

```
////////////////////////////////////  
////  
//// WISHBONE revB.2 compliant I2C Master controller Top-level ////  
////  
////  
//// 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_top.v,v 1.10 2003/09/01 10:34:38 rherveille Exp $  
//  
// $Date: 2003/09/01 10:34:38 $  
// $Revision: 1.10 $  
// $Author: rherveille $  
// $Locker: $  
// $State: Exp $  
//  
// Change History:  
//      $Log: i2c_master_top.v,v $  
//      Revision 1.10  2003/09/01 10:34:38  rherveille  
//      Fix a blocking vs. non-blocking error in the wb_dat output mux.  
//  
//      Revision 1.9   2003/01/09 16:44:45  rherveille  
//      Fixed a bug in the Command Register declaration.  
//  
//      Revision 1.8   2002/12/26 16:05:12  rherveille  
//      Small code simplifications  
//  
//      Revision 1.7   2002/12/26 15:02:32  rherveille  
//      Core is now a Multimaster I2C controller  
//  
//      Revision 1.6   2002/11/30 22:24:40  rherveille  
//      Cleaned up code
```

```
//
//          Revision 1.5  2001/11/10 10:52:55  rherveille
//          Changed PRER reset value from 0x0000 to 0xffff, conform specs.
//

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

`include "i2c_master_defines.v"

module i2c_master_top(
    wb_clk_i, wb_rst_i, arst_i, wb_adr_i, wb_dat_i, wb_dat_o,
    wb_we_i, wb_stb_i, wb_cyc_i, wb_ack_o, wb_inta_o,
    scl_pad_i, scl_pad_o, scl_padoen_o, sda_pad_i, sda_pad_o, sda_padoen_o );

    // parameters
    parameter ARST_LVL = 1'b1; // asynchronous reset level

    //
    // inputs & outputs
    //

    // wishbone signals
    input      wb_clk_i;      // master clock input
    input      wb_rst_i;      // synchronous active high reset
    input      arst_i;        // asynchronous reset
    input  [2:0] wb_adr_i;     // lower address bits
    input  [7:0] wb_dat_i;    // databus input
    output [7:0] wb_dat_o;    // databus output
    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_inta_o;     // interrupt request signal output

    reg [7:0] wb_dat_o;
    reg wb_ack_o;
    reg wb_inta_o;

    // I2C signals
    // i2c clock line
    input  scl_pad_i;         // SCL-line input
    output scl_pad_o;         // SCL-line output (always 1'b0)
    output scl_padoen_o;     // SCL-line output enable (active low)

    // i2c data line
    input  sda_pad_i;         // SDA-line input
    output sda_pad_o;         // SDA-line output (always 1'b0)
    output sda_padoen_o;     // SDA-line output enable (active low)

    //
    // variable declarations
    //

    // registers
    reg [15:0] prer; // clock prescale register
    reg [ 7:0] ctr;  // control register
    reg [ 7:0] txr;  // transmit register
    wire [ 7:0] rxr; // receive register
    reg [ 7:0] cr;   // command register
    wire [ 7:0] sr;  // status register
```

```
// done signal: command completed, clear command register
wire done;

// core enable signal
wire core_en;
wire ien;

// status register signals
wire irxack;
reg rxack; // received acknowledge from slave
reg tip; // transfer in progress
reg irq_flag; // interrupt pending flag
wire i2c_busy; // bus busy (start signal detected)
wire i2c_al; // i2c bus arbitration lost
reg al; // status register arbitration lost bit

//
// module body
//

// generate internal reset
wire rst_i = arst_i ^ ARST_LVL;

// generate wishbone signals
wire wb_wacc = wb_cyc_i & wb_stb_i & wb_we_i;

// generate acknowledge output signal
always @(posedge wb_clk_i)
    wb_ack_o <= #1 wb_cyc_i & wb_stb_i & ~wb_ack_o; // because timing is always
honored

// assign DAT_O
always @(posedge wb_clk_i)
begin
