EE405(B), Fall 2022

Electronics Design Lab. <Advanced Digital System Design>

- Student Names: Hyemin Lee
- Student IDs: 20910533

- LAB Project Number: Lab4, Lab5

1. Goal / Specification

A. Lab 4

The purpose of this lab was to design a simple pipelined SIMD processor for dot product. Single Instruction Multiple Data (SIMD) processor is a type of parallel processing hardware. It supports simultaneous computation with multiple data. Pipelined processor divides instruction into several stages and execute non-overlapping stages simultaneously. Therefore, with SIMD processor and pipelined processor, large amount of data can be processed in parallel at the same time. In this lab, we developed a processor that computes dot product of two 16 length vectors while controlling data dependency. Three memories were used to store input, weight, and output vectors.

B. Lab 5

The purpose of this lab was to design and implement a simple GEMM accelerator for matrix multiplication, accelerator the matrix multiplication on real FPGA hardware. This lab covered all the labs we've conducted before, especially the lab 3 and lab 4. In addition, memory bank concept was used to simultaneously read 16 values from input and weight memories. APB command designated control register values and initial value of input and weight memories. After adder tree completes dot product, the result value is saved in output memory and read by APB command.

2. Architecture / Design

A. Lab 4

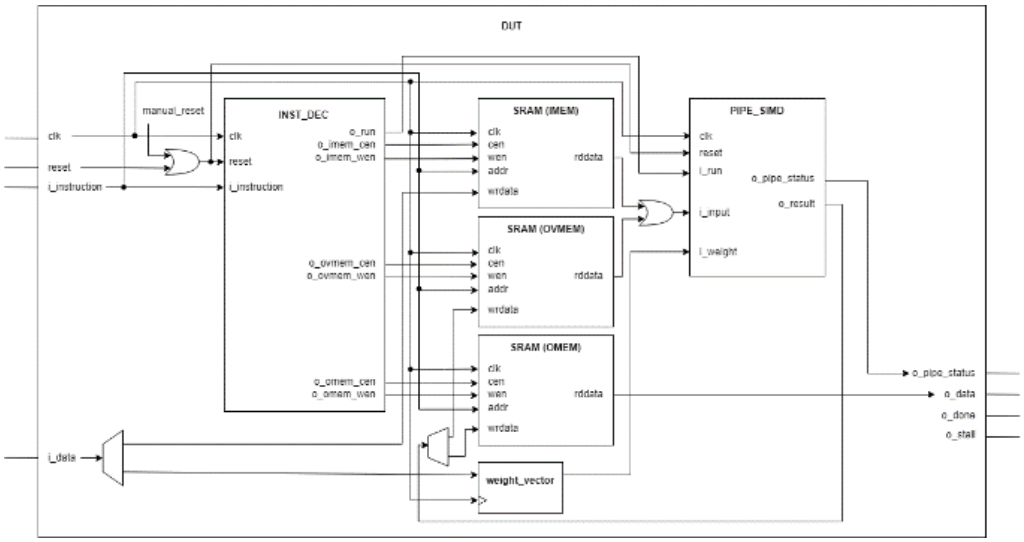
i. Description

There are four modules: DUT, INST_DEC, PIPE_SIMD, and SRAM. As testbench inserts instruction and input data, DUT and INST_DEC decodes the instruction with opcode and determines what operation they will execute among 7 operations. If the instruction was op0, it resets every register in the system and returns to idle state after 2 cycles. In case of op1, op2, and op6, it reads data from two different memories, conduct dot product, and save the value to the other memory. INST_DEC raise enable signals (control path) for each memory while DUT transfers data and calculated result to proper destination (data path). As INST_DEC enables input data from memories and DUT sends the data to PIPE_SIMD, PIPE_SIMD calculates dot product of two input data in 4 cycles. In total, 6 cycles are used for op1, op2, and op6. If the instruction was op3, INST_DEC enables input memory and DUT inserts input data to the memory. After 2 cycles, input memory value will be updated. If the instruction was

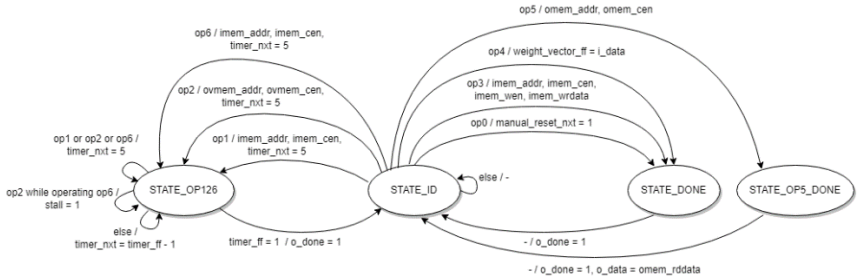
op4, it reads data from weight memory and fetches into weight register in 2 cycles. Lastly, in case of op5, it reads data from output memory and returns the value in 2 cycles.

Here, since op6 writes result value to output vector memory and op2 reads data from output vector memory, data dependency exists. Therefore, if op2 was inserted while operating op6, stall is raised and all the pipeline operation stops processing the new input instructions until the previous op6 is done.

