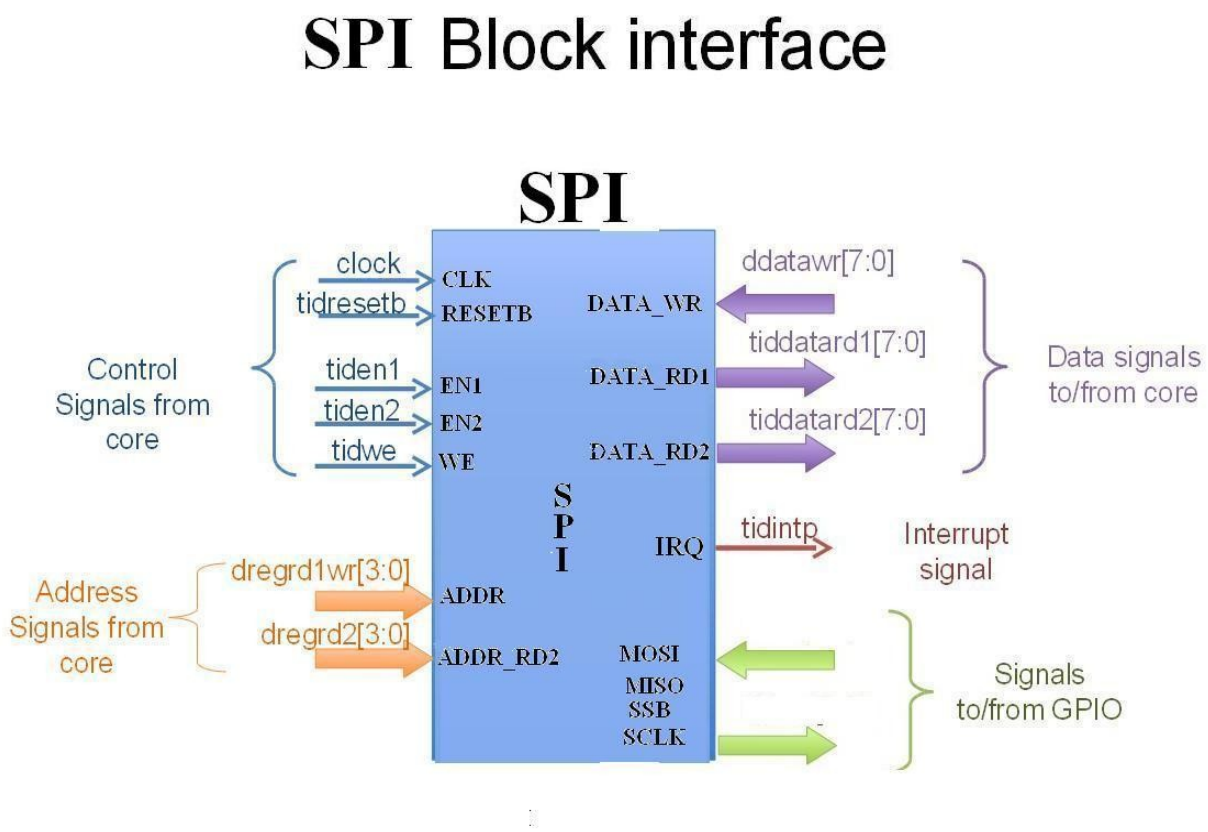
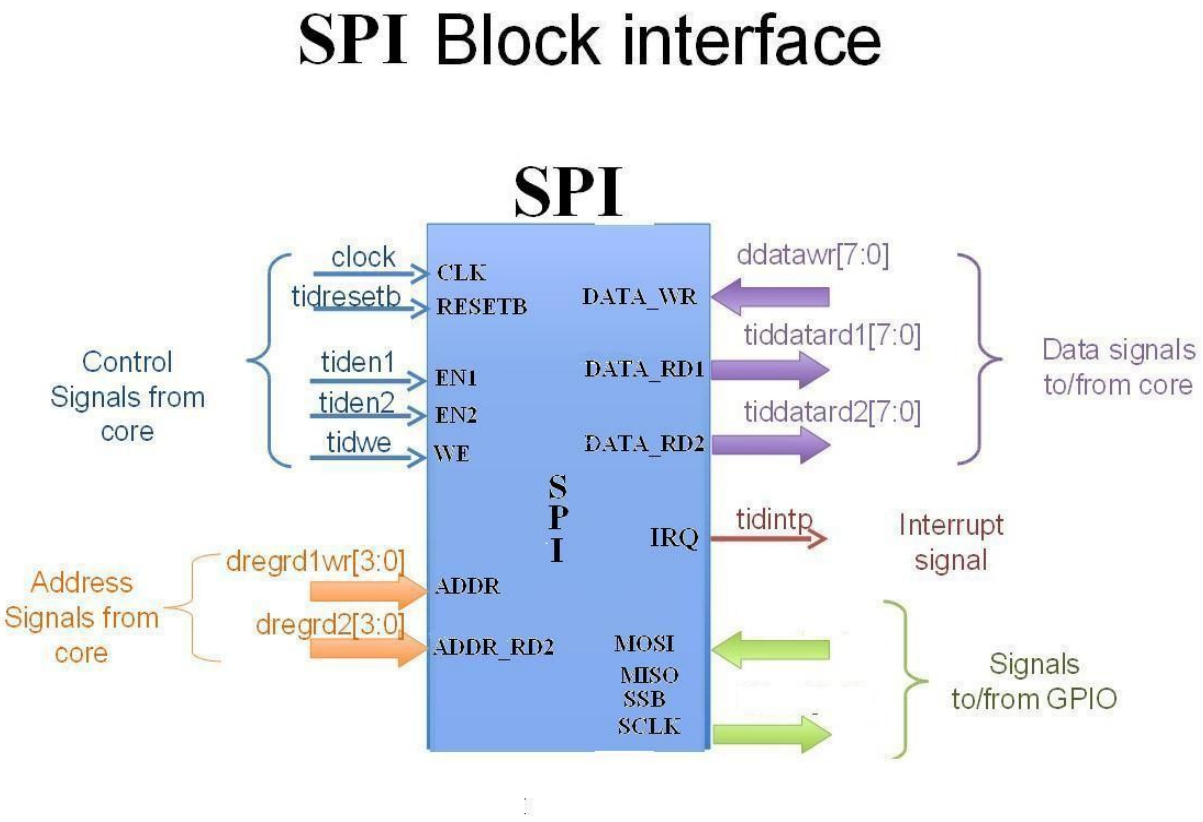
EYE-C SPI BLOCK DESIGN AND VERIFICATION SPECIFICATION



