Top Module Simulation Shell Script

```
vfiles="DDR_TOP_tb.v \

      ddr.v\
      altddio.v\
      ALTERA_DEVICE_FAMILIES.v\
      ../DDR_TOP.v \
      ../ddram_ctrl.v\
      ../ddr_init.v\
      ../ddr_read_write.v
      ../mux_2dir_reg.v\
      ../mux_4dir_reg.v\
      ../reg_1stg.v"


iverilog $vfiles -DSIMULATION -I../ -o tmp.vvp


vvp tmp.vvp


rm -f tmp.vvp
```

This script defines all the modules used by the simulation in vfiles, the compiles, synthesizes and runs a simulation using iVerilog. You must be running a version of Linux and have gtkwave and iVerilog executable from the console to use this.

DDR_TOP_tb.v – testbench for the top file

```
/* This is the simulation for the top file. It simulates the user pressing the reset
button on the
DE2 board. Note that we are simulating 8-bit RAM. DDR.v has a compilation error if 16-
bit is chosen.
Please contact Micron you want to fix this :)
*/


`include "defs.v"

module test ();

    reg ddr_rst, CLOCK_50;

    wire test_done;

    initial begin
        $dumpfile("ddr_top.vcd"); //GTKWave outputfile
        $dumpvars(0,test); //dump all signals into .vcd
//        #(`DDR_CLK_PERIOD * 50) $finish;
    end
    initial begin
        CLOCK_50  = 0;
        ddr_rst  = 1;
        #(`DDR_CLK_PERIOD*3);
        ddr_rst = 0;
    end


    always CLOCK_50 = #(`DDR_CLK_PERIOD/2) ~CLOCK_50;

assign test_done = GPIO_0[0];


always @(test_done)
       if(test_done)
              #(`DDR_CLK_PERIOD*20) $finish; //finish simulation 20 cycles after
read/write is complete



wire   [35:0] GPIO_0;                                    //     GPIO Connection 0
wire   [35:0] GPIO_1;                                    //     GPIO Connection 1

wire  [12:0] Addr;
assign Addr =
{                GPIO_1[16], //addr 12
                                GPIO_1[17], //11
                                GPIO_1[7], //10
                                GPIO_1[18], //9
                                GPIO_1[19], //8
                                GPIO_1[20], //7
                                GPIO_1[21], //6
                                GPIO_1[22], //5
                                GPIO_1[23], //4
                                GPIO_1[2], //3
                                GPIO_1[4], //2
                                GPIO_1[5], //1
```

```
                                GPIO_1[6]  //addr 0
                            };


wire [15:0] Dq;
assign  Dq = {      GPIO_0[21], //d15
                                GPIO_0[20], //14
                                GPIO_0[19], //13
                                GPIO_0[18], //12
                                GPIO_0[17], //11
                                GPIO_0[22], //10
                                GPIO_0[23], //9
                                GPIO_0[24], //8
                                GPIO_0[9], //7
                                GPIO_0[10], //6
                                GPIO_0[11], //5
                                GPIO_0[16], //4
                                GPIO_0[15], //3
                                GPIO_0[14], //2
                                GPIO_0[13], //1
                                GPIO_0[12] //0
                            }; //bidirectional data io


wire [1:0] bank;
assign bank = {GPIO_1[8],GPIO_1[9]} ; //ba1-0

wire CS,RAS,CAS,WE,LDM,LDQS;

assign              {CS, RAS,CAS,WE,LDM,LDQS} =
          { GPIO_0[3], //cs
                    GPIO_0[4], //ras
                    GPIO_0[5], //cas
                    GPIO_0[6], //we
                    GPIO_0[7], //LDM
                    GPIO_0[8] //LDQS
                                            };


wire UDM, UDQS, CKn, CK, CKE;
assign {CKE, CK, CKn, UDM,UDQS} = GPIO_0[29:25] ;

//wire CS,RAS,CAS,WE,LDM,LDQS;



DDR_TOP fpga
       (
             ////////////////////       Clock Input         ////////////////////
             .CLOCK_27(),                                   //     27 MHz
             .CLOCK_50(CLOCK_50),                                   //     50 MHz
             ////////////////////       Push Button         ////////////////////
             .KEY(),                                        //
       Pushbutton[3:0]
             ////////////////////       7-SEG Dispaly////////////////////
             .HEX0(),                                       //     Seven Segment
Digit 0
             .HEX1(),                                       //     Seven Segment
Digit 1
             .HEX2(),                                       //     Seven Segment
Digit 2
             .HEX3(),                                       //     Seven Segment
Digit 3
```

```
            .ddr_rst(ddr_rst),
            .GPIO_0(GPIO_0),                                        //    GPIO
Connection 0
            .GPIO_1(GPIO_1)                                         //    GPIO
Connection 1
      );


      ddr ddram
            (
            .Clk   (CK),
            .Clk_n (CKn),
            .Cke   (CKE), //not used??
            .Cs_n  (CS), //not used??
