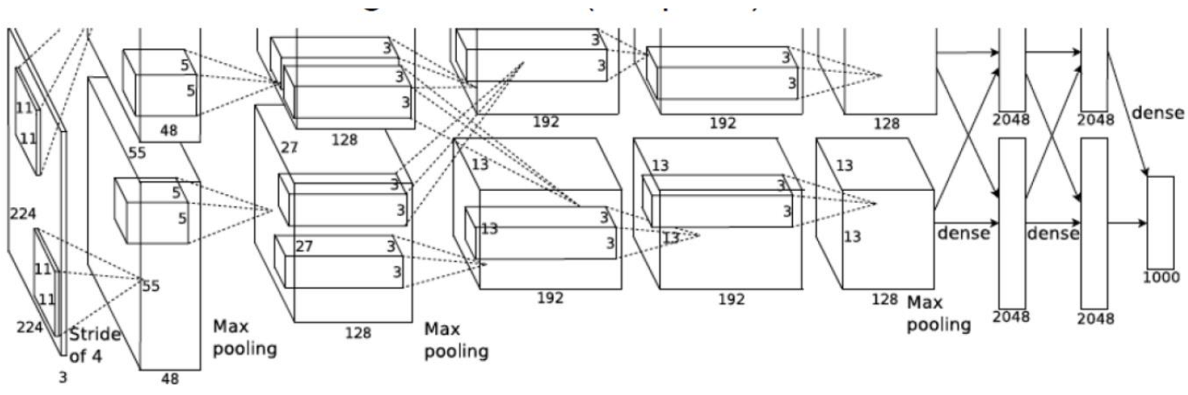
Hardware System Design Final Exam

2021.05.31

Student ID:
Name:

Q1. Convolution lowering, Systolic Array (10 points)

The following figure shows AlexNet. Assume convolution lowering.



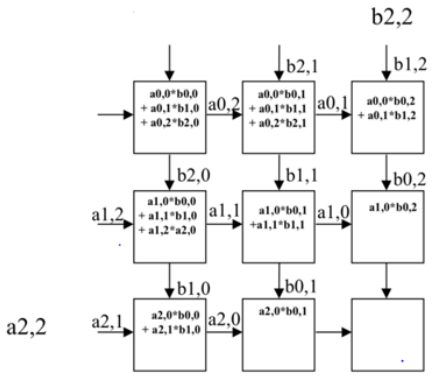
Problem 1-1. Convolution Lowering (3 points)

Calculate the size (width and height) of input activation matrix for 1st convolution layer (224x224x3 → 55x55x48).

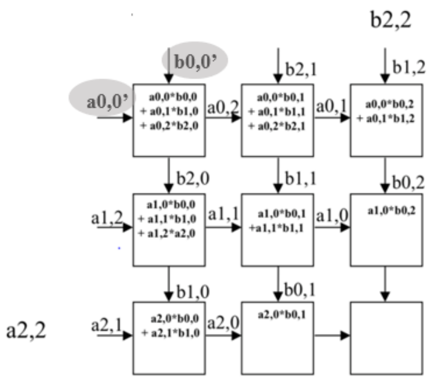
[Answer]
Width: 55x55
Height: 11x11x3

Problem 1-2. Systolic Array (7 points)

Assume a 128x128 systolic array having 128x128 PEs and output stationary, i.e., both input activation and weight tiles (128x128 each) are provided to systolic array and each PE calculates one element of output tile. For instance, recall the systolic example of 3x3 matrix multiplication we studied in the lecture. The following figure shows a snapshot of systolic array execution at cycle T.



Cycle T (without pipelining)

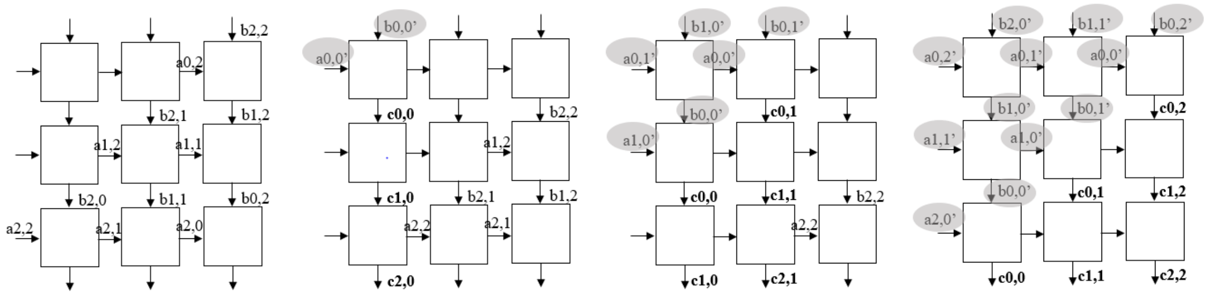


Cycle T (with pipelining)

The left-hand side of the above figure shows the case that pipelining is not applied while the right-hand side one shows the pipeline case. Their key difference is highlighted by two grey ovals on the top left corner of the right-hand side figure. They represent the first input pairs of the next input tiles. By performing such a pipeline operation, multiple pairs of input tiles can be multiplied finally to produce an output tile.

Even though the pipeline operation is applied, when an output tile is being read out of the systolic array, the input of next input tiles is being delayed as explained below.

After obtaining an output tile, the elements of output tile can be read out of the systolic array as the following figures exemplify. Assume that the last pair of input tiles are being multiplied at cycle T as shown above. As the figures show, the elements of output tile can be read out just after the last inputs enter the systolic array, i.e., cycle T+2. At this cycle, the elements of the first column of output tile (c0,0, c1,0, c2,0) starts to be read out downwards in a systolic array manner (one hop per cycle). Note that the first inputs of the next input tiles also enter the systolic array as the grey ovals show. In the next cycle (T+3), the 2nd column of output tile starts to be read out while the first column is being read out and new inputs enter the systolic array. In T+4, the last column of output tile also starts to be read out while the computation of the next input tiles is being performed.