VERSION HISTORY

Date	Version	Author	Comments
12/07/10	0.1	Carlos Oppus	Initial SPI design
02/02/11	0.2	Carlos Oppus	SPI module based on Z8 SPI module Master SPI -- working
03/04/11	0.3	Carlos Oppus	SPI module based on Z8 SPI module and ZUF Core specifications -- initial
04/05/11	1	Carlos Oppus	SPI module based on Z8 SPI module and ZUF Core specifications – complete Master SPI implementation
05/05/11	1.1	Carlos Oppus	Updated SPI module based on Z8 SPI module and ZUF Core specifications – complete Master SPI implementation Supported changes in ZUP core
05/09/11	1.5	Carlos Oppus	Updated SPI module based on Z8 SPI module and ZUF Core specifications – complete Master/Slave (bidirectional) SPI implementation Supported changes in ZUP core

SPI BLOCK INTERFACE



OVERVIEW

ASPI module based on Z8 SPI module

Features of the SPI include:

- Full-duplex, synchronous, character-oriented communication
- Four-wire interface
- Data transfers rates up to a maximum of one-half the system clock frequency
- Error detection
- Dedicated Baud Rate Generator

The four basic SPI signals are:

- MISO (Master In, Slave Out)
- MOSI (Master Out, Slave In)
- SCK (SPI Serial Clock)
- SS (Slave Select)

These SPI signals are discussed in the following paragraphs. Each signal is described in both MASTER and SLAVE modes.

Master In, Slave Out The Master In, Slave Out (MISO) pin is configured as an input in a master device and as an output in a slave device. It is one of the two lines that transfer serial data, with the most-significant bit sent first. The MISO pin of a slave device is placed in a high-impedance state if the slave is not selected. When the SPI is not enabled, this signal is in a high-impedance state.

Master Out, Slave In The Master Out, Slave In (MOSI) pin is configured as an output in a master device and as an input in a slave device. It is one of the two lines that transfer serial data, with the most-significant bit sent first. When the SPI is not enabled, this signal is in a high-impedance state.

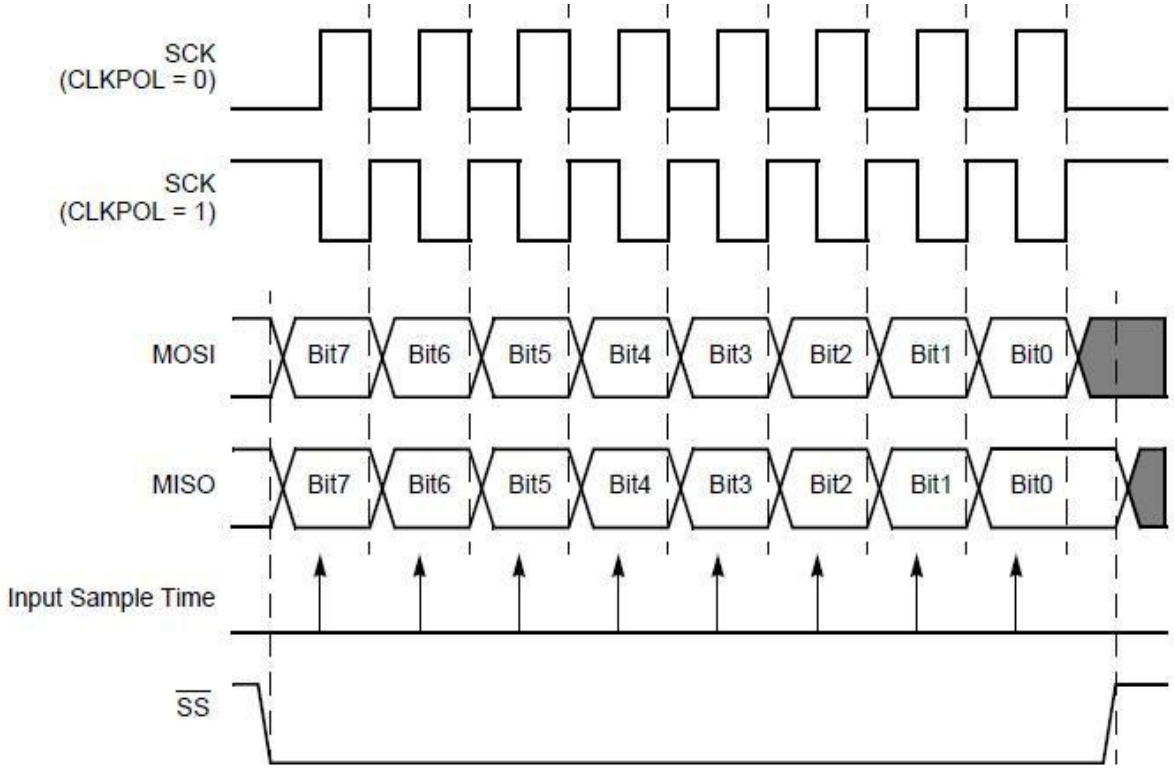
Slave Select The active Low Slave Select (SS) input signal is used to select the SPI as a slave device. It must be Low prior to all data communication and must stay Low for the duration of the data transfer. The SS input signal must be High for the SPI to operate as a master device. If the SS signal goes Low in Master mode, a Mode Fault error flag (MODF) is set in the SPI_SR register. For more information, see SPI Status Register on page 211. When the clock phase (CPHA) is set to 0, the shift clock is the logical

OR of SS with SCK. In this clock phase mode, SS must go High between successive characters in an SPI message. When CPHA is set to 1, SS remains Low for several SPI characters. In cases where there is only one SPI slave, its SS line could be tied Low as long as CPHA is set to 1. For more information on CPHA, see SPI Control Register on page 210.

Serial Clock The Serial Clock (SCK) is used to synchronize data movement both in and out of the device via its MOSI and MISO pins. The master and slave are each capable of exchanging a byte of data during a sequence of eight clock cycles. Because SCK is generated by the master, the SCK pin becomes an input on a slave device. The SPI contains an internal divide-by-two clock divider. In MASTER mode, the SPI serial clock is one-half the fre-quency of the clock signal created by the SPI's Baud Rate Generator

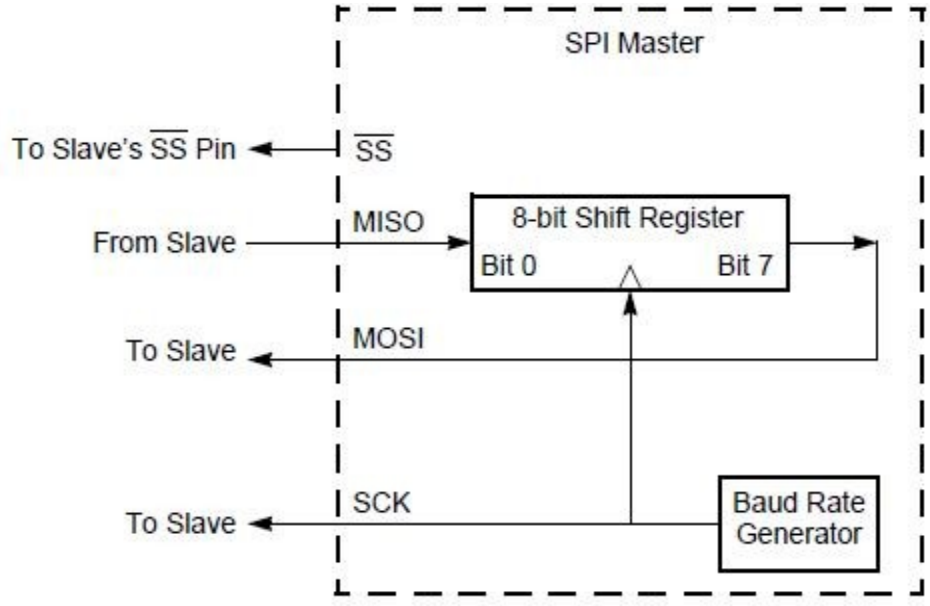
Table 62. SPI Clock Phase (PHASE) and Clock Polarity (CLKPOL) Operation

