```
/*******************************************************************************
            RTL version of the Interface of the ALTRO chip
********************************************************************************

The ALTRO module instantiates a INTERFACE module that is divided in 4 sub-
blocks:

-INTCTRL    : The state machines controlling the interface
-INTDEC           : The instruction decoder
-BUSINT           : The Bus interface
-INTEXEC    : The instruction decoder

The code bellow is a hierarchical one.
The first three modules are a glitch filter (GLITCHF) and two synchronization
modules
needed for the interface sub-blocks.
The INTCTRL module has 2 state machines, Hamming protected (INTRDOH and
INTRECH).

*******************************************************************************/


//--------------------GLITCHF-------------------------//

module glitchf (x2, clk, rstb, x1);

input clk, x2, rstb;
output      x1;

wire  w0;
reg   r1, r2, r3;


assign w0 = (r1 & r2);
assign x1 = ~(r3 | ~w0);



always@(posedge clk or negedge rstb)

if(!rstb)
      begin
            r1 = 0;
            r2 = 0;
            r3 = 0;
      end
else
      begin
            r1 <= x2;
            r2 <= r1;
            r3 <= r2;
      end

endmodule
```

```
//---------------------SYNC221 and SYNC21-------------------------//

module sync221 (x2, clk, clk2, rstb, x1);

input clk, clk2, x2, rstb;
output      x1;

wire  w0, w1, w2, w3;
reg   r1, r2, r3, r4, x1;


assign w0 = (r1 & r2);
assign w1 = ~(r3 | ~w0);
assign w2 = (~x1 & (w1 | r4));
assign w3 = (~x1 & r4);


always@(posedge clk or negedge rstb)
if(!rstb)   x1 = 0;
else        x1 <= w3;




always@(posedge clk2 or negedge rstb)

if(!rstb)
      begin
            r1 = 0;
            r2 = 0;
            r3 = 0;
            r4 = 0;
      end
else
      begin
            r1 <= x2;
            r2 <= r1;
            r3 <= r2;
            r4 <= w2;

      end


endmodule



module sync21 (x2, clk, clk2, rstb, x1, busyb);

input clk, clk2, x2, rstb;
output      x1, busyb;

wire  w0, w1;
reg   r1, r2, r3, x1;

assign w0 = ((r3 | ~( r2 | ~r1)) & ~x1);
assign w1 = ( r3 & ~x1);
assign busyb = (~x1 & ~r3);


always@(posedge clk or negedge rstb)
if(!rstb)   x1 = 0;
else        x1 <= w1;



always@(posedge clk2 or negedge rstb)

if(!rstb)
      begin
            r1 = 0;
            r2 = 0;
            r3 = 0;
      end
else
      begin
            r1 <= x2;
            r2 <= r1;
            r3 <= w0;
      end

endmodule


//-----------------------------------------INTRECH------------------------------
--------------

module intrech
(clk2,rstb,cstb,cs,done,chrdo,valid,rdo_done,idle,decode,exec,en1,ack,en2,exrdo,
h_err,h_abt);


input       clk2,rstb;
input       cstb, cs;
input    done,chrdo,valid,rdo_done;

output      idle,
                  decode,
                  exec,
                  en1,
                  ack,
                  en2,
                  exrdo;

output      h_err,
                  h_abt;

reg         idle,
                  decode,
                  exec,
                  en1,
                  ack,
                  en2,
                  exrdo;
```

```
reg [5:0]   next_st, st;
reg               h_err, h_abt;


// State definition

parameter   s_idle       = 6'b000000,           // 0
                        s_decode1 = 6'b000111,        // 1
                        s_decode2 = 6'b011001,        // 2
                        s_exec    = 6'b011110,        // 3
                        s_en1     = 6'b101010,        // 4
                        s_ack     = 6'b101101,        // 5
                        s_en2     = 6'b110011,        // 6
                        s_rdo        = 6'b110100;           // 7


always @(posedge clk2 or negedge rstb)
begin
      if (!rstb)
            st = s_idle;
      else
            st = next_st;
end

always @(st or cstb or cs or done or chrdo or valid or rdo_done)
begin

            next_st = st;
// Good States

      case (st)

            s_idle:                       //0
             begin
                  h_err  = 0;
                  h_abt  = 0;
                  idle   = 1;
                  decode = 0;
                  exec   = 0;
                  en1    = 0;
                  ack    = 0;
