Hardware System Design Final Exam

2017. 6. 13

ID :

Name :

Q1. (10) Assume that the following code runs on ARM CPU of Zynq FPGA used in our practice.

```
1   int foo = open("/dev/mem", O_RDWR);
2   int *fpga_bram = mmap(NULL, SIZE * sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED, foo,
0x40000000);
3   for (i = 0; i < SIZE; i++)
4         *(fpga_bram + i) = (i * 2);
```

Q1.1 (2) Explain what happens on the page table when executing line 2.

(Ans) The page table entry (PTE) of the virtual address, fpga_bram is assigned the physical address of 0x40000000.

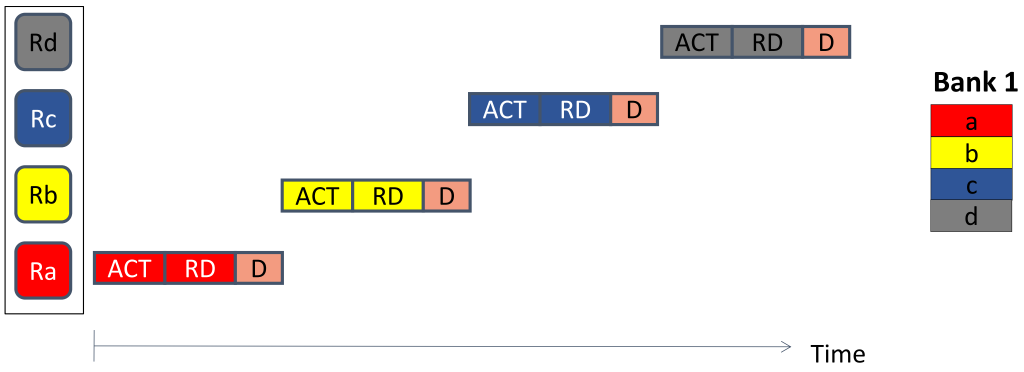
Q1.2 (2) Explain what happens on the TLB when executing line 4.

(Ans) TLB is looked up with the virtual address of fpga_bram+i*sizeof(int). In case of TLB hit, the associated physcial page number (the address of physical page) is read from the TLB to form the physical address which is used to access L1 data cache. In case of TLB miss, the MMU accesses the page table on main memory, fetches the associated PTE from main memory, and inserts it to TLB after returning the required physical page number.

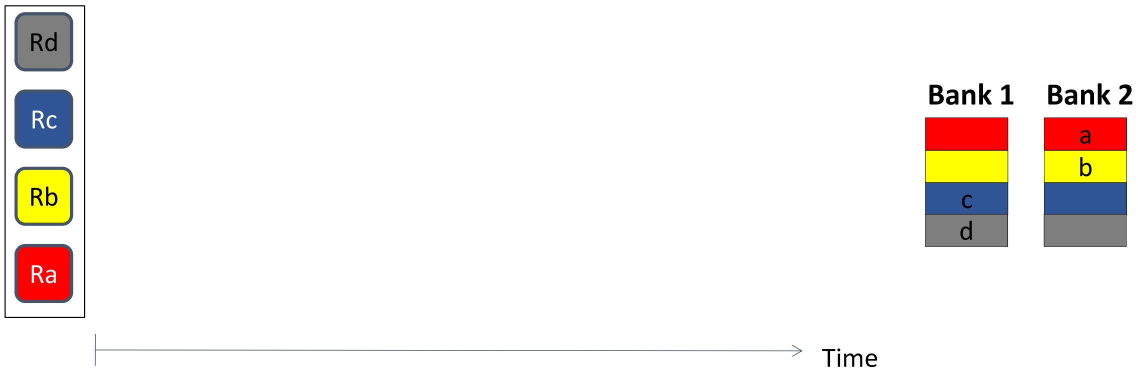
Q1.3 (6) Assume that the BRAM (on the programmable logic) has an AXI interface and is connected to an AXI bus. Explain what happens on the AXI interface of BRAM when executing line 4.

(Ans) We assume the following actions are taken at clock rising edge. On AW (write address) channel, the AXI bus forwards a write request (received from the CPU) to the BRAM. To do that, the bus sets aw_addr to the physical address (calculated with the TLB) and raises aw_valid. The write request is received by the BRAM when aw_ready is ‘1’. The AXI bus also forwards write data (received from the CPU) to the BRAM on W (write) channel by raising w_valid after setting w_data to the write data. The write data is received by the BRAM when w_ready is ‘1’. After all data requests are received by the BRAM, BRAM issues BRESP/BVALID signal as successful write response.