PHASE	CLKPOL	SCK Transmit Edge	SCK Receive Edge	SCK Idle State
0	0	Falling	Rising	Low
0	1	Rising	Falling	High
1	0	Rising	Falling	Low
1	1	Falling	Rising	High



SPI Timing When PHASE is 0

BLOCK DIAGRAM



SPI Configured as a Master in a Single Master, Single Slave System

PORTS

Name	Mode	Size	Description	Source	Destination
CLK	Input	1 bit	master clock (ZUP master clock)	ZUP Core	SPI
RESETB	Input	1 bit	low enabled reset, master reset of ZUP	ZUP Core	SPI
ADDR	Input	[3:0]	4-bit address lines of Read1 & Write	ZUP Core	SPI
ADDR_RD2	Input	[3:0]	4-bit address lines of Read2	ZUP Core	SPI
WE	Input	1 bit	Write Enable	ZUP Core	SPI
EN1	Input	1 bit	Output Enable for read1	ZUP Core	SPI
EN2	Input	1 bit	Output Enable for read2	ZUP Core	SPI
DATA_WR	Input	[7:0]	Output data bus	ZUP Core	
MISO	inout	1 bit	serial data input of SPI, Master In/Slave Out serial data of SPI, also known as SDI	ZUP Core	
SS	inout	1 bit	low enabled Slave Select of the SPI, also known as CS (chip select)	SPI module	SPI Slave Device
SCLK	inout	1 bit	serial clock of the SPI	SPI module	SPI Slave Device

MOSI	inout	1 bit	serial data output of SPI, Master Out/Slave In serial data of SPI, also known as SDO	SPI module	SPI Slave Device
DATA_RD1	Output	[7:0]	Input read1 data bus	SPI module	ZUP Core
DATA_RD2	Output	[7:0]	Input read2 data bus	SPI module	ZUP Core
IRQ	Output	1 bit	Interrupt signal	SPI module	ZUP Core

REGISTERS

SPI Data Register (SPIDATA)

BITS	7	6	5	4	3	2	1	0
FIELD	DATA							
RESET	X							
R/W	R/W							
ADDR	F60H							

DATA—Data
Transmit and/or receive data.

SPI Control Register (SPICTL)

BITS	7	6	5	4	3	2	1	0
FIELD	IRQE	STR	BIRQ	PHASE	CLKPOL	WOR	MMEN	SPIEN
RESET	0							
R/W	R/W							
ADDR	F61H							

SPI Status Register (SPISTAT)

BITS	7	6	5	4	3	2	1	0
FIELD	IRQ	OVR	COL	ABT	Reserved		TXST	SLAS
RESET	0							1
R/W	R/W*				R			
ADDR	F62H							

Note: R/W* = Read access. Write a 1 to clear the bit to 0.

SPI Mode Register (SPIMODE)

BITS	7	6	5	4	3	2	1	0
FIELD	Reserved		DIAG	NUMBITS[2:0]			SSIO	SSV
RESET	0							
R/W	R		R/W					
ADDR	F63H							

SPI Diagnostic State Register (SPIDST)

BITS	7	6	5	4	3	2	1	0
FIELD	SCKEN	TCKEN	SPISTATE					
RESET	0							
R/W	R							
ADDR	F64H							

SPI Baud Rate High Byte Register (SPIBRH)

BITS	7	6	5	4	3	2	1	0
FIELD	BRH							
RESET	1							
R/W	R/W							
ADDR	F66H							

BRH = SPI Baud Rate High Byte
Most significant byte, BRG[15:8], of the SPI Baud Rate Generator's reload value.

SPI Baud Rate Low Byte Register (SPIBRL)

BITS	7	6	5	4	3	2	1	0
FIELD	BRL							
RESET	1							
R/W	R/W							
ADDR	F67H							

BRL = SPI Baud Rate Low Byte
Least significant byte, BRG[7:0], of the SPI Baud Rate Generator's reload value.

LIMITATIONS

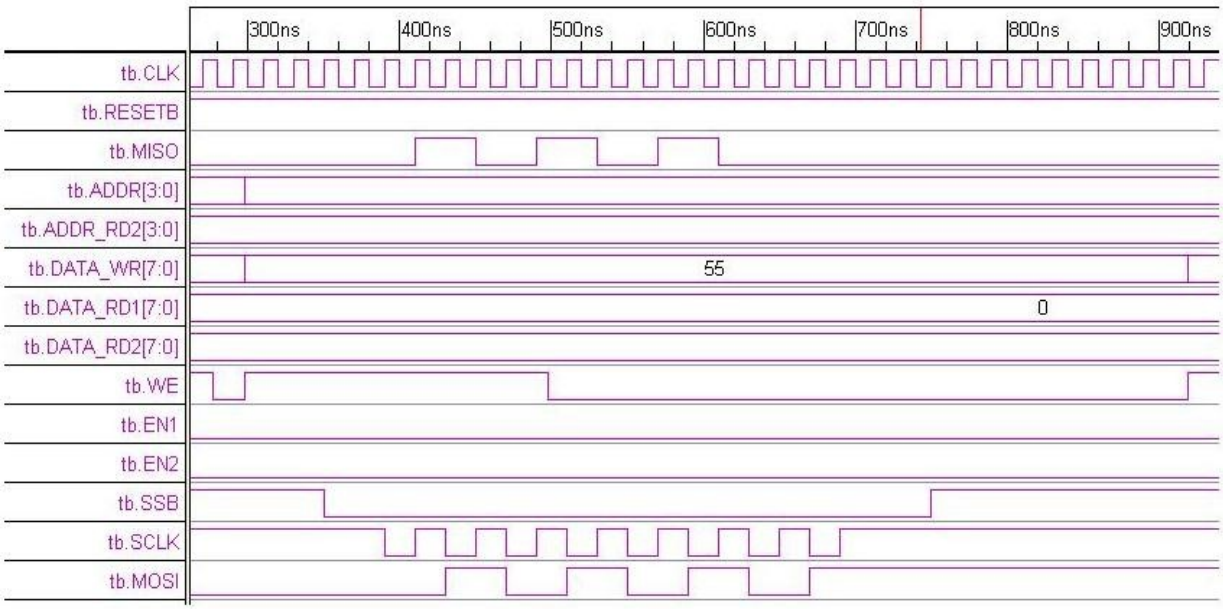
At the moment the SPI module is only configured as a Master SPI.

