```
////////////////////////////////////////////////////////////////////
////                                                            ////
////  WISHBONE rev.B2 compliant I2C Master bit-controller       ////
////                                                            ////
////                                                            ////
////  Author: Richard Herveille                                 ////
////          richard@asics.ws                                  ////
////          www.asics.ws                                      ////
////                                                            ////
////  Downloaded from: http://www.opencores.org/projects/i2c/   ////
////                                                            ////
////////////////////////////////////////////////////////////////////
////                                                            ////
//// Copyright (C) 2001 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: i2c_master_bit_ctrl.v,v 1.14 2009-01-20 10:25:29 rherveille Exp $
//
//  $Date: 2009-01-20 10:25:29 $
//  $Revision: 1.14 $
//  $Author: rherveille $
//  $Locker:  $
//  $State: Exp $
//
// Change History:
//             $Log: $
//             Revision 1.14  2009/01/20 10:25:29  rherveille
//             Added clock synchronization logic
//             Fixed slave_wait signal
//
//             Revision 1.13  2009/01/19 20:29:26  rherveille
//             Fixed synopsys miss spell (synopsis)
//             Fixed cr[0] register width
//             Fixed ! usage instead of ~
//             Fixed bit controller parameter width to 18bits
//
//             Revision 1.12  2006/09/04 09:08:13  rherveille
//             fixed short scl high pulse after clock stretch
//             fixed slave model not returning correct '(n)ack' signal
//
//             Revision 1.11  2004/05/07 11:02:26  rherveille
//             Fixed a bug where the core would signal an arbitration lost (AL bit set), when another master
controls the bus and the other master generates a STOP bit.
//
//             Revision 1.10  2003/08/09 07:01:33  rherveille
//             Fixed a bug in the Arbitration Lost generation caused by delay on the (external) sda line.
//             Fixed a potential bug in the byte controller's host-acknowledge generation.
//
//             Revision 1.9  2003/03/10 14:26:37  rherveille
//             Fixed cmd_ack generation item (no bug).
//
//             Revision 1.8  2003/02/05 00:06:10  rherveille
//             Fixed a bug where the core would trigger an erroneous 'arbitration lost' interrupt after being
reset, when the reset pulse width < 3 clk cycles.
//
//             Revision 1.7  2002/12/26 16:05:12  rherveille
//             Small code simplifications
```

```
//
//              Revision 1.6  2002/12/26 15:02:32  rherveille
//              Core is now a Multimaster I2C controller
//
//              Revision 1.5  2002/11/30 22:24:40  rherveille
//              Cleaned up code
//
//              Revision 1.4  2002/10/30 18:10:07  rherveille
//              Fixed some reported minor start/stop generation timing issuess.
//
//              Revision 1.3  2002/06/15 07:37:03  rherveille
//              Fixed a small timing bug in the bit controller.\nAdded verilog simulation environment.
//
//              Revision 1.2  2001/11/05 11:59:25  rherveille
//              Fixed wb_ack_o generation bug.
//              Fixed bug in the byte_controller statemachine.
//              Added headers.
//


//
////////////////////////////////////
// Bit controller section
////////////////////////////////////
//
// Translate simple commands into SCL/SDA transitions
// Each command has 5 states, A/B/C/D/idle
//
// start:   SCL ~~~~~~~~~~|____
//  SDA ~~~~~~~~|______
//       x | A | B | C | D | i
//
// repstart SCL ____/~~~~|___
//  SDA __/~~~|______
//       x | A | B | C | D | i
//
// stop SCL ____/~~~~~~~~
//  SDA ==|____/~~~~~
//       x | A | B | C | D | i
//
//- write   SCL ____/~~~~|____
//  SDA ==X=========X=
