Hardware System Design Final Exam

2017. 6. 13

ID :

Name :

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

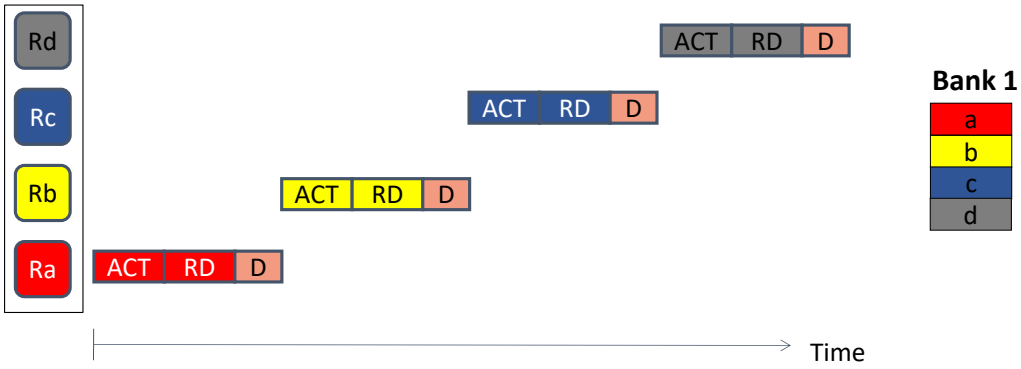
line	code
1	`int foo = open("/dev/mem", O_RDWR);`
2	`int *fpga_bram = mmap(NULL, SIZE * sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED, foo, 0x40000000);`
3	`for (i = 0; i < SIZE; i++)`
4	`*(fpga_bram + i) = (i * 2);`

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

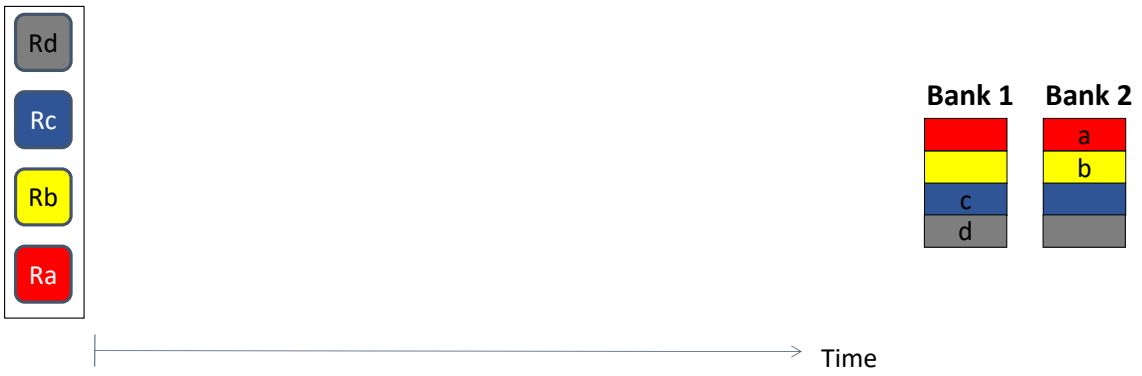
Q1.2 (2pts) Explain what happens on the TLB when executing line 4.

Q1.3 (6pts) 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.

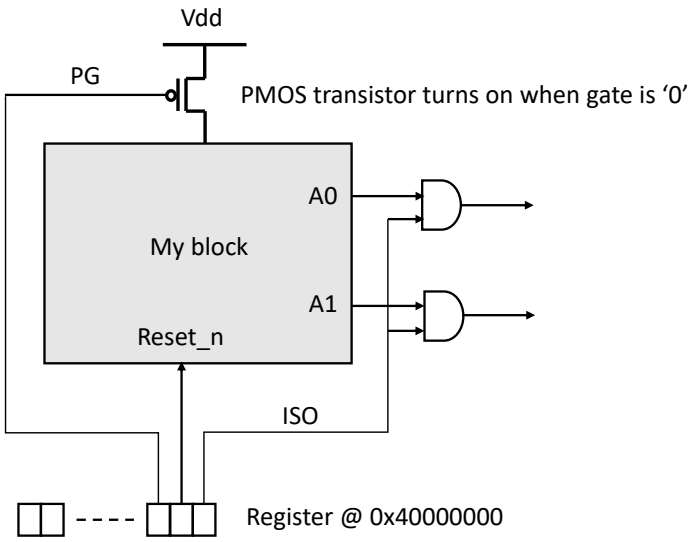
Q2. (7pts) Assume four read requests, Ra (oldest), Rb, Rc, and Rd (youngest) to access data a, b, c, and d on Bank 1, respectivley, arrive at the same time at the DRAM memory controller. The following figure shows how they are served in a first-come-first-serve (FIFO) manner.