    case (wb_adr_i) // synopsis full_case parallel_case
        3'b000: wb_dat_o <= #1 prer[7:0];
        3'b001: wb_dat_o <= #1 prer[15:8];
        3'b010: wb_dat_o <= #1 ctr;
        3'b011: wb_dat_o <= #1 rxr; // write is transmit register (txr)
        3'b100: wb_dat_o <= #1 sr; // write is command register (cr)
        3'b101: wb_dat_o <= #1 txr;
        3'b110: wb_dat_o <= #1 cr;
        3'b111: wb_dat_o <= #1 0; // reserved
    endcase
end

// generate registers
always @(posedge wb_clk_i or negedge rst_i)
    if (!rst_i)
        begin
            prer <= #1 16'hffff;
            ctr <= #1 8'h0;
            txr <= #1 8'h0;
        end
    else if (wb_rst_i)
        begin
            prer <= #1 16'hffff;
            ctr <= #1 8'h0;
            txr <= #1 8'h0;
        end
    else
        if (wb_wacc)
```

```
        case (wb_adr_i) // synopsis full_case parallel_case
            3'b000 : prer [ 7:0] <= #1 wb_dat_i;
            3'b001 : prer [15:8] <= #1 wb_dat_i;
            3'b010 : ctr      <= #1 wb_dat_i;
            3'b011 : txr      <= #1 wb_dat_i;
        endcase

// generate command register (special case)
always @(posedge wb_clk_i or negedge rst_i)
    if (~rst_i)
        cr <= #1 8'h0;
    else if (wb_rst_i)
        cr <= #1 8'h0;
    else if (wb_wacc)
        begin
            if (core_en & (wb_adr_i == 3'b100) )
                cr <= #1 wb_dat_i;
            end
        else
            begin
                if (done | i2c_al)
                    cr[7:4] <= #1 4'h0;           // clear command bits when done
                                                    // or when arbitration lost
                cr[2:1] <= #1 2'b0;           // reserved bits
                cr[0]   <= #1 2'b0;           // clear IRQ_ACK bit
            end

// decode command register
wire sta = cr[7];
wire sto = cr[6];
wire rd  = cr[5];
wire wr  = cr[4];
wire ack = cr[3];
wire iack = cr[0];

// decode control register
assign core_en = ctr[7];
assign ien = ctr[6];

// hookup byte controller block
i2c_master_byte_ctrl byte_controller (
    .clk      ( wb_clk_i      ),
    .rst      ( wb_rst_i      ),
    .nReset   ( rst_i         ),
    .ena      ( core_en       ),
    .clk_cnt  ( prer          ),
    .start    ( sta           ),
    .stop     ( sto           ),
    .read     ( rd            ),
    .write    ( wr            ),
    .ack_in   ( ack           ),
    .din      ( txr           ),
    .cmd_ack  ( done          ),
    .ack_out  ( irxack        ),
    .dout     ( rxr           ),
    .i2c_busy ( i2c_busy      ),
    .i2c_al   ( i2c_al        ),
    .scl_i    ( scl_pad_i     ),
    .scl_o    ( scl_pad_o     ),
    .scl_oen  ( scl_padoen_o  ),
    .sda_i    ( sda_pad_i     ),
    .sda_o    ( sda_pad_o     ),
```

```
.sda_oen ( sda_padoen_o )
);

// status register block + interrupt request signal
always @(posedge wb_clk_i or negedge rst_i)
  if (!rst_i)
    begin
      al      <= #1 1'b0;
      rxack   <= #1 1'b0;
      tip     <= #1 1'b0;
      irq_flag <= #1 1'b0;
    end
  else if (wb_rst_i)
    begin
      al      <= #1 1'b0;
      rxack   <= #1 1'b0;
      tip     <= #1 1'b0;
      irq_flag <= #1 1'b0;
    end
  else
    begin
      al      <= #1 i2c_al | (al & ~sta);
      rxack   <= #1 irxack;
      tip     <= #1 (rd | wr);
      irq_flag <= #1 (done | i2c_al | irq_flag) & ~iack; // interrupt request
flag is always generated
    end

// generate interrupt request signals
always @(posedge wb_clk_i or negedge rst_i)
  if (!rst_i)
    wb_inta_o <= #1 1'b0;
  else if (wb_rst_i)
    wb_inta_o <= #1 1'b0;
  else