ii. Hardware Block Diagram



iii. FSM

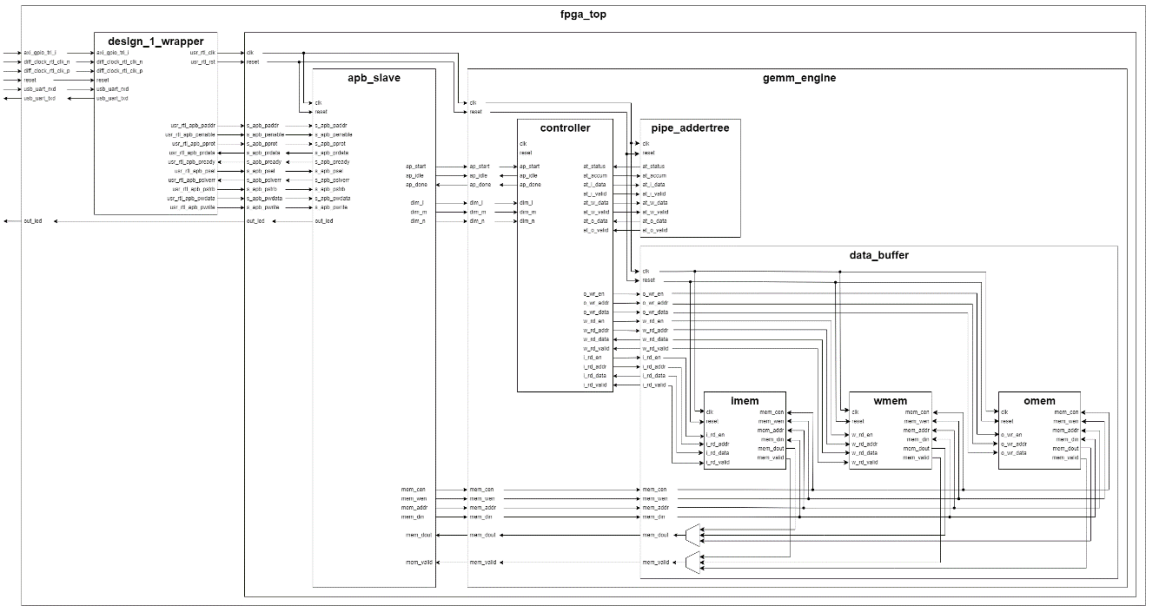


B. Lab 5

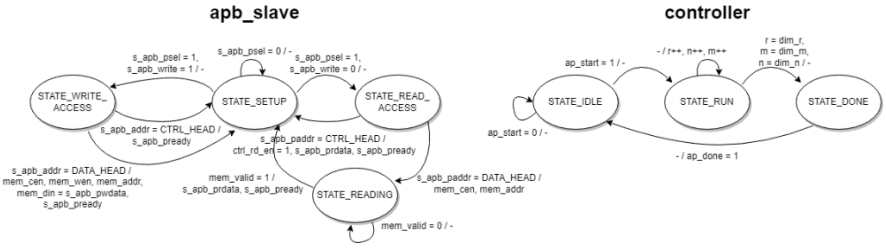
i. Description

The host generates orders using APB_master and sends the orders to RTL kernel via APB_slave. APB_slave decodes the data received, and setup control registers or access memories according to the order. If memory is accessed, corresponding LED will be turned on. Input and weight memory have 16 banks, enabling independent writing to each address as well as reading 16 values simultaneously. If APB_master initializes input and weight memory and sets dim_l, dim_m, and dim_n, it raises start signal to operate GEMM engine. If controller receives start signal, it reads 16 length vector from input and weight memory and executes dot product using pipe_addertree. 6 cycles later, pipe_addertree returns calculated value. Then controller saves the value to output memory. After all the calculation is finished, done signal is raised to the host. Here, pipe_addertree is operated as pipeline processor, allows high throughput of the engine.

