**Hardware System Design Final Exam**

2020.06.09 13:30~14:30

(6 questions, 20 points each, x pages)

\*\*\*IMPORTANT: You have to choose 5 out of 6 questions to answer. If you write answers to 6 questions, the highest scoring question will be excluded.\*\*\*

Student ID:

Name:

**Q1. Choose either 1-A or 1-B then answer it. (20 points)**

**Problem 1-A. (Quantization)**

Quantization is one of the most frequently used methods to efficiently execute the neural network. Explain what quantization is and describe two possible advantages and disadvantages in applying quantization to the neural network. (2 advantages, 2 disadvantages)

[Answer]

Quantization is… (Reasonable한 설명이면 정답처리) ***<10점>***

Advantage: (Reasonable한 설명이면 정답처리) ***<각 2.5점>***

Disadvantage: (Reasonable한 설명이면 정답처리) ***<각 2.5점>***

**Problem 1-B. (Weight Pruning)**

Weight pruning is another frequently used method to efficiently run the neural network. Explain what weight pruning is and describe two possible advantages and disadvantages in applying weight pruning to the neural network. (2 advantages, 2 disadvantages)

[Answer]

Pruning is… (Reasonable한 설명이면 정답처리) ***<10점>***

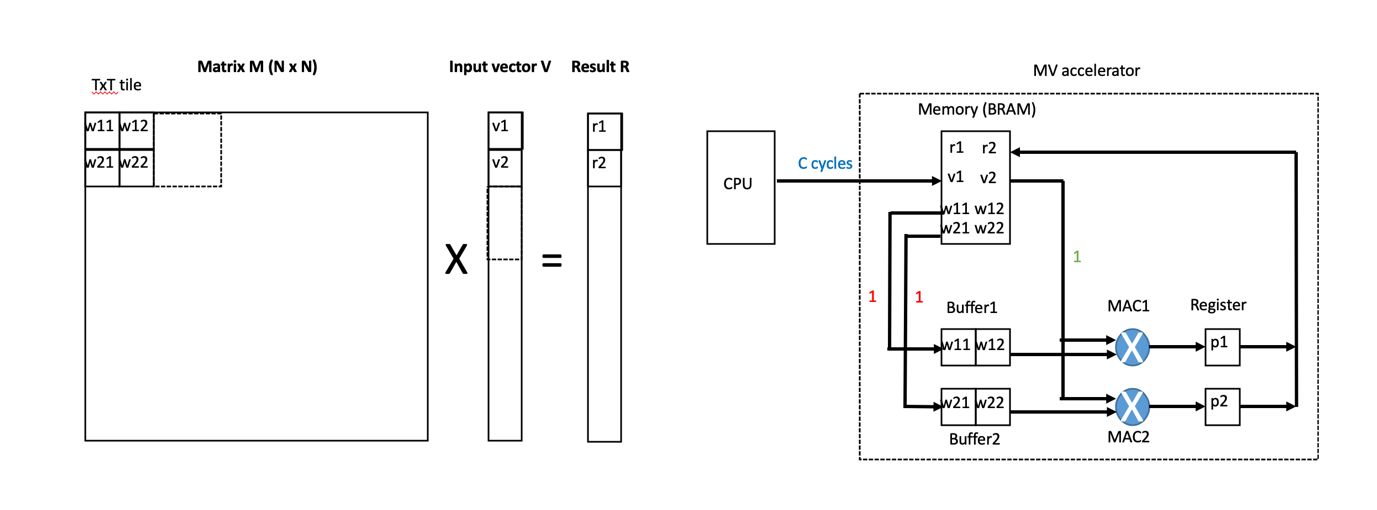
Advantage: (Reasonable한 설명이면 정답처리) ***<각 2.5점>***

Disadvantage: (Reasonable한 설명이면 정답처리) ***<각 2.5점>***

**Q2. Choose either 2-A or 2-B then answer it. (20 points)**

**Problem 2-A. (Layerwise Quantization)**

In last week’s lab, TA explained layerwise quantization for project v2. Describe what layerwise quantization is, why it is better than quantization you implement in project v1, and how you can implement layerwise quantization using the diagram below.



[Answer]