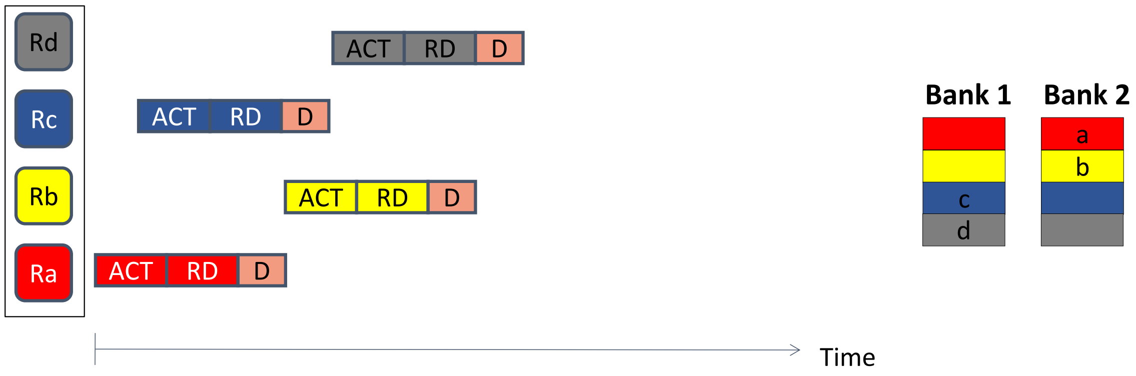
Q2. (7) Assume four read requests, Ra (oldest), Rb, Rc, and Rd (youngest) to access data a, b, c, and d on bank1, respectivley, arrive at the same time at the DRAM memory controller. The following figure shows how they are served in a first-come-first-serve (FIFO) manner.



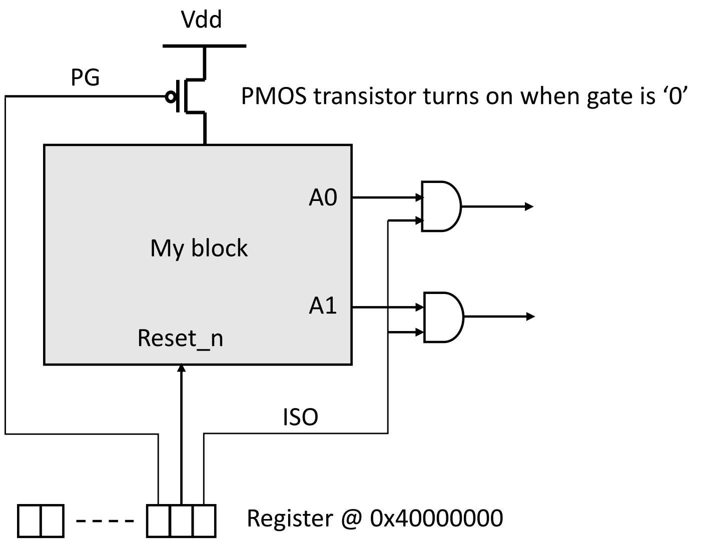
Assume that data a and b are re-allocated, i.e., moved to two distinct rows on Bank 2 as shown on the right-hand side of the following figure. Explain how we can improve memory access scheduling by exploiting bank parallelism. Draw your solution with your explanation. Hint: At any instant, one one bank can provide data to the output pins of DRAM.



(Ans) As the following figure shows, two sets of requests, {Ra, Rb} and {Rc, Rd} can be served in parallel because their target banks are different. For each bank, the 2nd request, e.g., Rb can be served only after the 1st one, e.g., Ra is completely served.



Q3. (8) Assume that we want to enable power gating to a hardware block named "My block". As the following figure shows, we need to use a power switch (PMOS transistor at the top) and isolation cells (two AND gates for two outputs A0 and A1). The control signals for power gating, PG, reset (reset is performed when Reset_n is low) and isolation cell, ISO are connected to three least significant bits of a control register at address 0x40000000.

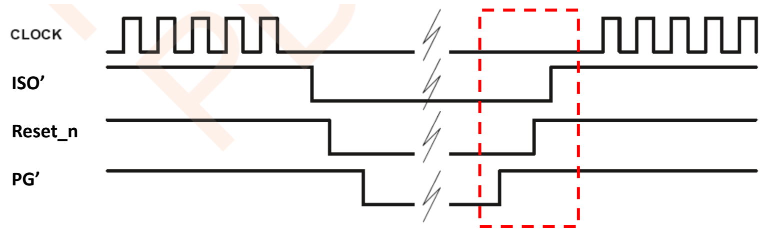


We can use a function, set_reg(address, value) to set a value on the register. For instance, in order to set the least significant bit of the register, we can call the following function.

set_reg(0x40000000, 0x00000001);

Write a software code which turns on My block and makes it ready for operation before clock is applied.

