

```
////////////////////////////////////  
////  
//// OpenCores                      MC68HC11E based SPI interface  ////  
////  
//// Author: Richard Herveille      ////  
////      richard@asics.ws          ////  
////      www.asics.ws              ////  
////  
////////////////////////////////////  
////  
//// Copyright (C) 2002 Richard Herveille  ////  
////      richard@asics.ws          ////  
////  
//// This source file may be used and distributed without  ////  
//// restriction provided that this copyright statement is not  ////  
//// removed from the file and that any derivative work contains  ////  
//// the original copyright notice and the associated disclaimer.////  
////  
////      THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY  ////  
//// EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED  ////  
//// TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS  ////  
//// FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL THE AUTHOR  ////  
//// OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,  ////  
//// INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES  ////  
//// (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE  ////  
//// GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR  ////  
//// BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF  ////  
//// LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT  ////  
//// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT  ////  
//// OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE  ////  
//// POSSIBILITY OF SUCH DAMAGE.  ////  
////  
////////////////////////////////////
```

```
// CVS Log  
//  
// $Id: simple_spi_top.v,v 1.5 2004/02/28 15:59:50 rherveille Exp $  
//  
// $Date: 2004/02/28 15:59:50 $  
// $Revision: 1.5 $  
// $Author: rherveille $  
// $Locker: $  
// $State: Exp $  
//  
// Change History:  
//      $Log: simple_spi_top.v,v $  
//      Revision 1.5  2004/02/28 15:59:50  rherveille  
//      Fixed SCK_O generation bug.  
//      This resulted in a major rewrite of the serial interface engine.  
//  
//      Revision 1.4  2003/08/01 11:41:54  rherveille  
//      Fixed some timing bugs.  
//  
//      Revision 1.3  2003/01/09 16:47:59  rherveille  
//      Updated clkcnt size and decoding due to new SPR bit assignments.  
//  
//      Revision 1.2  2003/01/07 13:29:52  rherveille  
//      Changed SPR bits coding.  
//  
//      Revision 1.1.1.1  2002/12/22 16:07:15  rherveille  
//      Initial release  
//  
//
```

```
//
// Motorola MC68HC11E based SPI interface
//
// Currently only MASTER mode is supported
//

// synopsys translate_off
`include "timescale.v"
// synopsys translate_on

module simple_spi_top(
  // 8bit WISHBONE bus slave interface
  input wire      clk_i,          // clock
  input wire      rst_i,          // reset (asynchronous active low)
  input wire      cyc_i,          // cycle
  input wire      stb_i,          // strobe
  input wire [1:0] adr_i,         // address
  input wire      we_i,           // write enable
  input wire [7:0] dat_i,         // data input
  output reg [7:0] dat_o,         // data output
  output reg      ack_o,          // normal bus termination
  output reg      inta_o,         // interrupt output

  // SPI port
  output reg      sck_o,          // serial clock output
  output wire     mosi_o,         // MasterOut SlaveIN
  input wire      miso_i         // MasterIn SlaveOut
);

//
// Module body
//
reg [7:0] spcr; // Serial Peripheral Control Register ('HC11 naming)
wire [7:0] spsr; // Serial Peripheral Status register ('HC11 naming)
