0] latch; // latch signal for storing the data in shift register |
| input | [3:0] byte\_sel; // byte select signals for storing the data in shift |

register

input [`SPI\_CHAR\_LEN\_BITS-1:0] len; // data len in bits (minus one)

input lsb; // lbs first on the line input go; // start stansfer input pos\_edge; // recognize posedge of sclk input neg\_edge; // recognize negedge of sclk input rx\_negedge; // s\_in is sampled on negative edge input tx\_negedge; // s\_out is driven on negative edge output tip; // transfer in progress output last; // last bit

input [31:0] p\_in; // parallel in

output [`SPI\_MAX\_CHAR-1:0] p\_out; // parallel out

|  |  |  |
| --- | --- | --- |
| input | s\_clk; | // serial clock |
| input | s\_in; | // serial in |
| output | s\_out; | // serial out |

reg s\_out; reg tip;

reg [`SPI\_CHAR\_LEN\_BITS:0] cnt; // data bit count reg [`SPI\_MAX\_CHAR-1:0] data; // shift register wire [`SPI\_CHAR\_LEN\_BITS:0] tx\_bit\_pos; // next bit position wire [`SPI\_CHAR\_LEN\_BITS:0] rx\_bit\_pos; // next bit position

wire rx\_clk; // rx clock enable wire tx\_clk; // tx clock enable assign p\_out = data;

assign tx\_bit\_pos = lsb ? {!(|len), len} - cnt : cnt -

{{`SPI\_CHAR\_LEN\_BITS{1'b0}},1'b1}; assign rx\_bit\_pos = lsb ? {!(|len), len} - (rx\_negedge ? cnt + {{`SPI\_CHAR\_LEN\_BITS{1'b0}},1'b1} : cnt) :

(rx\_negedge ? cnt : cnt - {{`SPI\_CHAR\_LEN\_BITS{1'b0}},1'b1});

assign last = !(|cnt);

assign rx\_clk = (rx\_negedge ? neg\_edge : pos\_edge) && (!last || s\_clk); assign tx\_clk = (tx\_negedge ? neg\_edge : pos\_edge) && !last;

// Character bit counter always @(posedge clk or posedge rst) begin

if(rst)

cnt <= #Tp {`SPI\_CHAR\_LEN\_BITS+1{1'b0}};

else

begin

if(tip)

cnt <= #Tp pos\_edge ? (cnt - {{`SPI\_CHAR\_LEN\_BITS{1'b0}}, 1'b1}) : cnt;

else cnt <= #Tp !(|len) ? {1'b1, {`SPI\_CHAR\_LEN\_BITS{1'b0}}} : {1'b0, len}; end

end

// Transfer in progress always @(posedge clk or posedge rst) begin

if(rst)

tip <= #Tp 1'b0;

else if(go && ~tip)

tip <= #Tp 1'b1;

else if(tip && last && pos\_edge)

tip <= #Tp 1'b0;

end

// Sending bits to the line always @(posedge clk or posedge rst) begin

if (rst)

s\_out <= #Tp 1'b0;

else s\_out <= #Tp (tx\_clk ) ? data[tx\_bit\_pos[`SPI\_CHAR\_LEN\_BITS-1:0]] : s\_out; end

// Receiving bits from the line always @(posedge clk or posedge rst) begin

if (rst)

data <= #Tp {`SPI\_MAX\_CHAR{1'b0}};

`ifdef SPI\_MAX\_CHAR\_128 else if (latch[0] && !tip)

begin

if (byte\_sel[3]) data[31:24] <= #Tp p\_in[31:24];

if (byte\_sel[2]) data[23:16] <= #Tp p\_in[23:16];

if (byte\_sel[1]) data[15:8] <= #Tp p\_in[15:8];

if (byte\_sel[0])

data[7:0] <= #Tp p\_in[7:0];

end

else if (latch[1] && !tip)

begin

if (byte\_sel[3]) data[63:56] <= #Tp p\_in[31:24];

if (byte\_sel[2]) data[55:48] <= #Tp p\_in[23:16];

if (byte\_sel[1]) data[47:40] <= #Tp p\_in[15:8];

if (byte\_sel[0])

data[39:32] <= #Tp p\_in[7:0];

end

else if (latch[2] && !tip)

begin

if (byte\_sel[3]) data[95:88] <= #Tp p\_in[31:24];

if (byte\_sel[2]) data[87:80] <= #Tp p\_in[23:16];

if (byte\_sel[1]) data[79:72] <= #Tp p\_in[15:8];