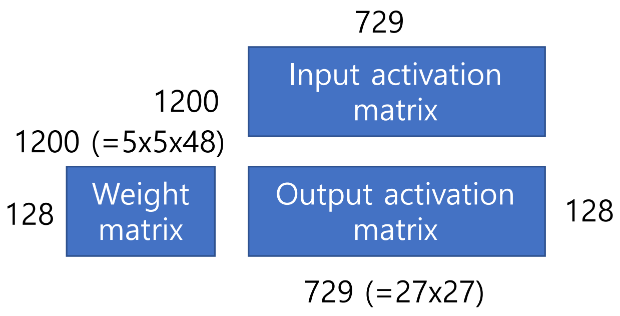


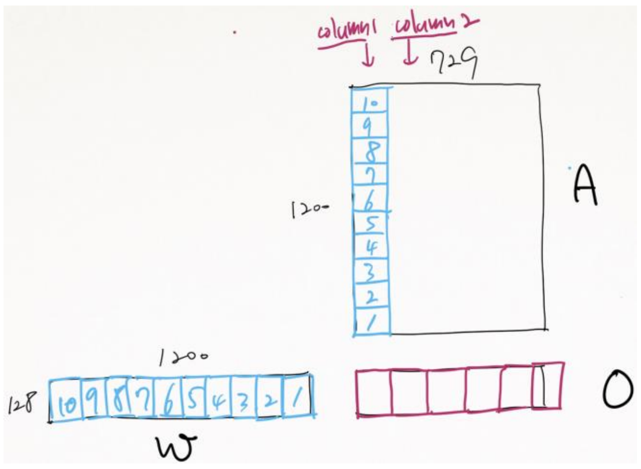
Cycle T+1 T+2 T+3 T+4

Assuming the above pipelined operation, calculate the total number of cycles to execute the 2nd convolution layer (27x27x48 → 27x27x128) assuming zero cycle for max pooling.

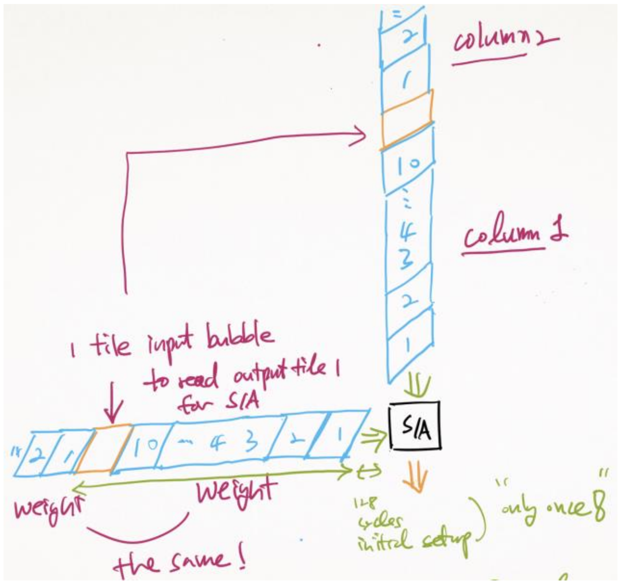
Hint: First, calculate how many input pairs of tiles are needed to compute an output tile, and then consider the number of output tiles. Don't forget that, as explained above, we cannot take input while the tile output is being read out. Finally, note that there is an initial setup cycle of systolic array for the first input of matrix-matrix multiplication.

[Answer]
The matrix size is shown below.





There are six output tiles (each, 128x128). In order to calculate one output tile of 128x128, 10 tile multiplication operations are performed. Those 10 operations can be pipelined, i.e., 10 pairs of input activation and weight tiles can enter the systolic array consecutively. After multiplying 10 pairs of input tiles, one output tile is read out from the systolic array, which takes **127 cycles** (orange bubble in the following figure). Thus, the calculation of one output tile takes **(11x128 – 1)** cycles. Total cycles to obtain the output matrix is **8569 cycles** (=**initial 127 cycles** + 6 output tile x **(11x128 – 1)** cycles/output tile = **(67x128 – 7)** cycles).



Q2. Verilog Language (10 points)

Problem 2-1. (2 points)

Correct the following code to obtain correctly synthesizable code for selection (MUX) function.

```
module mux (
        input [1:0] select,
        input [3:0] data,
        output reg y
);
        always @(select or data) begin
                case (select)
                        2'b00: y = data[select];
                        2'b01: y = data[select];
                        2'b10: y = data[select];
                endcase
        end
endmodule
```

[Answer]

```
module mux (
        input [1:0] select,
        input [3:0] data,
        output reg y
);
        always @(select or data) begin
                case (select)
                        2'b00: y = data[select];
                        2'b01: y = data[select];
                        2'b10: y = data[select];
                        default: y = data[select];
                endcase
        end
endmodule
```

Problem 2-2. (3 points)

Check to see if there is an error in the following code of n-bit adder. If there is an error, correct it.

```
module function_a(
        input [3:0] x,
        input [3:0] y,
        input c_in,
        output reg c_out,
        output reg [3:0] sum

);
     parameter N=4;
     integer i;
     reg co;

     always @(x or y or c_in) begin
          co = c_in;
          for (i=0; i<N; i=i+1)
               {co, sum[i]} = x[i] + y[i] + co;
          c_out = co;
     end
endmodule
```

[Answer] No error

Problem 2-3. (5 points)
Write a testbench code to simulate the n-bit adder in 2.2. Verify its functionality with the given inputs below.

x	y	cin
4’b1010	4’b1110	0
4’b0111	4’b1000	1
4’b0101	4’b0001	0

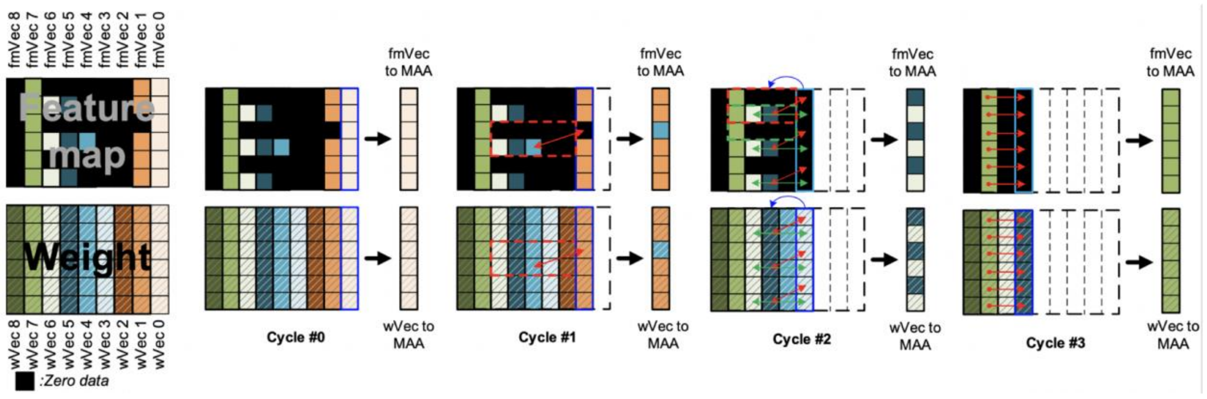
[Answer]

```
module testbench;

    reg [3:0] x, y;
    reg cin;
    wire cout;
    wire [3:0] sum;

    function_a adder (x, y, cin, cout, sum);

    initial begin
        x <= 4’b1010;
        y <= 4’b1110;
        cin <= 1’b0;
        #10;
        x <= 4’b0111;
        y <= 4’b1000;
        cin <= 1’b1;
        #10;
        x <= 4’b0101;
        y <= 4’b0001;
        cin <= 0;
    end

endmodule
```