reg [7:0] sper; // Serial Peripheral Extension register
reg [7:0] treg, rreg; // Transmit/Receive register

// fifo signals
wire [7:0] rfdout;
reg      wfre, rfwe;
wire     rfre, rffull, rfempty;
wire [7:0] wfdout;
wire     wfwe, wffull, wfempty;

// misc signals
wire     tirq; // transfer interrupt (selected number of transfers done)
wire     wfov; // write fifo overrun (writing while fifo full)
reg [1:0] state; // statemachine state
reg [2:0] bcnt;

//
// Wishbone interface
wire wb_acc = cyc_i & stb_i; // WISHBONE access
wire wb_wr  = wb_acc & we_i; // WISHBONE write access

// dat_i
always @(posedge clk_i or negedge rst_i)
  if (~rst_i)
    begin
      spcr <= #1 8'h10; // set master bit
```

```
        sper <= #1 8'h00;
    end
    else if (wb_wr)
    begin
        if (adr_i == 2'b00)
            sPCR <= #1 dat_i | 8'h10; // always set master bit

            if (adr_i == 2'b11)
                sper <= #1 dat_i;
        end

// write fifo
assign wfwe = wb_acc & (adr_i == 2'b10) & ack_o & we_i;
assign wfov = wfwe & wffull;

// dat_o
always @(posedge clk_i)
    case(adr_i) // synopsys full_case parallel_case
        2'b00: dat_o <= #1 sPCR;
        2'b01: dat_o <= #1 sPSR;
        2'b10: dat_o <= #1 rfdout;
        2'b11: dat_o <= #1 sper;
    endcase

// read fifo
assign rfre = wb_acc & (adr_i == 2'b10) & ack_o & ~we_i;

// ack_o
always @(posedge clk_i or negedge rst_i)
    if (~rst_i)
        ack_o <= #1 1'b0;
    else
        ack_o <= #1 wb_acc & !ack_o;

// decode Serial Peripheral Control Register
wire    spie = sPCR[7]; // Interrupt enable bit
wire    spe  = sPCR[6]; // System Enable bit
wire    dwom = sPCR[5]; // Port D Wired-OR Mode Bit
wire    mstr = sPCR[4]; // Master Mode Select Bit
wire    cpol = sPCR[3]; // Clock Polarity Bit
wire    cpha = sPCR[2]; // Clock Phase Bit
wire [1:0] spr = sPCR[1:0]; // Clock Rate Select Bits

// decode Serial Peripheral Extension Register
wire [1:0] icnt = sper[7:6]; // interrupt on transfer count
wire [1:0] spre = sper[1:0]; // extended clock rate select

wire [3:0] espr = {spre, spr};

// generate status register
wire wr_spsr = wb_wr & (adr_i == 2'b01);

reg spif;
always @(posedge clk_i)
    if (~spe)
        spif <= #1 1'b0;
    else
        spif <= #1 (tirq | spif) & ~(wr_spsr & dat_i[7]);

reg wcol;
always @(posedge clk_i)
    if (~spe)
        wcol <= #1 1'b0;
```

```
else
    wcol <= #1 (wfov | wcol) & ~(wr_spsr & dat_i[6]);

assign spsr[7] = spif;
assign spsr[6] = wcol;
assign spsr[5:4] = 2'b00;
assign spsr[3] = wffull;
assign spsr[2] = wfempty;
assign spsr[1] = rffull;
assign spsr[0] = rfempty;

// generate IRQ output (inta_o)
always @(posedge clk_i)
    inta_o <= #1 spif & spie;

//
// hookup read/write buffer fifo
fifo4 #(7)
rfifo(
    .clk ( clk_i ),
    .rst ( rst_i ),
    .clr ( ~spe ),
    .din ( treg ),
    .we ( rfwe ),
    .dout ( rfdout ),
    .re ( rfre ),
    .full ( rffull ),
    .empty ( rfempty )
),
wfifo(
    .clk ( clk_i ),
    .rst ( rst_i ),
    .clr ( ~spe ),
    .din ( dat_i ),
    .we ( wfwe ),
    .dout ( wfdout ),
    .re ( wfre ),
    .full ( wffull ),
    .empty ( wfempty )
);

//
// generate clk divider
reg [11:0] clkcnt;
always @(posedge clk_i)
    if(spe & (|clkcnt & |state))
        clkcnt <= #1 clkcnt - 11'h1;
    else
        case (espr) // synopsys full_case parallel_case
            4'b0000: clkcnt <= #1 12'h0; // 2 -- original M68HC11 coding
            4'b0001: clkcnt <= #1 12'h1; // 4 -- original M68HC11 coding
            4'b0010: clkcnt <= #1 12'h3; // 16 -- original M68HC11 coding