ii. Hardware Block Diagram



iii. FSM

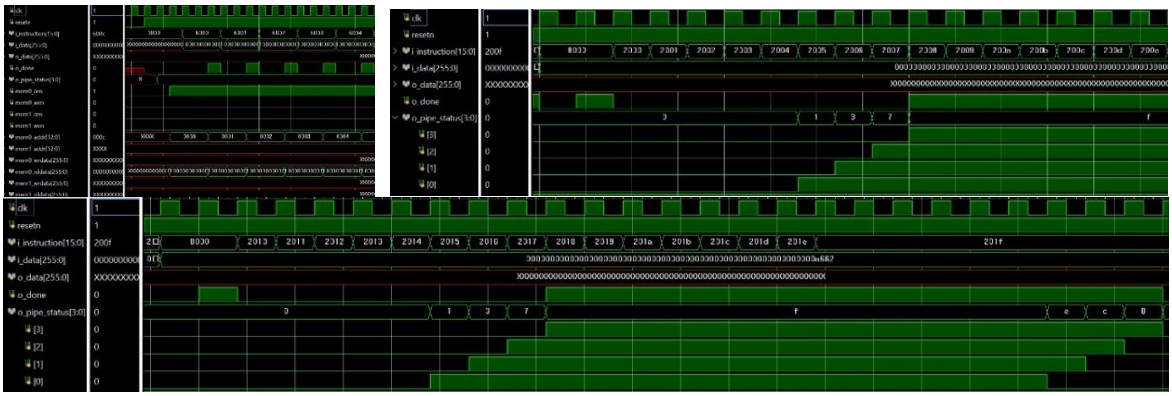


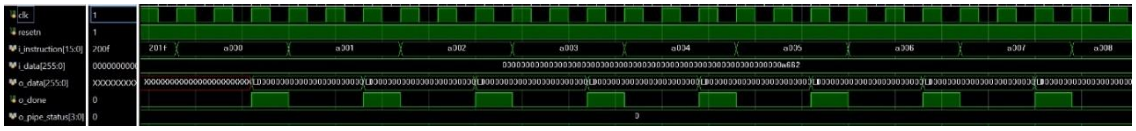
3. Experiment Results

A. Lab 4

i. Problem A - Wave form

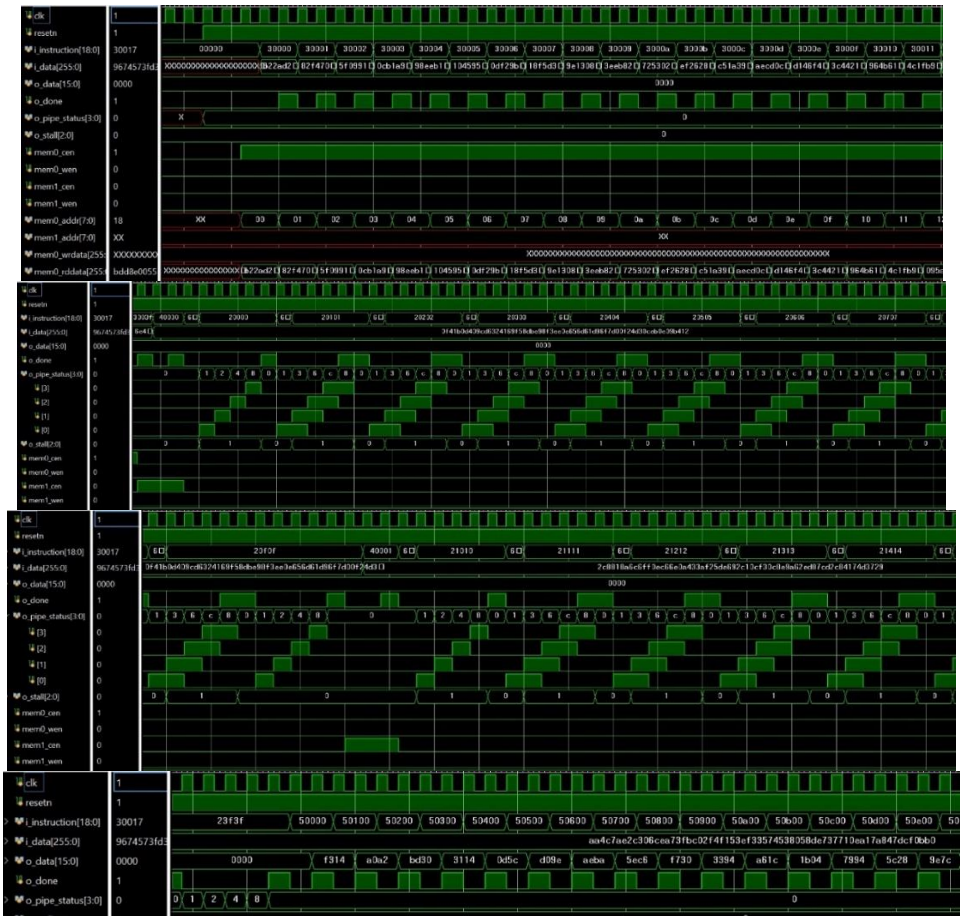
First, input memory was filled with initial values. Second, weight vector was written and pipelined SIMD was executed. Third, after 16 operations, weight vector was changed and another 16 operations followed. Lastly, it read data from output memory which is the result of dot product.





ii. Problem B - Wave form

First, input memory was filled with initial values. Second, weight vector was written and pipelined SIMD was executed. Here, since op2 is following op6, stall was raised for every op2 operation. Third, after 16 operations, weight vector was changed and another 16 operations followed. Lastly, after 64 operations, it read data from output memory which is the result of dot product.



iii. FPGA reports

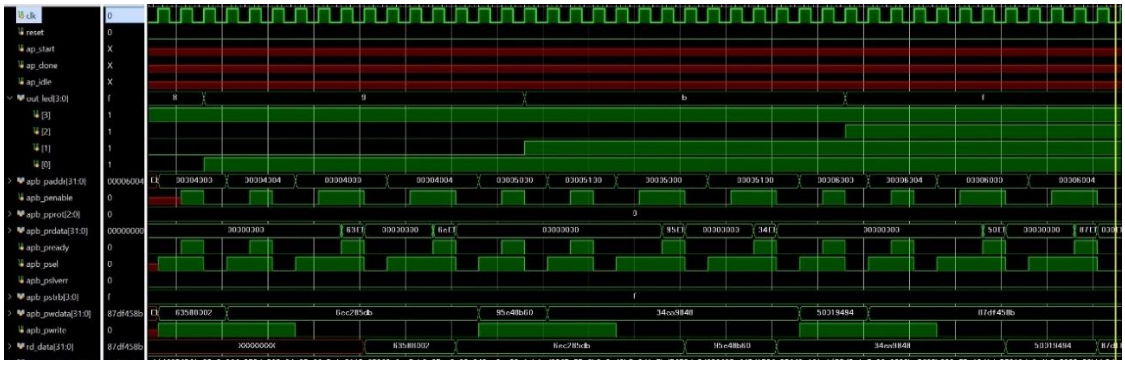
Utilization and power reports are showed like below. The utilization report analyzed the number of components used in the synthesis model, and the power report analyzed that of power consumption.

Resource	Utilization	Available	Utilization %
LUT	124	20800	0.60
FF	1368	41600	3.29
BRAM	68	50	136.00
DSP	23	90	25.56
IO	535	210	254.76

B. Lab 5

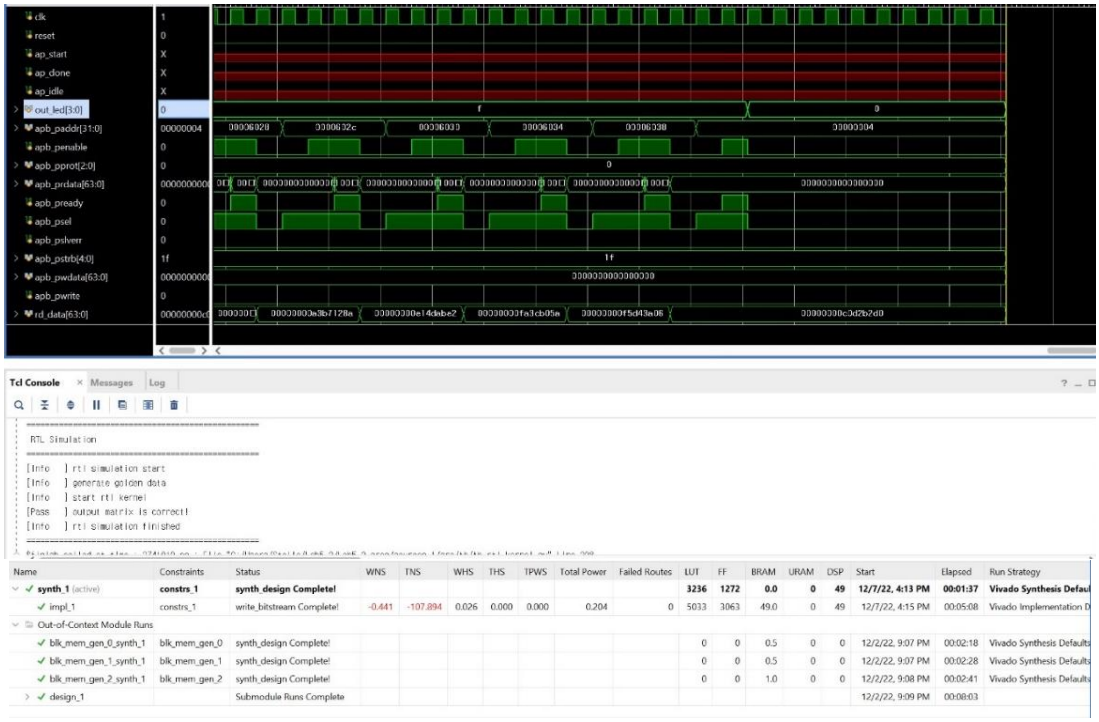
i. Problem A – Wave form & Demo picture

I wrote and read data to and from input memory, weight memory, and output memory, sequentially. Below waveform shows successful result and demo picture shows that each memory is accessed successfully.