VERIFICATION

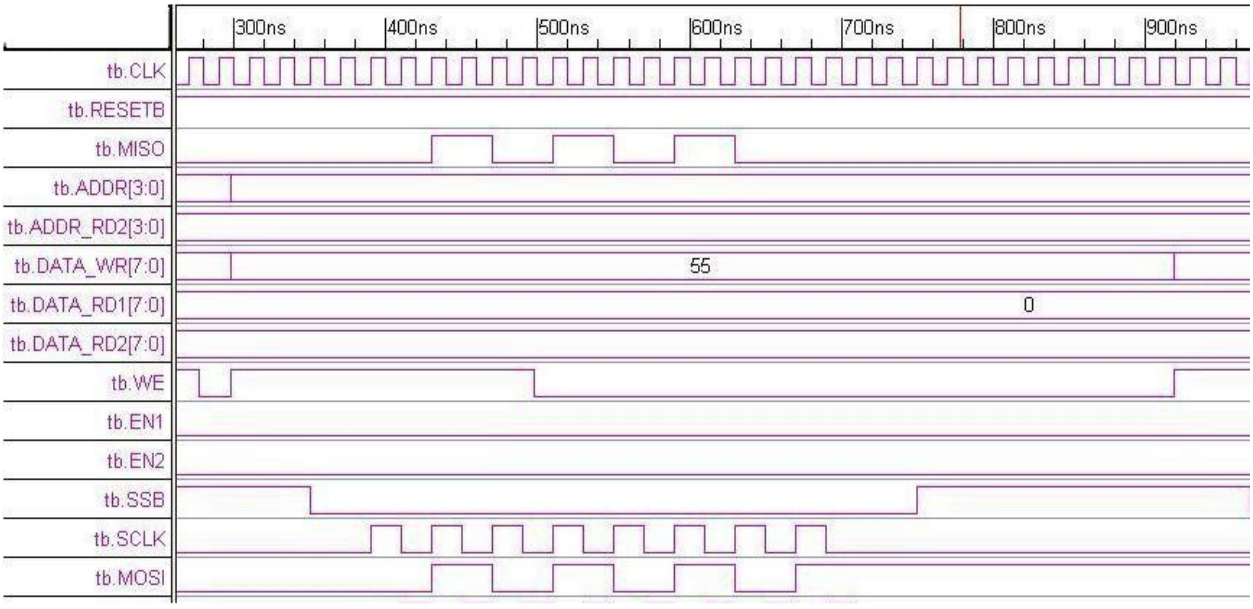
VERIFICATION ARCHITECTURE

Uses testbench.

Phase 0, Clkpol = 1



Phase 0, Clkpol = 0;



COVERAGE

List the scenarios covered by your simulations. Often, each scenario would correspond to a test.