                  en2    = 0;
                  exrdo  = 0;

                  if(cstb) next_st = s_decode1;
                  else        next_st = s_idle;
             end


            s_decode1:                    //1
             begin
                  h_err  = 0;
                  h_abt  = 0;
                  idle   = 0;
                  decode = 1;
                  exec   = 0;
                  en1    = 0;
                  ack    = 0;
                  en2    = 0;
                  exrdo  = 0;

                  if(cstb & cs)     next_st = s_decode2;
                  else                    next_st = s_idle;
             end

            s_decode2:                    //2
             begin
                  h_err  = 0;
                  h_abt  = 0;
                  idle   = 0;
                  decode = 0;
                  exec   = 0;
                  en1    = 0;
                  ack    = 0;
                  en2    = 0;
                  exrdo  = 0;

                  if (valid)
                        if (chrdo) next_st = s_en1;
                        else       next_st = s_exec;
                  else  next_st = s_idle;
             end

            s_exec:                       // 3
             begin
                  h_err  = 0;
                  h_abt  = 0;
                  idle   = 0;
                  decode = 0;
                  exec   = 1;
                  en1    = 0;
                  ack    = 0;
                  en2    = 0;
                  exrdo  = 0;

                  if (done) next_st = s_en1;
                  else      next_st = s_exec;
             end


            s_en1:                        // 4
             begin
                  h_err  = 0;
                  h_abt  = 0;
                  idle   = 0;
                  decode = 0;
                  exec   = 0;
                  en1    = 1;
                  ack    = 0;
                  en2    = 0;
                  exrdo  = 0;

                  next_st = s_ack;
```

```
            end


           s_ack:                        // 5
            begin
                 h_err  = 0;
                 h_abt  = 0;
                 idle   = 0;
                 decode = 0;
                 exec   = 0;
                 en1    = 0;
                 ack    = 1;
                 en2    = 0;
                 exrdo  = 0;

                 if (!cstb)  next_st = s_en2;
                 else        next_st = s_ack;
            end


           s_en2:                        // 6
            begin
                 h_err  = 0;
                 h_abt  = 0;
                 idle   = 0;
                 decode = 0;
                 exec   = 0;
                 en1    = 0;
                 ack    = 0;
                 en2    = 1;
                 exrdo  = 0;

                 if (!chrdo) next_st = s_idle;
                 else        next_st = s_rdo;
            end


           s_rdo:                        // 7
            begin
                 h_err  = 0;
                 h_abt  = 0;
                 idle   = 0;
                 decode = 0;
                 exec   = 0;
                 en1    = 0;
                 ack    = 0;
                 en2    = 0;
                 exrdo  = 1;

                 if (rdo_done)  next_st = s_idle;
                 else                    next_st = s_rdo;
            end


// Recoverable States

      //derived from s_idle              // 0
           6'b000001, 6'b000010, 6'b000100, 6'b001000, 6'b010000, 6'b100000:
            begin
                 h_err  = 1;
                 h_abt  = 0;
                 idle   = 1;
                 decode = 0;
                 exec   = 0;
                 en1    = 0;
                 ack    = 0;
                 en2    = 0;
                 exrdo  = 0;

                 if(cstb) next_st = s_decode1;
                 else        next_st = s_idle;
            end

     //derived from s_decode1                  // 1
           6'b000110, 6'b000101, 6'b000011, 6'b001111, 6'b010111, 6'b100111:
            begin
                 h_err  = 1;
                 h_abt  = 0;
                 idle   = 0;
                 decode = 1;
                 exec   = 0;
                 en1    = 0;
                 ack    = 0;
                 en2    = 0;