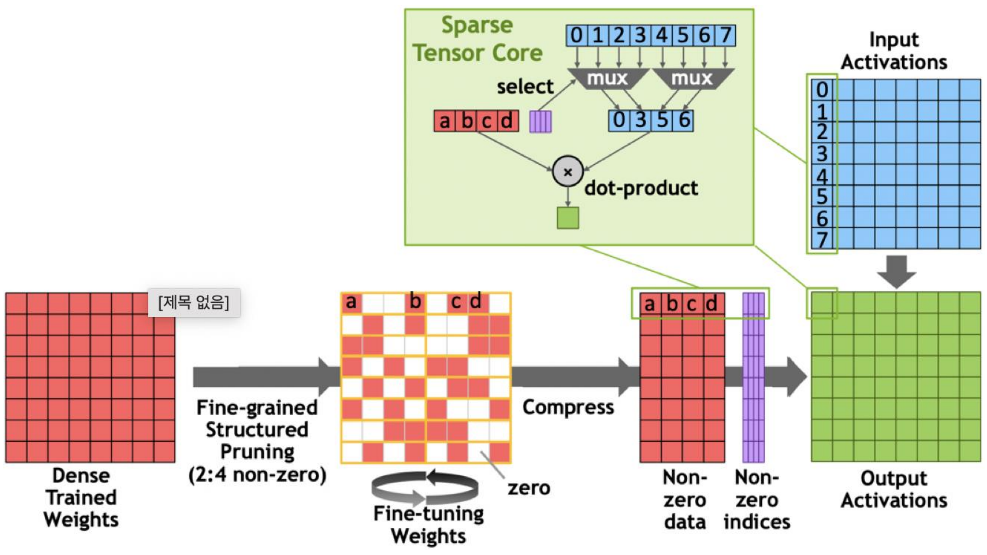
Q3. Zero-skipping (10 points)

Choose one of the following two commercial zero-skipping accelerators we studied in the lecture and, using the given figure, explain in detail how it works.

A. Samsung's zero input activation skipping accelerator



B. NVIDIA's Sparse Tensor Core (for zero weight skipping)



[Answer]

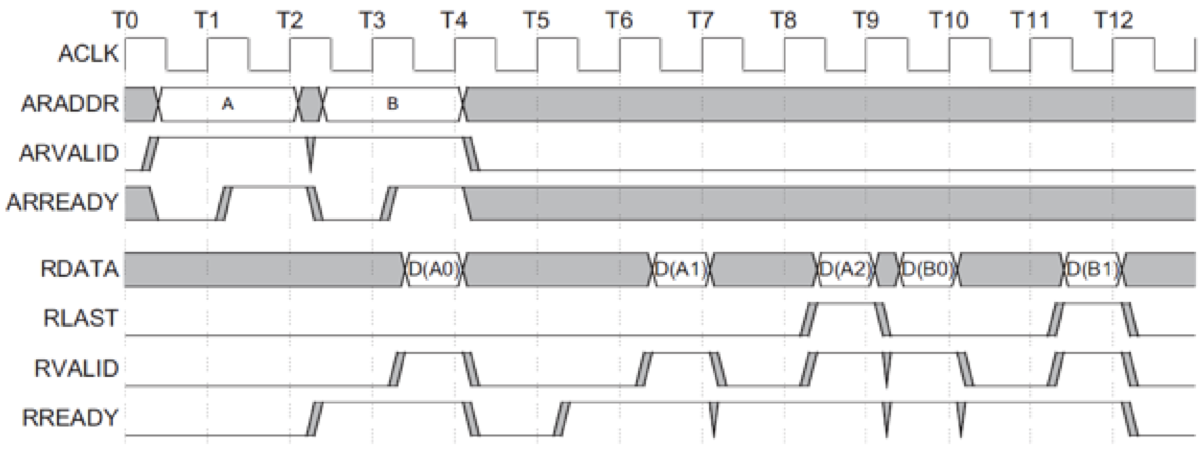
A. window 내에서 non-zero activation과 그에 대응하는 weight 가져와서 input arrays, fmVec, wVec 각각 만들어 dot product 한다는게 들어가면 OK.

B. 4개 weights 마다 2개를 pruning 하고, non-zero weight의 2bit index를 non-zero weight과 함께 가지고 있고, 이를 이용해 non-zero weights과 이들에 해당하는 input activations 간의 dot product을 수행하다고 하면 OK.

Q4. Bus and memory (10 points)

Problem 4-1. (5 points)

The following figure illustrates AXI bus operations on addresses A and B. Explain what happens at cycles 3 and 4 (T3 and T4), respectively.

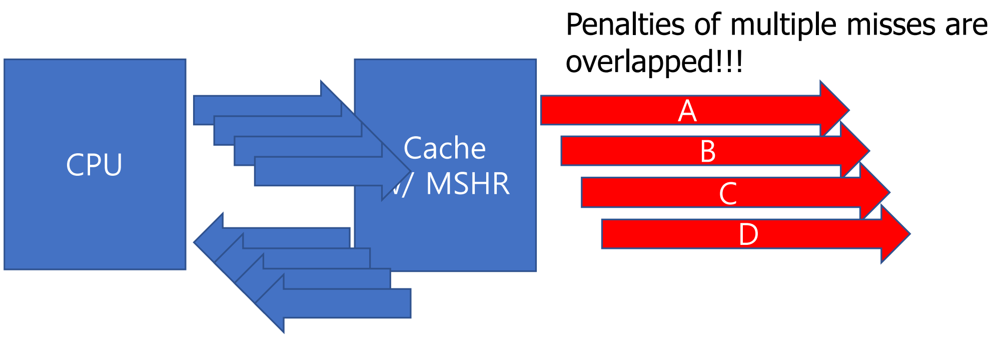


[Answer]

At T3, a read request is issued at read channel. However, it is not accepted by the slave interface because the slave is not ready (ARREADY = 0).
At T4, there are two events. In the read address channel, the read request (issued at T3) is now taken by the slave interface. In the read data channel, the first data of the previous request (to address A) is taken by the master interface.