CODE

```
/*
ASPI module
based on Z8 SPI module
SPI Master only mode

inputs:
  CLK       = master clock of Eye-C
  RESETB       = low enabled reset, master reset of Eye-C
  ADDR      = [3:0], 4-bit address lines
  WE        = write enable = 1 means ZUP core write operation
  EN1       = output enable = 1 means ZUP core read1 operation
  EN2       = output enable = 1 means ZUP core read2 operation
  DATA_WR  = [7:0] input from ZUP

  MISO      = Master In/Slave Out serial data of SPI, also known as SDI
```

```
outputs:
  SSB       = low enabled Slave Select of the SPI, also known as CS (chip select)
  SCLK      = serial clock of the SPI
  MOSI      = Master Out/Slave In serial data of SPI, also known as SDO
  DATA_IN1   = [7:0] output data to ZUP

Legend
  START     = high enabled start signal, to signify start of SPI transfer
  MODE      = follows the z8 SPI mode
  DATASIZE  = 9-bit = 0 to 511 (only 0 to 256 for z8 SPI), length of data to be transferred via SPI
  BRG       = 16-bit prescaler input of master clock in generating SCLK
         =    BRG   SCLK frequency
              0      (clk freq)/65536
              1      (clk freq)/2
              2      (clk freq)/4
              3      (clk freq)/6
              n      (clk freq)/(2 X n) and so on
  DATA_WRint  = 8-bit data, current data to be transferred via SPI
  DATA_IN1int   = 8-bit parallel data generated from 8 bits of MISO
  NEXT_DATA_WR  = active high with pulse width equal to 1 CLK period, use to alert CPU core that
              SPI block has produced SPI_DATA and is ready to get next DATA_WR byte
RWbar     = RWbar = 1 means ZUP core read operation,
                    RWbar = 0 means ZUP core write operation,.
*/

module ASPI_bidir(CLK, RESETB, ADDR, ADDR_RD2, DATA_WR, DATA_RD1, DATA_RD2, WE, EN1, EN2,
MISO, SSB, SCLK, MOSI);
//module ASPI(CLK, RESETB, START, MODE, DATASIZE, BRG, DATA_WR, MISO, SSB, SCLK, MOSI,
DATA_IN, NEXT_DATA_WR);
input CLK, RESETB;
input WE, EN1, EN2;//RWbar;
input [3:0] ADDR, ADDR_RD2;
input [7:0] DATA_WR;
output [7:0] DATA_RD1, DATA_RD2;

inout MISO, MOSI;
inout SSB;
inout SCLK;

reg  [7:0] DATA_IN1, DATA_IN2;
reg EN1r, EN2r, EN1a, EN2a;
assign DATA_RD1 = (EN1) ? DATA_IN1 : 8'h0;
assign DATA_RD2 = (EN2) ? DATA_IN2 : 8'h0;


reg START;
reg [1:0] MODE;
reg [8:0] DATASIZE;
reg [15:0] BRG;
reg  [7:0] DATA_WRint;
reg  [7:0] DATA_IN1int;
reg NEXT_DATA_WR;

reg mySSB;
wire sclk;
reg sclkint, sdo;
```

```
wire sclk_ne, sclk_pe;
reg [3:0] state;
initial state = 4'b0;
reg [3:0] statein;
initial statein = 4'b0;
reg [8:0] startcnt;
initial startcnt = 9'b0;
wire sclk1;
reg sclk2, sclk3;
wire sdi;
wire [16:0] PRESCALER;
wire SCKEN, TCKEN;
reg SPIEN, MMEN, WOR, BIRQ, IRQE, PHASE, CLKPOL, STR;
reg IRQtmp, OVR, COL, ABT, RES1, RES0, SLAS;
wire TXST;
reg TIMER_ENABLE, READY, SPI_MODE;
reg OVRX, COLX, ABTX, TXSTX;
reg  DIAG, SSIO,SSV;
reg [2:0] NUMBITS;
wire start_pex, start_pe1;
reg start_pe;


assign sclk1 = CLKPOL ? (sclk | (statein > 4'h7)) : ~(sclk | (statein > 4'h7)) ;
wire SCLK_INT, MISO_INT, MOSI_INT, SSB_INT;
assign SSB = (MMEN | SSIO) ? SSB_INT : 1'bz;
assign MISO_INT = MMEN ? MISO : MOSI_INT; //1'bz;
assign MISO = MMEN ? 1'bz : MOSI_INT;
assign MOSI = MMEN ? MOSI_INT : 1'bz;
assign SCLK = MMEN ? SCLK_INT : 1'bz;
assign SCLK_INT = PHASE ? sclk3 : sclk2;
assign MOSI_INT = sdo;
assign SSB_INT = (MMEN & SSIO) ? SSV : (MMEN ? mySSB : (SSIO ? SLAS : 1'bz));
assign sdi = MISO_INT;


assign TXST = !READY;

always @ (posedge CLK or negedge RESETB)
begin
  if (!RESETB)           READY <= 1'b1;
  else if (start_pex)    READY <= 1'b0;
  else if (NEXT_DATA_WR)  READY <= 1'b1;
end

//always @ (posedge CLK or negedge RESETB)
always @(RESETB or ADDR or WE)
begin
  if (!RESETB)
    begin //default values
      BRG <= 1'b1;
      START = 1'b0;
      STR <= 1'b0;
      DATASIZE <= 9'h1;
      DATA_WRint <= 8'h55;
      CLKPOL <= 1'b0;
```

```
      NUMBITS <= 3'b000;
      PHASE <= 1'b0;
      OVR <= 1'b0;
      TIMER_ENABLE <= 1'b1;
      MMEN <= 1'b1;
      SPI_MODE <= 1'b1;
      SPIEN <= 1'b1;
      IRQE <= 1'b1;
      BIRQ <= 1'b1;
      RES1 <= 1'b0;
      RES0 <= 1'b0;
      SLAS <= 1'b1;
      SSIO <= 1'b0;
      SSV  <= 1'b1;
      DATA_IN1 <= 8'h0;
      IRQtmp <= 1'b0;
    end
  else if (WE)
// ZUP core write operation
  begin
    //START = 1'b0;
    case (ADDR)
    4'h0 : begin
             if (mySSB) //SSB_INT)
               begin
                 DATA_WRint <= DATA_WR;
                 START = 1'b1;
               end
             else if (!START)
               OVR <= 1'b1;
           end
    4'h1 : begin
             {IRQE, STR, BIRQ, PHASE, CLKPOL, WOR, MMEN, SPIEN } <= DATA_WR;
             if (DATA_WR[6]) IRQtmp <= 1'b1; //if (STR)
             case ({IRQE,BIRQ})
               2'b00 : TIMER_ENABLE <= 1'b0;
               2'b01 : TIMER_ENABLE <= 1'b1;
               2'b10 : TIMER_ENABLE <= 1'b1;
               2'b11 : TIMER_ENABLE <= 1'b1;
             endcase
             if (MMEN) SPI_MODE <= 1'b1; //master mode
             else      SPI_MODE <= 1'b0; //slave mode
           end
    4'h2 : begin
             {IRQtmp, OVRX, COLX, ABTX, RES1, RES0, TXSTX, SLAS} <= DATA_WR;
             if (DATA_WR[7]) STR <= 1'b0; //if (IRQtmp)
           end
    4'h3 : begin
             {DIAG,NUMBITS,SSIO,SSV} = DATA_WR[5:0];
           end
    4'h5 : begin
           end
    4'h6 : begin
             BRG[15:8] <= DATA_WR;
           end
    4'h7 : begin
```

```
              BRG[7:0] <= DATA_WR;
             end
     default: begin
              START = 1'b0;
              IRQtmp <= 1'b0;
             end
     endcase
    end
   else
    begin
     START = 1'b0;
     IRQtmp <= 1'b0;
    end

end

wire IRQ;
edge_detect edgeIRQ(CLK, IRQtmp, 1'b1, IRQ);

//----read1 operation ------
//always @ (posedge CLK or negedge RESETB)