What layerwise quantization is ***<6점>***

* Quantize the weight & input of a neural network layer by layer

Why it is better than quantization you implement in project v1 ***<7점>***

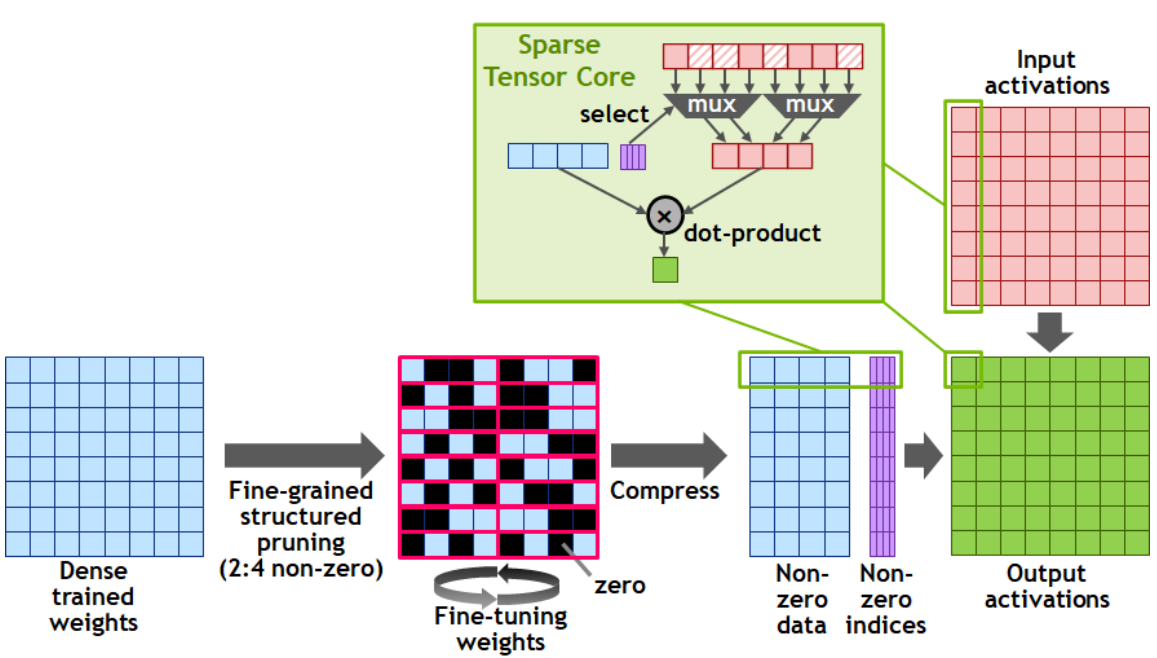
* It is better than project v1 because we do not execute redundant quantization operation caused by redundant value transfer from cpu to fpga (Because we execute quantization operation in cpu, before transferring data to fpga)

How you can implement layerwise quantization using the diagram below ***<7점>***

* 1. Quantize all the weights & inputs required to calculate a layer of a neural network
* 2. Transfer all the quantized weights & inputs required to calculate a layer of a neural network from cpu to fpga
* 3. Tiling the weights & inputs in bram to perform the operation corresponding to one layer of a neural network

**Problem 2-B. (Zero Skipping)**

The following figure explains how NVIDIA A100 exploits zero skipping. As one can see, the weight matrix is pruned with a 2-out-of-4 non-zero pattern (structured sparsity). After weight pruning, non-zero values of weight matrix and corresponding indices are obtained. As a result, A100 can double the math throughput by zero skipping.



In this architecture, assume the non-zero indices are represented by a bit-vector. Write a code snippet (a few lines of code) in Verilog to select input activations associated with non-zero weights within two clock cycles.

(Hint: Consider a case that the non-zero indices are 4’b1010, e.g., bitVec = 4’b1010, which means the 1st and 3rd activations are non-zero while 2nd and 4th activations are zero.)

[Answer]

다양한 답이 가능함

아래는 대표적인 예시:

Example segment:



1. Generate bit vector for target segment

bitVec: 4’b1010 🡪 여기까지는 문제에서 주어짐

2. Use priority encoder twice:

Nonzero\_Idx[1] = |bitVec[3:2]

Nonzero\_Idx[0] = bitVec[3] || bitVec[1]

1st use of priority encoder to find first nonzero index 🡪 1st nonzero index: 11

Remove first nonzero from bit vector 🡪 new bitVec: 4’b0010

2nd use of priority encoder to find second nonzero index 🡪 2nd nonzero index: 01

Mux corresponding data

Policy

Priority encoder를 두 번 사용한다. Priority encoder의 동작 설명까지 자세히 설명(위): 20점 만점

스스로 gate level 서킷을 작성해서 2cyc (1cyc 두 단계) 안에 index 2개 추출 성공: 20점 만점

(어쨌거나 일반적인 segment에 대해 답이 맞으면 만점)