Problem 4-2. (5 points)

The following figure illustrates how multiple cache misses (to the DRAM main memory) can be handled at the same time in the non-blocking cache. Explain, in order to realize such parallel operations, what is needed for each of bus, memory access scheduling and data placement on the DRAM main memory. (5pt)



[Answer]

Multiple outstanding requests on the bus
Placing data across banks (not on a single bank)
Access scheduling (e.g., FR-FCFS) in memory controller (MC)

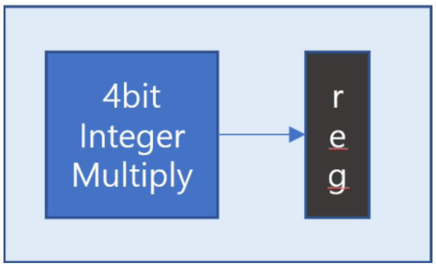
Q5. Lab04 – Adder array, Lab05 - PE (10 points)

We have implemented a MAC with a register in each PE in the lab practice, where a single PE computes the inner product of two input vectors.
However, you can also accumulate the output value by using an Adder Tree. The element-wise multiplication is done by several PEs, and the adder tree accumulates the output values.
In this problem, you are going to design an array of multipliers and an adder tree.

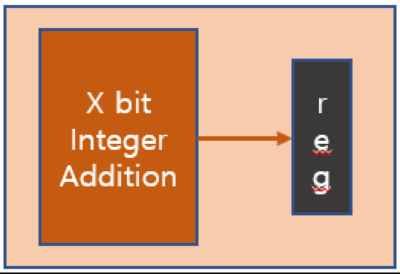
Problem 5-1. Overflow (2 points)

Fill in the blanks indicating the lowest possible bitwidth of the output so that there is no overflow.
(Assume unsigned Integer)

```
module mult#(
        parameter BITWIDTH = 4
    )
    (
        input [BITWIDTH-1:0] ain,
        input [BITWIDTH-1:0] bin,
        input clk,
        input reset,
        output reg [ _______: _______] out


    );

    always @(posedge clk or negedge reset) begin
        if(!reset) begin
            out <= 'b0;
        end
        else begin
            out <= ain * bin;
        end
    end
endmodule
```



```
module add#(
        parameter BITWIDTH = 8
    )
    (
        input [BITWIDTH-1:0] ain,
        input [BITWIDTH-1:0] bin,
        input clk,
        input reset,
        output reg [ _______: _______] out


    );

    always @(posedge clk or negedge reset) begin
        if(!reset) begin
            out <= 'b0;
        end
        else begin
            out <= ain + bin;
        end
    end
endmodule
```

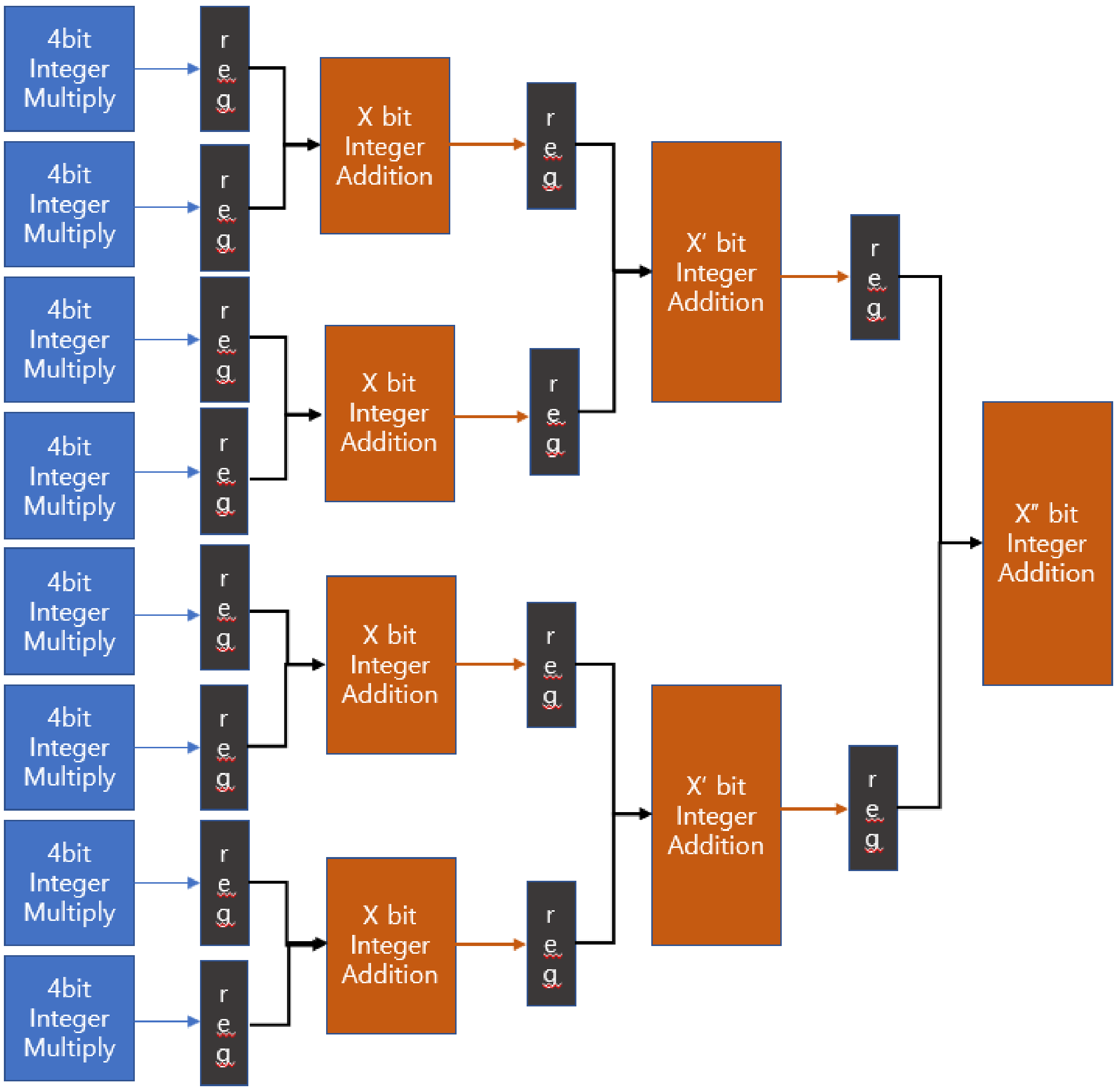


[Answer]
[2*BITWIDTH - 1: 0] / [BITWIDTH : 0]