            .Ras_n  (RAS),
      .Cas_n  (CAS),
      .We_n   (WE),
`ifdef DDR16BIT // user must choose whether to implement 16 bit or 8 bit ddr
                  // currently, the micron DDR.v module has a compilation error when
using 16-bit DDR
                  // For simulation purposes we use 8-bit ram instead
      .Dq     (Dq),
`else
      .Dq     (Dq[7:0]),
`endif
       .Addr   (Addr),
       .Ba     (bank),
       .Dm     (LDM), //data mask
       .Dqs    (LDQS) //data strobe
   );

endmodule
```

DDR_TOP.v – Top file

```
module DDR_TOP
      (
            ////////////////////    Clock Input       ////////////////////
            CLOCK_27,                                 //    27 MHz
            CLOCK_50,                                 //    50 MHz
            ////////////////////    Push Button       ////////////////////
            KEY,                                      //    Pushbutton[3:0]
            ////////////////////    7-SEG Dispaly     ////////////////////
            HEX0,                                     //    Seven Segment
Digit 0
            HEX1,                                     //    Seven Segment
Digit 1
            HEX2,                                     //    Seven Segment
Digit 2
            HEX3,                                     //    Seven Segment
Digit 3
            ddr_rst, //****REMOVE WHEN COMPILING FOR FPGA****
            GPIO_0,                                   //    GPIO
Connection 0
            GPIO_1                                    //    GPIO
Connection 1
      );

//////////////////////      Clock Input       //////////////////////
input             CLOCK_27;                   //    27 MHz
```

```
input             CLOCK_50;                     //    50 MHz

input [3:0] KEY;                           //    Pushbutton[3:0]

input ddr_rst;
//////////////////////      7-SEG Dispaly     //////////////////////
output      [6:0] HEX0;                      //    Seven Segment Digit 0
output      [6:0] HEX1;                      //    Seven Segment Digit 1
output      [6:0] HEX2;                      //    Seven Segment Digit 2
output      [6:0] HEX3;                      //    Seven Segment Digit 3


inout [35:0]      GPIO_0;                          //    GPIO Connection 0
inout [35:0]      GPIO_1;                          //    GPIO Connection 1

//    Turn on all display
assign      HEX0        =     7'h00;
assign      HEX1        =     7'h00;
assign      HEX2        =     7'h00;
assign      HEX3        =     7'h00;


//    All inout port turn to tri-state

//Linking wires from expansion headers to FPGA in/out

wire  [12:0] DDR_addr_w;
assign {              GPIO_1[16], //addr 12
                              GPIO_1[17], //11
                              GPIO_1[7], //10
                              GPIO_1[18], //9
                              GPIO_1[19], //8
                              GPIO_1[20], //7
                              GPIO_1[21], //6
                              GPIO_1[22], //5
                              GPIO_1[23], //4
                              GPIO_1[2], //3
                              GPIO_1[4], //2
                              GPIO_1[5], //1
                              GPIO_1[6]  //addr 0
                        } = DDR_addr_w;

//assign GPIO_0[1] = 1b'1;
//assign GPIO_0[0] = 1b'0;

wire [1:0] bank_w;
assign {GPIO_1[8],GPIO_1[9]} = bank_w ; //ba1-0

reg RAS_w, CAS_w, WE_w;

wire CS_w, LDM_w,LDQS_w;


assign CS_w = 0;

assign            { GPIO_0[3], //cs
                  GPIO_0[4], //ras
```

```
                  GPIO_0[5], //cas
                  GPIO_0[6], //we
                  GPIO_0[7], //LDM
                  GPIO_0[8] //LDQS
                                          } = {CS_w,
RAS_w,CAS_w,WE_w,LDM_w,LDQS_w};

wire [15:0] data_io_w;
assign {    GPIO_0[21], //d15
                              GPIO_0[20], //14
                              GPIO_0[19], //13
                              GPIO_0[18], //12
                              GPIO_0[17], //11
                              GPIO_0[22], //10
                              GPIO_0[23], //9
                              GPIO_0[24], //8
                              GPIO_0[9], //7
                              GPIO_0[10], //6
                              GPIO_0[11], //5