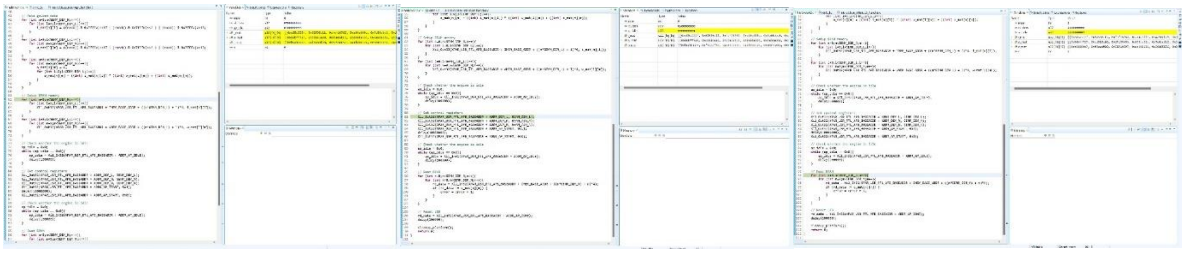
ii. Problem B - Wave form & Synthesis and Implementation result

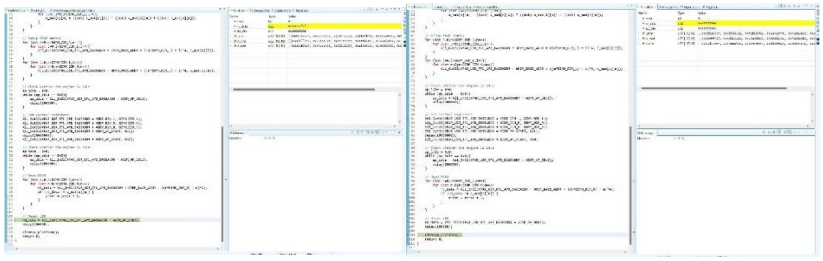
Below wave form shows successful run of the test case. However, in synthesis and implementation result, timing error occurred. Since the RTL code worked well in FPGA board, it seems it is just because of highly strict standard of simulation system. The error occurred in the first stage of adder tree, which reads data from memory and calculates multiplication of 16 length vector in a single cycle. If you want to fix this problem, you can divide this stage into two stages.



iii. Problem C – GDB and host code & Demo picture

Problem C examines the entire operation of GEMM by FPGA board. Overall testing code is the same with the problem B where dimensions were all 16 in this case. Demo picture shows sequential LED change. We can observe that the whole system works well.

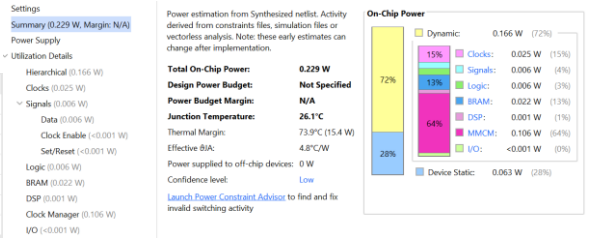




iv. FPGA reports

Utilization and power reports of problem A, B, and C are showed like below, sequentially. The utilization report analyzed the number of components used in the synthesis model, and the power report analyzed that of power consumption.

Resource	Utilization	Available	Utilization %
LUT	5335	20800	25.65
LUTRAM	192	9600	2.00
FF	3770	41600	9.06
BRAM	49	50	98.00
DSP	49	90	54.44
IO	13	210	6.19
MMCM	1	5	20.00



4. Discussion

A. [Lab 4] Is the more stages in the pipeline processor configuration, the better? If not, how can we find the optimal structure?

More stages allow processor to run more stages simultaneously. Therefore, it will increase throughput and clock speed. However, since the number of pipeline registers increase, area and power cost of the processor will increase. And deeply pipelined processors are easily exposed to data hazard issue. For these reasons, first we should choose the number of stages within the area and budget constraint. If the number of stages are fixed, we should choose dividing points where the least data dependencies occur.

B. [Lab 4] How hardware automatically sense data dependency and provide optimized bubble cycles?

Since data dependency is determined only with instruction orders, hardware can easily detect data dependency at each stage by accessing other stage's instruction. Optimized bubble cycles can be determined by the distance between the stages that makes data dependency.

C. [Lab 5] Describe your memory-mapped system with address field.

Among 14bit of mem_addr, since 2 bits are used to determine memory type (imem, wmem, omem) and another 2 bits are for word, 10 bits are used for real addressing. At imem and wmem, 4 bits among 10 bits are used to select bank and remaining 6 bits are used as address inside the bank. Since we can identify individual memory, bank, and address inside the bank, writing to individual entity is possible, while it is also possible to read 16 entities at the same time since there are 16 banks, 16 BRAMs. In case of omem, there is only one bank so 10 bits are all used for addressing inside the bank.

D. [Lab 5] Describe each module of your own RTL code.

rtl_kernel connects apb_slave and gemm_engine. apb_slave receives apb signals from apb_master, and write or read to or from control registers (dim_l, dim_m, dim_n, ap_start, ap_done, ap_idle) or memories (imem, wmem, omem). gemm_engine connects controller, data_buffer, and pipe_addertree. controller receives dim_l, dim_m, dim_n, ap_start from apb_slave, and generates i_rd_~, w_rd_~, o_wr_~ signals to control memories and pipe_addertree. As data_buffer receives signals from controller or directly from apb_slave, it transfers them to imem, wmem, and omem. Each memory writes given value if the command is write or read data from designated address if the command is read. pipe_addertree receives data all the way down from memory, data_buffer, and controller. It executes dot product of 16 length vector with 6 cycle latency and returns the result in pipelined way.