Priority encoder를 사용해 nonzero 위치의 index를 추출한다 라고만 작성: 5점

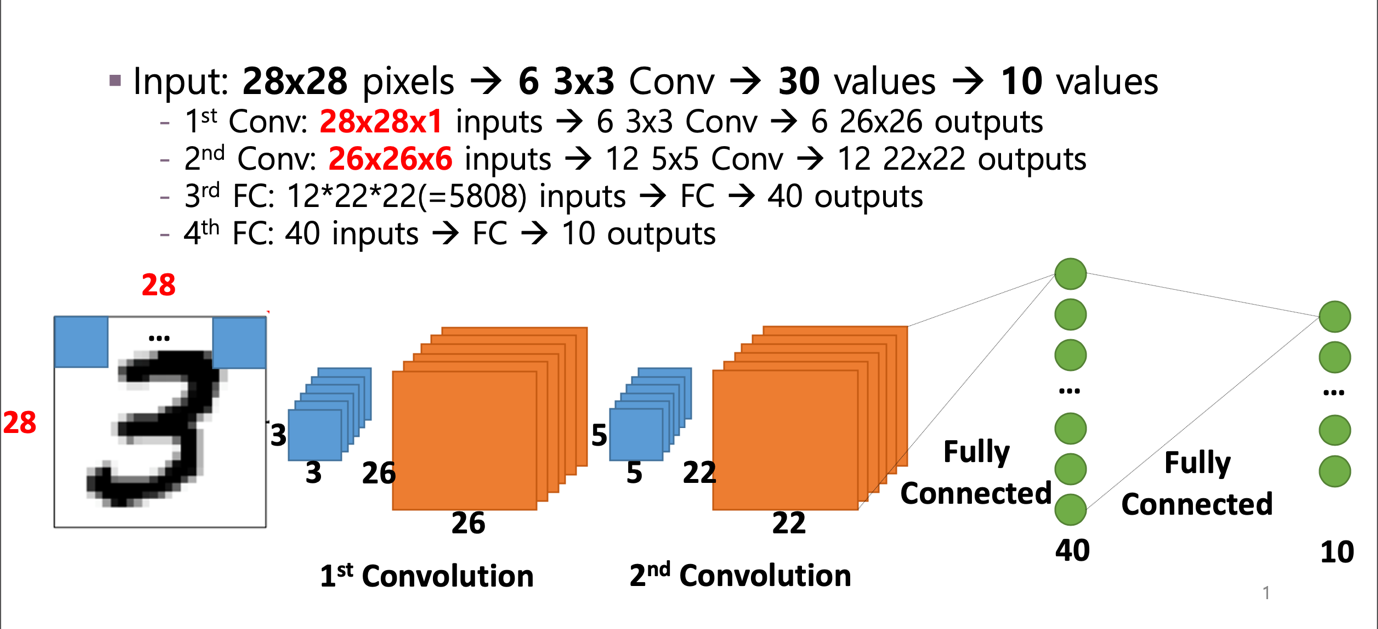
Priority encoder를 각각의 nonzero에 대해 두 번 사용한다 라고만 작성: 10점

Priority encoder의 동작 설명(위와 같이): 7점

Bit vector를 작성 또는 답(index)이 틀린 경우: 0점

**Q3. (Neural Network Operation) (20 points)**

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



**Problem 3-1. (10 points)**

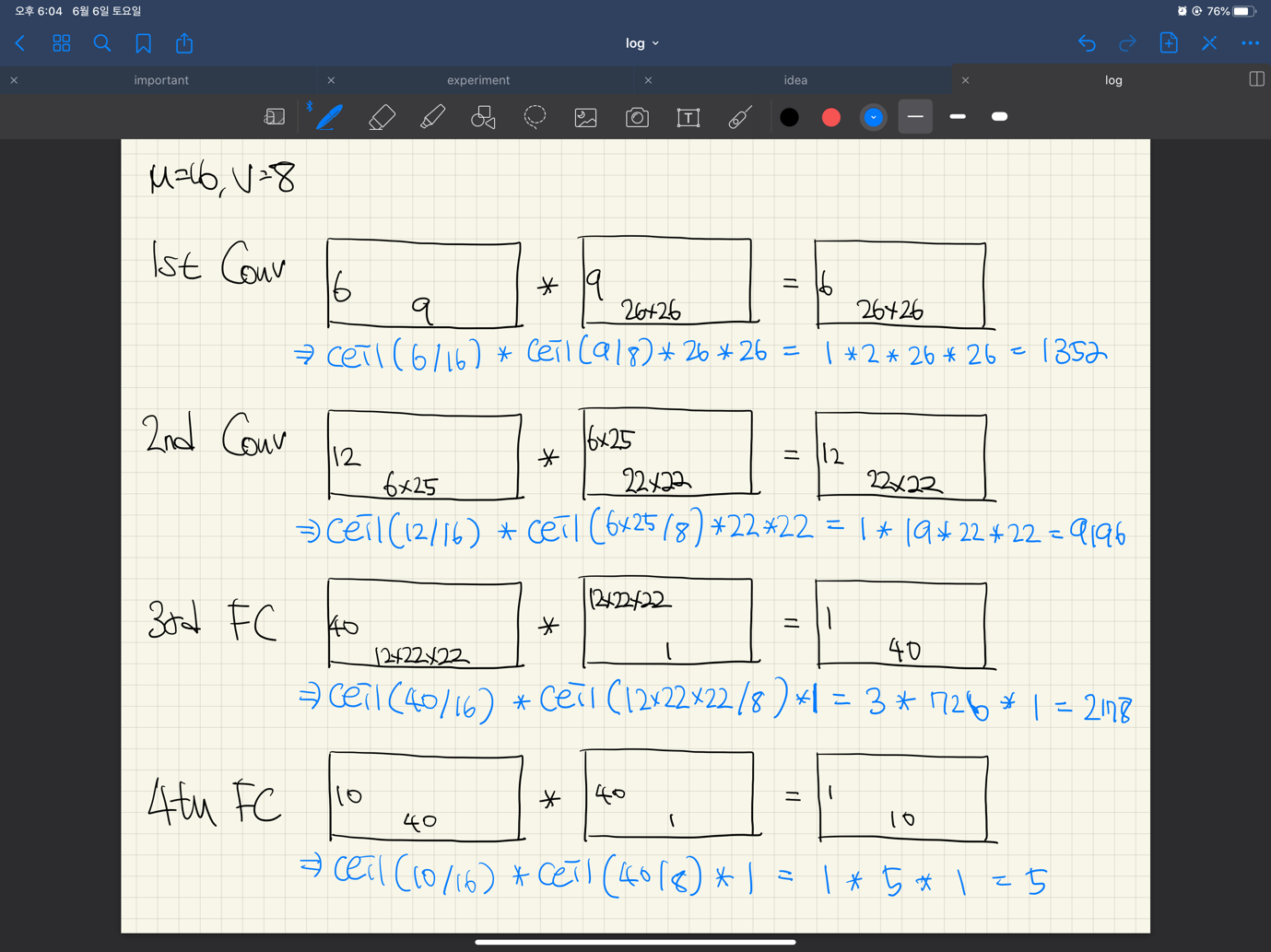
Calculate the total number of block operations where M=16, and V=8.