Problem 5-2. Integer 4-bit multipliers + Adder Tree (8 points)

The following figure shows the block design of Integer 4-bit multipliers and an adder tree.
Complete the code of the adder tree that satisfies the conditions below according to the block design

- Assume unsigned integer
- Use the lowest possible bitwidth where overflow does not occur for all the wires and registers.
- Use at least one generate-for statement.
- You can choose to use or not use the modules 'mult' and 'add' in Problem A
- Synchronous to clock
- Code should be synthesizable



```
module adder_tree#(
        parameter BITWIDTH = 4,
        parameter TREE_WIDTH = 8
    )
    (
        input [BITWIDTH-1:0] ain0, ain1, ain2, ain3, ain4, ain5, ain6, ain7,
        input [BITWIDTH-1:0] bin0, bin1, bin2, bin3, bin4, bin5, bin6, bin7,
        input clk,
        input reset,
        output reg [2*BITWIDTH+2 : 0] out
    );

    wire [BITWIDTH-1:0] ain [7:0];
    wire [BITWIDTH-1:0] bin [7:0];
    wire [2*BITWIDTH-1:0] temp_out1[7:0];
    reg [2*BITWIDTH:0] temp_out2[3:0];
    reg [2*BITWIDTH+1:0] temp_out3[1:0];

    assign{ain[0], ain[1], ain[2], ain[3], ain[4], ain[5], ain[6], ain[7]} = {ain0, ain1, ain2, ain3, ain4, ain5, ain6,
ain7};
    assign{bin[0], bin[1], bin[2], bin[3], bin[4], bin[5], bin[6], bin[7]} = {bin0, bin1, bin2, bin3, bin4, bin5, bin6,
bin7};

    always @(posedge clk or negedge reset) begin
        if(!reset) begin
            out <= 'b0;
            temp_out2[0] <= 'b0;
            temp_out2[1] <= 'b0;
            temp_out2[2] <= 'b0;
            temp_out2[3] <= 'b0;
            temp_out3[0] <= 'b0;
            temp_out3[1] <= 'b0;
        end
        else begin
            temp_out2[0] <= temp_out1[0] + temp_out1[1];
            temp_out2[1] <= temp_out1[2] + temp_out1[3];
            temp_out2[2] <= temp_out1[4] + temp_out1[5];
            temp_out2[3] <= temp_out1[6] + temp_out1[7];
            temp_out3[0] <= temp_out2[0] + temp_out2[1];
            temp_out3[1] <= temp_out2[2] + temp_out2[3];
            out <= temp_out3[0] + temp_out3[1];
        end
    end
    genvar i;
    generate
        for(i = 0; i < TREE_WIDTH; i = i + 1) begin : mult
            mult #(.BITWIDTH(BITWIDTH)) u_mult(
                .ain(ain[i]),
                .bin(bin[i]),
                .clk(clk),
                .reset(reset),
                .out(temp_out1[i])
            );
        end
    endgenerate
endmodule
```

Q6. Lab05 – BRAM (10 points)

Problem 6-1. (1 point)

Fill in the blank as we learned BRAM in Lab5. The requirements of how BRAM should perform are specified below.

<Requirements>

BRAM takes 2 cycle to read, 1 cycle to write.
Do not use delay operation (ex: #10)

- BRAM_ADDR: Value of address that comes from outside of BRAM.
- BRAM_CLK: clock.
- BRAM_WRDATA: BRAM's data input port.
- BRAM_RDDATA: BRAM's data output port.
- BRAM_EN: if ‘BRAM_EN’ is 1, BRAM read or write, and ‘mem[addr]’ is printed out to ‘BRAM_RDDATA'.
- BRAM_WE: if ‘BRAM_WE[i]’ is 1, ‘BRAM_WRDATA[8*i-1:8*(i-1)]’ is stored to ‘mem[addr][8*i-1:8*(i-1)]’.
- BRAM_RST: if ‘BRAM_RST’ is 1, ‘BRAM_RDDATA’ prints out 0.

*** No additional register is needed. Just use the defined variables.**

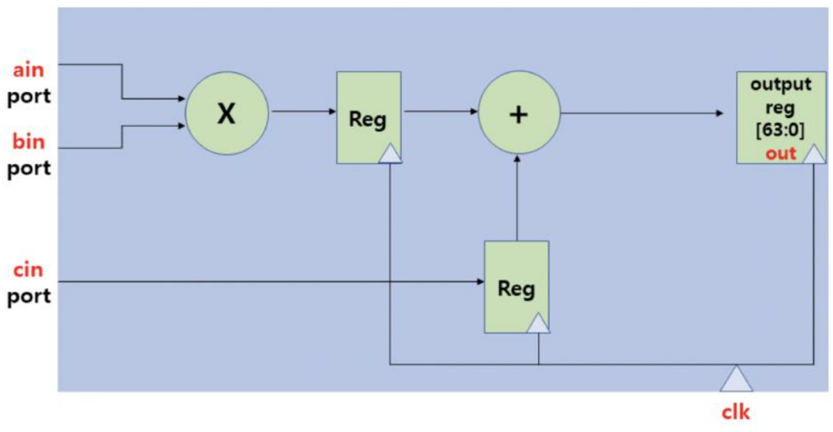
```
`timescale 1ns / 1ps
module my_bram # (
    parameter integer BRAM_ADDR_WIDTH = 15, // 4x8192
    parameter INIT_FILE = "input.txt",
    parameter OUT_FILE = "output.txt"
)(
    input wire [BRAM_ADDR_WIDTH-1:0] BRAM_ADDR,
    input wire BRAM_CLK,
    input wire [31:0] BRAM_WRDATA,
    output reg [31:0] BRAM_RDDATA,
    input wire BRAM_EN,
    input wire BRAM_RST,
    input wire [3:0] BRAM_WE,
    input wire done
);

    reg [31:0] mem[0:11];
    wire [BRAM_ADDR_WIDTH-3:0] addr = BRAM_ADDR[BRAM_ADDR_WIDTH-1:2];
    reg [31:0] dout;

    initial begin
        if (INIT_FILE != "") begin
            $readmemh(INIT_FILE, mem);
        end
        wait(done)
            $writememh(OUT_FILE, mem);
    End

    always @(posedge BRAM_CLK)
        if (BRAM_WE[0] && BRAM_EN)
            mem[addr][7:0] <= BRAM_WRDATA[7:0];

    always @(posedge BRAM_CLK)
        if (BRAM_WE[1] && BRAM_EN)
            mem[addr][15:8] <= BRAM_WRDATA[15:8];
        always @(posedge BRAM_CLK)
            if (BRAM_WE[2] && BRAM_EN)
                mem[addr][23:16] <= BRAM_WRDATA[23:16];

        always @(posedge BRAM_CLK)
            if (BRAM_WE[3] && BRAM_EN)
                mem[addr][31:24] <= BRAM_WRDATA[31:24];

always @(posedge BRAM_CLK) begin
            dout <= mem[addr];
            if (BRAM_RST)
                BRAM_RDDATA <= 'd0;
            else if (BRAM_EN)
                BRAM_RDDATA <= dout;
        end
 endmodule
```