always @(RESETB or ADDR or EN1) // or DATA_IN1int or IRQ or OVR or COL or ABT or RES1 or RES0 or
TXST or SLAS or SCKEN or TCKEN or state or BRG)
begin
  if (!RESETB)
    begin //default values
      DATA_IN1 <= 8'h0;
    end
   else if (EN1)
// ZUP core read operation
     case (ADDR)
     4'h0 : begin
              DATA_IN1 <= DATA_IN1int;
             end
     4'h1 : begin
             end
     4'h2 : begin
              DATA_IN1 <= {IRQ, OVR, COL, ABT, RES1, RES0, TXST, SLAS};
             end
     4'h3 : begin
              DATA_IN1 <= {2'b00,1'b1,NUMBITS,SSIO,SSB_INT}; //{2'b00,DIAG,NUMBITS,SSIO,SSV};
             end
     4'h4 : begin
              DATA_IN1 <= {SCKEN, TCKEN, 2'b00,state};
             end
     4'h5 : begin
             end
     4'h6 : begin
               DATA_IN1 <= BRG[15:8];
             end
     4'h7 : begin
              DATA_IN1 <= BRG[7:0];
             end

     default: begin
              DATA_IN1 <= 8'h0;
```

```
              START = 1'b0;
             end
    endcase
   else
    begin
     DATA_IN1 <= 8'h0;
    end

end

//--- end of read1 ----

//----read2 operation ------
//always @ (posedge CLK or negedge RESETB)
always @(RESETB or ADDR or EN2)
begin
  if (!RESETB)
    begin //default values
      DATA_IN2 <= 8'h0;
    end
   else if (EN2)
// ZUP core read2 operation
     case (ADDR_RD2)
     4'h0 : begin
              DATA_IN2 <= DATA_IN1int;
             end
     4'h1 : begin
             end
     4'h2 : begin
              DATA_IN2 <= {IRQ, OVR, COL, ABT, RES1, RES0, TXST, SLAS};
             end
     4'h3 : begin
              DATA_IN2 <= {2'b00,1'b1,NUMBITS,1'b1,SSB_INT}; //{2'b00,DIAG,NUMBITS,SSIO,SSV};
             end
     4'h4 : begin
              DATA_IN2 <= {SCKEN, TCKEN, 2'b00,state};
             end
     4'h5 : begin
           end
     4'h6 : begin
               DATA_IN2 <= BRG[15:8];
             end
     4'h7 : begin
              DATA_IN2 <= BRG[7:0];
             end

     default: begin
              DATA_IN2 <= 8'h0;
             end
   endcase
   else
    begin
     DATA_IN2 <= 8'h0;
    end

end
```

```
//----- end of read2 --------

assign PRESCALER = (BRG == 0) ? 17'h0FFFF : (BRG-1);

wire myrst1;
assign myrst1 = ~(~RESETB | mySSB);
reg myrst2, myrst;
always @(posedge CLK)
begin
  myrst2 <= myrst1;
  myrst  <= myrst1 | myrst2;
end

initial sclk2 = 1'b1; //MODE[0];
always @(posedge CLK or negedge myrst)
begin
  if (!myrst) sclk2 <= CLKPOL;
  else        sclk2 <= sclk1;
end

initial sclk3 =1'b1; // MODE[0];
always @(posedge CLK or negedge myrst)
begin
  if (!myrst)                 sclk3 <= CLKPOL;
  else if (sclk_ne | sclk_pe)   sclk3 <= sclk2;
end

edge_detect edgestartx(CLK, START, 1'b1, start_pex);
edge_detect edgestart(CLK, (state == 4'h9), 1'b1, start_pe1);

initial start_pe = 1'b0;
always @(posedge CLK)
  if (!RESETB)      start_pe <= 1'b0;
  else          start_pe <= start_pex;

always @(posedge CLK or negedge myrst)
  if (!myrst)           startcnt <= 9'b0;
  else if (start_pe1)     startcnt <= startcnt + 9'b1;

initial mySSB = 1'b1;
always @(posedge CLK)
  if (!RESETB)                mySSB <= 1'b1;
  else if (startcnt == DATASIZE) mySSB <= 1'b1;
  else if (start_pe)            mySSB <= 1'b0;

initial NEXT_DATA_WR = 1'b0;
always @(posedge CLK or negedge myrst)
  if (!myrst)   NEXT_DATA_WR <= 1'b0;
  else          NEXT_DATA_WR <= start_pe1;


reg [16:0] timer;
initial timer = 9'b0;
always @(posedge CLK or negedge myrst1)
  if (!myrst1)                  timer <= 17'b0;
```

```
  else if (timer == (PRESCALER))     timer <= 17'b0;
  else                               timer <= timer + 17'b1;

initial sclkint = 1'b1;
always @(posedge CLK or negedge myrst1)
  if (!myrst1)                       sclkint <= 1'b1;
  else if (timer == (PRESCALER))     sclkint <= ~sclkint;

assign sclk = sclkint;

edge_detect edgesclkne(CLK, sclk, 1'b0, sclk_ne);
edge_detect edgesclkpe(CLK, sclk, 1'b1, sclk_pe);

initial sdo = 1'b0;
always @(posedge CLK or negedge myrst1)
  if (!myrst1) state <= 4'h0;
  else if (sclk_ne)
  case (state)
    4'h0 : begin
           case (NUMBITS)
             3'b000 : begin sdo <= DATA_WRint[7];  state <= 4'h1;  end
             3'b001 : begin sdo <= DATA_WRint[0];  state <= 4'h8; end
             3'b010 : begin sdo <= DATA_WRint[1];  state <= 4'h1; end
             3'b011 : begin sdo <= DATA_WRint[2];  state <= 4'h1; end
             3'b100 : begin sdo <= DATA_WRint[3];  state <= 4'h1;  end
             3'b101 : begin sdo <= DATA_WRint[4];  state <= 4'h1; end
             3'b110 : begin sdo <= DATA_WRint[5];  state <= 4'h1; end
             3'b111 : begin sdo <= DATA_WRint[6];  state <= 4'h1; end
           endcase
          end
    4'h1 : begin
           case (NUMBITS)
             3'b000 : begin sdo <= DATA_WRint[6];  state <= 4'h2;  end
             3'b001 : begin state <= 4'h8; end
             3'b010 : begin sdo <= DATA_WRint[0];  state <= 4'h8; end
             3'b011 : begin sdo <= DATA_WRint[1];  state <= 4'h2; end
             3'b100 : begin sdo <= DATA_WRint[2];  state <= 4'h2;  end
             3'b101 : begin sdo <= DATA_WRint[3];  state <= 4'h2; end
             3'b110 : begin sdo <= DATA_WRint[4];  state <= 4'h2; end
             3'b111 : begin sdo <= DATA_WRint[5];  state <= 4'h2; end
           endcase
          end
    4'h2 : begin
           case (NUMBITS)
             3'b000 : begin sdo <= DATA_WRint[5];  state <= 4'h3;  end
             3'b001 : begin state <= 4'h8; end
             3'b010 : begin state <= 4'h8; end
             3'b011 : begin sdo <= DATA_WRint[0];  state <= 4'h8; end
             3'b100 : begin sdo <= DATA_WRint[1];  state <= 4'h3;  end
             3'b101 : begin sdo <= DATA_WRint[2];  state <= 4'h3; end
             3'b110 : begin sdo <= DATA_WRint[3];  state <= 4'h3; end