                              GPIO_0[16], //4
                              GPIO_0[15], //3
                              GPIO_0[14], //2
                              GPIO_0[13], //1
                              GPIO_0[12]  //0
                         } = data_io_w; //bidirectional data io

wire UDM_w, UDQS_w, CKn_w, CK_w, CKE_w;
wire CK_w_90, CK_w_270;
assign      GPIO_0[29:25]
      = {CKE_w, CK_w, CKn_w, UDM_w,UDQS_w};

wire [31:0] data_out;  // to the ddr
wire [31:0] data_in;   // from the ddr
//wire ddr_clk, reset;
//assign reset = ~KEY[0]; //reset = button 0

wire inclock = CK_w;
wire outclock = CK_w;

assign CKn_w = ~CK_w;
//assign CK_w = CLOCK_50;
//assign GPIO_0[0] = CLOCK_50;


//assign data_in[31:16] = 16'hffff;
//assign data_in[15:0] = 16'h0000;


    // ren is 3 clocks early, to allow for output pipeline precharde
    // we delay ren for 3 clocks for the dqs testing
    reg ren, ren0, ren1;
      wire reno;
    always @(posedge CK_w or posedge ~pll_clock_locked)
      if (~pll_clock_locked) {ren, ren0, ren1} = 0;
      else         {ren, ren0, ren1} = {ren0, ren1, reno};

    reg ddrio_oe;
```

```
     always @(posedge CK_w or posedge ~pll_clock_locked)
      if (~pll_clock_locked) ddrio_oe = 0;
      else                   ddrio_oe = ren;


//     reg dqs_cmd;
//     //wire dqs;
//     //assign dqs = dts == 1 ? 1'bZ : dqs_cmd;
//
//    assign LDQS_w = dts == 1 ? 1'bZ : dqs_cmd;
//    assign UDQS_w = dts == 1 ? 1'bZ : dqs_cmd;



//     always @(posedge ren) begin
//          @(posedge CK_w);
//                ddr_ext_dat_o = 8'haa;
//                dqs_cmd = 1;
//          @(negedge CK_w);
//                ddr_ext_dat_o = 8'h55;
//                dqs_cmd = 0;
//    end



//io to/from DDR memory

// bit 17 is used as DQS
wire dqs_in_h, dqs_in_h;

`ifdef SIMULATION //required for simulation to run properly
altddio_bidir
#(.width(17),
  .intended_device_family("CYCLONE II")
 ) ddrio (
`else
altddio io (
`endif
          .datain_h({1'b0, 16'h0000}),   // required port, input data to be
output of padio port at the
                       // rising edge of outclock
          .datain_l({1'b1, 16'h1234}),   // required port, input data to be
output of padio port at the
                       // falling edge of outclock
          .inclock(inclock),    // required port, input reference clock to
sample data by
            .outclock(outclock),   // required port, input reference clock to
register data output
          .inclocken(1'b1),  // inclock enable
          .outclocken(1'b1), // outclock enable

            //.aclr(reset),       // asynchronous clear
          .oe(ddrio_oe),         // output enable for padio port
          .dataout_h({dqs_in_h, data_out[31:16]}),  // data sampled from the
padio port at the rising edge of inclock
```

```
          .dataout_l({dqs_in_l, data_out[15:0]}),  // data sampled from the
padio port at the falling edge of
          .padio({LDQS_w, data_io_w})     // bidirectional DDR port
      ); //bidirectional data to and from DDR ram

wire pll_clock_locked; //output from pll

//DDR clk pll
`ifdef SIMULATION
assign CK_w = CLOCK_50;
assign pll_clock_locked = !ddr_rst;
`else
write_clk_pll pll1 (
      .inclk0(CLOCK_50),
      .pllena(1'b1), //always enable
      .c0(CK_w),
      .c1(CK_w_90),
      .c2(CK_w_270),
      .areset(ddr_rst),
      .locked(pll_clock_locked) );
`endif

    wire [1:0] ddr_cmd_rw;
    wire  ddr_cmd_init;

    wire [10:0] ddr_cmd_addr; // A11, 12 is always set to 0 in ddrctrl
    wire [2:0]  ddr_cmd_rcw;

    wire [2:0]  ddr_len;
    wire [12:0] ddr_row;
    wire [1:0]  ddr_ba, ba_init, ba_rw;



    wire [12:0] ddr_addr;

    wire [2:0] ddr_rcw;
    wire ddr_dts;

    /* UNUSED
    reg dts, dts0, dts1;
    always @(posedge CK_w or posedge ~pll_clock_locked)
      if (~pll_clock_locked) {dts, dts0, dts1} = 0;
      else         {dts, dts0, dts1} = {dts0, dts1, ddr_dts};*/




    // -----  IO PADS & REGISTERS ------

    // pad regs
    reg [7:0] ddr_ext_dat_o;


      assign ddr_ba = init_active ? ba_init : ba_rw;
```

```
reg [12:0]  ddr_ext_addr;
reg [1:0]   ddr_ext_ba;
always @(posedge CK_w)
  {ddr_ext_addr, ddr_ext_ba} <= {ddr_addr, ddr_ba};

  assign DDR_addr_w = ddr_ext_addr;
  assign bank_w = ddr_ext_ba;


always @(posedge CK_w)
  {RAS_w,CAS_w,WE_w} <= ddr_rcw; // iob register