(Note that M is the number of PEs in PE array, and V is the number of elements in input vector)

[Answer]

Total = 12731 = 1352 + 9196 + 2178 + 5

***<Layer별로 Calculation 맞으면 각 2점씩, 모두 맞으면 10점>***



**Problem 3-2. (10 points)**

Total number of block operations is the same in both cases of M=16, V=8 and M=8, V=16.

Answer TRUE or FALSE, and explain why.

(Note that M is the number of PEs in PE array, and V is the number elements in input vector)

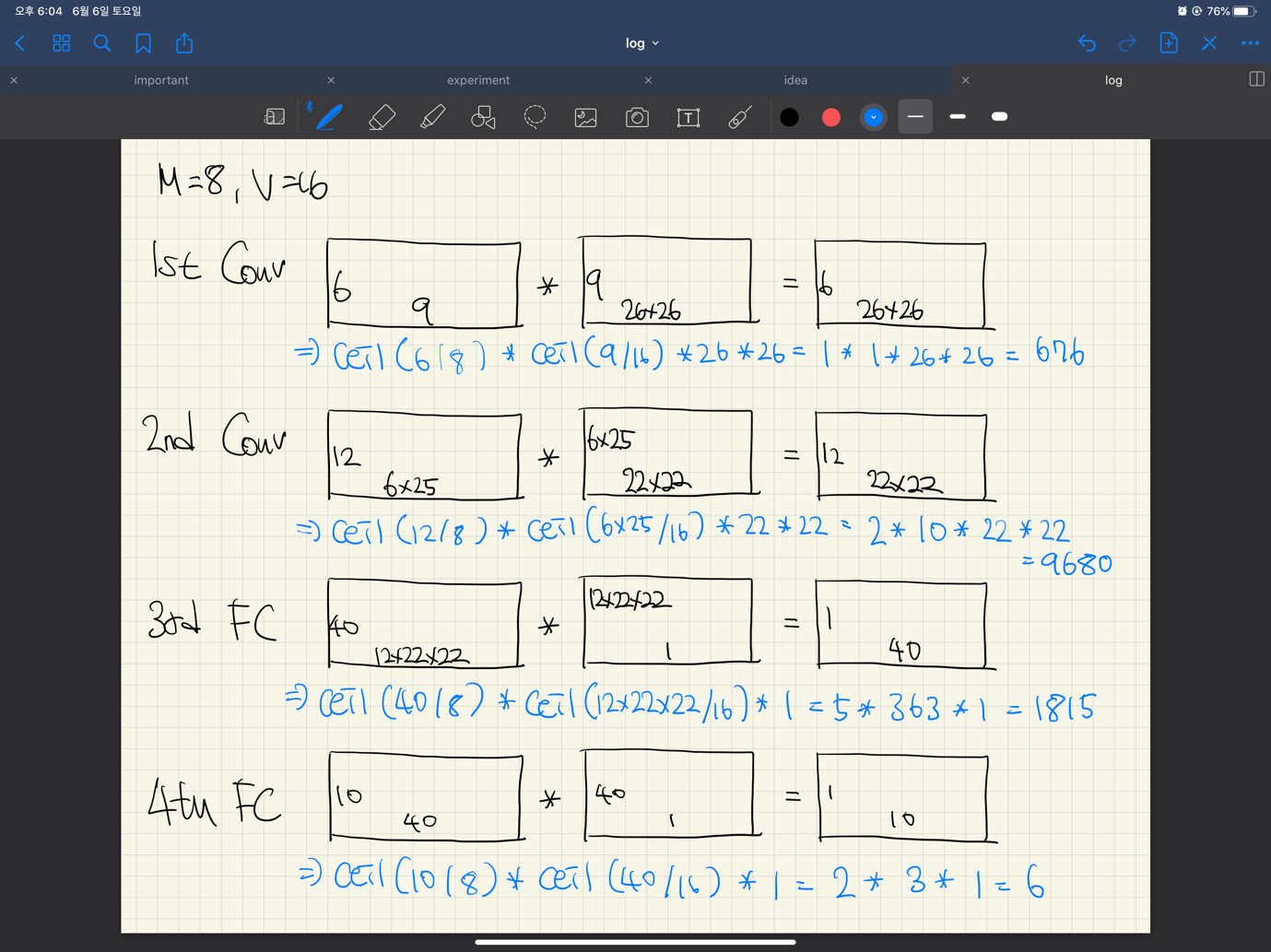
[Answer]