Problem 6-2. (2 points)

Implement the synthesizable integer multiply-adder module as specified below. No additional register is needed.

- Requirements

i.	The intermediate value for calculation and result should be stored in reg type variable, and they should be synchronous to clock.
ii.	Your module should act like the block diagram below. The block diagram shows only ain, bin, cin, out, and clk.
iii.	If 'valid' input is 1 for one cycle with ain, bin and cin, when the corresponding calculated output is stored to 'out', 'dvalid' output should be 1 for one cycle.
iv.	The registers should act synchronously to clk. (using *always @(posedge clk)*)

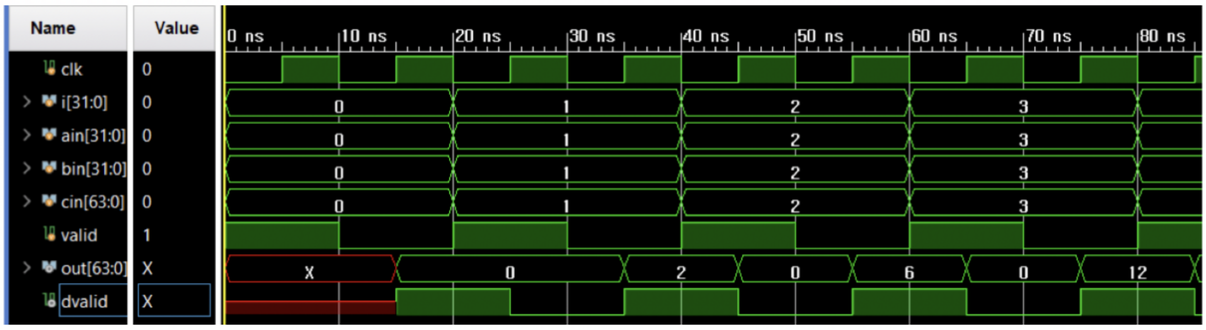


```
module my_integer_mult_add(
        input wire [31:0] ain,
        input wire [31:0] bin,
        input wire [63:0] cin,
        input wire valid,
        input wire clk,
        output reg dvalid,
        output reg [63:0] out
);
        reg [63:0] temp_mult;
        reg temp_valid;
        reg [63:0] temp_cin;

        always @(posedge clk) begin
```

```
                mult_temp <= ({32{valid}}& ain) * ({32{valid}} &bin);
                temp_cin <= ({64{valid}}&cin);
                temp_valid <= valid;
                out <= temp_cin + temp_mult;
                dvalid <= temp_valid;
```

```
        end
endmodule
```

Hint: Check the waveform below. The values are shown as decimal.



Problem 6-3. (7 points)
The testbench code which use the **6-1**'s BRAM and **6-2**'s integer mult-add.

```
module check_multadd #(
        parameter integer BRAM_ADDR_WIDTH = 6
    );
    //BRAM_in/output
    reg [BRAM_ADDR_WIDTH-1:0] BRAM_ADDR;
    reg [31:0] res_reg;
    wire [31:0] BRAM_RDDATA;

    reg clk;
    reg BRAM_EN;
    reg [3:0] BRAM_WE;
    reg BRAM_RST;
    wire done;

    my_bram   #(BRAM_ADDR_WIDTH,   "input.txt",
"output.txt") BRAM_test(
            .BRAM_ADDR(BRAM_ADDR),
            .BRAM_CLK(clk),
            .BRAM_WRDATA(res_reg),
            .BRAM_RDDATA(BRAM_RDDATA),
            .BRAM_EN(BRAM_EN),
            .BRAM_RST(BRAM_RST),
            .BRAM_WE(BRAM_WE),
            .done(done)
    );
    //multadd in/output
    reg[31:0] ain;
    reg[31:0] bin;
    reg[63:0] cin;
    reg valid;
    wire dvalid;
    wire [63:0] res;

    my_integer_mult_add int32_multadd(
            .ain(ain),
            .bin(bin),
            .cin(cin),
            .valid(valid),
            .clk(clk),
            .out(res),
            .dvalid(dvalid)
    );

    reg [31:0] counter_dvalid;
    reg[31:0] cal_register [11:0];
    integer i,j ;

    //initialize
    initial begin
            BRAM_RST <= 0;
            BRAM_WE <= 0;
        BRAM_EN <= 1;
        clk <= 0;
        cin <= 0;
        valid <= 0;
        counter_dvalid <= 0;
    end

    //BRAM address value
    initial begin
        for (i=0; i<16; i=i+1) begin
            BRAM_ADDR <= (i%12) * 4;
            #8;
        end
    end

    //calculate
    initial begin
        #20;
        for (j=0; j<12; j=j+1) begin
            cal_register[j] <= BRAM_RDDATA;
            #10;
        end

        for (j=0; j<4; j=j+1) begin
            valid <=1;
            ain <= cal_register[3*j];
            bin <= cal_register[3*j+1];
            cin <= cal_register[3*j+2];
            #10;
            valid <= 0;
            #10;
        end
    end

//write BRAM
    always@(posedge dvalid) begin
        res_reg <= res;
        BRAM_WE <= 4'b0101;
        BRAM_ADDR <= (counter_dvalid) <<2;
        #10;
        BRAM_WE <= 4'b0000;
        counter_dvalid <= counter_dvalid + 1;
    end

    assign done = (counter_dvalid==4'b100);

    always #5 clk = ~clk;

endmodule
```

```
3
2
121
300
5
2
210
8
1
D
300
4
```