assign {UDM_w, LDM_w} = {1'b0, 1'b0}; //data mask always on *active low*


  wire init_active, ddr_ctrl_done;

   ddr_init init
  (
        .clk(CK_w),
        .rst(~pll_clock_locked),
        .ddr_ctrl_done(ddr_ctrl_done),

        .addr_o(ddr_cmd_addr), // addr[12:11] are added in the ddrctrl
        .ba_o(ba_init),
        .rcw_o(ddr_cmd_rcw),    // ras, cas, we
        .cmd_o(ddr_cmd_init),         // only one bit is required
        .init_active_o(init_active),
        .cke_o (CKE_w)

  );

  ddr_read_write rw
  (
        .ddr_cmd_o(ddr_cmd_rw),
        .ba_o(ba_rw),
        .burst_len_o(ddr_len),  // (number of words in burst) / 2
        .row_o(ddr_row),
        .column_o(),
        .test_done(GPIO_0[0]),

        .clk(CK_w),
        .rst(~pll_clock_locked),
        .init_active (init_active),
        .ddr_ctrl_done(ddr_ctrl_done)
  );


ddram_ctrl ddramctrl
  (
   // arbiter interface
   .arb_cmd_i   (ddr_cmd_rw),  //used to send read/write commands
   .cpu_cmd_i   (ddr_cmd_init), //used to send extended commands
  // external command
   .cmd_addr_i (ddr_cmd_addr),  // address for CMD_EXT
   .cmd_rcw_i  (ddr_cmd_rcw),   // ras, cas, we CMD_EXT
```

```
        .row_i   (ddr_row),
        .len_i   (ddr_len),
        .done_o  (ddr_ctrl_done),


        .ren_o   (reno), //used for DQS generation
        .wen_o   (), //not used

        // ddram interface
        .dts_o  (ddr_dts),  // data buffers tristate - NOT USED in this
project but might be useful for someone else
        .addr_o (ddr_addr),
        .rcw_o  (ddr_rcw), // ras, cas, we

        // system
        .CLK (CK_w),
        .RST (~pll_clock_locked)

        );

endmodule
```

DDRAM_CTRL.v

```
//
// file : ddram_ctrl.v
//

`include  "defs.v"

//
// commands ({cpu_cmd, arb_cmd}) :
//   000 : CMD_IDLE
//   001 : CMD_REFRESH
//   010 : CMD_WR
//   011 : CMD_RD
//   1xx : CMD_EXT, addr_ext_i & cmd_rcw_i are used
//


module ddram_ctrl
  (
   // arbiter interface
   input  wire  [1:0] arb_cmd_i,
   input  wire        cpu_cmd_i,  // 0 to 1 transition executes cpu cmd
   input  wire  [2:0] cmd_rcw_i,  // CMD_EXT ras, cas, we
   input  wire [10:0] cmd_addr_i, // CMD_EXT address
   input  wire [12:0] row_i,
   input  wire  [2:0] len_i,      // (length in columns) = 2*len_i
   output wire        done_o,


   // data in/out read/write enable
   output wire        ren_o,
```

```
   output wire        wen_o,

   // ddram interface
   output wire        dts_o,  // data buffers tristate
   output wire [12:0] addr_o,
   output wire  [2:0] rcw_o,  // ras, cas, we (command)

   // system
   input  wire        CLK,
   input  wire        RST

   );


    // state encodings

    parameter st_idle   = 12'b0000_0100_1111;  // 04f idle / done / addr =
row
    //-------
    parameter st_extcmd = 12'b0000_0111_0111;  // 077 external command
    parameter st_extdn  = 12'b0010_0100_1111;  // 24f idle / done / addr =
row
   //-------
    parameter st_w_act  = 12'b0000_0000_0011;  // 003 write activate
    parameter st_w_nop1 = 12'b0000_0001_0111;  // 017 single nop
    parameter st_w_nop2 = 12'b0010_0001_0111;  // 217 single nop
    parameter st_write  = 12'b0001_0001_0100;  // 114 write command
    parameter st_w_brst = 12'b0001_0000_0111;  // 107 write burst, nop's with
ren_o
    //-------
    parameter st_r_act  = 12'b0010_0100_0011;  // 243 read activate
    parameter st_r_nop1 = 12'b0010_0101_0111;  // 257 single nop
    parameter st_r_nop2 = 12'b0000_0101_0111;  // 057 single nop
    parameter st_read   = 12'b0010_1101_0101;  // 2d5 read command
    parameter st_r_brst = 12'b0010_1100_0111;  // 2c7 read burst, nop's with
wen_o
    parameter st_refr   = 12'b1000_0100_0001;  // 041 refresh
    parameter st_rfwtdn = 12'b1000_0100_0111;  // 041 wait rf done
    //-------
    parameter st_term   = 12'b0000_0100_0010;  // 042 //046 rd/wr burst
terminate



    wire [2:0] cmd = {cpu_cmd_i, arb_cmd_i[1:0]};

    // dts_o must be delayed for 2 clk, to account for VI fifo read cycle
    // and output (IO PAD) register
    wire  dts;
    reg dts_r0, dts_r1;

    always @(posedge CLK or posedge RST)
      if (RST) {dts_r1, dts_r0} <= 0;
      else     {dts_r1, dts_r0} <= {dts_r0, dts};

    assign dts_o = dts_r1;
```

```
    // 3 clk wen_o delay for cas latency + 3 clk for control & data pad regs.