if (byte\_sel[0])

data[71:64] <= #Tp p\_in[7:0];

end

else if (latch[3] && !tip)

begin

if (byte\_sel[3]) data[127:120] <= #Tp p\_in[31:24];

if (byte\_sel[2]) data[119:112] <= #Tp p\_in[23:16];

if (byte\_sel[1]) data[111:104] <= #Tp p\_in[15:8]; if (byte\_sel[0])

data[103:96] <= #Tp p\_in[7:0];

end

`else

`ifdef SPI\_MAX\_CHAR\_64 else if (latch[0] && !tip)

begin

if (byte\_sel[3]) data[31:24] <= #Tp p\_in[31:24];

if (byte\_sel[2]) data[23:16] <= #Tp p\_in[23:16];

if (byte\_sel[1]) data[15:8] <= #Tp p\_in[15:8];

if (byte\_sel[0])

data[7:0] <= #Tp p\_in[7:0];

end

else if (latch[1] && !tip)

begin

if (byte\_sel[3]) data[63:56] <= #Tp p\_in[31:24];

if (byte\_sel[2]) data[55:48] <= #Tp p\_in[23:16];

if (byte\_sel[1]) data[47:40] <= #Tp p\_in[15:8];

if (byte\_sel[0])

data[39:32] <= #Tp p\_in[7:0];

end

`else else if (latch[0] && !tip) begin

`ifdef SPI\_MAX\_CHAR\_8 if (byte\_sel[0])

data[`SPI\_MAX\_CHAR-1:0] <= #Tp p\_in[`SPI\_MAX\_CHAR-1:0];

`endif

`ifdef SPI\_MAX\_CHAR\_16 if (byte\_sel[0]) data[7:0] <= #Tp p\_in[7:0];

if (byte\_sel[1])

data[`SPI\_MAX\_CHAR-1:8] <= #Tp p\_in[`SPI\_MAX\_CHAR-1:8];

`endif

`ifdef SPI\_MAX\_CHAR\_24 if (byte\_sel[0]) data[7:0] <= #Tp p\_in[7:0];

if (byte\_sel[1]) data[15:8] <= #Tp p\_in[15:8];

if (byte\_sel[2])

data[`SPI\_MAX\_CHAR-1:16] <= #Tp p\_in[`SPI\_MAX\_CHAR-1:16];

`endif

`ifdef SPI\_MAX\_CHAR\_32

if (byte\_sel[0]) data[7:0] <= #Tp p\_in[7:0];

if (byte\_sel[1]) data[15:8] <= #Tp p\_in[15:8];

if (byte\_sel[2]) data[23:16] <= #Tp p\_in[23:16];

if (byte\_sel[3])

data[`SPI\_MAX\_CHAR-1:24] <= #Tp p\_in[`SPI\_MAX\_CHAR-1:24];

`endif end

`endif

`endif else

data[rx\_bit\_pos[`SPI\_CHAR\_LEN\_BITS-1:0]] <= #Tp rx\_clk ? s\_in :

data[rx\_bit\_pos[`SPI\_CHAR\_LEN\_BITS-1:0]];

end endmodule

SPI\_slave.v

`include "timescale.v" module spi\_slave (sck,

cs, mosi, miso

);

input sck; input cs; input mosi; output miso;

reg [7:0]interna\_reg; //reg [7:0] slv\_reg=8'hFF; reg [7:0] slv\_reg=8'b10100010; reg [7:0] in\_reg=0; reg [7:0] rec\_reg=0; reg [7:0] count=0;

assign miso = (cs==1'b0) ? slv\_reg[7] : 1'bz; always@(negedge sck) begin

if (!cs)

slv\_reg<=slv\_reg<<1'b1;

end

always@(posedge sck) begin

if (!cs)

rec\_reg<={rec\_reg[6:0],mosi};

end

always@(posedge sck) begin

if (!cs) count<=count+1;

else count=0;

end

always@(posedge sck) if (count==7)

begin in\_reg<={rec\_reg[6:0],mosi};

end

Endmodule

SPI\_top.v

`include "spi\_defines.v" `include "timescale.v"

module spi\_top

(

// Wishbone signals wb\_clk\_i, wb\_rst\_i, wb\_adr\_i, wb\_dat\_i, wb\_dat\_o, wb\_sel\_i, wb\_we\_i, wb\_stb\_i, wb\_cyc\_i, wb\_ack\_o, wb\_err\_o, wb\_int\_o,