//       x | A | B | C | D | i
//
//- read    SCL ____/~~~~|____
```

```
//  SDA XXXX=====XXXX
//      x | A | B | C | D | i
//

// Timing:    Normal mode     Fast mode
////////////////////////////////////////////////////////////////////////
// Fscl       100KHz          400KHz
// Th_scl     4.0us           0.6us   High period of SCL
// Tl_scl     4.7us           1.3us   Low period of SCL
// Tsu:sta    4.7us           0.6us   setup time for a repeated start condition
// Tsu:sto    4.0us           0.6us   setup time for a stop conditon
// Tbuf       4.7us           1.3us   Bus free time between a stop and start condition
//

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

`include "i2c_master_defines.v"

module i2c_master_bit_ctrl (
    input           clk,     // system clock
    input           rst,     // synchronous active high reset
    input           nReset,  // asynchronous active low reset
    input           ena,     // core enable signal

    input     [15:0] clk_cnt,  // clock prescale value

    input     [ 3:0] cmd,     // command (from byte controller)
    output reg       cmd_ack,  // command complete acknowledge
    output reg       busy,     // i2c bus busy
    output reg       al,       // i2c bus arbitration lost

    input           din,
    output reg       dout,

    input           scl_i,    // i2c clock line input
    output          scl_o,    // i2c clock line output
    output reg       scl_oen,  // i2c clock line output enable (active low)
    input           sda_i,    // i2c data line input
    output          sda_o,    // i2c data line output
    output reg       sda_oen   // i2c data line output enable (active low)
);
```

```
//
// variable declarations
//

reg [ 1:0] cSCL, cSDA;     // capture SCL and SDA
reg [ 2:0] fSCL, fSDA;     // SCL and SDA filter inputs
reg        sSCL, sSDA;     // filtered and synchronized SCL and SDA inputs
reg        dSCL, dSDA;     // delayed versions of sSCL and sSDA
reg        dscl_oen;       // delayed scl_oen
reg        sda_chk;        // check SDA output (Multi-master arbitration)
reg        clk_en;         // clock generation signals
reg        slave_wait;     // slave inserts wait states
reg [15:0] cnt;            // clock divider counter (synthesis)
reg [13:0] filter_cnt;     // clock divider for filter


// state machine variable
reg [17:0] c_state; // synopsys enum_state

//
// module body
//

// whenever the slave is not ready it can delay the cycle by pulling SCL low
// delay scl_oen
always @(posedge clk)
  dscl_oen <= #1 scl_oen;

// slave_wait is asserted when master wants to drive SCL high, but the slave pulls it low
// slave_wait remains asserted until the slave releases SCL
always @(posedge clk or negedge nReset)
  if (!nReset) slave_wait <= 1'b0;
  else         slave_wait <= (scl_oen & ~dscl_oen & ~sSCL) | (slave_wait & ~sSCL);

// master drives SCL high, but another master pulls it low
// master start counting down its low cycle now (clock synchronization)
wire scl_sync  = dSCL & ~sSCL & scl_oen;


// generate clk enable signal
always @(posedge clk or negedge nReset)
  if (~nReset)
  begin
```

```
      cnt    <= #1 16'h0;
      clk_en <= #1 1'b1;
    end
    else if (rst || ~|cnt || !ena || scl_sync)
    begin
      cnt    <= #1 clk_cnt;
      clk_en <= #1 1'b1;
    end
    else if (slave_wait)
    begin
      cnt    <= #1 cnt;
      clk_en <= #1 1'b0;
    end
    else
    begin
      cnt    <= #1 cnt - 16'h1;
      clk_en <= #1 1'b0;
    end


  // generate bus status controller

  // capture SDA and SCL
  // reduce metastability risk
  always @(posedge clk or negedge nReset)
    if (!nReset)
    begin
      cSCL <= #1 2'b00;
      cSDA <= #1 2'b00;
    end
    else if (rst)
    begin
      cSCL <= #1 2'b00;
      cSDA <= #1 2'b00;
    end
    else
    begin
      cSCL <= {cSCL[0],scl_i};
      cSDA <= {cSDA[0],sda_i};
    end


  // filter SCL and SDA signals; (attempt to) remove glitches
  always @(posedge clk or negedge nReset)
```

```
  if      (!nReset    ) filter_cnt <= 14'h0;