FALSE

M=8, V=16 total = 12177 = 676 + 9680 + 1815 + 6

***<TRUE FALSE 맞으면 5점,***

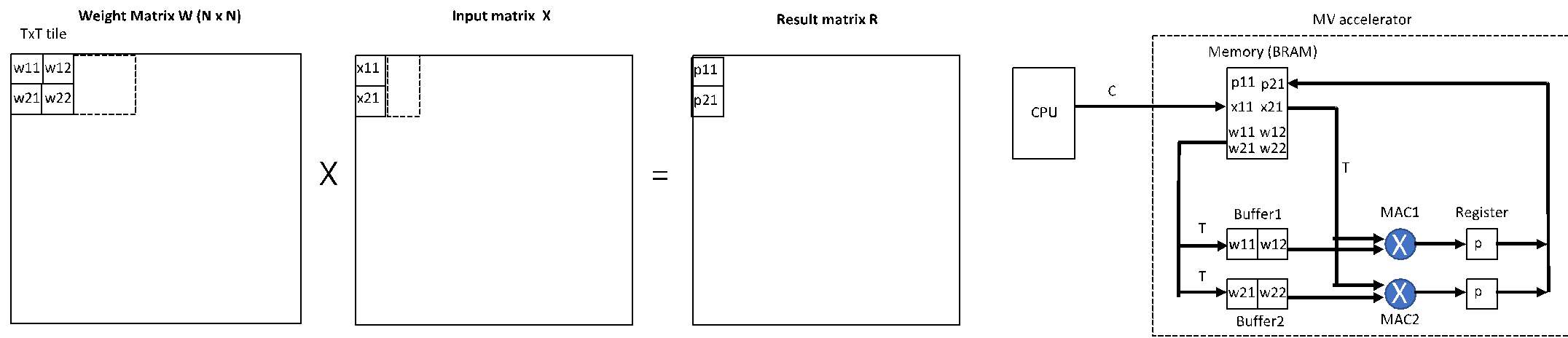
***>***



**Q4. (Matrix-vector Multiplication in Hardware) (20 points)**

The following figure (left) shows a tile-based matrix-matrix multiplication. Assume that we use MxV weight tile and Vx1 input tile. Assume that we use our accelerator shown in the following figure (right). The figure shows the number of execution cycles on each operation for the multiplication of a weight tile and an input tile. The communication (for the transfer of a pair of weight (MxV) and input tiles (Vx1)) from CPU to the FPGA memory (BRAM) of MV accelerator takes C, and D cycles each, which gives C+D cycles for the transfer of a pair of weight (MxV) and input (Vx1) tiles. Weight transfer from the FPGA memory (BRAM) to *each* PE buffer takes V cycles, which gives M\*V cycles for the transfer of entire MxV weight tile from the FPGA memory to M PE buffers. Broadcast of input data from the FPGA memory to MAC units and computation takes V cycles to consume a Vx1 input tile.

The dimensions of weight (W), input (X), and result (R) matrices are IxJ, JxK, and IxK, respectively. We assume that J is a multiple of V, V is a multiple of M, and C is a multiple of M. Returning the result from MV accelerator to CPU does not take any cycle.

****

**Problem 4-1. Case (A) (8 points)**

Calculate the total execution cycle of matrix multiplication of weight W and input X to obtain matrix R. (You can use / operator to express its quotient, and use % operator to express its remainder.)

[Answer]

Total execution cycles = ((P) \* (M/Q) \* (N/T)) \* ((C+D) + (Q\*T) + T)

+ ((P) \* (1) \* (N/T)) \* ((C\*(M%Q)/Q + D) + (T\*(M%Q)/Q) + T)

Red: Execution cycles for the quotient of weight matrix

Each weight tile is transferred P times and we have (M/Q) \* (N/T) such weight tiles. Also we need (C+D) cycles for transferring a pair of weight tile and input tile from cpu to fpga, (Q\*T) cycles for transferring a weight tile from bram to buffers, and T cycles for a input tile broadcasting and computing.

Blue: Execution cycles for the remainder of weight matrix

Each weight tile is transferred P times and we have (N/T) such weight tiles. Also we need ((C\*(M%Q)/Q + D) cycles for transferring a pair of weight tile and input tile from cpu to fpga, (T\*(M%Q)/Q) cycles for transferring a weight tile from bram to buffers, and T cycles for a input tile broadcasting and computing.

|  |
| --- |
|  |
|  |

Quotient of weight matrix (M/Q)

Remainder of weight matrix (M%Q)

***<빨간색 계산 4점, 파란색 계산 4점>***

**Problem 4-2. Case (B) (12 points)**

A weight tile can be reused by the MV accelerator over multiple tile-based MV computations. When a weight tile is reused, it can be stored in the FPGA memory (BRAM and buffers in the figure) of MV accelerator.