// SPI signals ss\_pad\_o, sclk\_pad\_o, mosi\_pad\_o, miso\_pad\_i

);

parameter Tp = 1;

// Wishbone signals

input wb\_clk\_i; // master clock input input wb\_rst\_i; // synchronous active high reset input [4:0] wb\_adr\_i; // lower address bits

input [32-1:0] wb\_dat\_i; // databus input output [32-1:0] wb\_dat\_o; // databus output

|  |  |  |
| --- | --- | --- |
| input | [3:0] wb\_sel\_i; | // byte select inputs |
| input | wb\_we\_i; | // write enable input |
| input | wb\_stb\_i; | // stobe/core select signal |
| input | wb\_cyc\_i; | // valid bus cycle input |
| output | wb\_ack\_o; | // bus cycle acknowledge output |
| output | wb\_err\_o; | // termination w/ error |
| output | wb\_int\_o; | // interrupt request signal output |

// SPI signals output [`SPI\_SS\_NB-1:0] ss\_pad\_o; // slave select

|  |  |  |
| --- | --- | --- |
| output | sclk\_pad\_o; | // serial clock |
| output | mosi\_pad\_o; | // master out slave in |
| input | miso\_pad\_i; | // master in slave out |

reg [32-1:0] wb\_dat\_o; reg wb\_ack\_o; reg wb\_int\_o;

// Internal signals

reg [`SPI\_DIVIDER\_LEN-1:0] divider; // Divider register reg [`SPI\_CTRL\_BIT\_NB-1:0] ctrl; // Control and status register reg [`SPI\_SS\_NB-1:0] ss; // Slave select register reg [32-1:0] wb\_dat; // wb data out wire [`SPI\_MAX\_CHAR-1:0] rx; // Rx register wire rx\_negedge; // miso is sampled on negative edge wire tx\_negedge; // mosi is driven on negative edge

wire [`SPI\_CHAR\_LEN\_BITS-1:0] char\_len; // char len

wire go; // go

wire lsb; // lsb first on line wire ie; // interrupt enable wire ass; // automatic slave select

wire spi\_divider\_sel; // divider register select wire spi\_ctrl\_sel; // ctrl register select wire [3:0] spi\_tx\_sel; // tx\_l register select wire spi\_ss\_sel; // ss register select wire tip; // transfer in progress wire pos\_edge; // recognize posedge of sclk wire neg\_edge; // recognize negedge of sclk wire last\_bit; // marks last character bit

// Address decoder assign spi\_divider\_sel = wb\_cyc\_i & wb\_stb\_i & (wb\_adr\_i[`SPI\_OFS\_BITS] ==

`SPI\_DEVIDE); assign spi\_ctrl\_sel = wb\_cyc\_i & wb\_stb\_i & (wb\_adr\_i[`SPI\_OFS\_BITS] ==

`SPI\_CTRL);

|  |  |
| --- | --- |
| assign spi\_tx\_sel[0]  `SPI\_TX\_0); | = wb\_cyc\_i & wb\_stb\_i & (wb\_adr\_i[`SPI\_OFS\_BITS] == |
| assign spi\_tx\_sel[1]  `SPI\_TX\_1); | = wb\_cyc\_i & wb\_stb\_i & (wb\_adr\_i[`SPI\_OFS\_BITS] == |
| assign spi\_tx\_sel[2]  `SPI\_TX\_2); | = wb\_cyc\_i & wb\_stb\_i & (wb\_adr\_i[`SPI\_OFS\_BITS] == |
| assign spi\_tx\_sel[3]  `SPI\_TX\_3); | = wb\_cyc\_i & wb\_stb\_i & (wb\_adr\_i[`SPI\_OFS\_BITS] == |

assign spi\_ss\_sel = wb\_cyc\_i & wb\_stb\_i & (wb\_adr\_i[`SPI\_OFS\_BITS] == `SPI\_SS);

// Read from registers always @(wb\_adr\_i or rx or ctrl or divider or ss) begin

case (wb\_adr\_i[`SPI\_OFS\_BITS]) `ifdef SPI\_MAX\_CHAR\_128

`SPI\_RX\_0: wb\_dat = rx[31:0];

`SPI\_RX\_1: wb\_dat = rx[63:32];

`SPI\_RX\_2: wb\_dat = rx[95:64];

`SPI\_RX\_3: wb\_dat = {{128-`SPI\_MAX\_CHAR{1'b0}}, rx[`SPI\_MAX\_CHAR-1:96]};