(Ans) The software code implements the highlighted part (in dashed red box) of the following figure.



set_reg(0x40000000, 0x00000000); // We turn on the power and reset My block while keeping ISO low.

set_reg(0x40000000, 0x00000002); // Finishes reset operation by setting reset_n as 1.

set_reg(0x40000000, 0x00000003); // Isolation cell is disabled. Thus, the output can be driven by My block.

A4.1 배점 4점

```
module custom_block (
input [1:0] sel,
input [3:0] in,
output [3:0] out
);
reg [3:0] temp;
assign out=temp;
always @(*)
  case (sel)
    2'b01: temp <= {0, 0, in[1], 0};
    2'b10: temp <= {0, in[2], 0, 0};
    default: temp <= {in[0], in[1], in[2], in[3]};
  endcase
endmodule
```

A4.2 배점 4점

```
module tb ();
reg [1:0] sel;
reg [3:0] in;
reg [3:0] out;

custom_block ( .sel(sel), .in(in), .out(out) );
initial begin
in=4'b1010;
#10 sel=2'b00;
#10 sel=2'b01;
#10 sel=2'b10;
#10 sel=2'b11;
end
endmodule
```

A4.3 배점 3점

```
module demux (
input [1:0] sel,
input [3:0] in,
output [3:0] out3,
output [3:0] out2,
output [3:0] out1,
output [3:0] out0,
);
```

```
assign out3 = (sel==3) ? in : 4’b0000;
assign out2 = (sel==2) ? in : 4’b0000;
assign out1 = (sel==1) ? in : 4’b0000;
assign out0 = (sel==0) ? in : 4’b0000;
endmodule
```

A4.4 배점 4점

```
module up_counter (
input clk,
input rstn,
output [3:0] count
);
reg [3:0] temp;
assign count = temp;
always @(posedge clk or negedge rstn)
  if(!rstn) temp <= ‘d0;
  else temp <= temp+1;
endmodule
```

A5. 배점 15점

```
module auto_oven(clk, start, temp_ok, done, load, heat, unload, beep);
input clk, start, temp_ok, done;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;
// state encoding using parameter syntax
parameter IDLE = 'b000;
parameter PREHEAT = 'b001;
parameter LOAD = 'b010;
parameter COOK = 'b011;
parameter EMPTY = 'b100;

//State register block
always @(posedge clk)
    state <= #10 next_state;

// next state logic
always @(state or start or temp_ok or done) begin
    next_state = state; // default to stay in current state
    case (state)
        IDLE: if (start) next_state = PREHEAT;
        PREHEAT: if (temp_ok) next_state = LOAD;
        LOAD: next_state = COOK;
        COOK: if (done) next_state = EMPTY;
        EMPTY: next_state = IDLE;
        default: next_state = IDLE;
    endcase
end

// Output logic
always @(state) begin
    if (state == LOAD) load = 1;
    else load = 0;
    if (state == EMPTY) unload = 1;
    else unload = 0;
    if (state == EMPTY) beep = 1;
    else beep = 0;
    if (state == PREHEAT || state == LOAD || state == COOK) heat = 1;
    else heat = 0;
end
endmodule
```

A6. 배점 15점

```
module seq_detector (
    input       clk,
    input       rstn,
    input       x,
    output reg   z
    );

   reg   [1:0]   present_state, next_state;
   parameter s0 = 2'b00;
   parameter s1 = 2'b01;
   parameter s2 = 2'b10;
   parameter s3 = 2'b11;

// part 1:  initialize to state s0 and update present state register
    always @(posedge clk or negedge rstn)
        if(!rstn) present_state <= s0;
        else present_state <= next_state;

// part 2: determine next state
   always @(present_state or x)
        case(present_state)
            s0: next_state = x? s1 : s0;
            s1: next_state = x? s1 : s2;
            s2: next_state = x? s3 : s0;
            s3: next_state = x? s1 : s2;
      endcase

// part 3: evaluate output z
   always @(present_state or x)
      case (present_state)
         s0: z = x? 1'b0 : 1'b0;
         s1: z = x? 1'b0 : 1'b0;
         s2: z = x? 1'b0 : 1'b0;
         s3: z = x? 1'b0 : 1'b1;
      endcase
endmodule
```