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, only one bank can provide data to the output pins of DRAM.



Q3. (8pts) 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.

Q4. (Basic syntax, 15pts) Fill the blank and complete the code without exceeding blank lines (No partial score).

Q4.1 (4pts) Use 'case' statement and design block to meet function table shown below.

FUNCTION TABLE

sel	out[3]	out[2]	out[1]	out[0]
2'b00	in[0]	in[1]	in[2]	in[3]
2'b01	0	0	in[1]	0
2'b10	0	in[2]	0	0
2'b11	in[0]	in[1]	in[2]	in[3]

```
module custom_block (
input [1:0] sel,
input [3:0] in,
output [3:0] out
);
_____________________________________________________
_____________________________________________________
always @(__________)
  case (__________)
    _____________________________________________________
    _____________________________________________________
    _____________________________________________________
  endcase
endmodule
```

Q4.2 (4pts) Design testbench to test module designed in **Q1.1**. You should test every possible 'sel' with any 'in'.

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

custom_block (_____________________________________________________);
initial begin
_____________________________________________________
_____________________________________________________
_____________________________________________________
_____________________________________________________
_____________________________________________________
end
endmodule
```

Q4.3 (3pts) Use ‘assign’ statement and design a 1-to-4 demultiplexer.

```
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,
);
_____________________________________________________
_____________________________________________________
_____________________________________________________
_____________________________________________________
endmodule
```

Q4.4 (4pts) Design 4-bit up-counter with asynchronous reset.

```
module up_counter (
input clk,  //clock signal
input rstn, //reset, active low
output [3:0] count
);
_____________________________________________________
_____________________________________________________
always @(_________________________________)
  _____________________________________________________
  _____________________________________________________
endmodule
```

Q5. (Automatic oven – Moore machine, 15pts) Design a Verilog module for the automatic oven FSM as a Moore machine style –module named '*auto_oven'*. The automatic oven has following inputs.

Inputs

- **clk**: clock signal.
- **start**: start signal, active high.
- **temp_ok**: signal for temperature, active high indicates that **PREHEAT** is done.
- **done**: signal for finishing cooking, active high indicates finishing cooking.

Outputs are determined as following table.

STATES AND OUTPUTS OF THE AUTOMATIC OVEN

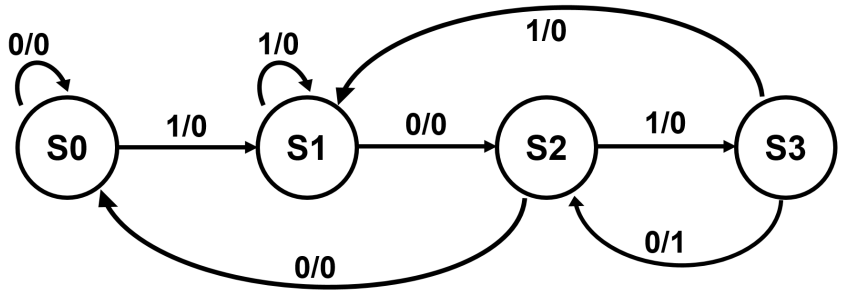
State	load	heat	unload	beep
IDLE	0	0	0	0
PREHEAT	0	1	0	0
LOAD	1	1	0	0
COOK	0	1	0	0
EMPTY	0	0	1	1

The automatic oven has five states. Each state operates as follows. When a user cook using this machine (**IDLE**), it first preheats (**PREHEAT**). After preheating, the user should put a food in the oven (**LOAD**), and then it starts cook (**COOK**). When cooking is done, the user should unload the food (**EMPTY**). Finally, oven changes its state into **IDLE** and waits for new commend. You should use 'parameter' syntax to make your own state encoding. Fill the blank and complete the following code.

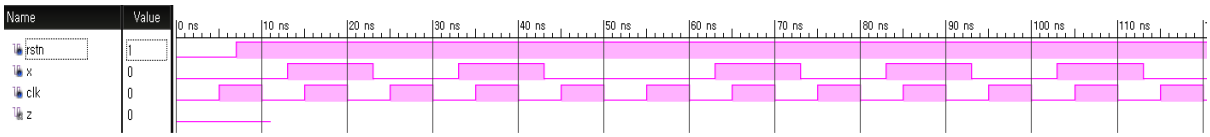
```
module  ____________________________________________________
input clk, start, temp_ok, done;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;

// state encoding using 'parameter' syntax
_____________________________
_____________________________
_____________________________
_____________________________
_____________________________