<**input.txt** that the testbench code uses as the contents of BRAM module>

(a) Write all values in hexadecimal in the cal_register at the time when BRAM_WE is set to 4'b0101 for the first time. (2 points)

cal_register's index	value
0	0x00000003
1	0x00000002
2	0x00000300
3	0x00000005
4	0x00000002
5	0x00000210
6	0x00000001
7	0x0000000d
8	0x00000300
9	0x00000004
10	0x00000002
11	0x00000121

(b) Write all values in the file 'output.txt' after wire 'done' is set to 1 and the BRAM module's the writememh function is executed. You do not need to fill zeros in front of values. (5 points)

```
00000006
0000001a
0000010d
00000329
00000005
00000002
00000210
00000008
00000001
0000000d
00000300
00000004
```

Q7. Lab06 - PE Controller (10 points)

```
module pe_controller (
      input start, areset, aclk,
      input [31:0] rddata,
      output [4:0] rdaddr,
      output reg [31:0] out,
      output done
);

  reg valid;
  wire dvalid;
  reg [31:0] psum;
  wire [31:0] dout;
  reg [1:0] present_state, next_state;
  reg [4:0] cnt_LOAD, cnt_CALC, cnt_DONE;
localparam S_IDLE = 2'd0, S_LOAD = 2'd1, S_CALC = 2'd2, S_DONE = 2'd3;

  reg [31:0] GlobalBuffer [0:31];

  assign rdaddr = cnt_LOAD;
  assign done = (present_state == S_DONE);

  always @(posedge aclk)
      present_state <= (!areset) ? S_IDLE : next_state;

  always @(*)
      case (present_state)
          S_IDLE: next_state <= (start) ? S_LOAD : S_IDLE;
          S_LOAD: next_state <= (cnt_LOAD == 31) ? S_CALC : S_LOAD;
          S_CALC: next_state <= (cnt_CALC == 15 && dvalid) ? S_DONE : S_CALC;
          S_DONE: next_state <= (cnt_DONE == 4) ? S_IDLE : S_DONE;
  endcase

  always @(posedge aclk) begin  ··· (1)
      cnt_LOAD <= (present_state == S_LOAD) ? cnt_LOAD + 1 : 0;
      cnt_CALC <= (present_state == S_CALC && dvalid) ? cnt_CALC + 1 : 0;
      cnt_DONE <= (present_state == S_DONE) ? cnt_DONE + 1 : 0;

  end

  always @(posedge aclk)
      GlobalBuffer[cnt_LOAD] <= (present_state == S_LOAD) ? rddata : GlobalBuffer[cnt_LOAD];
```

always @(posedge aclk) begin ··· **(2)**

```
if (cnt_CALC == 15)
   valid <= 0;
else if (present_state == S_CALC)
   valid <= dvalid;
else if (next_state == S_CALC)
   valid <= 1;
else
   valid <= 0;
```

end

always @(posedge aclk) begin ··· **(3)**

```
if (cnt_CALC == 15 && dvalid)
   out <= dout;
else if (next_state == S_DONE)
   out <= out;
else
   out <= 0;
```

end

always @(posedge aclk) begin ··· **(4)**

```
if (!areset | done)
     psum <= 'd0;
else if (dvalid)
   psum <= dout;
```

end

```
floating_point_MAC MAC(
   .aclk(aclk),
   .aresetn(areset),
   .s_axis_a_tvalid(valid),
   .s_axis_b_tvalid(valid),
   .s_axis_c_tvalid(valid),
    .s_axis_a_tdata(GlobalBuffer[cnt_CALC]),
    .s_axis_b_tdata(GlobalBuffer[cnt_CALC + 16]),
    .s_axis_c_tdata(psum),
    .m_axis_result_tvalid(dvalid),
    .m_axis_result_tdata(dout)
  );
endmodule
```

Problem 7-1.

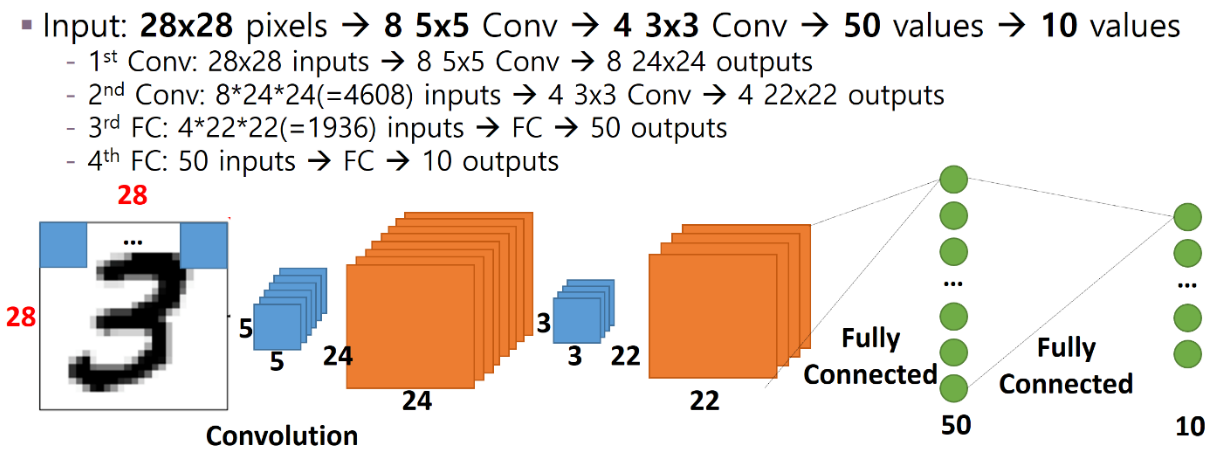
The above figure shows the code of PE controller that you have implemented in lab 6.
The FSM follows the following state transitions: *S_IDLE -> S_LOAD -> S_CALC -> S_DONE.*

Fill in the boxes (1) ~(4) in which the **(1) counters**(cnt_LOAD, cnt_CALC, cnt_DONE)**, (2) valid, (3) out, (4) psum** is/are controlled so that the code is synthesizable and satisfy the functionalities below. The other registers must not be controlled in each box. Also, no initialization of additional wires nor registers is allowed.