E. [Lab 5] Our GEMM accelerator proceeds the operation with a single adder-tree. Describe how order of the matrix multiplication changes when adding an extra adder-tree to increase parallelism of the accelerator.

If we add an extra adder-tree, we can do two of 16 length vector dot product simultaneously. In other words, we can perform 32 length vector dot product without accumulating at the end of the pipe_addertree. Therefore, throughput of dot product of longer vector will be increased.

F. [Lab 5] Due to the limit on the number of BRAMs in Arty A7, the size of the matrix that can be calculated at once is limited. Explain how the host code should be change when a larger matrix operation is performed using the current accelerator architecture. Also, explain the order in which input, weight, and output should be loaded to the GEMM accelerator.

It is same with the bonus point task. First, set the largest matrix the accelerator can support (let this size is (l,m,n) and this is a single tile) and calculate the output result. After one matrix multiplication is finished, move the tile to l side (l+1~2l,m,n) and reset the accelerator and calculate the result again. After moving to the end of the l side, increase m side, i.e. (l,m+1~2m,n) and continue calculation. Therefore, input, weight, and output memory will be accessed repeatedly to calculate the next tile.

5. Appendix: Include your code here

A. Lab 4

i. inst_dec

```
// Filename     : inst_dec.sv
// Author       :
//                                Seokchan Song  < ssong0410@kaist.ac.kr >
// ----------------------------------------------------------------
// Description:
// Instruction decoder
// (recommended) 3-bit opcode + 16-bit operand
// {18-16: opcode, 15-8: read mem addr, 7-0: write mem addr}

// operation 0 : reset.
// operation 1 : read data from IMEM and run inner product with weight vector.
//             Store result in OMEM. Operand must contain {IMEM ADDR} and {OMEM ADDR}.
// operation 2 : read data from {OMEM_VECTOR} and run inner product with weight vector.
//             Store result in OMEM. Operand must contain {OMEM_VECTOR ADDR} and {OMEM ADDR}
// operation 3 : Fetch input from tb_imem to IMEM write i_data into IMEM. operand will be {IMEM
ADDR}.
// operation 4 : Fetch weight from tb_wmem to weight register.
// operation 5 : read data from output OMEM and assign to o_data.  operand will be {OMEM ADDR}
// operation 6 : 16 input vectors are sequentially read from a given {IMEM ADDR} to perform an
inner product with a weight vector.
//                   Store result in {OMEM_VECTOR ADDR}. Operand must contain {IMEM ADDR} and
{OMEM_VECTOR ADDR}


// cen : cell enable siignal for each sram. LOW ACTIVATE SIGNAL
// wen : write enable siignal for each sram. LOW ACTIVATE SIGNAL
// o_run : activate signal for pipelined SIMD
// o_stall : stall signal for pipe_SIMD. only for week 2

// -FHDR----------------------------------------------------------

`timescale 1ns / 1ps

import PS_pkg::*;
module INST_DEC
(
   input  logic                                                                clk,
   input  logic                                                                reset,


   input  logic [INST_WIDTH -1:0]                i_instruction,

   //OUT_SIGNALS : Add signals for your design if you need
        output logic                                                                  o_imem_cen,
        output logic                                                                  o_imem_wen,

   output logic                                                                o_omem_cen,
        output logic                                                                  o_omem_wen,

   output logic                                                                o_ovmem_cen,
        output logic
        o_ovmem_wen,

   output logic                                                                o_run,
        output logic                                                                   o_stall
);

/* TO DO: Design your INST DECODER here.
       Your implementation should follow provided method. */
enum logic [2:0] {
   STATE_ID,
   STATE_RESET,
   STATE_OP126,
   STATE_DONE,
   STATE_OP5_DONE
} state_ff, state_nxt;
```

```
logic state_ff, o_imem_cen_ff, o_imem_wen_ff, o_omem_cen_ff, o_ovmem_cen_ff, o_ovmem_wen_ff;
logic state_nxt;
logic [INST_WIDTH -1:0]            i_instruction_ff, i_instruction_nxt;

// Pipeline registers for operation 1,2,6
logic    o_run_op126_ff,    o_omem_cen_op126_ff,    o_omem_wen_op126_ff,    o_ovmem_cen_op126_ff,
o_ovmem_wen_op126_ff;
logic [3:0] timer_ff, timer_nxt;
logic [INST_WIDTH -1:0]                 DOT_PRODUCT_1_run_ff,              DOT_PRODUCT_2_run_ff,
DOT_PRODUCT_3_run_ff, DOT_PRODUCT_4_run_ff, DONE_run_ff;
logic [INST_WIDTH -1:0]                 DOT_PRODUCT_1_run_nxt,            DOT_PRODUCT_2_run_nxt,
DOT_PRODUCT_3_run_nxt, DOT_PRODUCT_4_run_nxt, DONE_run_nxt;
logic [INST_WIDTH -1:0]                 DOT_PRODUCT_1_inst_ff,            DOT_PRODUCT_2_inst_ff,
DOT_PRODUCT_3_inst_ff, DOT_PRODUCT_4_inst_ff, DONE_inst_ff;
logic [INST_WIDTH -1:0]                 DOT_PRODUCT_1_inst_nxt,          DOT_PRODUCT_2_inst_nxt,
DOT_PRODUCT_3_inst_nxt, DOT_PRODUCT_4_inst_nxt, DONE_inst_nxt;
logic o_stall_ff;
logic [2:0] o_stall_count_ff,o_stall_count_nxt;

assign o_imem_cen = o_imem_cen_ff;
assign o_imem_wen = o_imem_wen_ff;
assign o_ovmem_cen = o_ovmem_cen_ff| o_ovmem_cen_op126_ff;
assign o_ovmem_wen = o_ovmem_wen_ff| o_ovmem_wen_op126_ff;
assign o_omem_cen = o_omem_cen_ff | o_omem_cen_op126_ff;
assign o_omem_wen = o_omem_wen_op126_ff;
assign o_run = o_run_op126_ff;
assign o_stall = o_stall_ff;

/* TO DO: Write sequential code for your INST DECODER here. */
always_ff @(posedge clk) begin
    if (~o_stall_ff) begin
        state_ff <= state_nxt;
        i_instruction_ff <= i_instruction_nxt;

        // Op1,2,6 pipeline register
        timer_ff <= timer_nxt;
        DOT_PRODUCT_1_run_ff <= DOT_PRODUCT_1_run_nxt;
        DOT_PRODUCT_1_inst_ff <= DOT_PRODUCT_1_inst_nxt;
    end else begin
        o_stall_count_ff <= o_stall_count_nxt;
        DOT_PRODUCT_1_run_ff <= DOT_PRODUCT_1_run_nxt;
        DOT_PRODUCT_1_inst_ff <= DOT_PRODUCT_1_inst_nxt;
    end
end

/* TO DO: Write combinational code for your INST DECODER here. */
always_comb begin
    if (reset) begin
        i_instruction_nxt = 0;
        state_nxt = STATE_ID;
        timer_nxt = 0;
        DOT_PRODUCT_1_run_nxt = 0;
        DOT_PRODUCT_1_inst_nxt = 0;
        o_imem_cen_ff = 0;
        o_imem_wen_ff = 0;
        o_omem_cen_ff = 0;
        o_stall_ff = 0;
    end else if (o_stall_ff) begin
        o_stall_count_nxt = o_stall_count_ff - 1;
        DOT_PRODUCT_1_run_nxt = 0;
        DOT_PRODUCT_1_inst_nxt = 0;
        if (o_stall_count_ff == 0) begin
            o_stall_ff = 0;
            timer_nxt = 5;
            DOT_PRODUCT_1_run_nxt = 1;