             3'b111 : begin sdo <= DATA_WRint[4];  state <= 4'h3; end
           endcase
          end
    4'h3 : begin
           case (NUMBITS)
```

```
          3'b000 : begin sdo <= DATA_WRint[4];  state <= 4'h4;  end
          3'b001 : begin state <= 4'h8; end
          3'b010 : begin state <= 4'h8; end
          3'b011 : begin state <= 4'h8; end
          3'b100 : begin sdo <= DATA_WRint[0];  state <= 4'h8;  end
          3'b101 : begin sdo <= DATA_WRint[1];  state <= 4'h4; end
          3'b110 : begin sdo <= DATA_WRint[2];  state <= 4'h4; end
          3'b111 : begin sdo <= DATA_WRint[3];  state <= 4'h4; end
        endcase
      end
4'h4 : begin
        case (NUMBITS)
          3'b000 : begin sdo <= DATA_WRint[3];  state <= 4'h5;  end
          3'b001 : begin state <= 4'h8; end
          3'b010 : begin state <= 4'h8; end
          3'b011 : begin state <= 4'h8; end
          3'b100 : begin state <= 4'h8;  end
          3'b101 : begin sdo <= DATA_WRint[0];  state <= 4'h8; end
          3'b110 : begin sdo <= DATA_WRint[1];  state <= 4'h5; end
          3'b111 : begin sdo <= DATA_WRint[2];  state <= 4'h5; end
        endcase
      end
4'h5 : begin
        case (NUMBITS)
          3'b000 : begin sdo <= DATA_WRint[2];  state <= 4'h6;  end
          3'b001 : begin state <= 4'h8; end
          3'b010 : begin state <= 4'h8; end
          3'b011 : begin state <= 4'h8; end
          3'b100 : begin state <= 4'h8;  end
          3'b101 : begin state <= 4'h8; end
          3'b110 : begin sdo <= DATA_WRint[0];  state <= 4'h8; end
          3'b111 : begin sdo <= DATA_WRint[1];  state <= 4'h6; end
        endcase
      end
4'h6 : begin
        case (NUMBITS)
          3'b000 : begin sdo <= DATA_WRint[1];  state <= 4'h7;  end
          3'b001 : begin state <= 4'h8; end
          3'b010 : begin state <= 4'h8; end
          3'b011 : begin state <= 4'h8; end
          3'b100 : begin state <= 4'h8;  end
          3'b101 : begin state <= 4'h8; end
          3'b110 : begin state <= 4'h8; end
          3'b111 : begin sdo <= DATA_WRint[0];  state <= 4'h8; end
        endcase
      end
4'h7 : begin
        case (NUMBITS)
          3'b000 : begin sdo <= DATA_WRint[0];  state <= 4'h8;  end
          3'b001 : begin state <= 4'h8; end
          3'b010 : begin state <= 4'h8; end
          3'b011 : begin state <= 4'h8; end
          3'b100 : begin state <= 4'h8;  end
          3'b101 : begin state <= 4'h8; end
          3'b110 : begin state <= 4'h8; end
          3'b111 : begin state <= 4'h8; end
```

```
          endcase
        end
   4'h8 : begin state <= 4'h9; end
   4'h9 : begin state <= 4'h0; end
   default: state <= 4'h0;
  endcase


initial DATA_IN1int = 8'h0;
always @(posedge CLK or negedge myrst)
  if (!myrst)         statein <= 4'h0;
  else if (start_pex)  DATA_IN1int <= 8'h0;
  else  if (sclk_pe)
  case (statein)
   4'h0 : begin
          case (NUMBITS)
            3'b000 : begin DATA_IN1int[7] <= sdi;  statein <= 4'h1;  end
            3'b001 : begin DATA_IN1int[0] <= sdi;  statein <= 4'h8;  end
            3'b010 : begin DATA_IN1int[1] <= sdi;  statein <= 4'h1;  end
            3'b011 : begin DATA_IN1int[2] <= sdi;  statein <= 4'h1;  end
            3'b100 : begin DATA_IN1int[3] <= sdi;  statein <= 4'h1;  end
            3'b101 : begin DATA_IN1int[4] <= sdi;  statein <= 4'h1;  end
            3'b110 : begin DATA_IN1int[5] <= sdi;  statein <= 4'h1;  end
            3'b111 : begin DATA_IN1int[6] <= sdi;  statein <= 4'h1;  end
          endcase
        end
   4'h1 : begin
          case (NUMBITS)
            3'b000 : begin DATA_IN1int[6] <= sdi;  statein <= 4'h2;  end
            3'b001 : begin statein <= 4'h8;  end
            3'b010 : begin DATA_IN1int[0] <= sdi;  statein <= 4'h8;  end
            3'b011 : begin DATA_IN1int[1] <= sdi;  statein <= 4'h2;  end
            3'b100 : begin DATA_IN1int[2] <= sdi;  statein <= 4'h2;  end
            3'b101 : begin DATA_IN1int[3] <= sdi;  statein <= 4'h2;  end
            3'b110 : begin DATA_IN1int[4] <= sdi;  statein <= 4'h2;  end
            3'b111 : begin DATA_IN1int[5] <= sdi;  statein <= 4'h2;  end
          endcase
        end
   4'h2 : begin
          case (NUMBITS)
            3'b000 : begin DATA_IN1int[5] <= sdi;  statein <= 4'h3;  end
            3'b001 : begin statein <= 4'h8;  end
            3'b010 : begin statein <= 4'h8;  end
            3'b011 : begin DATA_IN1int[0] <= sdi;  statein <= 4'h8;  end
            3'b100 : begin DATA_IN1int[1] <= sdi;  statein <= 4'h3;  end
            3'b101 : begin DATA_IN1int[2] <= sdi;  statein <= 4'h3;  end
            3'b110 : begin DATA_IN1int[3] <= sdi;  statein <= 4'h3;  end
            3'b111 : begin DATA_IN1int[4] <= sdi;  statein <= 4'h3;  end
          endcase
        end
   4'h3 : begin
          case (NUMBITS)
            3'b000 : begin DATA_IN1int[4] <= sdi;  statein <= 4'h4;  end
            3'b001 : begin statein <= 4'h8;  end
            3'b010 : begin statein <= 4'h8;  end
            3'b011 : begin statein <= 4'h8;  end
```

```
          3'b100 : begin DATA_IN1int[0] <= sdi;  statein <= 4'h8;  end
          3'b101 : begin DATA_IN1int[1] <= sdi;  statein <= 4'h4;  end
          3'b110 : begin DATA_IN1int[2] <= sdi;  statein <= 4'h4;  end
          3'b111 : begin DATA_IN1int[3] <= sdi;  statein <= 4'h4;  end
        endcase
      end
4'h4 : begin
        case (NUMBITS)
          3'b000 : begin DATA_IN1int[3] <= sdi;  statein <= 4'h5;  end
          3'b001 : begin statein <= 4'h8;  end
          3'b010 : begin statein <= 4'h8;  end
          3'b011 : begin statein <= 4'h8;  end
          3'b100 : begin statein <= 4'h8;  end
          3'b101 : begin DATA_IN1int[0] <= sdi;  statein <= 4'h8;  end
          3'b110 : begin DATA_IN1int[1] <= sdi;  statein <= 4'h5;  end
          3'b111 : begin DATA_IN1int[2] <= sdi;  statein <= 4'h5;  end