    wire       wen;
    reg wen_r4, wen_r3, wen_r2, wen_r1, wen_r0;
    always @(posedge CLK or posedge RST)
      if (RST) {wen_r4, wen_r3, wen_r2, wen_r1, wen_r0} <= 0;
      else     {wen_r4, wen_r3, wen_r2, wen_r1, wen_r0} <= {wen_r3, wen_r2,
wen_r1, wen_r0, wen};

    assign wen_o = wen_r4;

   // address mux
    wire [1:0] amxs;  // address mux select


    wire [12:0] full_addr;

`ifdef AUTO_PRECHARGE
    assign      full_addr = {2'b00, 1'b1, 10'h000};  // start from the column
0
`else
    assign      full_addr = {2'b00, 1'b0, 10'h000};  // start from the column
0
`endif

    // FIXME add ce
    mux_4dir_reg #(13) addrmux
      (
       .O  (addr_o),
       .I0 (row_i),              // row
       .I1 (full_addr),          // 0, A10, column
       .I2 (13'b0),              // NOT USED
       .I3 ({2'b0, cmd_addr_i}), // address for CMD_EXT
       .S  (amxs),
       .CLK(CLK),
       .RST(RST)
       );

    // ras, cas, we mux
    wire [2:0] rcw; // {ras, cas, we} from the state machine
    wire       rcwsel = amxs[1];
    mux_2dir_reg #(3, 3'b111) rcwmux
      (.O({rcw_o}), .I0(rcw), .I1(cmd_rcw_i), .S(rcwsel), .CLK(CLK),
.RST(RST));

    // register done
    wire done;
    reg_1stg #(1,0) donereg
      (.O1(done_o), .I(done), .CLK(CLK), .RST(RST) ) ;

    // column counter (burst length)
    reg [2:0]  clmcnt;
    wire       clmdn;
    assign     clmdn = &(clmcnt[2:0]);

    always @(posedge CLK or posedge RST)
      if (RST)           clmcnt <= 0;
      else if (done)     clmcnt <= len_i + 1;
```

```
      else               clmcnt <= clmcnt - 1;

    wire       rf, rfdn;

    // delay for rfdn
    reg rf_r5, rf_r4, rf_r3, rf_r2, rf_r1, rf_r0;
    always @(posedge CLK or posedge RST)
      if (RST) { rf_r5, rf_r4, rf_r3, rf_r2, rf_r1, rf_r0  } <= 0;
      else     { rf_r5, rf_r4, rf_r3, rf_r2, rf_r1, rf_r0  } <= { rf_r4,
rf_r3, rf_r2, rf_r1, rf_r0, rf };

      assign rfdn = rf_r5;




    // FSM
    // state         | rf |spare |ren||wen|dts|amxs||done| rcw
    //-----------------------------------------------
    //  st_idle      |  0 |  00  | 0 || 0 | 1 | 00 || 1  | 111
    //
    //  st_extcmd    |  0 |  00  | 0 || 0 | 1 | 11 || 0  | 111
    //  st_extdn     |  0 |  01  | 0 || 0 | 1 | 00 || 1  | 111 // similar to
idle
    //
    //  st_w_act     |  0 |  00  | 0 || 0 | 0 | 00 || 0  | 011 // rcw =
active
    //  st_w_nop1    |  0 |  00  | 0 || 0 | 0 | 01 || 0  | 111
    //  st_w_nop2    |  0 |  01  | 0 || 0 | 0 | 01 || 0  | 111
    //  st_write     |  0 |  00  | 1 || 0 | 0 | 01 || 0  | 100 // rcw = write
    //  st_w_brst    |  0 |  00  | 1 || 0 | 0 | 00 || 0  | 111
    //
    //  st_r_act     |  0 |  01  | 0 || 0 | 1 | 00 || 0  | 011 // rcw =
active
    //  st_r_nop1    |  0 |  01  | 0 || 0 | 1 | 01 || 0  | 111
    //  st_r_nop2    |  0 |  00  | 0 || 0 | 1 | 01 || 0  | 111
    //  st_read      |  0 |  01  | 0 || 1 | 1 | 01 || 0  | 101 // rcw = read
    //  st_r_brst    |  0 |  01  | 0 || 1 | 1 | 00 || 0  | 111
    //
    //  st_refr      |  1 |  00  | 0 || 0 | 1 | 00 || 0  | 001 // rcw
    //  st_rfwtdn    |  1 |  00  | 0 || 0 | 1 | 00 || 0  | 111
    //
    //  st_term      |  0 |  00  | 0 || 0 | 1 | 00 || 0  | 010 /* 110 ? */
    //  st_nop2      |  0 |  11  | 0 || 0 | 1 | 01 || 0  | 111
    //
    //




    reg [11:0] state, next;

    wire [1:0] spare;

    assign     {rf, spare[1:0],  ren_o,/* */wen, dts, amxs[1:0],/* */done,
rcw[2:0]} = state[11:0];
```

```
always @(posedge CLK or posedge RST)
  if (RST) state <= st_idle;
  else     state <= next;

always @(state or clmdn or cmd[2:0] or rfdn)
  case (state)

    st_idle:
      case (cmd[2:0])
        3'b000:  next = st_idle;  // no cmd
        3'b001:  next = st_refr;  // refresh
        3'b010:  next = st_w_act; // write
        3'b011:  next = st_r_act; // read
        3'b100:  next = st_extcmd;
        3'b101:  next = st_extcmd;
        3'b110:  next = st_extcmd;
        3'b111:  next = st_extcmd;
      endcase // case(cmd[1:0])

    st_refr:     next = st_rfwtdn;
    st_rfwtdn:
      case (rfdn)
        1'b0:    next = st_rfwtdn;
        1'b1:    next = st_idle;
      endcase // case(rfdn)

    st_extcmd:   next = st_extdn;

    st_extdn:
      case (cmd[2])
        1'b1:    next = st_extdn;
        1'b0:    next = st_idle;
      endcase

    st_w_act:    next = st_w_nop1;
    st_w_nop1:   next = st_w_nop2;
    st_w_nop2:   next = st_write;
    st_write:
      case(clmdn)
        1'b0:    next = st_w_brst;
        1'b1:    next = st_term;

      endcase // case(clmdn)
    st_w_brst:
      case(clmdn)
        1'b0:    next = st_w_brst;
        1'b1:    next = st_term;

      endcase // case(clmdn)

    st_r_act:    next = st_r_nop1;
    st_r_nop1:   next = st_r_nop2;
    st_r_nop2:   next = st_read;
    st_read:
      case(clmdn)
        1'b0:    next = st_r_brst;
```

```
            1'b1:    next = st_term;

          endcase // case(clmdn)
        st_r_brst:
          case(clmdn)
            1'b0:    next = st_r_brst;
            1'b1:    next = st_term;

