Hardware System Design Mid-term Exam

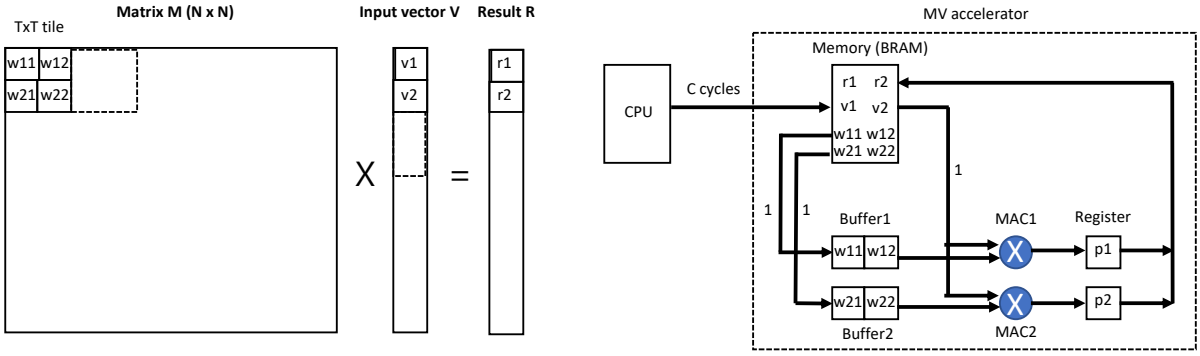
2019. 04. 16, Total <50pts>

ID :

Name :

Q1. (Matrix-vector multiplication in hardware, 10 pts)

The following figure shows how the hardware MV acceleator performs a multiplication of NxN matrix and vector with TxT (2x2 in the figure) tile. Assume the case that the CPU has all the data of weight matrix M and vector V. The CPU sends, to the memory (BRAM in the figure) of MV acceleartor, each tile of weight matrix (w11, w12, w21, and w22 in the example) and vector (v1 and v2). Transfering a tile data from CPU to MV accelerator takes C cycles (no dependency on data size) as shown in the figure. After receiving the tile data, the MV accelerator first writes the matrix elements to the buffers of MAC units. Writes to T buffers (Buffer1 and Buffer2 in this example) are performed in parallel. It takes one cycle to write a matrix element to each buffer as the cycle number '1' (one cycle / write) denoted on the corresponding arrows in the figure. Thus, in order to write T matrix elements for each buffer, it takes T clock cycles. After writing matrix elements to the buffers, the MV accelerator broadcasts a vector element (e.g., v1) to all the MAC units (MAC1 and MAC2). It also takes T clock cycles to broadcast all the vector elements of the tile. On each cycle of the broadcast, upon receiving the vector element, the MAC unit performs a multiplication (of matrix element and vector element, e.g., w11 and v1 on MAC1) and accumulcation (adding the multiplication results to the previous accumulation result) to produce a partial sum. Then, the MAC unit stores the partial sum (e.g., p1) in its register. Note that the execution of the broadcast of a vector element and MAC operation takes only one clock cycle.
After finishing the computation of the current tile, the MV accelerator receives, from the CPU, a new tile data (matrix elements and vector elements, e.g., w13, w14, w23, w24, v3 and v4) and perform computation in the same manner until the completion of matrix-vector multiplication. Note that, in order to handle TxT tile, the MV accelerator is equipped with T MAC units (each of which has its own buffer and register). In this example, we do not consider the execution cycles of moving the computation results (elements of vector R) from the registers to the memory and finally to the CPU.



(1) (5 pts) Obtain the equation (represented in terms of N, T, and C) to calculate the total number of execution cycles for matrix M x vector V multiplication shown above. Assume N%T = 0 (N is a multiple of T).

(2) (5 pts) Assume N = 100 and C = 10 cycles. Assuming that the execution cycle of M*V on the CPU is N^2, i.e., 10,000 cycles, we want to obtain two times speedup, i.e., less than 5,000 cycles by running the MV accelerator. What is the tile size, T to achieve two times speedup? (Use the equation obtained in (1) and ignore the condition N%T=0. Assume the memory (BRAM) can keep the data of a tile and the final results.)

Q2. (Verilog Implementation, 10 pts)

(1) (3 pts) Implement a 1-bit half adder.

```
module half_adder_1b (
    input _____________________________________,
    output _____________________________________
);
    // Implement HERE









endmodule
```

(2) (3 pts) Implement a 1-bit full adder using, i.e., instantiating half adders (i.e., `half_adder_1b`) from Q2. (1).

```
module full_adder_1b (
    input _____________________________________,
    output _____________________________________
);
    // Implement HERE
```

```
endmodule
```

(3) (4 pts) Implement a 4-bit full adder using 1-bit half adders (i.e., `half_adder_1b`) or 1-bit full adders (i.e., `full_adder_1b`) that you implemented above.

```
module full_adder_4b (
    input [3:0] _____________________________,
    input _____________________________,
    output [3:0] _____________________________,
    output _____________________________
);
    // Implement HERE









endmodule
```

Q3. (Synthesizable code, 10 pts)

In hardware implementations in Verilog, unintentional latch inference that occurs during synthesis can produce unpredictable bugs in the system. The Verilog codes below are part of the code of combinational logic or flip-flop, each of which implements the truth table on the right.