Assume the communication cycle is constant, i.e., C cycles whether both weight and input tiles are transferred or only input tile is transferred. The communication cycle is still C cycles whether both full size tile (MxV) is transferred or partial size tile (size is less than MxV) is transferred together with the input tile.

We also assume that only a pair of weight tile and input tile can be stored in the local memory (=FPGA memory + PE buffer) of MV accelerator.

Calculate the minimum total execution cycle of W\*X matrix multiplication when the weight tile is reused.

[Answer]

Total execution cycles = ((M/Q) \* (N/T) \* (C + Q\*T)) + ((1) \* (N/T) \* (C + (T\*(M%Q)/Q))) + ((M/Q + 1) \* (N/T) \* (C + T)) \* P

Red: Weight tile transfer (for quotient) from CPU to local memory/buffer

Since the weight tile is reused by the MV accelerator, weight tile transfer from cpu to local memory, which consumes C cycles, occurs (M/Q) \* (N/T) times. For each new weight tile, transferring it from the local memory of MV accelerator to the buffer takes Q\*T cycles.

Blue: Weight tile transfer (for remainder) from CPU to local memory/buffer

Since the weight tile is reused by the MV accelerator, weight tile transfer from cpu to local memory, which consumes C cycles, occurs (N/T) times. For each new weight tile, transferring it from the local memory of MV accelerator to the buffer takes (T\*(M%Q)/Q) cycles.

Purple: input tile broadcast and computation

For each new weight tile, we need to send P input tiles from cpu to the local memory of MV accelerator, each consuming C cycles. For each input tile, computation takes T cycles. Note that the weights are reused by the MV accelerator. Thus, for each new input tile, we do not transfer the weights from the local memory to the buffer since they are already stored in the buffer. Thus, we need only to broadcast input data to MAC units, which takes T clock cycles for an input tile of 1XT. Thus, it takes C+T cycles for each input tile and P \* (C+T) cycles for each new weight tile. We have ((M/Q + 1) \* (N/T) such weight tile.

***<빨간색 계산 4점, 파란색 계산 4점, 보라색 계산 4점>***

**Q5. (Verilog Simulation) (20 points)**

module my\_con#(

parameter VECTOR\_SIZE = 4,

parameter L\_RAM\_SIZE = 6

)

(

input start,

output done,

input aclk,

input aresetn,

output [L\_RAM\_SIZE-1:0] rdaddr,

input [31:0] rddata,

output reg [31:0] wrdata

);

wire [31:0] ain;

wire [31:0] din;

wire valid;

wire dvalid;

wire [31:0] dout;

wire [L\_RAM\_SIZE-1:0] addr;

wire we\_global;

wire we\_local;

// Global buffer

reg [31:0] gdout;

(\* ram\_style = "block" \*) reg [31:0] globalmem [0:2\*\*VECTOR\_SIZE - 1];

always @(posedge aclk)

if (we\_global)

globalmem[addr] <= rddata;

else

gdout <= globalmem[addr];

// FSM

wire load\_done;

wire calc\_done;

wire done\_done;

reg [3:0] curr\_state, next\_state;

localparam S\_IDLE = 4'd0;

localparam S\_LOAD = 4'd1;

localparam S\_CALC = 4'd2;

localparam S\_DONE = 4'd3;

always @(posedge aclk)

if (!aresetn)

next\_state <= S\_IDLE;

else

case (next\_state)

S\_IDLE:

next\_state <= (start)? S\_LOAD : S\_IDLE;

S\_LOAD:

next\_state <= (load\_done)? S\_CALC : S\_LOAD;

S\_CALC:

next\_state <= (calc\_done)? S\_DONE : S\_CALC;

S\_DONE:

next\_state <= (done\_done)? S\_IDLE : S\_DONE;

default:

next\_state <= S\_IDLE;

endcase

always @(posedge aclk)

if (!aresetn)

curr\_state <= S\_IDLE;

else

curr\_state <= next\_state;

// COUNTER

reg[VECTOR\_SIZE+1:0] counter;

wire counter\_load\_init = (curr\_state == S\_IDLE) && (next\_state == S\_LOAD);

wire counter\_en = (next\_state == S\_LOAD) || dvalid || (next\_state == S\_DONE);

localparam N\_LOAD = 2\*\*(VECTOR\_SIZE+2) - 1;

localparam N\_CALC = 2\*\*VECTOR\_SIZE - 1;

localparam N\_DONE = 5 - 1;

// Counter behavior

always @(posedge aclk)

if (!aresetn)

counter <= 'd0;

else

if (counter\_load\_init)

counter <= N\_LOAD;

else if (load\_done)

counter <= N\_CALC;

else if (calc\_done)

counter <= N\_DONE;

else if (counter\_en)

counter <= counter - 1;

// WE signal assignment

assign we\_local = (curr\_state == S\_LOAD && counter[L\_RAM\_SIZE-1])? 'd1 : 'd0;

assign we\_global = (curr\_state == S\_LOAD && next\_state == S\_LOAD && !counter[L\_RAM\_SIZE-1])? 'd1 : 'd0;

