## Module-1. ddr_user_if.v

```
module ddr_user_if (
    input  wire        clk,
    input  wire        rst_n,
    // User Side Interface
    input  wire [31:0] app_addr,
    input  wire [2:0]  app_cmd,   // 000=Read, 001=Write
    input  wire        app_en,
    output reg         app_rdy,
    input  wire [63:0] app_wdf_data,
    input  wire        app_wdf_wren,
    output reg         app_wdf_rdy,
    output reg [63:0]  app_rd_data,
    output reg         app_rd_data_valid
);

    // Internal pipeline registers for Read Latency
    // We use a 3-bit pipe to ensure data and valid signals align perfectly
    reg [2:0] rd_valid_pipe;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            app_rdy          <= 1'b0;
            app_wdf_rdy      <= 1'b0;
            app_rd_data      <= 64'b0;
        app_rd_data_valid <= 1'b0;
        rd_valid_pipe     <= 3'b000;
    end else begin
        // 1. Handshaking Logic
        // In a simple model, we are always ready after reset.
        app_rdy     <= 1'b1;
        app_wdf_rdy <= 1'b1;

        // 2. Read Latency Pipeline
        // Shift the "Read Enable" signal through the pipe
        // Bit 0: Request sampled
        // Bit 1: Processing
        // Bit 2: Data Out
        rd_valid_pipe <= {rd_valid_pipe[1:0], (app_en && app_cmd == 3'b000)};

        // 3. Synchronized Data and Valid Output
        // app_rd_data_valid and app_rd_data now update on the same clock edge
        app_rd_data_valid <= rd_valid_pipe[2];

        if (rd_valid_pipe[2]) begin
            app_rd_data <= 64'hDEADBEEFCAFEBABE; // Simulated Data from Memory
        end else begin
            app_rd_data <= 64'h0; // Clear bus when not in use
        end
    end
end
```

```
endmodule
```

## Testbench- ddr_user_if_tb.v

```
module ddr_user_if_tb;

   // Clock and Reset

   reg        clk;

   reg        rst_n;


   // Command/Address Interface

   reg [31:0]  app_addr;

   reg [2:0]   app_cmd;

   reg        app_en;

   wire       app_rdy;


   // Write Data Interface

   reg [63:0]  app_wdf_data;

   reg        app_wdf_wren;

   wire       app_wdf_rdy;


   // Read Data Interface

   wire [63:0] app_rd_data;

   wire       app_rd_data_valid;


   // UUT Instantiation (Explicit Mapping)

   ddr_user_if uut (

      .clk             (clk),
    .rst_n            (rst_n),
    .app_addr         (app_addr),
    .app_cmd          (app_cmd),
    .app_en           (app_en),
    .app_rdy          (app_rdy),
    .app_wdf_data     (app_wdf_data),
    .app_wdf_wren     (app_wdf_wren),
    .app_wdf_rdy      (app_wdf_rdy),
    .app_rd_data      (app_rd_data),
    .app_rd_data_valid (app_rd_data_valid)
);

// Robust Clock Generation (100MHz)
initial clk = 0;
always #5 clk = ~clk;

initial begin
    // --- 1. Initialization ---
    rst_n       = 0;
    app_en      = 0;
    app_wdf_wren = 0;
    app_addr    = 32'b0;
    app_cmd     = 3'b0;
    app_wdf_data = 64'b0;

    // Release Reset
```

```
#50 rst_n = 1;

// Wait for UUT to be ready
wait(app_rdy && app_wdf_rdy);
repeat(2) @(posedge clk);

// --- 2. Aligned Write Cycle ---
// Best practice: Drive signals on the edge but ensure
// they are stable for the next rising edge
@(posedge clk);
if (app_rdy && app_wdf_rdy) begin
   app_addr     <= 32'h0000_1000;
   app_cmd      <= 3'b001; // Write Command
   app_en       <= 1'b1;
   app_wdf_data <= 64'h1122334455667788;
   app_wdf_wren <= 1'b1;
end

@(posedge clk);
app_en       <= 1'b0;
app_wdf_wren <= 1'b0;

repeat(5) @(posedge clk); // Idle gap

// --- 3. Read Cycle ---
@(posedge clk);
```

```
        if (app_rdy) begin
          app_addr <= 32'h0000_1000;
          app_cmd  <= 3'b000; // Read Command
          app_en   <= 1'b1;
        end

        @(posedge clk);
        app_en   <= 1'b0;

        // --- 4. Result Observation ---
        // Wait for the valid strobe (considering 2-3 cycle latency)
        wait(app_rd_data_valid);
        $display("Time: %t | Read Data Validated: %h", $time, app_rd_data);

        repeat(5) @(posedge clk);

        $display("Testbench Completed Successfully.");
        $finish;
    end

endmodule
```