  else if (rst || !ena ) filter_cnt <= 14'h0;
  else if (~|filter_cnt) filter_cnt <= clk_cnt >> 2; //16x I2C bus frequency
  else                   filter_cnt <= filter_cnt -1;


always @(posedge clk or negedge nReset)
  if (!nReset)
  begin
     fSCL <= 3'b111;
     fSDA <= 3'b111;
  end
  else if (rst)
  begin
     fSCL <= 3'b111;
     fSDA <= 3'b111;
  end
  else if (~|filter_cnt)
  begin
     fSCL <= {fSCL[1:0],cSCL[1]};
     fSDA <= {fSDA[1:0],cSDA[1]};
  end


// generate filtered SCL and SDA signals
always @(posedge clk or negedge nReset)
  if (~nReset)
  begin
     sSCL <= #1 1'b1;
     sSDA <= #1 1'b1;

     dSCL <= #1 1'b1;
     dSDA <= #1 1'b1;
  end
  else if (rst)
  begin
     sSCL <= #1 1'b1;
     sSDA <= #1 1'b1;

     dSCL <= #1 1'b1;
     dSDA <= #1 1'b1;
  end
  else
  begin
```

```
    sSCL <= #1 &fSCL[2:1] | &fSCL[1:0] | (fSCL[2] & fSCL[0]);
    sSDA <= #1 &fSDA[2:1] | &fSDA[1:0] | (fSDA[2] & fSDA[0]);

    dSCL <= #1 sSCL;
    dSDA <= #1 sSDA;
  end

// detect start condition => detect falling edge on SDA while SCL is high
// detect stop condition => detect rising edge on SDA while SCL is high
reg sta_condition;
reg sto_condition;
always @(posedge clk or negedge nReset)
  if (~nReset)
  begin
    sta_condition <= #1 1'b0;
    sto_condition <= #1 1'b0;
  end
  else if (rst)
  begin
    sta_condition <= #1 1'b0;
    sto_condition <= #1 1'b0;
  end
  else
  begin
    sta_condition <= #1 ~sSDA &  dSDA & sSCL;
    sto_condition <= #1  sSDA & ~dSDA & sSCL;
  end


// generate i2c bus busy signal
always @(posedge clk or negedge nReset)
  if      (!nReset) busy <= #1 1'b0;
  else if (rst    ) busy <= #1 1'b0;
  else              busy <= #1 (sta_condition | busy) & ~sto_condition;


// generate arbitration lost signal
// aribitration lost when:
// 1) master drives SDA high, but the i2c bus is low
// 2) stop detected while not requested
reg cmd_stop;
always @(posedge clk or negedge nReset)
  if (~nReset)
    cmd_stop <= #1 1'b0;
```

```
  else if (rst)
     cmd_stop <= #1 1'b0;
  else if (clk_en)
     cmd_stop <= #1 cmd == `I2C_CMD_STOP;

always @(posedge clk or negedge nReset)
  if (~nReset)
     al <= #1 1'b0;
  else if (rst)
     al <= #1 1'b0;
  else
     al <= #1 (sda_chk & ~sSDA & sda_oen) | (|c_state & sto_condition & ~cmd_stop);


// generate dout signal (store SDA on rising edge of SCL)
always @(posedge clk)
  if (sSCL & ~dSCL) dout <= #1 sSDA;


// generate statemachine

// nxt_state decoder
parameter [17:0] idle    = 18'b0_0000_0000_0000_0000;
parameter [17:0] start_a = 18'b0_0000_0000_0000_0001;
parameter [17:0] start_b = 18'b0_0000_0000_0000_0010;
parameter [17:0] start_c = 18'b0_0000_0000_0000_0100;
parameter [17:0] start_d = 18'b0_0000_0000_0000_1000;
parameter [17:0] start_e = 18'b0_0000_0000_0001_0000;
parameter [17:0] stop_a  = 18'b0_0000_0000_0010_0000;
parameter [17:0] stop_b  = 18'b0_0000_0000_0100_0000;
parameter [17:0] stop_c  = 18'b0_0000_0000_1000_0000;
parameter [17:0] stop_d  = 18'b0_0000_0001_0000_0000;
parameter [17:0] rd_a    = 18'b0_0000_0010_0000_0000;
parameter [17:0] rd_b    = 18'b0_0000_0100_0000_0000;
parameter [17:0] rd_c    = 18'b0_0000_1000_0000_0000;
parameter [17:0] rd_d    = 18'b0_0001_0000_0000_0000;
parameter [17:0] wr_a    = 18'b0_0010_0000_0000_0000;
parameter [17:0] wr_b    = 18'b0_0100_0000_0000_0000;
parameter [17:0] wr_c    = 18'b0_1000_0000_0000_0000;
parameter [17:0] wr_d    = 18'b1_0000_0000_0000_0000;

always @(posedge clk or negedge nReset)
  if (!nReset)
  begin
```

```
    c_state <= #1 idle;
    cmd_ack <= #1 1'b0;
    scl_oen <= #1 1'b1;
    sda_oen <= #1 1'b1;
    sda_chk <= #1 1'b0;
end
else if (rst | al)
begin
    c_state <= #1 idle;
    cmd_ack <= #1 1'b0;
    scl_oen <= #1 1'b1;
    sda_oen <= #1 1'b1;
    sda_chk <= #1 1'b0;
end
else
begin
    cmd_ack  <= #1 1'b0; // default no command acknowledge + assert cmd_ack only 1clk cycle

    if (clk_en)
        case (c_state) // synopsys full_case parallel_case
            // idle state
            idle:
            begin