`else

`ifdef SPI\_MAX\_CHAR\_64

|  |  |
| --- | --- |
| `SPI\_RX\_0: | wb\_dat = rx[31:0]; |
| `SPI\_RX\_1: | wb\_dat = {{64-`SPI\_MAX\_CHAR{1'b0}}, rx[`SPI\_MAX\_CHAR-1:32]}; |
| `SPI\_RX\_2: | wb\_dat = 32'b0; |
| `SPI\_RX\_3:  `else | wb\_dat = 32'b0; |
| `SPI\_RX\_0: | wb\_dat = {{32-`SPI\_MAX\_CHAR{1'b0}}, rx[`SPI\_MAX\_CHAR-1:0]}; |
| `SPI\_RX\_1: | wb\_dat = 32'b0; |
| `SPI\_RX\_2: | wb\_dat = 32'b0; |
| `SPI\_RX\_3:  `endif  `endif | wb\_dat = 32'b0; |
| `SPI\_CTRL: | wb\_dat = {{32-`SPI\_CTRL\_BIT\_NB{1'b0}}, ctrl}; |

`SPI\_DEVIDE: wb\_dat = {{32-`SPI\_DIVIDER\_LEN{1'b0}}, divider};

`SPI\_SS: wb\_dat = {{32-`SPI\_SS\_NB{1'b0}}, ss}; default: wb\_dat = 32'bx;

endcase

end

// Wb data out always @(posedge wb\_clk\_i or posedge wb\_rst\_i) begin

if (wb\_rst\_i)

wb\_dat\_o <= #Tp 32'b0; else wb\_dat\_o <= #Tp wb\_dat; end

// Wb acknowledge always @(posedge wb\_clk\_i or posedge wb\_rst\_i) begin

if (wb\_rst\_i)

wb\_ack\_o <= #Tp 1'b0; else

wb\_ack\_o <= #Tp wb\_cyc\_i & wb\_stb\_i & ~wb\_ack\_o;

end

// Wb error assign wb\_err\_o = 1'b0;

// Interrupt

always @(posedge wb\_clk\_i or posedge wb\_rst\_i) begin

if (wb\_rst\_i)

wb\_int\_o <= #Tp 1'b0;

else if (ie && tip && last\_bit && pos\_edge)

wb\_int\_o <= #Tp 1'b1;

else if (wb\_ack\_o)

wb\_int\_o <= #Tp 1'b0;

end

// Divider register always @(posedge wb\_clk\_i or posedge wb\_rst\_i) begin

if (wb\_rst\_i)

divider <= #Tp {`SPI\_DIVIDER\_LEN{1'b0}};

else if (spi\_divider\_sel && wb\_we\_i && !tip)

begin

`ifdef SPI\_DIVIDER\_LEN\_8 if (wb\_sel\_i[0])

divider <= #Tp wb\_dat\_i[`SPI\_DIVIDER\_LEN-1:0];

`endif

`ifdef SPI\_DIVIDER\_LEN\_16 if (wb\_sel\_i[0]) divider[7:0] <= #Tp wb\_dat\_i[7:0];

if (wb\_sel\_i[1])

divider[`SPI\_DIVIDER\_LEN-1:8] <= #Tp wb\_dat\_i[`SPI\_DIVIDER\_LEN-1:8];

`endif

`ifdef SPI\_DIVIDER\_LEN\_24 if (wb\_sel\_i[0]) divider[7:0] <= #Tp wb\_dat\_i[7:0];

if (wb\_sel\_i[1]) divider[15:8] <= #Tp wb\_dat\_i[15:8];

if (wb\_sel\_i[2])

divider[`SPI\_DIVIDER\_LEN-1:16] <= #Tp wb\_dat\_i[`SPI\_DIVIDER\_LEN-1:16]; `endif

`ifdef SPI\_DIVIDER\_LEN\_32 if (wb\_sel\_i[0]) divider[7:0] <= #Tp wb\_dat\_i[7:0];

if (wb\_sel\_i[1]) divider[15:8] <= #Tp wb\_dat\_i[15:8];

if (wb\_sel\_i[2]) divider[23:16] <= #Tp wb\_dat\_i[23:16];

if (wb\_sel\_i[3])

divider[`SPI\_DIVIDER\_LEN-1:24] <= #Tp wb\_dat\_i[`SPI\_DIVIDER\_LEN-1:24];