                 exrdo  = 0;

                 if(cstb & cs)     next_st = s_decode2;
                 else                    next_st = s_idle;
            end

     //derived from s_decode2            // 2
           6'b011000, 6'b011011, 6'b011101, 6'b010001, 6'b001001, 6'b111001:
            begin
                 h_err  = 1;
                 h_abt  = 0;
                 idle   = 0;
                 decode = 0;
                 exec   = 0;
                 en1    = 0;
                 ack    = 0;
                 en2    = 0;
                 exrdo  = 0;

                 if (chrdo) next_st = s_en1;
                 else       next_st = s_exec;
            end

     //derived from s_exec          // 3
           6'b011111, 6'b011100, 6'b011010, 6'b010110, 6'b001110, 6'b111110:
            begin
                 h_err  = 1;
                 h_abt  = 0;
                 idle   = 0;
                 decode = 0;
```

```
                 exec   = 1;
                 en1    = 0;
                 ack    = 0;
                 en2    = 0;
                 exrdo  = 0;

                 if (done) next_st = s_en1;
                 else      next_st = s_exec;
            end

    //derived from s_en1          // 4
           6'b101011, 6'b101000, 6'b101110, 6'b100010, 6'b111010, 6'b001010:
            begin
                 h_err  = 1;
                 h_abt  = 0;
                 idle   = 0;
                 decode = 0;
                 exec   = 0;
                 en1    = 1;
                 ack    = 0;
                 en2    = 0;
                 exrdo  = 0;

                 next_st = s_ack;
            end

    //derived from s_ack          // 5
           6'b101100, 6'b101111, 6'b101001, 6'b100101, 6'b111101, 6'b001101:
            begin
                 h_err  = 1;
                 h_abt  = 0;
                 idle   = 0;
                 decode = 0;
                 exec   = 0;
                 en1    = 0;
                 ack    = 1;
                 en2    = 0;
                 exrdo  = 0;

                 if (!cstb)  next_st = s_en2;
                 else        next_st = s_ack;
            end

    //derived from s_en2          // 6
           6'b110010, 6'b110001, 6'b110111, 6'b111011, 6'b100011, 6'b010011:
            begin
                 h_err  = 1;
                 h_abt  = 0;
                 idle   = 0;
                 decode = 0;
                 exec   = 0;
                 en1    = 0;
                 ack    = 0;
                 en2    = 1;
                 exrdo  = 0;

                 if (!chrdo) next_st = s_idle;
```

```
                  else        next_st = s_rdo;
             end

      //derived from s_rdo                // 7
            6'b110101, 6'b110110, 6'b110000, 6'b111100, 6'b100100, 6'b010100:
             begin
                  h_err  = 1;
                  h_abt  = 0;
                  idle   = 0;
                  decode = 0;
                  exec   = 0;
                  en1    = 0;
                  ack    = 0;
                  en2    = 0;
                  exrdo  = 1;

                  if (rdo_done)  next_st = s_idle;
                  else                    next_st = s_rdo;
             end


// Irrecoverable states

      //other
            default:
             begin
                  h_err  = 0;
                  h_abt  = 1;
                  idle   = 0;
                  decode = 0;
                  exec   = 0;
                  en1    = 0;
                  ack    = 0;
                  en2    = 0;
                  exrdo  = 0;
                  next_st  = s_idle;
             end

      endcase

end

endmodule



//------------------------------INTRDOH--------------------------------

module intrdoh (clk2,rstb,exrdo,ch_red,rdo,waitst,done,h_err,h_abt);


input       clk2,rstb;
input       exrdo,ch_red;

output      rdo,
                  waitst,
                  done;
```

```
output      h_err,
               h_abt;

reg         rdo,
                  waitst,
                  done;

reg [4:0]   next_st, st;
reg            h_err, h_abt;


// State definition

      parameter   s_idle   = 5'b00000,          // 00
                        s_rdo    = 5'b00111,          // 01
                        s_wait   = 5'b11001,          // 10