    wb_inta_o <= #1 irq_flag && ien; // interrupt signal is only generated when IEN
(interrupt enable bit is set)

// assign status register bits
assign sr[7]   = rxack;
assign sr[6]   = i2c_busy;
assign sr[5]   = al;
assign sr[4:2] = 3'h0; // reserved
assign sr[1]   = tip;
assign sr[0]   = irq_flag;

endmodule

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

`include "i2c_master_defines.v"

module i2c_master_byte_ctrl (
  clk, rst, nReset, ena, clk_cnt, start, stop, read, write, ack_in, din,
  cmd_ack, ack_out, dout, i2c_busy, i2c_al, scl_i, scl_o, scl_oen, sda_i, sda_o,
  sda_oen );

  //
```

```
// inputs & outputs
//
input clk;      // master clock
input rst;     // synchronous active high reset
input nReset;  // asynchronous active low reset
input ena;     // core enable signal

input [15:0] clk_cnt; // 4x SCL

// control inputs
input start;
input stop;
input read;
input write;
input ack_in;
input [7:0] din;

// status outputs
output cmd_ack;
reg cmd_ack;
output ack_out;
reg ack_out;
output i2c_busy;
output i2c_al;
output [7:0] dout;

// I2C signals
input scl_i;
output scl_o;
output scl_oen;
input sda_i;
output sda_o;
output sda_oen;

//
// Variable declarations
//

// statemachine
parameter [4:0] ST_IDLE = 5'b0_0000;
parameter [4:0] ST_START = 5'b0_0001;
parameter [4:0] ST_READ = 5'b0_0010;
parameter [4:0] ST_WRITE = 5'b0_0100;
parameter [4:0] ST_ACK = 5'b0_1000;
parameter [4:0] ST_STOP = 5'b1_0000;

// signals for bit_controller
reg [3:0] core_cmd;
reg core_txd;
wire core_ack, core_rxd;

// signals for shift register
reg [7:0] sr; //8bit shift register
reg shift, ld;

// signals for state machine
wire go;
reg [2:0] dcnt;
wire cnt_done;

//
// Module body
```

```
//
// hookup bit_controller
i2c_master_bit_ctrl bit_controller (
    .clk      ( clk      ),
    .rst      ( rst      ),
    .nReset   ( nReset   ),
    .ena      ( ena      ),
    .clk_cnt  ( clk_cnt  ),
    .cmd      ( core_cmd ),
    .cmd_ack  ( core_ack ),
    .busy     ( i2c_busy ),
    .al       ( i2c_al   ),
    .din      ( core_txd ),
    .dout     ( core_rxd ),
    .scl_i    ( scl_i    ),
    .scl_o    ( scl_o    ),
    .scl_oen  ( scl_oen  ),
    .sda_i    ( sda_i    ),
    .sda_o    ( sda_o    ),
    .sda_oen  ( sda_oen  )
);

// generate go-signal
assign go = (read | write | stop) & ~cmd_ack;

// assign dout output to shift-register
assign dout = sr;

// generate shift register
always @(posedge clk or negedge nReset)
    if (!nReset)
        sr <= #1 8'h0;
    else if (rst)
        sr <= #1 8'h0;
    else if (ld)
        sr <= #1 din;
    else if (shift)
        sr <= #1 {sr[6:0], core_rxd};

// generate counter
always @(posedge clk or negedge nReset)
    if (!nReset)
        dcnt <= #1 3'h0;
    else if (rst)
        dcnt <= #1 3'h0;
    else if (ld)
        dcnt <= #1 3'h7;
    else if (shift)
        dcnt <= #1 dcnt - 3'h1;

assign cnt_done = ~(|dcnt);

//
// state machine
//
reg [4:0] c_state; // synopsis enum_state

always @(posedge clk or negedge nReset)
    if (!nReset)
        begin
            core_cmd <= #1 `I2C_CMD_NOP;
            core_txd <= #1 1'b0;
```

```
        shift    <= #1 1'b0;
        ld      <= #1 1'b0;
        cmd_ack <= #1 1'b0;
        c_state <= #1 ST_IDLE;
        ack_out <= #1 1'b0;
    end
else if (rst | i2c_al)
begin
    core_cmd <= #1 `I2C_CMD_NOP;
    core_txd <= #1 1'b0;
    shift    <= #1 1'b0;
    ld      <= #1 1'b0;
    cmd_ack <= #1 1'b0;
    c_state <= #1 ST_IDLE;
    ack_out <= #1 1'b0;
end
else
begin
    // initially reset all signals
    core_txd <= #1 sr[7];
    shift    <= #1 1'b0;
    ld      <= #1 1'b0;
    cmd_ack <= #1 1'b0;