## Module-2. ddr_addr_decode.v

```
module ddr_addr_decode (
    input  wire [31:0] addr,

    output wire [13:0] row,
    output wire [2:0]  bank,
    output wire [9:0]  col
);

    // Address slicing
    assign row  = addr[31:18];
    assign bank = addr[17:15];
    assign col  = addr[14:5];

endmodule
```

**Testbench- ddr_add_decode_tb.v**

```
module ddr_addr_decode_tb;

    reg  [31:0] addr;
    wire [13:0] row;
    wire [2:0]  bank;
    wire [9:0]  col;

    // DUT
    ddr_addr_decode dut (
        .addr(addr),
        .row(row),
        .bank(bank),
        .col(col)
    );
```

```
initial begin
  // -------------------------
  // TEST 1
  // -------------------------
  addr = 32'h1234_5678;
  #1;
  if (row  !== addr[31:18] ||
     bank !== addr[17:15] ||
     col  !== addr[14:5])
     $display("      TEST 1 FAILED");
  else
     $display("      TEST 1 PASSED");

  // -------------------------
  // TEST 2
  // -------------------------
  addr = 32'h8765_4321;
  #1;
  if (row  !== addr[31:18] ||
     bank !== addr[17:15] ||
     col  !== addr[14:5])
     $display("      TEST 2 FAILED");
  else
     $display("      TEST 2 PASSED");

  // -------------------------
```

```
        // TEST 3: Bank boundary
        // -------------------------
        addr = 32'h0003_8000; // bank should change
        #1;
        if (bank !== addr[17:15])
          $display("      TEST 3 FAILED");
        else
          $display("      TEST 3 PASSED");

        $display("      Address Decode Tests Complete");
        $finish;
    end

endmodule
```

## Module-3. ddr_bank_state.v

```
module ddr_bank_state (
    input  wire        clk,
    input  wire        rst,
    input  wire        req_valid,
    input  wire [2:0]  req_bank,
    input  wire [13:0] req_row,
    output reg         act_cmd,
    output reg         pre_cmd,
    output reg  [2:0]  cmd_bank,
    output reg  [13:0] cmd_row,
    output wire       row_open,

    output wire       row_hit

);


    // Internal State: 8 Banks (for 3-bit req_bank)

    reg [13:0] open_rows [0:7];

    reg [7:0]  bank_active;

    integer i;


    // Check if the requested bank is open and if it's the correct row

    assign row_open = bank_active[req_bank];

    assign row_hit  = row_open && (open_rows[req_bank] == req_row);


    always @(posedge clk or posedge rst) begin

        if (rst) begin

            act_cmd  <= 1'b0;

            pre_cmd  <= 1'b0;

            cmd_bank <= 3'b0;

            cmd_row  <= 14'b0;

            bank_active <= 8'b0;

            for (i = 0; i < 8; i = i + 1) open_rows[i] <= 14'b0;

        end else if (req_valid) begin

            cmd_bank <= req_bank;

            cmd_row  <= req_row;


            if (!row_open) begin
            // Case 1: Bank is closed. Issue ACTIVATE.
            act_cmd <= 1'b1;
            pre_cmd <= 1'b0;
            bank_active[req_bank] <= 1'b1;
            open_rows[req_bank]   <= req_row;
          end else if (!row_hit) begin
            // Case 2: Bank is open but wrong row. Issue PRECHARGE then ACTIVATE.
            pre_cmd <= 1'b1;
            act_cmd <= 1'b0; // In a real controller, ACT happens after PRE delay
            open_rows[req_bank]   <= req_row;
          end else begin
            // Case 3: Row Hit.
            act_cmd <= 1'b0;
            pre_cmd <= 1'b0;
          end
        end else begin
          act_cmd <= 1'b0;
          pre_cmd <= 1'b0;
        end
    end
endmodule
```