          endcase // case(clmdn)

        default:     next = st_idle;

      endcase // case(state)



endmodule // ddram_ctrl
```

DDRINIT.v

```
`include "defs.v"



module ddr_init
(
      input wire clk,
      input wire rst,
      input wire ddr_ctrl_done,

      output wire [10:0] addr_o, // addr[12:11] are added in the ddrctrl
      output wire [1:0] ba_o,
      output wire [2:0] rcw_o,  // ras, cas, we
      output wire cmd_o,         // only one bit is required
      output wire init_active_o,
      output wire cke_o

);

      reg ddr_cmd, init_active;
      reg [1:0] ddr_ba;
      reg [2:0] ddr_cmd_rcw;
      reg [1:0] Cke_cmd;
      reg [10:0] ddr_cmd_addr;



      assign cmd_o = ddr_cmd;
      assign init_active_o = init_active;
      assign ba_o = ddr_ba;
      assign rcw_o = ddr_cmd_rcw;
      assign addr_o = ddr_cmd_addr;
      assign cke_o  = Cke_cmd;
```

```
    parameter CMD_IDLE = 1'b0,
              CMD_EXT  = 1'b1;

      //counters for wait states
      reg [8:0] counter1, counter3;
      reg [3:0] counter2;

      reg [5:0] state, prev_state;
      parameter st_start = 5'd0, st_idle1 = 5'd1, st_pre1 = 5'd2, st_count9 =
5'd3,
                  st_setemr = 5'd4, st_setmr = 5'd5, st_pre2 = 5'd6,
st_auto1 = 5'd7,
                  st_auto2 = 5'd8, st_auto3 = 5'd9, st_count200 = 5'd10,
st_wait = 5'd11,
                  st_done = 5'd12;

always @(posedge clk)
begin
      if(rst)
      begin
            state <= st_start;
            init_active <= 1'b1;
            ddr_ba <= 2'b0;
            {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} <= 15'b0;
            Cke_cmd <= 2'b0;
            counter1 <= 9'd0;
            counter2 <= 4'd0;
      end
      else begin
            case (state)
                  st_start: //init initialized - first thing is wait 500
cycles
                  begin
                        if(counter1 == 500)
                        begin
                              state <= st_idle1;
                              counter1 <= 9'd0;
                        end else
                        begin
                              counter1 <= counter1 + 9'b1;
                    end
                  end //end st_start

                st_idle1: //bring CKE high and execute idle command
                  begin
                        Cke_cmd <= 2'b01;
                        ddr_cmd <= CMD_IDLE;
                        state <= st_pre1;
                  end //end st_idle 1

                  st_count9:
                  begin
                        if(counter2 == 4'd9) //done counting 9 go to next
state
                        begin
                              counter2 <= 4'b0;
```

```
                              case(prev_state)
                                    st_pre1: state <= st_setemr;
                                    st_setemr: state <= st_setmr;
                                    st_setmr: state <= st_pre2;
                                    st_pre2: state <= st_auto1;
                                    st_auto1: state <= st_auto2;
                                    st_auto2: state <= st_auto3;
                              endcase
                        end
                        else begin
                              ddr_cmd <= CMD_IDLE;
                              counter2 <= counter2 + 4'b1;
                        end

                  end //end st_count9

                  st_pre1: //pre charge all banks
                  begin
                        {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} <= {CMD_EXT,
3'b010, 11'b100_0000_0000};
                        state <= st_count9;
                        counter2 <= 4'b0;
                        prev_state <= state;
                  end  //end st_pre1

                  st_setemr:
                  begin
                        ddr_ba <= 2'b01; //select extended mr, set it to all
zeros
                        {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = {CMD_EXT,
3'b000, 11'b000_0000_0000};
                        state <= st_count9;
                        prev_state <= state;
                  end //end setemr

                  st_setmr: //set mode register, defines normal operating
parameters
                  begin
                         ddr_ba = 2'b00;
                        // addr[N:7] - mode: ..00000 - normal op, ....10
normal op, reset  dll
                        // addr[6:4] - cas latency: 010 - 2, 011 - 3 (for -5B
only), 110 - 2.5
                        // addr[3]   - burst type:  0 - sequential, 1 -
interleave
                        // addr[2:0] - burst len: 001 - 2, 010 - 4, 011 - 8
                        {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = {CMD_EXT,
3'b000, 11'b0010_010_0_001};
                        state <= st_count9;
                        prev_state <= state;
                  end //end set mr

                  st_pre2:
                  begin
                         {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} <= {CMD_EXT,
3'b010, 11'b100_0000_0000};