            DOT_PRODUCT_1_inst_nxt = i_instruction;
            o_ovmem_cen_ff = 1;
        end
    end else begin
        i_instruction_nxt = i_instruction;
        state_nxt = state_ff;
        timer_nxt = 0;
```

```
DOT_PRODUCT_1_run_nxt = 0;
DOT_PRODUCT_1_inst_nxt = 0;
o_imem_cen_ff = 0;
o_imem_wen_ff = 0;
o_omem_cen_ff = 0;

case(state_ff)
    STATE_ID: begin
        if (i_instruction_ff != i_instruction) begin
            case(i_instruction[18:16])
                // operation 0
                3'b000: begin
                    state_nxt = STATE_ID;
                end
                // operation 1
                3'b001: begin
                    state_nxt = STATE_OP126;
                    timer_nxt = 5;
                    DOT_PRODUCT_1_run_nxt = 1;
                    DOT_PRODUCT_1_inst_nxt = i_instruction;
                    o_imem_cen_ff = 1;
                end
                // operation 2
                3'b010: begin
                    state_nxt = STATE_OP126;
                    timer_nxt = 5;
                    DOT_PRODUCT_1_run_nxt = 1;
                    DOT_PRODUCT_1_inst_nxt = i_instruction;
                    o_ovmem_cen_ff = 1;
                end
                // operation 3
                3'b011: begin
                    state_nxt = STATE_DONE;
                    o_imem_wen_ff = 1;
                    o_imem_cen_ff = 1;
                end
                // operation 4
                3'b100: begin
                    state_nxt = STATE_DONE;
                end
                // operation 5
                3'b101: begin
                    state_nxt = STATE_OP5_DONE;
                    o_omem_cen_ff = 1;
                end
                // operation 6
                3'b110: begin
                    state_nxt = STATE_OP126;
                    timer_nxt = 5;
                    DOT_PRODUCT_1_run_nxt = 1;
                    DOT_PRODUCT_1_inst_nxt = i_instruction;
                    o_imem_cen_ff = 1;
                end
            endcase
        end
    end

    STATE_DONE: begin
        state_nxt = STATE_ID;
    end

    // Pipelined Op1,2,6
    STATE_OP126: begin
        if (i_instruction_ff != i_instruction) begin
            case(i_instruction[18:16])
                // operation 1
                3'b001: begin
                    timer_nxt = 5;
                    DOT_PRODUCT_1_run_nxt = 1;
                    DOT_PRODUCT_1_inst_nxt = i_instruction;
                    o_imem_cen_ff = 1;
                end
                // operation 2
```

```
                    3'b010: begin
                        if (DOT_PRODUCT_1_inst_ff[18:16] == 3'b110) begin
                            o_stall_ff = 1;
                            o_stall_count_nxt = 3;
                        end else if (DOT_PRODUCT_2_inst_ff[18:16] == 3'b110) begin
                            o_stall_ff = 1;
                            o_stall_count_nxt = 2;
                        end else if (DOT_PRODUCT_3_inst_ff[18:16] == 3'b110) begin
                            o_stall_ff = 1;
                            o_stall_count_nxt = 1;
                        end else if (DOT_PRODUCT_4_inst_ff[18:16] == 3'b110) begin
                            o_stall_ff = 1;
                            o_stall_count_nxt = 0;
                        end else begin
                            timer_nxt = 5;
                            DOT_PRODUCT_1_run_nxt = 1;
                            DOT_PRODUCT_1_inst_nxt = i_instruction;
                            o_ovmem_cen_ff = 1;
                        end
                    end
                    // operation 6
                    3'b110: begin
                        timer_nxt = 5;
                        DOT_PRODUCT_1_run_nxt = 1;
                        DOT_PRODUCT_1_inst_nxt = i_instruction;
                        o_imem_cen_ff = 1;
                    end
                endcase
            end else begin
                timer_nxt = timer_ff - 1;
                DOT_PRODUCT_1_run_nxt = 0;
                DOT_PRODUCT_1_inst_nxt = i_instruction;
            end
            if (timer_ff == 1) state_nxt = STATE_ID;
        end

        STATE_OP5_DONE: begin
            state_nxt = STATE_ID;
        end
    endcase
  end
end

// Pipelined Operation 1,2,6
always_ff @(posedge clk) begin
  if (reset) begin
      DOT_PRODUCT_2_run_ff <= DOT_PRODUCT_2_run_nxt;
      DOT_PRODUCT_2_inst_ff <= DOT_PRODUCT_2_inst_nxt;
      DOT_PRODUCT_3_run_ff <= DOT_PRODUCT_3_run_nxt;
      DOT_PRODUCT_3_inst_ff <= DOT_PRODUCT_3_inst_nxt;
      DOT_PRODUCT_4_run_ff <= DOT_PRODUCT_4_run_nxt;
      DOT_PRODUCT_4_inst_ff <= DOT_PRODUCT_4_inst_nxt;
      DONE_run_ff <= DONE_run_nxt;
      DONE_inst_ff <= DONE_inst_nxt;
  end
  if (state_ff == STATE_OP126) begin
      // cycle 2
      DOT_PRODUCT_2_run_ff <= DOT_PRODUCT_2_run_nxt;
      DOT_PRODUCT_2_inst_ff <= DOT_PRODUCT_2_inst_nxt;
      // cycle 3
      DOT_PRODUCT_3_run_ff <= DOT_PRODUCT_3_run_nxt;
      DOT_PRODUCT_3_inst_ff <= DOT_PRODUCT_3_inst_nxt;
      // cycle 4
      DOT_PRODUCT_4_run_ff <= DOT_PRODUCT_4_run_nxt;
      DOT_PRODUCT_4_inst_ff <= DOT_PRODUCT_4_inst_nxt;
      // cycle 5
      DONE_run_ff <= DONE_run_nxt;
      DONE_inst_ff <= DONE_inst_nxt;
      // cycle 6
  end
end

always_comb begin
```

```
    if (reset) begin
        DOT_PRODUCT_2_run_nxt = 0;
        DOT_PRODUCT_3_run_nxt = 0;
        DOT_PRODUCT_4_run_nxt = 0;
        DOT_PRODUCT_2_inst_nxt = 0;
        DOT_PRODUCT_3_inst_nxt = 0;
        DOT_PRODUCT_4_inst_nxt = 0;
        DONE_run_nxt = 0;
        DONE_inst_nxt = 0;
        o_run_op126_ff = 0;
        o_ovmem_cen_op126_ff = 0;
        o_ovmem_wen_op126_ff = 0;
        o_omem_cen_op126_ff = 0;
        o_omem_wen_op126_ff = 0;
    end else begin
        // cycle 2
        DOT_PRODUCT_2_run_nxt = DOT_PRODUCT_1_run_ff;
        DOT_PRODUCT_2_inst_nxt = DOT_PRODUCT_1_inst_ff;
        o_run_op126_ff = DOT_PRODUCT_1_run_ff;
        // cycle 3
        DOT_PRODUCT_3_run_nxt = DOT_PRODUCT_2_run_ff;
        DOT_PRODUCT_3_inst_nxt = DOT_PRODUCT_2_inst_ff;
        // cycle 4
        DOT_PRODUCT_4_run_nxt = DOT_PRODUCT_3_run_ff;
        DOT_PRODUCT_4_inst_nxt = DOT_PRODUCT_3_inst_ff;
        // cycle 5
        DONE_run_nxt = DOT_PRODUCT_4_run_ff;
        DONE_inst_nxt = DOT_PRODUCT_4_inst_ff;
        if (DOT_PRODUCT_4_inst_ff[18:16] == 3'b001 || DOT_PRODUCT_4_inst_ff[18:16] == 3'b010) begin
            // Operation 1,2
            o_omem_cen_op126_ff = DOT_PRODUCT_4_run_ff;
            o_omem_wen_op126_ff = DOT_PRODUCT_4_run_ff;
        end else if (DOT_PRODUCT_4_inst_ff[18:16] == 3'b110) begin
            // Opertion 6
            o_ovmem_cen_op126_ff = DOT_PRODUCT_4_run_ff;
            o_ovmem_wen_op126_ff = DOT_PRODUCT_4_run_ff;
        end else begin
            o_omem_wen_op126_ff = 0;
            o_ovmem_wen_op126_ff = 0;
        end
        // cycle 6

    end
end


endmodule
```