    case (c_state) // synopsis full_case parallel_case
    ST_IDLE:
        if (go)
            begin
                if (start)
                    begin
                        c_state <= #1 ST_START;
                        core_cmd <= #1 `I2C_CMD_START;
                    end
                else if (read)
                    begin
                        c_state <= #1 ST_READ;
                        core_cmd <= #1 `I2C_CMD_READ;
                    end
                else if (write)
                    begin
                        c_state <= #1 ST_WRITE;
                        core_cmd <= #1 `I2C_CMD_WRITE;
                    end
                else // stop
                    begin
                        c_state <= #1 ST_STOP;
                        core_cmd <= #1 `I2C_CMD_STOP;

                        // generate command acknowledge signal
                        cmd_ack <= #1 1'b1;
                    end

                ld <= #1 1'b1;
            end

    ST_START:
        if (core_ack)
            begin
                if (read)
                    begin
                        c_state <= #1 ST_READ;
                        core_cmd <= #1 `I2C_CMD_READ;
                    end
            end
        end
    end
end
```

```
        else
            begin
                c_state <= #1 ST_WRITE;
                core_cmd <= #1 `I2C_CMD_WRITE;
            end

            ld <= #1 1'b1;
        end

ST_WRITE:
    if (core_ack)
        if (cnt_done)
            begin
                c_state <= #1 ST_ACK;
                core_cmd <= #1 `I2C_CMD_READ;
            end
        else
            begin
                c_state <= #1 ST_WRITE;           // stay in same state
                core_cmd <= #1 `I2C_CMD_WRITE; // write next bit
                shift <= #1 1'b1;
            end
        end

ST_READ:
    if (core_ack)
        begin
            if (cnt_done)
                begin
                    c_state <= #1 ST_ACK;
                    core_cmd <= #1 `I2C_CMD_WRITE;
                end
            else
                begin
                    c_state <= #1 ST_READ;           // stay in same state
                    core_cmd <= #1 `I2C_CMD_READ; // read next bit
                end

                shift <= #1 1'b1;
                core_txd <= #1 ack_in;
            end
        end

ST_ACK:
    if (core_ack)
        begin
            if (stop)
                begin
                    c_state <= #1 ST_STOP;
                    core_cmd <= #1 `I2C_CMD_STOP;
                end
            else
                begin
                    c_state <= #1 ST_IDLE;
                    core_cmd <= #1 `I2C_CMD_NOP;

                    // generate command acknowledge signal
                    cmd_ack <= #1 1'b1;
                end

                // assign ack_out output to bit_controller_rxd (contains last
received bit)
                ack_out <= #1 core_rxd;

                // generate command acknowledge signal
```

```

//          cmd_ack  <= #1 1'b1;
          core_txd <= #1 1'b1;
        end
      else
        core_txd <= #1 ack_in;

      ST_STOP:
        if (core_ack)
          begin
            c_state <= #1 ST_IDLE;
            core_cmd <= #1 `I2C_CMD_NOP;

            // generate command acknowledge signal
            cmd_ack <= #1 1'b1;
          end
        endcase
      end
endmodule

//
////////////////////////////////////////////////////////////////////
// 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
////////////////////////////////////////////////////////////////////