```

```
                        state <= st_count9;
                        prev_state <= state;
                end //end pre2

                st_auto1: //auto referesh all banks
                begin
                       {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} <= {CMD_EXT,
3'b001, 11'b000_0000_0000};
                        state <= st_count9;
                        prev_state <= state;
                end //end auto1

                st_auto2: //auto referesh all banks
                begin
                       {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} <= {CMD_EXT,
3'b001, 11'b000_0000_0000};
                        state <= st_count9;
                        prev_state <= state;
                end //end auto2

                st_auto3: //auto referesh all banks
                begin
                       {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} <= {CMD_EXT,
3'b001, 11'b000_0000_0000};
                        state <= st_count200;
                        counter3 <= 9'd0;
                end //end auto3

                st_count200: //wait 200 cycles
                begin
                      if(counter3 == 9'd200)
                      begin
                            state <= st_wait;
                      end else
                      begin
                            ddr_cmd <= CMD_IDLE;
                            counter3 <= counter3 + 9'd1;
                      end
                end //end count 200

                st_wait:
                begin
                      if(ddr_ctrl_done)
                      begin
                            init_active <= 1'b0;
                            state <= st_done;
                      end
                end



          endcase //end case


      end //end if

      /*
```

```
            initial begin

                init_active = 1;
              ddr_ba = 0;
              {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = 0;
                Cke_cmd = 0;

                wait (!rst);

              #1;

             // #(`DDR_CLK_PERIOD * 500); // wait 10 us ! after ddr_clk is
functional
              //bring CKE high and execute idle command for 1 cycle

              Cke_cmd = 1;
              ddr_cmd = CMD_IDLE;
              #(`DDR_CLK_PERIOD * 1);
              // precharge all banks
              $display ("precharge all banks at %t", $time);
              {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = {CMD_EXT,  3'b010,
11'b100_0000_0000};
              #(`DDR_CLK_PERIOD * 1) ddr_cmd = CMD_IDLE;// REMOVE CMD !!!
              #(`DDR_CLK_PERIOD * 9);

              //set extended mode register
              ddr_ba = 2'b01; //select extended mr
              {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = {CMD_EXT,  3'b000,
11'b000_0000_0000};
               #(`DDR_CLK_PERIOD * 1)
                ddr_cmd = CMD_IDLE;// REMOVE CMD !!!
              #(`DDR_CLK_PERIOD * 9);

              //set mode register
              ddr_ba = 2'b00;
              // addr[9:7] - mode: ..00000 - normal op, ....10 normal op,
reset  dll
              // addr[6:4] - cas latency: 010 - 2, 011 - 3 (for -5B only),
110 - 2.5
              // addr[3]   - burst type:  0 - sequential, 1 - interleave
              // addr[2"0] - burst len: 001 - 2, 010 - 4, 011 - 8
               {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = {CMD_EXT,  3'b000,
11'b0010_010_0_001};
              #(`DDR_CLK_PERIOD * 1) ddr_cmd = CMD_IDLE;// REMOVE CMD !!!
              #(`DDR_CLK_PERIOD * 9);
              // precharge all banks

                  // $dumpon;

              $display ("precharge all banks at %t", $time);
              {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = {CMD_EXT,  3'b010,
11'b100_0000_0000};
              #(`DDR_CLK_PERIOD * 1) ddr_cmd = CMD_IDLE;// REMOVE CMD !!!
              #(`DDR_CLK_PERIOD * 9);
              // autorefresh
              {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = {CMD_EXT,  3'b001,
11'b000_0000_0000};
```

```
              #(`DDR_CLK_PERIOD * 1) ddr_cmd = CMD_IDLE;// REMOVE CMD !!!
              #(`DDR_CLK_PERIOD * 9);
              // autorefresh
              {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = {CMD_EXT,  3'b001,
11'b000_0000_0000};
              #(`DDR_CLK_PERIOD * 1) ddr_cmd = CMD_IDLE;// REMOVE CMD !!!
              #(`DDR_CLK_PERIOD * 9);
              // autorefresh
              {ddr_cmd, ddr_cmd_rcw, ddr_cmd_addr} = {CMD_EXT,  3'b001,
11'b000_0000_0000};
              #(`DDR_CLK_PERIOD * 1) ddr_cmd = CMD_IDLE;// REMOVE CMD !!!
              #(`DDR_CLK_PERIOD * 9);

              //$dumpoff;

                  #(`DDR_CLK_PERIOD * 200);

              //$dumpon;
              wait (ddr_ctrl_done);

                  init_active = 0;
                  $finish;
            end*/
end
endmodule
```

DDR_READ_WRITE.v

```
/* Author: Denis Pyatkov, May 2008
A sample read/write module for the DDR RAM controller. Performs 2 writes
followed by 2 reads.