                case (cmd) // synopsys full_case parallel_case
                    `I2C_CMD_START: c_state <= #1 start_a;
                    `I2C_CMD_STOP:  c_state <= #1 stop_a;
                    `I2C_CMD_WRITE: c_state <= #1 wr_a;
                    `I2C_CMD_READ:  c_state <= #1 rd_a;
                    default:        c_state <= #1 idle;
                endcase

                scl_oen <= #1 scl_oen; // keep SCL in same state
                sda_oen <= #1 sda_oen; // keep SDA in same state
                sda_chk <= #1 1'b0;    // don't check SDA output
            end

            // start
            start_a:
            begin
                c_state <= #1 start_b;
                scl_oen <= #1 scl_oen; // keep SCL in same state
                sda_oen <= #1 1'b1;    // set SDA high
                sda_chk <= #1 1'b0;    // don't check SDA output
            end
```

```
start_b:
begin
    c_state <= #1 start_c;
    scl_oen <= #1 1'b1; // set SCL high
    sda_oen <= #1 1'b1; // keep SDA high
    sda_chk <= #1 1'b0; // don't check SDA output
end

start_c:
begin
    c_state <= #1 start_d;
    scl_oen <= #1 1'b1; // keep SCL high
    sda_oen <= #1 1'b0; // set SDA low
    sda_chk <= #1 1'b0; // don't check SDA output
end

start_d:
begin
    c_state <= #1 start_e;
    scl_oen <= #1 1'b1; // keep SCL high
    sda_oen <= #1 1'b0; // keep SDA low
    sda_chk <= #1 1'b0; // don't check SDA output
end

start_e:
begin
    c_state <= #1 idle;
    cmd_ack <= #1 1'b1;
    scl_oen <= #1 1'b0; // set SCL low
    sda_oen <= #1 1'b0; // keep SDA low
    sda_chk <= #1 1'b0; // don't check SDA output
end

// stop
stop_a:
begin
    c_state <= #1 stop_b;
    scl_oen <= #1 1'b0; // keep SCL low
    sda_oen <= #1 1'b0; // set SDA low
    sda_chk <= #1 1'b0; // don't check SDA output
end

stop_b:
```

```
begin
    c_state <= #1 stop_c;
    scl_oen <= #1 1'b1; // set SCL high
    sda_oen <= #1 1'b0; // keep SDA low
    sda_chk <= #1 1'b0; // don't check SDA output
end

stop_c:
begin
    c_state <= #1 stop_d;
    scl_oen <= #1 1'b1; // keep SCL high
    sda_oen <= #1 1'b0; // keep SDA low
    sda_chk <= #1 1'b0; // don't check SDA output
end

stop_d:
begin
    c_state <= #1 idle;
    cmd_ack <= #1 1'b1;
    scl_oen <= #1 1'b1; // keep SCL high
    sda_oen <= #1 1'b1; // set SDA high
    sda_chk <= #1 1'b0; // don't check SDA output
end

// read
rd_a:
begin
    c_state <= #1 rd_b;
    scl_oen <= #1 1'b0; // keep SCL low
    sda_oen <= #1 1'b1; // tri-state SDA
    sda_chk <= #1 1'b0; // don't check SDA output
end

rd_b:
begin
    c_state <= #1 rd_c;
    scl_oen <= #1 1'b1; // set SCL high
    sda_oen <= #1 1'b1; // keep SDA tri-stated
    sda_chk <= #1 1'b0; // don't check SDA output
end

rd_c:
begin
    c_state <= #1 rd_d;
```

```
    scl_oen <= #1 1'b1; // keep SCL high
    sda_oen <= #1 1'b1; // keep SDA tri-stated
    sda_chk <= #1 1'b0; // don't check SDA output
end

rd_d:
begin
    c_state <= #1 idle;
    cmd_ack <= #1 1'b1;
    scl_oen <= #1 1'b0; // set SCL low
    sda_oen <= #1 1'b1; // keep SDA tri-stated
    sda_chk <= #1 1'b0; // don't check SDA output
end

// write
wr_a:
begin
    c_state <= #1 wr_b;
    scl_oen <= #1 1'b0; // keep SCL low
    sda_oen <= #1 din;  // set SDA
    sda_chk <= #1 1'b0; // don't check SDA output (SCL low)
end

wr_b:
begin
    c_state <= #1 wr_c;
    scl_oen <= #1 1'b1; // set SCL high
    sda_oen <= #1 din;  // keep SDA
    sda_chk <= #1 1'b0; // don't check SDA output yet
                        // allow some time for SDA and SCL to settle
end

wr_c:
begin
    c_state <= #1 wr_d;
    scl_oen <= #1 1'b1; // keep SCL high
    sda_oen <= #1 din;
    sda_chk <= #1 1'b1; // check SDA output
end

wr_d:
begin
    c_state <= #1 idle;
    cmd_ack <= #1 1'b1;
```

```
                scl_oen <= #1 1'b0; // set SCL low
                sda_oen <= #1 din;
                sda_chk <= #1 1'b0; // don't check SDA output (SCL low)
            end

        endcase
    end


  // assign scl and sda output (always gnd)
  assign scl_o = 1'b0;
  assign sda_o = 1'b0;

endmodule
```