- The last three stages have their own counters which are incremented to trigger state transition at appropriate times.
- During *S_LOAD*, the global buffer loads data every cycle.
- During *S_CALC*, a pair of data is multiplied and the output is accumulated in the register "psum" until the FSM transits into *S_DONE*. The computation results do not depend on the computation latency of the MACs with proper control of the register "valid".
- During *S_DONE*, the final output is stored in the register "out" until the FSM transits into *S_IDLE*. The register holds zero during the other states.

Q8. Lab08 - Neural Network Operation (10 points)

Here is an example of neural network that classifies handwritten digits. As shown below, the network consists of two convolution layers and two fully connected layers. Note that other details of each operation follow our previous practices.



- Input: **28x28** pixels → **8 5x5** Conv → **4 3x3** Conv → **50** values → **10** values
  - 1st Conv: 28x28 inputs → 8 5x5 Conv → 8 24x24 outputs
  - 2nd Conv: 8*24*24(=4608) inputs → 4 3x3 Conv → 4 22x22 outputs
  - 3rd FC: 4*22*22(=1936) inputs → FC → 50 outputs
  - 4th FC: 50 inputs → FC → 10 outputs

Convolution operation should be calculated as matrix matrix multiplication, with convolution lowering. Fully connected layer should be calculated as matrix vector multiplication.

For matrix matrix multiplication and matrix vector multiplication, we use tiling to calculate small part of large matrix. And each tile sizes are denoted as (M1, M2) and (M2, M3). For matrix vector multiplication, tile size of matrix and vector are denoted as (M, V) and (V).

Problem 8-1. (3 points)

Calculate the total number of block operations and number of multiply operations using tiling where M1=4, M2=8, M3=12, M=8, V=6.

Problem 8-2. (3 points)

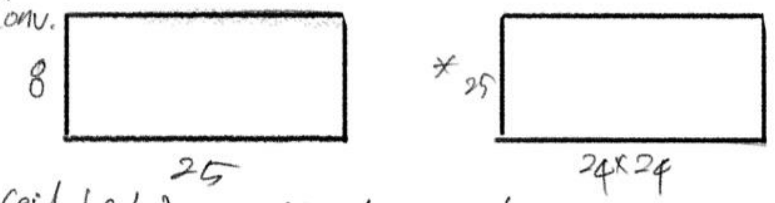
Calculate the number of multiply operations if we do not use tiling.

Problem 8-3. (4 points)

Is there any difference between the number of multiply operation in problem 8-1 and 8-2? Explain why. Explain the purpose of using tiling.

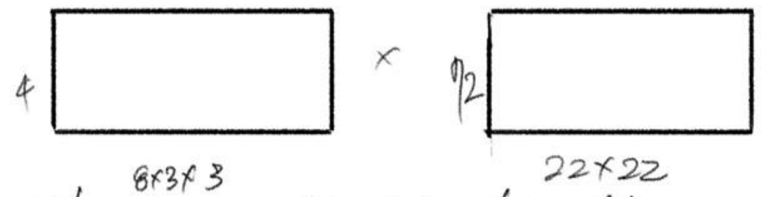
M1 = 4, M2 = 8, M3 = 12,   M = 8, V = 6.

1st. Conv.



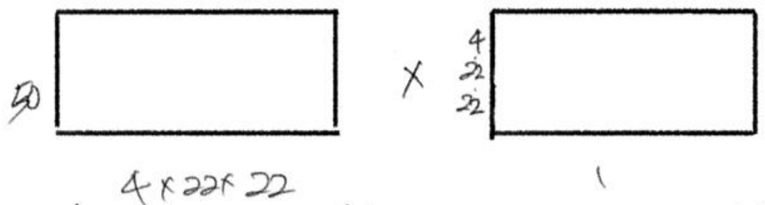
# blockops: $\text{ceil}\left(\underset{2}{8/4}\right) \times \text{ceil}\left(\underset{4}{25/8}\right) \times \text{ceil}\left(\underset{48}{24\times24/12}\right) = 384$

2nd Conv



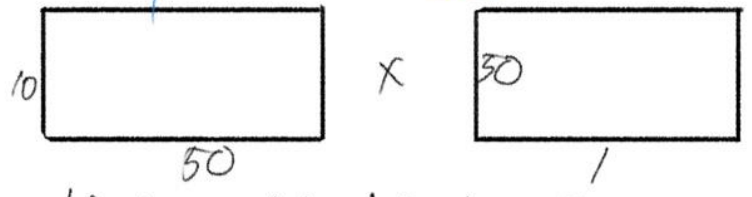
# block ops: $\text{ceil}\left(\underset{1}{4/4}\right) \times \text{ceil}\left(\underset{9}{72/8}\right) \times \text{ceil}\left(\underset{41}{22\times22/12}\right) = 369$

3rd FC



# block ops: $\text{ceil}\left(\underset{7}{50/8}\right) \times \text{ceil}\left(\underset{323}{4\times22\times22/6}\right) \times 1 = 2261$

4th FC



# block ops: $\text{ceil}\left(\underset{2}{10/8}\right) \times \text{ceil}\left(\underset{9}{50/6}\right) \times 1 = 18.$

total blockops : 3032 7

total Mul. ops : $\underbrace{(384+369)\times4\times8\times12}_{289,152} + \underbrace{(2261+18)\times48}_{109,392}$

b).

1st conv : $8\times25\times(24\times24) = 115200$

2nd Conv : $4\times72\times(22\times22) = 139392$

3rd FC : $50\times(4\times22\times22) = 96800$

4th FC : $10\times50 = 500$ ⏌ 351,892

c) a)이 더 많다. ⇒ tiling을 하면 남는 공간이 생길수 있는데 이 공간을 0으로 채우고 곱셈을 추가적으로 해야하기 때문이다.
tiling을 사용하는 이유는 곱셈 연산을 할때 행렬 크기가 너무커서 한번에 fit 하지 않기에 fit 하게 만들기 위함.
tiling을 통해서 cache hit rate를 높일수 있다.