## Testbench- ddr_bank_state_tb.v

```
module ddr_bank_state_tb;
    reg clk;
    reg rst;
    reg req_valid;
reg [2:0] req_bank;
reg [13:0] req_row;

wire act_cmd, pre_cmd;
wire [2:0] cmd_bank;
wire [13:0] cmd_row;
wire row_open, row_hit;

ddr_bank_state uut (
   .clk(clk), .rst(rst), .req_valid(req_valid),
   .req_bank(req_bank), .req_row(req_row),
   .act_cmd(act_cmd), .pre_cmd(pre_cmd),
   .cmd_bank(cmd_bank), .cmd_row(cmd_row),
   .row_open(row_open), .row_hit(row_hit)
);

// Clock generation (100MHz)
always #5 clk = ~clk;

initial begin
   // Initialize and Reset
   clk = 0; rst = 1; req_valid = 0; req_bank = 0; req_row = 0;
   #20 rst = 0;

   // T=30ns: Request Bank 0, Row 0x000A
   #10 req_valid = 1; req_bank = 0; req_row = 14'h000A;
```

```
        // T=50ns: Change Request to Bank 3, Row 0x000A (Bank Switch)
        #20 req_bank = 3; req_row = 14'h000A;

        // T=80ns: Change Request to Bank 3, Row 0x0014 (Row Miss)
        #30 req_row = 14'h0014;

        #50 $finish;
    end
endmodule
```

## Module-4. Ddr_timing_ctrl.v

```
module ddr_timing_ctrl (
    input  wire clk,
    input  wire rst,
    input  wire act_cmd,
    input  wire pre_cmd,
    input  wire rd_cmd,
    input  wire wr_cmd,
    output wire timing_ok
);

    // Timing parameters (in clock cycles)
    parameter TRCD = 3; // Activate to Read/Write delay
    parameter TRP  = 3; // Precharge to Activate delay

    reg [3:0] wait_cnt;
    // timing_ok is high only when the counter is 0
    assign timing_ok = (wait_cnt == 4'b0);

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            // FIX: Initialize counter to 0 to remove the red 'X'
            wait_cnt <= 4'b0;
        end else begin
            if (act_cmd) begin
                // Load counter for tRCD when ACT is issued
                wait_cnt <= TRCD;
            end else if (pre_cmd) begin
                // Load counter for tRP when PRE is issued
                wait_cnt <= TRP;
            end else if (wait_cnt > 0) begin
                // Count down every cycle
                wait_cnt <= wait_cnt - 1;
            end
        end
    end

endmodule
```

## Testbench- ddr_timing_ctrl_tb.v

```
module ddr_timing_ctrl_tb;
    reg clk;
    reg rst;
    reg act_cmd;
    reg pre_cmd;
    reg rd_cmd;
    reg wr_cmd;
    wire timing_ok;

    // Instantiate Unit Under Test
    ddr_timing_ctrl uut (
        .clk(clk), .rst(rst),
        .act_cmd(act_cmd), .pre_cmd(pre_cmd),
        .rd_cmd(rd_cmd), .wr_cmd(wr_cmd),
        .timing_ok(timing_ok)
    );

    // Clock Generation: 10ns period (100MHz)
    always #5 clk = ~clk;

    initial begin
        // Initialize Signals
        clk = 0; rst = 1;
        act_cmd = 0; pre_cmd = 0; rd_cmd = 0; wr_cmd = 0;

        // Reset Pulse
        #15 rst = 0;
```

```
        // T=30ns: Issue ACTIVATE Command
        #15 act_cmd = 1;
        #10 act_cmd = 0; // Pulse for 1 cycle

        // T=80ns: Issue PRECHARGE Command
        #40 pre_cmd = 1;
        #10 pre_cmd = 0; // Pulse for 1 cycle

        #50 $finish;
    end
endmodule
```