ii. pipe_SIMD

```
// Filename     : pipe_SIMD.sv
// Author       :
//                                  Seokchan Song  < ssong0410@kaist.ac.kr >
// -----------------------------------------------------------------
// Description:
//   Pipelined SIMD. It must consist 4 stage pipelined register. Refer to figure 2 of given
documentaiton.
//  o_pipe_status : Each bit indicates activation of the pipeline register. {ADD2, ADD1, ADD0,
MUL}

// -FHDR----------------------------------------------------------------

`timescale 1ns / 1ps

import PS_pkg::*;
module PIPE_SIMD
#(
    parameter DWIDTH             =  16  ,
    parameter VECTOR_LENGTH       =  16
)
(
    input  logic                                                                clk,
    input  logic                                                                reset,
```

```
    input  logic                            i_run      ,

    input  logic [DWIDTH-1:0]    i_input  [0:VECTOR_LENGTH-1],
    input  logic [DWIDTH-1:0]    i_weight [0:VECTOR_LENGTH-1],

         output logic [DWIDTH-1:0]               o_result    ,
    output logic [3:0]                     o_pipe_status
);

/* TO DO: Design your pipeliend SIMD here.
       Your implementation should follow provided method. */
// Declare Pipeline Registers
logic               MUL_run_ff, ADD0_run_ff, ADD1_run_ff, ADD2_run_ff;
logic               MUL_run_nxt, ADD0_run_nxt, ADD1_run_nxt, ADD2_run_nxt;

logic [DWIDTH*2-1:0]    MUL_data_ff      [0:VECTOR_LENGTH-1];
logic [DWIDTH*2-1:0]    MUL_data_nxt     [0:VECTOR_LENGTH-1];
logic [DWIDTH*2-1:0]  ADD0_data_ff     [0:VECTOR_LENGTH/2-1];
logic [DWIDTH*2-1:0]  ADD0_data_nxt    [0:VECTOR_LENGTH/2-1];
logic [DWIDTH*2-1:0]  ADD1_data_ff     [0:VECTOR_LENGTH/4-1];
logic [DWIDTH*2-1:0]  ADD1_data_nxt    [0:VECTOR_LENGTH/4-1];
logic [DWIDTH*2-1:0]  ADD2_data_ff     [0:VECTOR_LENGTH/8-1];
logic [DWIDTH*2-1:0]  ADD2_data_nxt    [0:VECTOR_LENGTH/8-1];

logic [DWIDTH*2-1:0]    o_result_ff;
logic [3:0]          o_pipe_status_ff;

assign o_result = o_result_ff[DWIDTH-1:0];
assign o_pipe_status = o_pipe_status_ff;

/* TO DO: Write sequential code for your SIMD here. */
always_ff @(posedge clk) begin
    if (reset) begin
        ADD0_run_ff <= ADD0_run_nxt;
        ADD1_run_ff <= ADD1_run_nxt;
        ADD2_run_ff <= ADD2_run_nxt;
    end else begin
        // Stage 1
        ADD0_run_ff <= ADD0_run_nxt;
        ADD0_data_ff <= ADD0_data_nxt;
        // Stage 2
        ADD1_run_ff <= ADD1_run_nxt;
        ADD1_data_ff <= ADD1_data_nxt;
        // Stage 3
        ADD2_run_ff <= ADD2_run_nxt;
        ADD2_data_ff <= ADD2_data_nxt;
        // Stage 4
    end
end

/* TO DO: Write combinational code for your SIMD here. */
always_comb begin
    if (reset) begin
        ADD0_run_nxt = 0;
        ADD1_run_nxt = 0;
        ADD2_run_nxt = 0;
        for (int i=0;i<VECTOR_LENGTH;i++) begin
            if (i<VECTOR_LENGTH/2) ADD0_data_nxt[i] = 0;
            if (i<VECTOR_LENGTH/4) ADD1_data_nxt[i] = 0;
            if (i<VECTOR_LENGTH/8) ADD2_data_nxt[i] = 0;
        end
    end else begin
        // Stage 1 - (1)
        MUL_run_ff = i_run;
        for (int i = 0; i < VECTOR_LENGTH; i++) begin
            MUL_data_ff[i] = i_input[i] * i_weight[i];
        end
        // Stage 1 - (2)
        ADD0_run_nxt = MUL_run_ff;
        for (int i = 0; i < VECTOR_LENGTH/2; i++) begin
            ADD0_data_nxt[i] = MUL_data_ff[i] + MUL_data_ff[VECTOR_LENGTH - i - 1];
        end
        // Stage 2
```

```
      ADD1_run_nxt = ADD0_run_ff;
      for (int i = 0; i < VECTOR_LENGTH/4; i++) begin
         ADD1_data_nxt[i] = ADD0_data_ff[i] + ADD0_data_ff[VECTOR_LENGTH/2 - i - 1];
      end
      // Stage 3
      ADD2_run_nxt = ADD1_run_ff;
      for (int i = 0; i < VECTOR_LENGTH/8; i++) begin
         ADD2_data_nxt[i] = ADD1_data_ff[i] + ADD1_data_ff[VECTOR_LENGTH/4 - i - 1];
      end
      // Stage 4
      o_result_ff = (ADD2_run_ff) ? ADD2_data_ff[0] + ADD2_data_ff[1] : 0;
      o_pipe_status_ff = {ADD2_run_ff,ADD1_run_ff,ADD0_run_ff,MUL_run_ff};
   end
end


endmodule
```