A7. 배점 15점

```
module clk_div (
    input   clk,
    input   rstn,
    output  clk_output
    );

    reg   [7:0]   divider_value; //program to "value - 1"
    reg   [7:0]   posedge_cnt;
    reg           rise_pulse_reg, neg_pulse_reg;

    always@(posedge clk or negedge rstn)
        if(!rstn)
            posedge_cnt <= {8{1'b0}};
        else if(posedge_cnt == divider_value)
            posedge_cnt <= {8{1'b0}};
         else
```

```
            posedge_cnt <= posedge_cnt+1;

    always@(posedge clk or negedge rstn)
        if(!rstn)
            rise_pulse_reg <= 1'b0;
        else if(posedge_cnt == divider_value[7:1])
            rise_pulse_reg <= 1'b1;
         else  if(posedge_cnt == divider_value)
            rise_pulse_reg <= 1'b0;

    always@(negedge clk or negedge rstn)
        if(!rstn)
            neg_pulse_reg <= 1'b0;
        else if(divider_value[0] ==1'b0)
            neg_pulse_reg <= rise_pulse_reg;

    assign clk_output = rise_pulse_reg | neg_pulse_reg;

endmodule
```

A8. 배점 15점

```
module car_alarm (
        input clk,
        input rstn,
        input user_lock,
        input user_unlock,
        input trespass,
        output light,
        output horn,
        output car_lock,
        output [1:0] state
);

parameter DISALARM='d0;
parameter SET='d1;
parameter ALARM='d2;
parameter ALERT='d3;

reg [1:0] curr_state;
reg [1:0] next_state;
reg alert_done;
reg light_r, horn_r, car_lock_r;

assign state = curr_state;
assign light = light_r;
assign horn = horn_r;
assign car_lock = car_lock_r;

//state transition
always @(posedge clk or negedge rstn)
  if(!rstn) curr_state<=DISALARM;
  else curr_state<=next_state;

//state decision
always @(*)
  case(curr_state)
    DISALARM:
          if(user_lock) next_state<=SET;
          else next_state<=curr_state;
        SET:
          if(user_unlock) next_state<=DISALARM;
          else if(timer==0) next_state<=ALARM;
          else next_state<=curr_state;
        ALARM:
          if(user_unlock) next_state<=DISALARM;
          else if(trespass) next_state<=ALERT;
          else next_state<=curr_state;
        ALERT:
          if(user_unlock) next_state<=DISALARM;
          else if(timer==0) next_state<=ALARM;
          else next_state<=curr_state;
        default:
          next_state<=DISALARM;
  endcase

//output decision
always @(posedge clk)
  case(curr_state)
    DISALARM:
          if(user_lock) begin
            light_r<='d1;
                horn_r<='d1;
          end
          else begin
            light_r<='d0;
                horn_r<='d0;
          end
        SET:
        begin
          light_r<='d0;
          horn_r<='d0;
        end
        ALARM:
        begin
          if (user_unlock) car_lock_r <= 0;
          else car_lock_r <= 1;
          light_r<=light_r;
          horn_r<=horn_r;
        end
        ALERT:
          if(user_unlock) begin
            light_r<='d0;
                horn_r<='d0;
          end
          else if(timer==0) begin
            light_r<='d1;
                horn_r<='d0;
          end
          else begin
            light_r<='d1;
                horn_r<='d1;
          end
        endcase

//SET
reg set_flag;
wire set_rst=!rstn||set_done;
wire set_en=(curr_state==DISALARM)&&(next_state==SET);
always @(posedge clk)
  if(set_rst) set_flag<='d0;
  else if(set_en) set_flag<='d1;
  else set_flag<=set_flag;

//ALERT
reg alert_flag;
wire alert_rst=!rstn||alert_done;
wire alert_en=(curr_state==ALARM)&&(next_state==ALERT);
```

```
always @(posedge clk)
  if(alert_rst) alert_flag<='d0;
  else if(alert_en) alert_flag<='d1;
  else alert_flag<=alert_flag;

//timer
reg [5:0] timer;
wire [5:0] ld_val = (set_en) ? 'd30:
                                        (alert_en) ? 'd60:'d0;

wire timer_ld=set_en || alert_en;
wire timer_en=set_flag || alert_flag;
wire timer_rst=!rstn || set_done || alert_done;

always @(posedge clk or negedge timer_rst)
  if(timer_rst) timer<='d0;
  else if(timer_ld) timer<=ld_val;
  else if(timer_en) timer<=timer-1;
  else timer<=timer;

//done signal
wire set_done=(set_flag)&&(timer==0);
wire alaert_done=(alert_flag)&&(timer==0);

endmodule
```