## Module-5. Ddr_cmd_fsm.v

```
module ddr_cmd_fsm (
    input  wire        clk,
    input  wire        rst_n,
    input  wire        start_init,
    input  wire        req_read,
    input  wire        req_write,
    output reg  [2:0]  current_state_out, // For debugging
    output reg         cmd_ready
);

    // State Encoding
    parameter IDLE      = 3'b000;
    parameter INIT      = 3'b001;
    parameter READY     = 3'b010;
parameter ACTIVATE  = 3'b011;
parameter RD_WR     = 3'b100;
parameter PRECHARGE = 3'b101;

reg [2:0] state, next_state;
reg [3:0] timer; // Simple counter for timing delays

// State Transition Logic
always @(posedge clk or negedge rst_n) begin
  if (!rst_n)
    state <= IDLE;
  else
    state <= next_state;
end

// Next State Logic
always @(*) begin
  next_state = state;
  case (state)
    IDLE: begin
      if (start_init) next_state = INIT;
    end
    INIT: begin
      if (timer == 4'd10) next_state = READY; // Simulated init delay
    end
    READY: begin
```

```
            if (req_read || req_write) next_state = ACTIVATE;
        end
        ACTIVATE: begin
            next_state = RD_WR;
        end
        RD_WR: begin
            next_state = PRECHARGE;
        end
        PRECHARGE: begin
            next_state = READY;
        end
        default: next_state = IDLE;
    endcase
end

// Output Logic and Timer
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        timer <= 0;
        cmd_ready <= 0;
        current_state_out <= IDLE;
    end else begin
        current_state_out <= state;

        if (state == INIT)
            timer <= timer + 1;
            else
                timer <= 0;

            cmd_ready <= (state == READY);
        end
    end

endmodule
```

## Testbench- ddr_cmd_fsm_tb.v

```
module ddr_cmd_fsm_tb();

    reg clk;
    reg rst_n;
    reg start_init;
    reg req_read;
    reg req_write;

    wire [2:0] current_state;
    wire cmd_ready;

    // Instantiate the Unit Under Test (UUT)
    ddr_cmd_fsm uut (
        .clk(clk),
        .rst_n(rst_n),
    .start_init(start_init),
    .req_read(req_read),
    .req_write(req_write),
    .current_state_out(current_state),
    .cmd_ready(cmd_ready)
);

// Clock Generation (100MHz)
always #5 clk = ~clk;

initial begin
    // Initialize Inputs
    clk = 0;
    rst_n = 0;
    start_init = 0;
    req_read = 0;
    req_write = 0;

    // Reset Sequence
    #20 rst_n = 1;

    // Start Initialization
    #10 start_init = 1;
    #10 start_init = 0;

    // Wait for READY state
```

```
        wait(cmd_ready == 1);

        $display("FSM is Ready at time %t", $time);


        // Issue a Read Request

        #10 req_read = 1;

        #10 req_read = 0;


        // Observe transitions through ACTIVATE -> RD_WR -> PRECHARGE

        #50;


        $display("Simulation Finished");

        $finish;

    end


    initial begin

        $monitor("Time=%0t | State=%b | Ready=%b", $time, current_state, cmd_ready);

    end
endmodule
```