                        s_done   = 5'b11110;          // 11

always @(posedge clk2 or negedge rstb)
begin
      if (!rstb)
            st = s_idle;
      else
            st = next_st;
end

always @(st or exrdo or ch_red)
begin

            next_st = st;
// Good States

      case (st)

            s_idle:                       //0
             begin
                  h_err    = 0;
                  h_abt    = 0;
                  rdo         = 0;
                  waitst   = 0;
                  done     = 0;

                  if(exrdo) next_st = s_rdo;
                  else         next_st = s_idle;
             end


            s_rdo:                  // 1
             begin
                  h_err    = 0;
                  h_abt    = 0;
                  rdo         = 1;
                  waitst   = 0;
                  done     = 0;

                  next_st = s_wait;
             end


            s_wait:                       // 2
             begin
                  h_err    = 0;
                  h_abt    = 0;
                  rdo         = 0;
                  waitst   = 1;
                  done     = 0;

                  if (ch_red) next_st = s_done;
                  else              next_st = s_wait;
             end


            s_done:                       // 3
             begin
                  h_err    = 0;
                  h_abt    = 0;
                  rdo         = 0;
                  waitst   = 0;
                  done     = 1;

                  next_st = s_idle;
             end

// Recoverable States

      //derived from s_idle              // 0
            5'b00001, 5'b00010, 5'b00100, 5'b01000, 5'b10000:
             begin
                  h_err    = 1;
                  h_abt    = 0;
                  rdo         = 0;
                  waitst   = 0;
                  done     = 0;

                  if(exrdo) next_st = s_rdo;
                  else         next_st = s_idle;
             end

      //derived from s_rdo               // 1
            5'b00110, 5'b00101, 5'b00011, 5'b01111, 5'b10111:
             begin
                  h_err    = 1;
                  h_abt    = 0;
                  rdo         = 1;
                  waitst   = 0;
                  done     = 0;

                  next_st = s_wait;
             end

      //derived from s_wait              // 2
            5'b11000, 5'b11011, 5'b11101, 5'b10001, 5'b01001:
             begin
```

```
                  h_err    = 0;
                  h_abt    = 0;
                  rdo         = 0;
                  waitst   = 1;
                  done     = 0;

                  if (ch_red) next_st = s_done;
                  else              next_st = s_wait;
             end

      //derived from s_done               // 3
            5'b11111, 5'b11100, 5'b11010, 5'b10110, 5'b01110:
             begin
                  h_err    = 0;
                  h_abt    = 0;
                  rdo         = 0;
                  waitst   = 0;
                  done     = 1;

                  next_st = s_idle;
             end


// Irrecoverable states

      //other
            default:
             begin
                  h_err    = 0;
                  h_abt    = 1;
                  rdo         = 0;
                  waitst   = 0;
                  done     = 0;

                  next_st = s_idle;
             end

      endcase

end

endmodule



//------------------------------BUSINT-------------------------------//


module busint (l2y_i, load, hadd, bd, write, trg, clk, clk2, rstb,
hadd_r, add_r, data_r, write_r, trg_r, l2y_r, cs, loadcs);

//Inputs

input       load,
            write,
            trg,
            clk,
            clk2,
            l2y_i,
            loadcs,
            rstb;
input [7:0] hadd;
input [39:0]      bd;


//Outputs

output            cs,
            write_r,
            trg_r,
            l2y_r;
output      [7:0] hadd_r;
output      [19:0]      add_r,
            data_r;


reg         write_r,
            w1, w0;
reg   [7:0] hadd_r, w2;
reg   [19:0]      data_r,
            add_r;

wire        busyb, busyb2, eq;


glitchf g0(trg, clk, rstb, trg_r);
sync221 s1(l2y_i, clk, clk2, rstb, l2y_r);

assign eq = (hadd_r == w2);
assign cs = ( ~w0 & ( eq | w1 ));