`endif end

end

// Ctrl register always @(posedge wb\_clk\_i or posedge wb\_rst\_i) begin

if (wb\_rst\_i)

ctrl <= #Tp {`SPI\_CTRL\_BIT\_NB{1'b0}};

else if(spi\_ctrl\_sel && wb\_we\_i && !tip)

begin

if (wb\_sel\_i[0])

//ctrl[7:0] <= #Tp wb\_dat\_i[7:0] | {7'b0, ctrl[0]}; //??? Acting as bug during

Verification ctrl[7:0] <= #Tp wb\_dat\_i[7:0]; //line modified from above line

if (wb\_sel\_i[1])

ctrl[`SPI\_CTRL\_BIT\_NB-1:8] <= #Tp wb\_dat\_i[`SPI\_CTRL\_BIT\_NB-1:8];

end

else if(tip && last\_bit && pos\_edge)

ctrl[`SPI\_CTRL\_GO] <= #Tp 1'b0;

end

assign rx\_negedge = ctrl[`SPI\_CTRL\_RX\_NEGEDGE]; assign tx\_negedge = ctrl[`SPI\_CTRL\_TX\_NEGEDGE]; assign go = ctrl[`SPI\_CTRL\_GO]; assign char\_len = ctrl[`SPI\_CTRL\_CHAR\_LEN]; assign lsb = ctrl[`SPI\_CTRL\_LSB]; assign ie = ctrl[`SPI\_CTRL\_IE]; assign ass = ctrl[`SPI\_CTRL\_ASS];

// Slave select register always @(posedge wb\_clk\_i or posedge wb\_rst\_i) begin

if (wb\_rst\_i)

ss <= #Tp {`SPI\_SS\_NB{1'b0}};

else if(spi\_ss\_sel && wb\_we\_i && !tip)

begin

`ifdef SPI\_SS\_NB\_8

if (wb\_sel\_i[0])

ss <= #Tp wb\_dat\_i[`SPI\_SS\_NB-1:0];

`endif

`ifdef SPI\_SS\_NB\_16 if (wb\_sel\_i[0]) ss[7:0] <= #Tp wb\_dat\_i[7:0];

if (wb\_sel\_i[1])

ss[`SPI\_SS\_NB-1:8] <= #Tp wb\_dat\_i[`SPI\_SS\_NB-1:8];

`endif

`ifdef SPI\_SS\_NB\_24 if (wb\_sel\_i[0]) ss[7:0] <= #Tp wb\_dat\_i[7:0];

if (wb\_sel\_i[1]) ss[15:8] <= #Tp wb\_dat\_i[15:8];

if (wb\_sel\_i[2])

ss[`SPI\_SS\_NB-1:16] <= #Tp wb\_dat\_i[`SPI\_SS\_NB-1:16];

`endif

`ifdef SPI\_SS\_NB\_32 if (wb\_sel\_i[0]) ss[7:0] <= #Tp wb\_dat\_i[7:0];

if (wb\_sel\_i[1]) ss[15:8] <= #Tp wb\_dat\_i[15:8];

if (wb\_sel\_i[2]) ss[23:16] <= #Tp wb\_dat\_i[23:16];

if (wb\_sel\_i[3])

ss[`SPI\_SS\_NB-1:24] <= #Tp wb\_dat\_i[`SPI\_SS\_NB-1:24];

`endif end

end assign ss\_pad\_o = ~((ss & {`SPI\_SS\_NB{tip & ass}}) | (ss & {`SPI\_SS\_NB{!ass}}));

spi\_clgen clgen (.clk\_in(wb\_clk\_i), .rst(wb\_rst\_i), .go(go), .enable(tip),

.last\_clk(last\_bit),

.divider(divider), .clk\_out(sclk\_pad\_o), .pos\_edge(pos\_edge),

.neg\_edge(neg\_edge));

spi\_shift shift (.clk(wb\_clk\_i), .rst(wb\_rst\_i), .len(char\_len[`SPI\_CHAR\_LEN\_BITS-1:0]),

.latch(spi\_tx\_sel[3:0] & {4{wb\_we\_i}}), .byte\_sel(wb\_sel\_i), .lsb(lsb),

.go(go), .pos\_edge(pos\_edge), .neg\_edge(neg\_edge),

.rx\_negedge(rx\_negedge), .tx\_negedge(tx\_negedge),

.tip(tip), .last(last\_bit),

.p\_in(wb\_dat\_i), .p\_out(rx),

.s\_clk(sclk\_pad\_o), .s\_in(miso\_pad\_i), .s\_out(mosi\_pad\_o)); endmodule

`Timescale.v

`timescale 1ns / 10ps