## TOP Module- ddr_top.v

```
// ddr_top.v
// Top-level that integrates the modules from your upload:
//  - ddr_user_if      (active-low reset: rst_n)
//  - ddr_addr_decode
//  - ddr_bank_state   (active-high reset: rst)
//  - ddr_cmd_fsm      (active-low reset: rst_n)
//  - ddr_timing_ctrl  (active-high reset: rst)
//
// NOTE: this top module does minimal glue logic to keep existing module code unchanged.
//       It pulses start_init once after reset release to trigger initialization in ddr_cmd_fsm.

module ddr_top (
    input  wire        clk,
    input  wire        rst_n,        // global active-low reset (external)

    // User application interface (same as your ddr_user_if)
    input  wire [31:0]  app_addr,
    input  wire [2:0]   app_cmd,      // 000 = read, 001 = write (per your file)
    input  wire        app_en,
    output wire        app_rdy,
    input  wire [63:0]  app_wdf_data,
    input  wire        app_wdf_wren,
    output wire        app_wdf_rdy,
    output wire [63:0]  app_rd_data,
    output wire        app_rd_data_valid,

    // debug / status outputs (optional)
    output wire [2:0]   dbg_state,
    output wire        dbg_timing_ok
);

    // Convert reset polarity for modules that expect active-high reset
    wire rst = ~rst_n; // active-high reset for bank_state & timing_ctrl

    // ------------------------------------------------------------------
    // Internal wires
    // ------------------------------------------------------------------
    wire [13:0] row;
    wire [2:0]  bank;
    wire [9:0]  col;

    wire       user_app_rdy;
    wire       user_app_wdf_rdy;
    wire [63:0] user_app_rd_data;
    wire       user_app_rd_valid;

    // command handshake
    wire       cmd_ready;
    wire [2:0]  fsm_state;

    // bank state outputs (commands decided by bank_state)
    wire       bank_act_cmd;
    wire       bank_pre_cmd;
    wire [2:0]  bank_cmd_bank;
    wire [13:0] bank_cmd_row;
    wire       row_open;
    wire       row_hit;
```

```
// timing controller inputs
reg        rd_cmd;
reg        wr_cmd;
wire       timing_ok;

// derived request valid: only forward request when user IF ready and cmd_fsm is READY
wire req_valid = app_en & user_app_rdy & cmd_ready;

// latch whether current request is a write (so RD/WR pulses can be asserted at RD_WR state)
reg pending_is_write;
always @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
      pending_is_write <= 1'b0;
   end else begin
      if (req_valid) begin
         pending_is_write <= (app_cmd == 3'b001);
      end
   end
end

// When FSM enters RD_WR state we generate rd_cmd/wr_cmd for a single cycle.
// FSM states in your file: RD_WR = 3'b100
localparam [2:0] FSM_RD_WR = 3'b100;

always @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
      rd_cmd <= 1'b0;
      wr_cmd <= 1'b0;
   end else begin
      // default de-assert
      rd_cmd <= 1'b0;
      wr_cmd <= 1'b0;

      if (fsm_state == FSM_RD_WR) begin
         if (pending_is_write)
            wr_cmd <= 1'b1;
         else
            rd_cmd <= 1'b1;
      end
   end
end

// ---------------------------------------------------------------
// Instantiations (from your uploaded files)
// ---------------------------------------------------------------
// User Interface (active-low reset)
ddr_user_if u_user_if (
   .clk              (clk),
   .rst_n            (rst_n),
   .app_addr         (app_addr),
   .app_cmd          (app_cmd),
   .app_en           (app_en),
   .app_rdy          (user_app_rdy),
   .app_wdf_data     (app_wdf_data),
   .app_wdf_wren     (app_wdf_wren),
   .app_wdf_rdy      (user_app_wdf_rdy),
   .app_rd_data      (user_app_rd_data),
   .app_rd_data_valid (user_app_rd_valid)
);

// expose user_if outputs
assign app_rdy          = user_app_rdy;
assign app_wdf_rdy      = user_app_wdf_rdy;
assign app_rd_data      = user_app_rd_data;
assign app_rd_data_valid = user_app_rd_valid;

// Address decode (combinational)
ddr_addr_decode u_addr_dec (
   .addr (app_addr),
   .row  (row),
   .bank (bank),
   .col  (col)
);

// Bank state: decides ACT / PRE based on req_valid + requested row/bank
ddr_bank_state u_bank_state (
   .clk      (clk),
   .rst      (rst),       // active-high reset as in your file
   .req_valid (req_valid),
   .req_bank  (bank),
   .req_row   (row),
   .act_cmd   (bank_act_cmd),
   .pre_cmd   (bank_pre_cmd),
   .cmd_bank  (bank_cmd_bank),
   .cmd_row   (bank_cmd_row),
   .row_open  (row_open),
   .row_hit   (row_hit)
);

// Command FSM (active-low reset)
// We'll pulse start_init once after reset release
```

```
    reg start_init;
    reg init_pulse_done;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            start_init <= 1'b0;
            init_pulse_done <= 1'b0;
        end else begin
            if (!init_pulse_done) begin
                // assert start_init for exactly one cycle right after reset release
                start_init <= 1'b1;
                init_pulse_done <= 1'b1;
            end else begin
                start_init <= 1'b0;
            end
        end
    end

    ddr_cmd_fsm u_cmd_fsm (
        .clk              (clk),
        .rst_n            (rst_n),
        .start_init       (start_init),
        .req_read         (req_valid & (app_cmd == 3'b000)),
        .req_write        (req_valid & (app_cmd == 3'b001)),
        .current_state_out (fsm_state),
        .cmd_ready        (cmd_ready)
    );

    // Timing controller (active-high reset)
    ddr_timing_ctrl #(
        .TRCD (3),
        .TRP  (3)
    ) u_timing (
        .clk     (clk),
        .rst     (rst),
        .act_cmd (bank_act_cmd),   // commands decided by bank_state
        .pre_cmd (bank_pre_cmd),
        .rd_cmd  (rd_cmd),
        .wr_cmd  (wr_cmd),
        .timing_ok (timing_ok)
    );

    // debug outputs
    assign dbg_state     = fsm_state;
    assign dbg_timing_ok = timing_ok;

endmodule
```