        endcase
      end
4'h5 : begin
        case (NUMBITS)
          3'b000 : begin DATA_IN1int[2] <= sdi;  statein <= 4'h6;  end
          3'b001 : begin statein <= 4'h8;  end
          3'b010 : begin statein <= 4'h8;  end
          3'b011 : begin statein <= 4'h8;  end
          3'b100 : begin statein <= 4'h8;  end
          3'b101 : begin statein <= 4'h8;  end
          3'b110 : begin DATA_IN1int[0] <= sdi;  statein <= 4'h8;  end
          3'b111 : begin DATA_IN1int[1] <= sdi;  statein <= 4'h6;  end
        endcase
      end
4'h6 : begin
        case (NUMBITS)
          3'b000 : begin DATA_IN1int[1] <= sdi;  statein <= 4'h7;  end
          3'b001 : begin statein <= 4'h8;  end
          3'b010 : begin statein <= 4'h8;  end
          3'b011 : begin statein <= 4'h8;  end
          3'b100 : begin statein <= 4'h8;  end
          3'b101 : begin statein <= 4'h8;  end
          3'b110 : begin statein <= 4'h8;  end
          3'b111 : begin DATA_IN1int[0] <= sdi;  statein <= 4'h8;  end
        endcase
      end
4'h7 : begin
        case (NUMBITS)
          3'b000 : begin DATA_IN1int[0] <= sdi;  statein <= 4'h8;  end
          3'b001 : begin statein <= 4'h8;  end
          3'b010 : begin statein <= 4'h8;  end
          3'b011 : begin statein <= 4'h8;  end
          3'b100 : begin statein <= 4'h8;  end
          3'b101 : begin statein <= 4'h8;  end
          3'b110 : begin statein <= 4'h8;  end
          3'b111 : begin statein <= 4'h8;  end
        endcase
      end
4'h8 : begin statein <= 4'h9; end
4'h9 : begin statein <= 4'h0; end
```

```
   default: statein <= 4'h0;
  endcase


endmodule

module edge_detect(CLK, sig, dirs, edges);
input CLK, sig, dirs;
output edges;

reg sn;
wire pe, ne;

always @(posedge CLK)
  sn <= ~sig;

assign pe = sig & sn;
assign ne = ~(sig | sn);

assign edges = (dirs) ? pe : ne;

endmodule


module tb;
reg CLK, RESETB;
wire MISO;
reg [3:0] ADDR, ADDR_RD2;
reg [7:0] DATA_WR;
wire [7:0] DATA_RD1, DATA_RD2;
reg WE, EN1, EN2, EN1x;
wire SSB, SCLK, MOSI;
reg MMEN, CLKPOL;


wire sclk_nex, sclk_pex;
reg MISOx, MOSIx, SSBx;

//ASPI x(CLK, RESETB, START, MODE, DATASIZE, BRG, DATA_WR, MISO, SSB, SCLK, MOSI, DATA_IN,
NEXT_DATA_WR);
ASPI x(CLK, RESETB, ADDR, ADDR_RD2, DATA_WR, DATA_RD1, DATA_RD2, WE, EN1, EN2, MISO,
SSB, SCLK, MOSI);
initial
begin
  CLK = 0; RESETB = 1; WE = 0; EN1 = 0; ADDR = 4'h7; DATA_WR = 0;  ADDR_RD2 = 4'h7; MISOx = 0;
  MMEN = 1;   CLKPOL = 1; EN1x = 0; MOSIx = 1;  SSBx = 1;
  #15  RESETB = 0;
  #20 RESETB = 1;
  #21  WE = 1; ADDR = 4'h6; DATA_WR = 0;
  #21  WE = 0;
  #21  WE = 1; ADDR = 4'h7; DATA_WR = 2;
  #21  WE = 0;
  #21  WE = 1; ADDR = 4'h3; DATA_WR = 8'b000_000_01; //{DIAG,NUMBITS,SSIO,SSV}
  #30 WE = 0;
  #21  WE = 1; ADDR = 4'h1; DATA_WR = {3'b001, 1'b0, CLKPOL, 3'b011};  //{IRQE, STR, BIRQ, PHASE,
CLKPOL, WOR, MMEN, SPIEN }
```

```
  #100 WE = 0;
  #21  WE = 1; ADDR = 4'h0; DATA_WR = 8'h55; //8'b00100011;
  #200 WE = 0;
  #821  WE = 1; ADDR = 4'h0; DATA_WR = 8'hAA; //8'b00100011;
  #50 WE = 0;
  #600 ;
  #250  WE = 1; ADDR = 4'h1; DATA_WR = {3'b001, 1'b0, CLKPOL, 3'b001}; //{IRQE, STR, BIRQ, PHASE,
CLKPOL, WOR, MMEN, SPIEN }
       MMEN = 0;
  #21   WE = 0;
  #21  WE = 1; ADDR = 4'h0; DATA_WR = 8'h55; //8'b00100011;
  #50  WE = 0;

  #21  EN1x = 1; ADDR = 4'h0; DATA_WR = 8'b00100011;
  #500 EN1x = 0;
 // #21  WE = 1; ADDR = 4'h1; DATA_WR = 8'b01000000;
  //#21  WE = 0;

  #2000 $finish;
end

always #10 CLK = ~CLK;

always @(posedge CLK)
begin
  EN2 <= tb.x.NEXT_DATA_WR;
  EN1 <= EN2 | EN1x;
end

reg clks;
initial clks = 0;
always #31 clks = ~clks;

reg sclkgen;
wire clks_pe;

//assign MISO_INT = MMEN ? MISO : MOSI_INT; //1'bz;
//assign MISO = MMEN ? 1'bz : MOSI_INT;
assign MISO = MMEN ? MISOx : 1'bz;

//assign MOSI = MMEN ? MOSI_INT : 1'bz;
assign MOSI = MMEN ? 1'bz : MOSIx;

assign SSB = (MMEN | tb.x.SSIO) ? 1'bz : SSBx;
assign SCLK = MMEN ? 1'bz : sclkgen;

edge_detect edgesclknex(CLK, SCLK, 1'b0, sclk_nex);
edge_detect edgesclkpex(CLK, SCLK, 1'b1, sclk_pex);
edge_detect edgesclks(CLK, clks, 1'b1, clks_pe);

reg [3:0] statex;
initial statex = 0;
always @(posedge CLK)
  if (!RESETB) begin statex <= 0; MISOx = 0; end
  else   if (sclk_nex)
```

```
  case (statex)
    0 : begin MISOx = 1;  statex <= 1; end
    1 : begin MISOx = 0;  statex <= 2; end
    2 : begin MISOx = 1;  statex <= 3; end
    3 : begin MISOx = 0;  statex <= 4; end
    4 : begin MISOx = 1;  statex <= 5; end
    5 : begin MISOx = 0;  statex <= 6; end
    6 : begin MISOx = 0;  statex <= 7; end
    7 : begin MISOx = 0;  statex <= 8; end
    8 : begin MISOx = 0;  statex <= 0; end
  endcase

reg [4:0] states;
initial states = 0;
always @(posedge CLK)
  if (!RESETB) begin states <= 0; sclkgen =  CLKPOL; SSBx <= 1; end
  else   if (clks_pe & !MMEN)
  case (states)
    0 : begin sclkgen = CLKPOL;  states <= 1; SSBx <= 1; end
    1 : begin sclkgen = CLKPOL;  states <= 2; SSBx <= 0; end
    2 : begin sclkgen = !CLKPOL;  states <= 3;  end
    3 : begin sclkgen = CLKPOL;  states <= 4; end
    4 : begin sclkgen = !CLKPOL;  states <= 5; end
    5 : begin sclkgen = CLKPOL;  states <= 6; end
    6 : begin sclkgen = !CLKPOL;  states <= 7; end
    7 : begin sclkgen = CLKPOL;  states <= 8; end
    8 : begin sclkgen = !CLKPOL;  states <= 9; end
    9 : begin sclkgen = CLKPOL;  states <= 10; end
    10 : begin sclkgen = !CLKPOL;  states <= 11; end
    11 : begin sclkgen = CLKPOL;  states <= 12; end
    12 : begin sclkgen = !CLKPOL;  states <= 13; end
    13 : begin sclkgen = CLKPOL;  states <= 14; end
    14 : begin sclkgen = !CLKPOL;  states <= 15; end
    15 : begin sclkgen = CLKPOL;  states <= 16; end
    16 : begin sclkgen = !CLKPOL;  states <= 17; end
    17 : begin sclkgen = CLKPOL;  states <= 18;   end
    18 : begin sclkgen = CLKPOL;  states <= 18; SSBx <= 1;  end

  endcase


endmodule
```