*/

module ddr_read_write
(

      output wire [1:0] ddr_cmd_o, //indicates to controller whehther read or
write is needed
      output wire [1:0] ba_o, //bank address
      output wire [2:0] burst_len_o,  // (number of words in burst) / 2
      output wire [12:0] row_o, //row number
      output wire [7:0] column_o, //column number
      output wire test_done, //ends simulation, IS NOT NEEDED WHEN CONTROLLER
IS ON FPGA

      input wire clk,
      input wire rst,
      input wire init_active, //lock for statemachine while RAM
initialization is performed
      input wire ddr_ctrl_done //notifies this module when controller has
returned to idle state
);


      assign burst_len_o = 3'h1; // two words for ddr
```

```
	reg [1:0] ba;
	reg [12:0] row;
	reg [7:0] column;

	assign ba_o = ba;
	assign row_o = row;
	assign column_o = column;

	// column/row/ba multiplexor
	wire mux_sel;

	// test two ddr locations
	always @(mux_sel)
		if (mux_sel == 1)
			{ba, row, column} = {2'b00, 13'h5, 8'h2};
		else
			{ba, row, column} = {2'b01, 13'h17, 8'h31};



	// state           | spare  | tst_done |   ddr_cmd | mux_sel |
	//-------------------------------------------------------------
	// st_idle         |   000  |    0     |   00    |    0
	// st_wr1          |   000  |    0     |   10    |    0
	// st_wr1_wt_ndone |   001  |    0     |   00    |    0
	// st_wr1_wt_done  |   010  |    0     |   00    |    0
	// st_wr2          |   000  |    0     |   10    |    1
	// st_wr2_wt_ndone |   000  |    0     |   00    |    1
	// st_wr2_wt_done  |   001  |    0     |   00    |    1
	// st_rd1          |   000  |    0     |   11    |    0
	// st_rd1_wt_ndone |   011  |    0     |   00    |    0
	// st_rd1_wt_done  |   100  |    0     |   00    |    0
	// st_rd2          |   000  |    0     |   11    |    1
	// st_rd2_wt_ndone |   010  |    0     |   00    |    1
	// st_rd2_wt_done  |   011  |    0     |   00    |    1
	// st_stop         |   000  |    1     |   00    |    0


	parameter st_idle         =  7'b000_0000; // 0x00 - hex number of state
for debugging in waveform viewer
	parameter st_wr1          =  7'b000_0100; // 0x04
	parameter st_wr1_wt_ndone =  7'b001_0000; // 0x10
	parameter st_wr1_wt_done  =  7'b010_0000; // 0x20
	parameter st_wr2          =  7'b000_0101; // 0x05
	parameter st_wr2_wt_ndone =  7'b000_0001; // 0x01
	parameter st_wr2_wt_done  =  7'b001_0001; // 0x11
	parameter st_rd1          =  7'b000_0110; // 0x06
	parameter st_rd1_wt_ndone =  7'b011_0000; // 0x30
	parameter st_rd1_wt_done  =  7'b100_0000; // 0x40
	parameter st_rd2          =  7'b000_0111; // 0x07
	parameter st_rd2_wt_ndone =  7'b010_0001; // 0x21
	parameter st_rd2_wt_done  =  7'b011_0001; // 0x31
	parameter st_stop         =  7'b000_1000; // 0x08

	reg [6:0] state, next;
```

```
assign {test_done, ddr_cmd_o, mux_sel} = state[3:0];


always @(posedge clk or posedge rst)
      if (rst) state <= st_idle;
      else     state <= next;

always @(state or init_active or ddr_ctrl_done)
case(state)
st_idle:
      if (init_active)   next = st_idle;
      else               next = st_wr1;

st_wr1:                next = st_wr1_wt_ndone;

st_wr1_wt_ndone:
      if (ddr_ctrl_done) next = st_wr1_wt_ndone;
      else               next = st_wr1_wt_done;

st_wr1_wt_done:
      if (ddr_ctrl_done) next = st_wr2;
      else               next = st_wr1_wt_done;

st_wr2:                next = st_wr2_wt_ndone;

st_wr2_wt_ndone:
      if (ddr_ctrl_done) next = st_wr2_wt_ndone;
      else               next = st_wr2_wt_done;

st_wr2_wt_done:
      if (ddr_ctrl_done) next = st_rd1;
      else               next = st_wr2_wt_done;

st_rd1:                next = st_rd1_wt_ndone;

st_rd1_wt_ndone:
      if (ddr_ctrl_done) next = st_rd1_wt_ndone;
      else               next = st_rd1_wt_done;

st_rd1_wt_done:
      if (ddr_ctrl_done) next = st_rd2;
      else               next = st_rd1_wt_done;

st_rd2:                next = st_rd2_wt_ndone;

st_rd2_wt_ndone:
      if (ddr_ctrl_done) next = st_rd2_wt_ndone;
      else               next = st_rd2_wt_done;

st_rd2_wt_done:
      if (ddr_ctrl_done) next = st_stop;
      else               next = st_rd2_wt_done;

st_stop: begin
      next = st_stop;
end
```

```
        default next = st_idle;
        endcase

endmodule
```