            4'b0011: clkcnt <= #1 12'hf; // 32 -- original M68HC11 coding
            4'b0100: clkcnt <= #1 12'h1f; // 8
            4'b0101: clkcnt <= #1 12'h7; // 64
            4'b0110: clkcnt <= #1 12'h3f; // 128
            4'b0111: clkcnt <= #1 12'h7f; // 256
            4'b1000: clkcnt <= #1 12'hff; // 512
            4'b1001: clkcnt <= #1 12'h1ff; // 1024
            4'b1010: clkcnt <= #1 12'h3ff; // 2048
            4'b1011: clkcnt <= #1 12'h7ff; // 4096
        endcase
```

```
// generate clock enable signal
wire ena = ~|clkcnt;

// transfer statemachine
always @(posedge clk_i)
  if (~spe)
    begin
      state <= #1 2'b00; // idle
      bcnt  <= #1 3'h0;
      treg  <= #1 8'h00;
      wfre  <= #1 1'b0;
      rfwe  <= #1 1'b0;
      sck_o <= #1 1'b0;
    end
  else
    begin
      wfre <= #1 1'b0;
      rfwe <= #1 1'b0;

      case (state) //synopsys full_case parallel_case
        2'b00: // idle state
          begin
            bcnt  <= #1 3'h7; // set transfer counter
            treg  <= #1 wfdout; // load transfer register
            sck_o <= #1 cpol; // set sck

            if (~wfempty) begin
              wfre <= #1 1'b1;
              state <= #1 2'b01;
              if (cpha) sck_o <= #1 ~sck_o;
            end
          end

        2'b01: // clock-phase2, next data
          if (ena) begin
            sck_o <= #1 ~sck_o;
            state <= #1 2'b11;
          end

        2'b11: // clock phasel
          if (ena) begin
            treg <= #1 {treg[6:0], miso_i};
            bcnt <= #1 bcnt -3'h1;

            if (~|bcnt) begin
              state <= #1 2'b00;
              sck_o <= #1 cpol;
              rfwe <= #1 1'b1;
            end else begin
              state <= #1 2'b01;
              sck_o <= #1 ~sck_o;
            end
          end

        2'b10: state <= #1 2'b00;
      endcase
    end

assign mosi_o = treg[7];

// count number of transfers (for interrupt generation)
```

```
reg [1:0] tcnt; // transfer count
always @(posedge clk_i)
  if (~spe)
    tcnt <= #1 icnt;
  else if (rfwe) // rfwe gets asserted when all bits have been transfered
    if (|tcnt)
      tcnt <= #1 tcnt - 2'h1;
    else
      tcnt <= #1 icnt;

assign tirq = ~|tcnt & rfwe;

endmodule
```

```
// synopsys translate_off
`include "timescale.v"
// synopsys translate_on
```

```
// 4 entry deep fast fifo
module fifo4(clk, rst, clr, din, we, dout, re, full, empty);
```

```
parameter dw = 7;
```

```
input  clk, rst;
input  clr;
input  [dw:0] din;
input  we;
output [dw:0] dout;
input  re;
output full, empty;
```

```
////////////////////////////////////
//
// Local Wires
//
```

```
reg [dw:0] mem[3:0];
reg [1:0] wp;
reg [1:0] rp;
wire [1:0] wp_p1;
wire [1:0] rp_p1;
wire full, empty;
reg gb;
```

```
////////////////////////////////////
//
// Misc Logic
//
```

```
always @(posedge clk or negedge rst)
  if(!rst) wp <= #1 2'h0;
  else
    if(clr) wp <= #1 2'h0;
    else
      if(we) wp <= #1 wp_p1;
```

```
assign wp_p1 = wp + 2'h1;
```

```
always @(posedge clk or negedge rst)
    if(!rst) rp <= #1 2'h0;
    else
        if(clr) rp <= #1 2'h0;
        else
            if(re) rp <= #1 rp_p1;

assign rp_p1 = rp + 2'h1;

// Fifo Output
assign dout = mem[ rp ];

// Fifo Input
always @(posedge clk)
    if(we) mem[ wp ] <= #1 din;

// Status
assign empty = (wp == rp) & !gb;
assign full = (wp == rp) & gb;

// Guard Bit ...
always @(posedge clk)
    if(!rst) gb <= #1 1'b0;
    else
        if(clr) gb <= #1 1'b0;
        else
            if((wp_p1 == rp) & we) gb <= #1 1'b1;
            else
                if(re) gb <= #1 1'b0;

endmodule
```