// Fsc1         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(
    clk, rst, nReset,
    clk_cnt, ena, cmd, cmd_ack, busy, al, din, dout,
    scl_i, scl_o, scl_oen, sda_i, sda_o, sda_oen
);

    //
    // inputs & outputs
    //
    input clk;
    input rst;
    input nReset;
    input ena;           // core enable signal

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

    input [3:0] cmd;
    output      cmd_ack; // command complete acknowledge
    reg cmd_ack;
    output      busy;    // i2c bus busy
    reg busy;
    output      al;      // i2c bus arbitration lost
    reg al;

    input din;
    output dout;
    reg dout;

    // I2C lines
    input scl_i;          // i2c clock line input
    output scl_o;         // i2c clock line output
    output scl_oen;       // i2c clock line output enable (active low)
    reg scl_oen;
    input sda_i;          // i2c data line input
    output sda_o;         // i2c data line output
    output sda_oen;       // i2c data line output enable (active low)
    reg sda_oen;

    //
    // variable declarations
    //

    reg sSCL, sSDA;       // synchronized SCL and SDA inputs
    reg dscl_oen;         // delayed scl_oen
    reg sda_chk;          // check SDA output (Multi-master arbitration)
    reg clk_en;           // clock generation signals
    wire slave_wait;
    // reg [15:0] cnt = clk_cnt; // clock divider counter (simulation)
    reg [15:0] cnt;       // clock divider counter (synthesis)

    //
    // 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;

assign slave_wait = dscl_oen && !sSCL;

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

// generate bus status controller
reg dSCL, dSDA;
reg sta_condition;
reg sto_condition;

// synchronize SCL and SDA inputs
// reduce metastability risc
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 scl_i;
        sSDA <= #1 sda_i;

        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
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
    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
// aribration 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) | (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 [16:0] idle      = 17'b0_0000_0000_0000_0000;
parameter [16:0] start_a  = 17'b0_0000_0000_0000_0001;
parameter [16:0] start_b  = 17'b0_0000_0000_0000_0010;
parameter [16:0] start_c  = 17'b0_0000_0000_0000_0100;
parameter [16:0] start_d  = 17'b0_0000_0000_0000_1000;
parameter [16:0] start_e  = 17'b0_0000_0000_0001_0000;
parameter [16:0] stop_a   = 17'b0_0000_0000_0010_0000;
parameter [16:0] stop_b   = 17'b0_0000_0000_0100_0000;
parameter [16:0] stop_c   = 17'b0_0000_0000_1000_0000;
parameter [16:0] stop_d   = 17'b0_0000_0001_0000_0000;
parameter [16:0] rd_a     = 17'b0_0000_0010_0000_0000;
parameter [16:0] rd_b     = 17'b0_0000_0100_0000_0000;
parameter [16:0] rd_c     = 17'b0_0000_1000_0000_0000;
parameter [16:0] rd_d     = 17'b0_0001_0000_0000_0000;
parameter [16:0] wr_a     = 17'b0_0010_0000_0000_0000;
parameter [16:0] wr_b     = 17'b0_0100_0000_0000_0000;
parameter [16:0] wr_c     = 17'b0_1000_0000_0000_0000;
parameter [16:0] wr_d     = 17'b1_0000_0000_0000_0000;

reg [16:0] c_state; // synopsis enum_state

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 | a1)
    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
    end

  if (clk_en)
    case (c_state) // synopsis full_case parallel_case
      // idle state
      idle:
        begin
          case (cmd) // synopsis 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'b1; // check SDA output
    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
```