always@(posedge clk2 or negedge rstb)
if(!rstb)
      begin
            write_r = 0;
            add_r = 0;
            data_r = 0;
            hadd_r = 0;
            w0 = 0;
            w1 = 0;
            w2 = 0;

      end
else
      begin

            if (load )
                  begin
                        write_r <= write;
                        data_r <= bd[19:0];
                        add_r <= bd[39:20];
                  end
            hadd_r <= hadd;

            if (loadcs )
             begin
              w0 <= bd[37];
              w1 <= bd[38];
              w2 <= bd[36:29];
             end
      end

endmodule



//--------------------- INTCTRL ---------------------------//


module intctrl (clk2,rstb,done,ch_red,cstb,cs,chrdo,valid,
load,exec,en1,ack,en2,rdo,waitst,h_err,h_abt,loadcs);

//Inputs

input cstb,
      cs,
      done,
      chrdo,
      valid,
      ch_red,
      rstb,
      clk2;


//Outputs

output      load,
      h_err,
      h_abt,
      exec,
      en1,
      ack,
      en2,
      rdo,
      waitst,
      loadcs;

//wires

wire  decode, exec, exrdo, h_err0, h_abt0, h_err1, h_abt1, idle;

//instantiations

intrdoh i0(clk2,rstb,exrdo,ch_red,rdo,waitst,rdo_done,h_err0,h_abt0);
intrech
i1(clk2,rstb,cstb,cs,done,chrdo,valid,rdo_done,idle,decode,exec,en1,ack,en2,exrd
o,h_err1,h_abt1);

assign h_err = ( h_err0 | h_err1 );
assign h_abt = ( h_abt0 | h_abt1 );
assign load = ( cs & decode );
assign loadcs = idle;

endmodule



//-------------------------------- INTDEC UNIT -------------------------------
------//



module intdec(add, write, cs, valid, chrdo, rg_wr, rg_rd, push, pop, swtrg,
trc_clr,
              err_clr, wr_bsl, rd_bsl, bcast, chadd, csr_sel, instr_err,
par_err);


//Inputs

input [19:0]      add;
input                   write;
input                   cs;


//Output

output                              valid,
                              chrdo,
                              rg_wr,
                              rg_rd,
                              push,
                              pop,
                              swtrg,
                              trc_clr,
                              err_clr,
                              wr_bsl,
                              rd_bsl,
                              bcast,
                              instr_err,
                              par_err;
output [3:0]      chadd;
output [4:0]      csr_sel;



wire        w1, w2, w3, w4, bcast_rd_err, bcast_err, bcast_err2, bcast_err3,
reg_err, bank1_err, bank2_err, bank3_err,bank0_sel, bank1_sel, bank2_sel,
bank3_sel, read, bcast3_sel, reg_err_new;



//Channel register select
assign csr_sel = add[4:0];
```

```
//Channel address
assign chadd = add[8:5];

//Broadcast
assign bcast = add[18];

// Parity error
assign par_err =
add[0]^add[1]^add[2]^add[3]^add[4]^add[5]^add[6]^add[7]^add[8]^add[9]^add[10]^ad
d[11]^add[12]^add[13]^add[14]^add[15]^add[16]^add[17]^add[18]^add[19];


//Read signal
assign read = !write;


//Valid if not instruction error
assign valid = !instr_err & !par_err;



//Assignements of bank registers

assign bank0_sel = (csr_sel[4:3]==2'b00);
assign bank1_sel = (csr_sel[4:3]==2'b01);
assign bank2_sel = (csr_sel[4:3]==2'b10);
assign bank3_sel = (csr_sel[4:3]==2'b11);



//Assignements Errors

//Instruction error = broadcast error OR register addressing error
assign instr_err = bcast_err | reg_err_new;


      //Broadcast error = broadcast and read OR broadcasting the wrong register
      assign bcast_err = (bcast_rd_err | bcast_err2 | bcast_err3);

            //Generation of the error signals for broadcast error
            assign bcast_rd_err = read & bcast;
            assign bcast_err2 = bank2_sel & bcast;
            assign bcast_err3 = bank3_sel & bcast & (csr_sel[2:0]==3'b010) &
write;


      //Register addressing error
      assign reg_err_new = reg_err & cs;                          // Register
error and chip select
      assign reg_err = (bank1_err | bank2_err | bank3_err);             //
Chosing one of the wrong banks

            //Defining each bank error
            assign bank1_err = bank1_sel & ((csr_sel[2:0]==3'b110) |
(csr_sel[2:0]==3'b111));
            assign bank2_err = bank2_sel & (write | (csr_sel[2]==1'b1) |
(csr_sel[1:0]==2'b11));