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

// next state logic
always @(_____________________________________) begin
    _____________________ // default to stay in current state (i.e., state)
    case (state)
        IDLE: if (start) next_state = PREHEAT;
        ___________________________________________
        ___________________________________________
        ___________________________________________
        ___________________________________________
        default: next_state = IDLE;
    endcase
end

// output logic
// you can use # of lines less than or equal to these underlined lines.
always @(_____________________________________) begin
    if (state == LOAD) load = 1;
    else load = 0;
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
end
endmodule
```

Q6. (Timing diagram, 15pts) Design a Verilog module for the sequence detector. The following figures are state diagram and timing diagram for sequence detector. The output, '*z*', of timing diagram is empty.

STATE DIAGRAM



TIMING DIAGRAM (COMPLETE THE OUTPUT SIGNAL, 'Z')



The sequence detector has following inputs and outputs.

Inputs

- **clk**: clock signal
- **rstn**: asynchronous negative reset signal
- **x**: an input bit in sequence

Outputs

- **z**: when the pattern (1010) is detected in a sequence of x, it turned to 1. Otherwise, it is 0.

Fill the blank on Verilog code(next page) and complete the timing diagram above (i.e., draw waveform of output signal, 'z').

Fill the blank on Verilog code and complete the timing diagram (i.e., draw waveform of output signal, *'z'*).

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

   reg   [1:0]   present_state, next_state;
   //state encoding using 'parameter' syntax
   _____________________________________________;
   _____________________________________________;
   _____________________________________________;
   _____________________________________________;

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

// part 2: determine next state
   always @(_____________________)
        case(_________________)
            s0: next_state = ____________________;
            s1: next_state = ____________________;
            s2: next_state = ____________________;
            s3: next_state = ____________________;
      endcase

// part 3: evaluate output 'z'
   always @(_____________________)
      case (_________________)
         s0: z = ____________________;
         s1: z = ____________________;
         s2: z = ____________________;
         s3: z = ____________________;
      endcase
endmodule
```

Q7. (Timing diagram, 15pts) Design a Verilog module for the generic clock divider. The module has following inputs and outputs.

Inputs

- **clk**: clock signal
- **rstn**: asynchronous negative reset signal

Outputs

- **clk_output**: clock signal generated by the module

The following figures show timing diagrams of three types of generic clock divider. *'divider_value'* stores (divider value-1). For example, when its value is 2'b10, the module divides the original clock by 3, with 50-50 duty cycle. *'posedge_cnt'* is used to count up to *'divider_value'*. Complete the code as the module follows these waveforms.

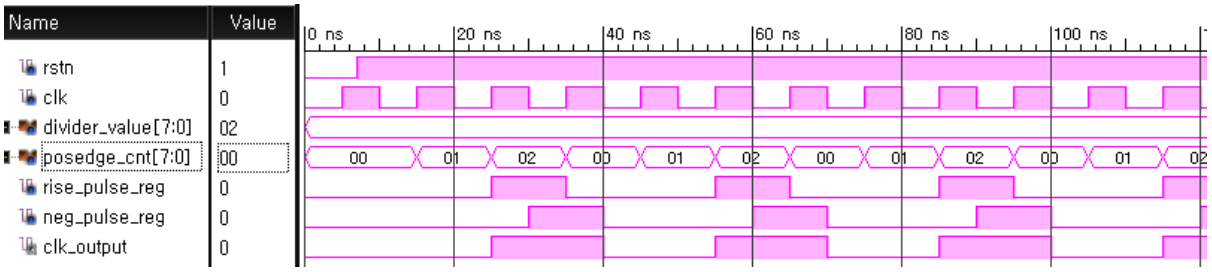


FIG. 1: TIMING DIAGRAM FOR CLOCK DIVIDE BY 3 WITH (50-50 DUTY CYCLE) (SET DIVIDER_VALUE = 2'B10)

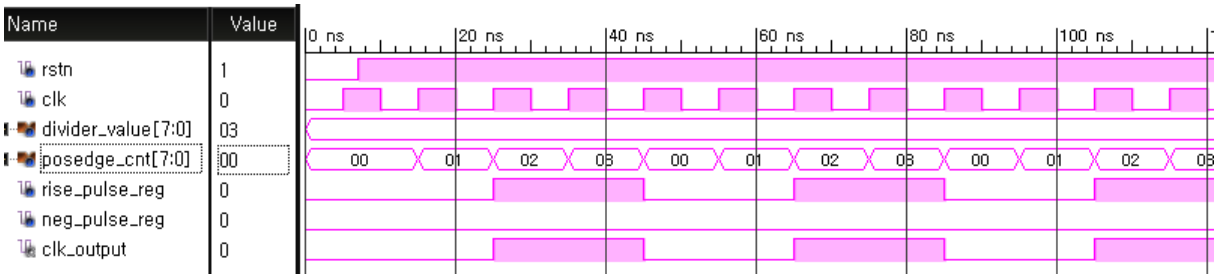


FIG. 2: TIMING DIAGRAM FOR CLOCK DIVIDE BY 4 WITH (50-50 DUTY CYCLE) (SET DIVIDER_VALUE = 2'B11)