// Drop the bits after VECTOR\_SIZE'th position so that both global & local buffer get the same addr value

assign rdaddr = (next\_state == S\_LOAD)? {{(L\_RAM\_SIZE-VECTOR\_SIZE){'b0}}, counter[VECTOR\_SIZE-1:0]} : 'd0;

assign addr = (next\_state == S\_CALC)? counter : rdaddr;

// Pass the data from global buffer to ain & rddata to din

assign ain = (next\_state == S\_CALC)? globalmem[addr] : 'd0;

assign din = (curr\_state == S\_LOAD)? rddata: 'd0;

// valid signal assignment

assign valid = ((curr\_state == S\_LOAD && next\_state == S\_CALC) || dvalid);

// State done signals

assign load\_done = curr\_state == S\_LOAD && counter == 'd0;

assign calc\_done = curr\_state == S\_CALC && counter == 'd0 && dvalid;

assign done\_done = curr\_state == S\_DONE && counter == 'd0;

// Final done signal

assign done = next\_state == S\_DONE;

// Pass the calculation result when the state is S\_DONE

always @(posedge aclk)

if (!aresetn)

wrdata <= 'd0;

else

if (calc\_done)

wrdata <= dout;

else

wrdata <= wrdata;

my\_pe #(

.L\_RAM\_SIZE(L\_RAM\_SIZE)

) u\_pe (

.aclk(aclk),

.aresetn(aresetn && (state != S\_DONE)),

.ain(ain),

.din(din),

.addr(addr),

.we(we\_local),

.valid(valid),

.dvalid(dvalid),

.dout(dout)

);

endmodule

**The code for a single PE controller is provided in pe\_con.v.**

The PE is operating in non-blocking mode (i.e., the valid signal cannot block the execution of an operation itself) with 16 cycle latency as in Lab session #06. The input data is a constant vector of 16 elements that all have the same value of 1. (That is, ain == din == 1.0.)

**Problem 5-1. (8 points)**

After reading the code carefully, complete the waveform on answer sheet 5.1. (Hint: fill in the signals when the state is S\_DONE first)

**Problem 5-2. (8 points)**

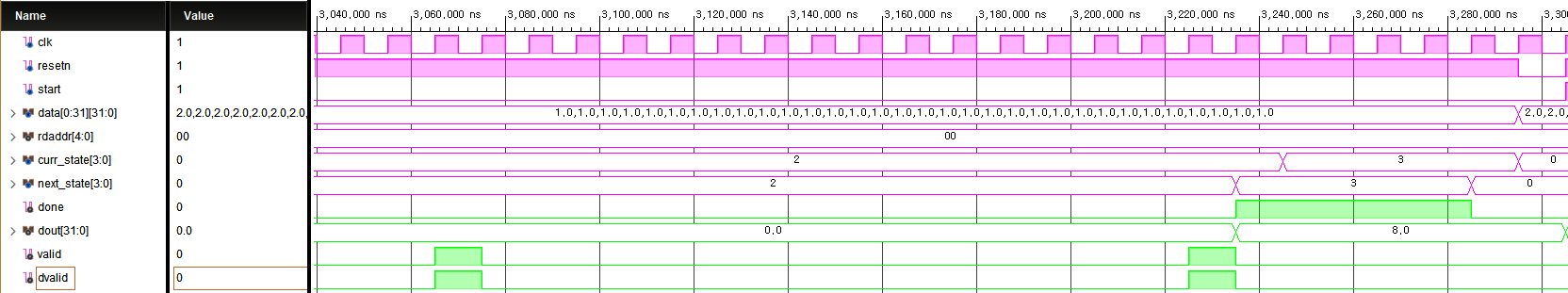
Identify the problem of pe\_con.v and fix the code in your own way.

**Problem 5-3. (4 points)**

With the correction you made, complete the waveform on answer sheet 5.3.

[Answer]

**Problem 5-1.**



Magenta: given / Green: to be filled

1) The final dout value must be 8.0 (the half of 16.0)

2) The timing of valid & dvalid must be correct

3) The timing of done signal must be correct

**Problem 5-2.**

Since the valid signal is activated simultaneously with dvalid signal, the partial sum within the PE is ignored once for every 2 operations. Therefore, the final result contains only the partial sums with odd parity.

The easiest way to fix this is to add a register that gives one-cycle dealy to dvalid signal, and modify valid assignment as below.

reg reg\_dvalid;

always @(posedge aclk)

if (!aresetn)

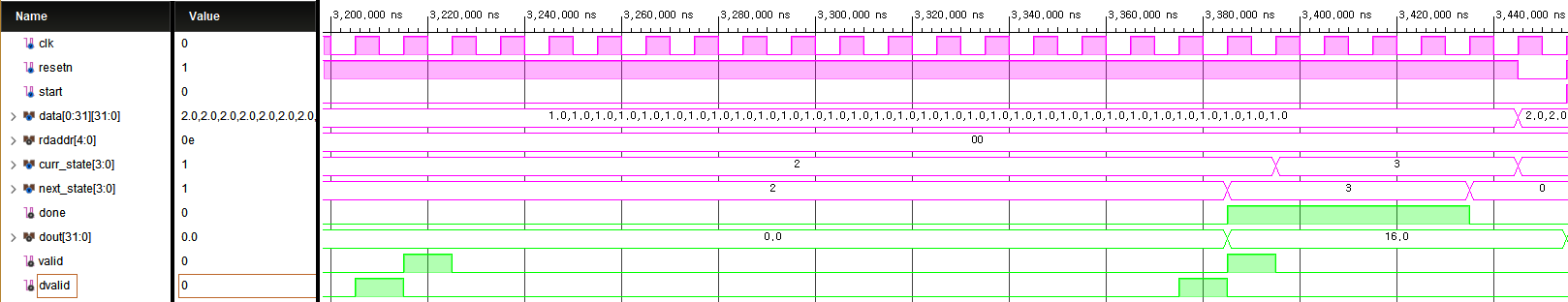
reg\_dvalid <= 0;