```

```
            assign bank3_err = bank3_sel & (read | (csr_sel[2:0]==3'b110) |
(csr_sel[2:0]==3'b111));




//Decoding the Instructions: bank3

assign w2 = valid & write & bank3_sel & cs;     /* Common Conditions:   - Chip
is selected
                                                        - No instruction error
                                                        - Write
                                                        - Bank 3 selection
                                    */


      assign push =           ((csr_sel[2:0]==3'b000) & w2);
      assign pop =            ((csr_sel[2:0]==3'b001) & w2);
      assign chrdo =          ((csr_sel[2:0]==3'b010) & w2);
      assign swtrg =          ((csr_sel[2:0]==3'b011) & w2);
      assign trc_clr =  ((csr_sel[2:0]==3'b100) & w2);
      assign err_clr =  ((csr_sel[2:0]==3'b101) & w2);



//Write baseline and read baseline

assign w3 = cs & bank0_sel & valid;       /* Common conditions:   - chip select

                                                  - Bank 0 selection

                                                  - No instruction error
                                    */

      assign wr_bsl = ((csr_sel[2:0]==3'b111) & write & w3);
      assign rd_bsl = ((csr_sel[2:0]==3'b111) & read & w3);



//Write register and read register

assign w4 = cs & (bank0_sel | bank1_sel | bank2_sel) & valid;

                                    /* Common conditions:   - chip select

      - All banks but bank 3

      - No instruction error
                                    */

      assign rg_wr = (write & w4);
      assign rg_rd = (read & w4);



endmodule
```

```
//------------------------------------------------INTEXEC-----------------------
---------------------------------//


module intexec (l2y_r, exec, en1, ackn, en2, waitst, clk2, rstb, clk, valid,
chrdo0, rg_rd0, rg_wr0, push0, pop0, swtrg, trc_clr0, err_clr0,
wr_bsl0, rd_bsl0, bcast, hwtrg, twx, done, ack, ack_en, dolo_en, trsf_en,
trg_overlap, rg_wr, push, pop, trg,
trc_clr, err_clr, wr_bsl, rd_bsl, tstout);

//Inputs

input       valid,
            rg_rd0,
            en1,
            ackn,
            en2,
            waitst,
            chrdo0,
            bcast,
            exec,
            clk,
            clk2,
            rstb,
            rg_wr0,
            push0,
            pop0,
            swtrg,
            hwtrg,
            trc_clr0,
            err_clr0,
            wr_bsl0,
            rd_bsl0,
            l2y_r,
            twx;

//Outputs

output            ack,
            dolo_en,
            ack_en,
            trsf_en,
            done,
            rg_wr,
            push,
            pop,
            trg,
            trg_overlap,
            trc_clr,
            err_clr,
            wr_bsl,
            rd_bsl,
            tstout;


wire        x1, busyb, w0;
reg         r1, r2;
```

```
assign ack = (valid & ackn);
assign w0 = (en1 | ackn | en2);
assign dolo_en = (valid & ((rg_rd0 & w0) | (waitst & chrdo0)));
assign ack_en = (valid & ~(bcast | !w0));
assign trsf_en = (valid & (waitst & chrdo0));
assign done = ~(!r1 | r2);
assign tstout = (ack | twx | x1);

assign rg_wr = (rg_wr0 & x1);
assign push = (push0 & x1) | l2y_r;
assign pop = (pop0 & x1);
assign trc_clr = (trc_clr0 & x1);
assign err_clr = (err_clr0 & x1);
assign wr_bsl = (wr_bsl0 & x1);
assign rd_bsl = (rd_bsl0 & x1);
assign trg = ~(twx | ~((swtrg & x1) | hwtrg));
assign trg_overlap = (twx & ((swtrg & x1) | hwtrg));


sync21 s(exec, clk, clk2, rstb, x1, busyb);


always@(posedge clk2 or negedge rstb)
if(!rstb)
      begin
            r1 = 0;
            r2 = 0;
      end
else
      begin
            r1 <= busyb;
            r2 <= r1;
      end


endmodule



//-----------------------------------------INTERFACE----------------------------
--------------------------//


module interface(clk, clk2, rstb, ch_red, hadd, bd, cstb, write, trg_i, twx,
h_abt, h_err, chrdo, ack,
ack_en, dolo_en, trsf_en, trg_overlap, csr_wr, push, pop, trg, trc_clr, err_clr,
wr_bsl, rd_bsl, instr_err,
bcast, chadd, csr_sel, hadd_o, csr_do, par_err, l2y_i, tstout);



//inputs assynchronous

input       rstb,
            write,
            l2y_i,
            trg_i;

//inputs with clk

input       clk,
            twx;