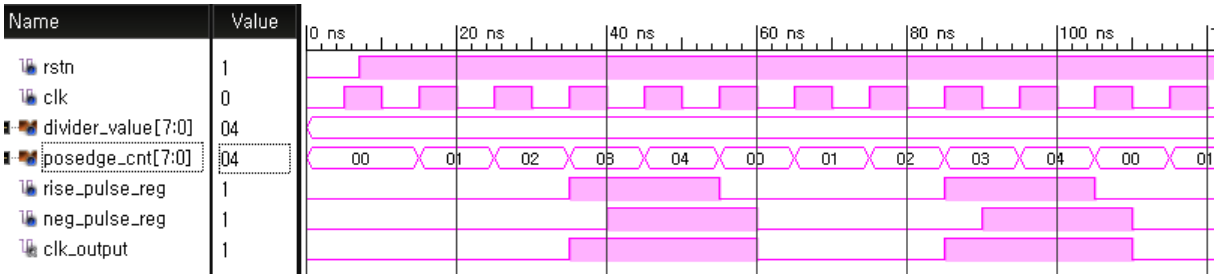


Fig. 3: Timing diagram for Clock divide by 5 with (50-50 duty cycle) (Set divider_value = 3'b100)

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

    reg   [7:0]   divider_value; //set to "value - 1". Ex) set 2'b10 for divide by 3
    reg   [7:0]   posedge_cnt;
    reg           rise_pulse_reg, neg_pulse_reg;

    always@(_____________________________)
        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@(_____________________________)
        if(!rstn)
            rise_pulse_reg <= _________;
        else if(posedge_cnt == ___________________)
            rise_pulse_reg <= _________;
        else if(posedge_cnt == ___________________)
            rise_pulse_reg <= _________;

    always@(_____________________________)
        if(!rstn)
            neg_pulse_reg <= __________;
        else if(_____________________________)
            neg_pulse_reg <= __________;

    assign clk_output = ___________________________________________;

endmodule
```

Q8. (Crime prevention alarm system for an automobile, 15pts) Design a Verilog module for the following simple crime prevention alarm system for an automobile. The alarm system has the following inputs and outputs.

Inputs

- **clk**: clock signal.
- **rstn**: asynchronous negative reset signal.
- **user_lock**: signal for locking door (user-side), active high indicates locking vehicle door.
- **user_unlock**: signal for unlocking door (user-side), active high indicates unlocking vehicle door.
- **trespass**: trespass signal, active high when someone come into vehicle illegally.

Outputs

- **light**: enable signal for vehicle light, active high
- **horn**: enable signal for horn, active high
- **car_lock**: signal for locking door (vehicle-side), active high.
- **state**: state.

The alarm system has four modes. Each mode operates as follows.

- **DISALARM**: alarm off mode where the doors are unlocked, and the reset state when *'rstn'* active. Whenever a *'user_unlock'* goes active in the **ALARM** or **ALERT** mode, current state is changed into this state. *'light'* and *'horn'* are both off.
- **SET**: alarm set mode when alarm is activated. **DISALARM** is changed into **SET** whenever a *'key_lock'* goes active. '*light'* and '*horn'* are turned on for a cycle to indicate the alarm is being enabled during the transition from **DISALARM** to **SET**. In order to enable the user to still enter the vehicle, delay timer counts 30 cycles once '*key_lock'* goes active. Then, state is changed into **ALARM** mode.
- **ALARM**: alarm active mode. After waiting in **SET** mode for 30 cycles, state is changed into **ALARM** state and '*car_lock'* is activated (i.e., lock the vehicle door). State is changed into **DISALARM** if '*user_unlock'* goes active. If not and a '*trespass'* goes active (i.e., someone come into car illegally), then state is changed into **ALERT** mode.
- **ALERT**: '*light'* and '*horn'* are turned on. In addition, a timer counts 60 cycles. If a '*user_unlock'* goes active, then state is changed into **DISALARM** mode. If the timer reaches 60 cycles, then transition back to the **ALARM** mode. At this time the '*light'* is turned on and the '*horn'* is off.

You should use counter to counts 30 cycles (in **ALARM**) and 60 cycles (in **ALERT**). Fill the blank and complete your own code freely **(Hint. Remind M × V multiplication code we designed at lab hour)**.

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