## Testbench- ddr_top_tb.v

```
module ddr_top_tb;
   reg clk;
   reg rst_n;

   reg  [31:0] app_addr;
   reg  [2:0]  app_cmd;
   reg         app_en;
   wire        app_rdy;
   reg  [63:0] app_wdf_data;
   reg         app_wdf_wren;
   wire        app_wdf_rdy;
   wire [63:0] app_rd_data;
   wire        app_rd_data_valid;

   wire [2:0] dbg_state;
   wire       dbg_timing_ok;

   // Instantiate top
   ddr_top uut (
      .clk              (clk),
      .rst_n            (rst_n),
      .app_addr         (app_addr),
      .app_cmd          (app_cmd),
      .app_en           (app_en),
      .app_rdy          (app_rdy),
      .app_wdf_data     (app_wdf_data),
      .app_wdf_wren     (app_wdf_wren),
      .app_wdf_rdy      (app_wdf_rdy),
      .app_rd_data      (app_rd_data),
      .app_rd_data_valid (app_rd_data_valid),
      .dbg_state        (dbg_state),
      .dbg_timing_ok    (dbg_timing_ok)
   );

   // 100 MHz clock
   initial clk = 0;
   always #5 clk = ~clk;

   initial begin
      // init
      rst_n = 0;
      app_addr = 32'd0;
```

```
        app_cmd  = 3'b0;
        app_en   = 1'b0;
        app_wdf_data = 64'd0;
        app_wdf_wren = 1'b0;

        #50;
        rst_n = 1; // release reset

        // wait a few cycles for start_init pulse to occur and FSM to reach READY
        #100;

        // 1) Write
        @(posedge clk);
        if (app_rdy && app_wdf_rdy) begin
          app_addr <= 32'h0000_1000;
          app_cmd  <= 3'b001; // write
          app_en   <= 1'b1;
          app_wdf_data <= 64'hA5A5_A5A5_F0F0_0F0F;
          app_wdf_wren <= 1'b1;
        end
        @(posedge clk);
        app_en <= 1'b0; app_wdf_wren <= 1'b0;

        // allow controller operate
        #200;

        // 2) Read same address
        @(posedge clk);
        if (app_rdy) begin
          app_addr <= 32'h0000_1000;
          app_cmd  <= 3'b000; // read
          app_en   <= 1'b1;
        end
        @(posedge clk);
        app_en <= 1'b0;

        // wait for read data valid
        wait (app_rd_data_valid == 1);
        $display("Read data at time %0t : %h", $time, app_rd_data);

        #100;
        $display("Top-level integration TB finished.");
        $finish;
    end

endmodule
```