//inputs with clk2

input       clk2,
            ch_red,
            cstb;
input [7:0] hadd;
input [39:0]      bd;




//outputs with clk

output            trg_overlap,
            csr_wr,
            push,
            pop,
            trg,
            trc_clr,
            err_clr,
            wr_bsl,
            rd_bsl,
            instr_err,
            par_err,
            bcast,
            tstout;
output      [3:0] chadd;
output      [4:0] csr_sel;
output      [7:0] hadd_o;
output      [19:0]      csr_do;



//outputs with clk2

output            h_abt,
            h_err,
            chrdo,
            ack,
            ack_en,
            dolo_en,
            trsf_en;


//registers and wires
reg         h_abt, h_err;

wire        rdo, hwtrg ,l2y_r;
wire  [19:0]      add_r;



//State machines
intctrl
i1(clk2,rstb,done,ch_red,cstb,cs,chrdo0,valid,load,exec,en1,ackn,en2,chrdo,waits
t,h_err0,h_abt0,
loadcs);


//Instruction decoder
intdec i2(add_r, write_r, cs, valid, chrdo0, rg_wr0, rg_rd0, push0, pop0, swtrg,
trc_clr0, err_clr0,
wr_bsl0, rd_bsl0, bcast0, chadd, csr_sel, instr_err0, par_err0);


//Bus  interface
busint b1(l2y_i, load, hadd, bd, write, trg_i, clk, clk2, rstb, hadd_o, add_r,
csr_do, write_r, trg_r,
l2y_r, cs, loadcs);


//Instructor executor
intexec i3 (l2y_r, exec, en1, ackn, en2, waitst, clk2, rstb, clk, valid,
chrdo0, rg_rd0, rg_wr0, push0, pop0, swtrg, trc_clr0, err_clr0,
wr_bsl0, rd_bsl0, bcast0, hwtrg, twx, done, ack, ack_en, dolo_en, trsf_en,
trg_overlap, csr_wr, push, pop, trg,
trc_clr, err_clr, wr_bsl, rd_bsl, tstout);


assign hwtrg = trg_r;
assign bcast = (~csr_sel[3] & ~csr_sel[4] & bcast0);  //Select only block 0

//Synchronizers

sync21 s0(par_err0, clk, clk2, rstb, x0, by0);
sync21 s1(par_err0, clk, clk2, rstb, x1, by1);
sync21 s2(par_err0, clk, clk2, rstb, x2, by2);

assign par_err = ((x0 & x1) | (x2 & x1) | (x0 & x2));

sync21 s3(instr_err0, clk, clk2, rstb, x3, by3);
sync21 s4(instr_err0, clk, clk2, rstb, x4, by4);
sync21 s5(instr_err0, clk, clk2, rstb, x5, by5);

assign instr_err = ((x3 & x4) | (x5 & x3) | (x5 & x4));


always@(posedge clk2 or negedge rstb)

if(!rstb)
      begin
            h_err = 0;
            h_abt = 0;
      end
else
```

```
      begin
            h_err <= h_err0;
            h_abt <= h_abt0;
      end


endmodule




//------------------------------------------ALTRO--------------------------------
----------------------//


module altro(clk, clk2, rstb, trg, l2y, cstb, writeb, ackb, ackb_en, dolob_en,
trsfb, trsfb_en, dstb, errorb, tstoutb, bd);



//inputs assynchronous

input       rstb,
            writeb,
            l2y,
            trg;

//inputs with clk
input       clk;

//inputs with clk2
input       clk2,
            cstb;

//Bidirectional Bus
inout [39:0]      bd;


//outputs with clk
output            tstoutb;


//outputs with clk2
output            ackb,
            trsfb,
            ackb_en,
            dolob_en,
            trsfb_en,
            dstb,
            errorb;


wire  [3:0] chadd;
wire  [4:0] csr_sel;
wire  [7:0] hadd_o;
wire  [19:0]      csr_do;
wire  [39:0]      bd, din;
wire        mixerror;
reg         ch_red, trsf, error;


interface i1(clk, clk2, rstb, ch_red, 8'h01, din, cstb, write, trg, 1'b0, h_abt,
h_err, chrdo,
ack, ack_en, dolo_en, trsf_en, trg_overlap, csr_wr, push, pop, trg_o, trc_clr,
err_clr, wr_bsl,
rd_bsl, instr_err, bcast, chadd, csr_sel, hadd_o, csr_do, par_err, l2y, tstout);


always @(chrdo)                // Simulate event manager
if(!rstb)

 begin
  ch_red = 1;
  trsf = 0;
 end

else

 begin
       ch_red = 0;
       #48 trsf = 1;
       #985 ch_red = 1;
       trsf = 0;
 end


always@(posedge mixerror or posedge err_clr or negedge rstb)
if(err_clr | !rstb)     error = 1'b0;
else if(mixerror) error <= 1'b1;


assign mixerror = (i1.i2.instr_err | par_err);
assign dstb = trsf? ~clk2:1'b0;
assign din = trsf_en? 40'h0:bd;
assign bd = (trsf_en & dolo_en)? 40'h5a5a5a5a5a:((!trsf_en & dolo_en)?
{20'hzzzzz,20'h55555}: 40'hzzzzzzzzzz);

//Inverting Inputs
assign ackb = !ack;
assign ackb_en = !ack_en;
assign trsfb = !trsf;
assign trsfb_en = !trsf_en;
assign tstoutb = !tstout;
assign dolob_en = !dolo_en;
assign errorb = !error;

//Inverting Outputs
assign write = !writeb;

endmodule
```