(1) (5 pts) Select the code where the unintentional latch inference occurred.

Answer :

(2) (5 pts) Modify the code so that it can be synthesized as originally intended.

ⓐ

```
always @(enable or data)

  if(enable) y = data;
```

Input	Output
enable	y
0	X
1	data

ⓑ

```
always @(posedge clk)

  if(enable) y = data;
```

Input		Output
clk	enable	y
posedge	0	y
	1	data

ⓒ

```
always @(select or data)

    case(select)

        2'b00: y = data[select];

        2'b01: y = data[select];

        2'b10: y = data[select];

    endcase
```

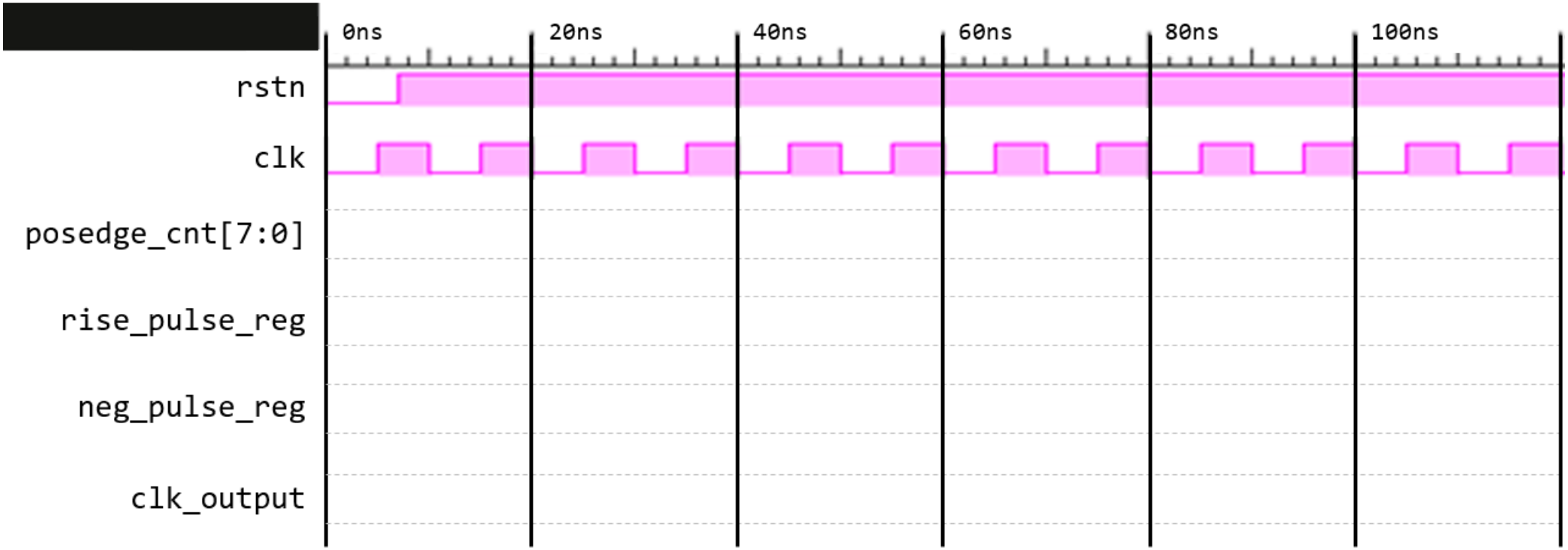
Input	Output
select	y
2’b00	data[0]
2’b01	data[1]
2’b10	data[2]
2’b11	X

(Empty Sheet)

Q4. (Simulation, 10 pts) Draw waveform of all the signals in the following Verilog code of signal generator.

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

    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 == 3'b100) 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 == 2'b10) rise_pulse_reg <= 1'b1;
        else if(posedge_cnt == 3'b100) rise_pulse_reg <= 1'b0;

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

    assign clk_output = rise_pulse_reg | neg_pulse_reg;

endmodule
```

Q4. (Answer Sheet)



Q5. (Simulation, 10 pts) Draw waveform of all the signals of the following Verilog design of alarm system.

```
module alarm_system (
        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 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) ? 'd10 :
                    ((alert_en) ? 'd10 : '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;
end

// Done signal
wire set_done = (set_flag) && (timer == 0);
wire alert_done = (alert_flag) && (timer == 0);


endmodule
```

Q5. (Answer Sheet)