else

reg\_dvalid <= dvalid;

assign valid = ((curr\_state == S\_LOAD && next\_state == S\_CALC) || dvalid);

**Problem 5-3.**



Magenta: given / Green: to be filled

1) The final dout value must be 16.0 (the half of 16.0)

2) The timing of valid & dvalid must be correct

3) The timing of done signal must be correct

**Q6. Integer 8-bit & Floating Point 32-bit Multiplier (20 points)**

**Problem 6-A. (integer 8-bit multiplier) (4 points)**

The following shows the skeleton of an integer 8-bit multiplier.

Design integer 8-bit multiplier. (Assume that multiplier is combinational logic and adopts unsigned-integer)

* Code

|  |
| --- |
| module multiplier\_int8(  input [ : ] ain,  input [ : ] bin,  output [ : ] out  );  endmodule |

[Answer]

채점기준

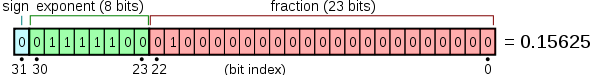
Input, output bit수 => 2점

Assign => 2점

|  |
| --- |
| module multiplier\_int8(  input [7:0] ain,  input [7:0] bin,  output [15:0] out  );  assign out = ain\*bin;  endmodule |

**Problem 6-B. (floating point 32-bit multiplier) (12 points)**

Consider floating point 32-bit (fp32) data format as follows.



The following shows the skeleton of a floating point 32-bit multiplier.

Design floating point 32-bit multiplier. (Assume that multiplier is combinational logic and ignores underflow.)

* Code

|  |
| --- |
| module multiplier\_fp32(  input [ : ] ain,  input [ : ] bin,  output [ : ] out  );        endmodule |

[Answer]

채점기준

Extra fields of A and B => 2점

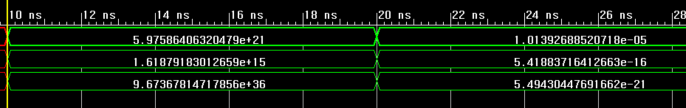
XOR sign bit => 2.5점

Multiply the fractions of A and B => 2.5점

Add exponents of A and B => 2.5점

Shift left one => 2.5점

|  |
| --- |
| module multiplier\_fp32 (  input [31:0] ain,  input [31:0] bin,  output [31:0] out  );  wire A\_s;  wire [7:0] A\_e;  wire [22:0] A\_f;  wire B\_s;  wire [7:0] B\_e;  wire [22:0] B\_f;  assign A\_s = ain[31];  assign A\_e = ain[30:23];  assign A\_f = {1'b1, ain[22:1]};  assign B\_s = bin[31];  assign B\_e = bin[30:23];  assign B\_f = {1'b1, bin[22:1]};  wire oProd\_s, oProd\_s\_out;  assign oProd\_s = A\_s ^ B\_s;  wire [45:0] pre\_prod\_frac;  assign pre\_prod\_frac = A\_f \* B\_f;  wire [8:0] pre\_prod\_exp;  assign pre\_prod\_exp = (A\_e - 9'd127) + (B\_e -9'd127);  wire [7:0] oProd\_e, oProd\_e\_out;  wire [22:0] oProd\_f\_out;  assign oProd\_e = (pre\_prod\_exp + 9'd127);    assign oProd\_s\_out = oProd\_s;  assign oProd\_e\_out = pre\_prod\_frac[45] ? oProd\_e + 9'd1 : oProd\_e;  assign oProd\_f\_out = pre\_prod\_frac[45] ? pre\_prod\_frac[44:22] : pre\_prod\_frac[43:21];  assign out = {oProd\_s\_out, oProd\_e\_out, oProd\_f\_out};  endmodule |



**Problem 6-C. (Integer & Floating Point on PE) (4 points)**

Consider using the above integer multiplier & floating point multiplier on our Processing Element (PE).

Which one will be more expensive in terms of execution time & resource? Also explain why.

[Answer]

Floating point multiply operation is more expensive.

Integer multiplier only operates with 8-bit multiply, while floating point multiplier needs several operations.

Floating point multiplier has more operations than Integer multiplier which is 1 XOR operation for sign, 23-bit multiply for fractions. 8-bit multiplication is common for both integer multiplier and floating point multiplier.

채점기준

Fp mult > integer => 2점

왜 fp mult > integer 인지 => 2점