iii. pipe_simd_top

```
// Filename     : pipe_simd_top.sv
// Author       :
//                                 Seokchan Song  < ssong0410@kaist.ac.kr >
// -----------------------------------------------------------------
// Description:
// Top unit for pipelined SIMD processor

// o_done : 1 when a value is given through o_data
// o_pipe_status : Each bit indicates activation of the pipeline register. {ADD2, ADD1, ADD0, MUL}

// -FHDR----------------------------------------------------------

`timescale 1ns / 1ps

import PS_pkg::*;
module DUT
(
   input  logic                                                                clk,
   input  logic                                                                reset,

   input  logic [INST_WIDTH -1:0]                    i_instruction   ,
   input  logic [VECTOR_LENGTH*DATA_WIDTH -1:0]   i_data         ,
   output logic [DATA_WIDTH -1:0]    o_data         ,

         output logic
o_done         ,
         output logic [3:0]                                      o_pipe_status   ,
         output logic [2:0]                           o_stall
);

/* TO DO: Declare your logic here. */

logic [INST_WIDTH -1:0]                  i_instruction_ff, i_instruction_nxt;
logic   [DATA_WIDTH-1:0]       weight_vector_ff       [0:VECTOR_LENGTH-1];
logic   [DATA_WIDTH-1:0]       weight_vector_nxt      [0:VECTOR_LENGTH-1];
logic   [DATA_WIDTH-1:0]       dot_input             [0:VECTOR_LENGTH-1];
logic
o_imem_wen,o_imem_cen,o_omem_wen,o_omem_cen,o_ovmem_wen,o_ovmem_cen,o_run;
logic
i_imem_wen,i_imem_cen,i_omem_wen,i_omem_cen,i_ovmem_wen,i_ovmem_cen,i_run;
logic                        reset_ff,manual_reset_ff,manual_reset_nxt,o_done_ff,o_done_op1_ff;
logic                                                          [DATA_WIDTH*VECTOR_LENGTH-1:0]
imem_rddata,imem_wrdata,omem_rddata,omem_wrdata,ovmem_rddata,ovmem_wrdata,o_data_ff;
logic   [DATA_WIDTH-1:0]                 result;
logic [ADDR_WIDTH-1:0]         imem_addr,omem_addr,ovmem_addr;
logic o_stall_ff;
logic [2:0] o_stall_count_ff,o_stall_count_nxt;

// Pipeline registers for operation 1
logic [3:0] timer_ff, timer_nxt;
logic [INST_WIDTH -1:0]                  DOT_PRODUCT_1_run_ff,              DOT_PRODUCT_2_run_ff,
DOT_PRODUCT_3_run_ff, DOT_PRODUCT_4_run_ff, DONE_run_ff;
```

```
logic [INST_WIDTH -1:0]                 DOT_PRODUCT_1_run_nxt,           DOT_PRODUCT_2_run_nxt,
DOT_PRODUCT_3_run_nxt, DOT_PRODUCT_4_run_nxt, DONE_run_nxt;
logic [INST_WIDTH -1:0]                 DOT_PRODUCT_1_inst_ff,           DOT_PRODUCT_2_inst_ff,
DOT_PRODUCT_3_inst_ff, DOT_PRODUCT_4_inst_ff, DONE_inst_ff;
logic [INST_WIDTH -1:0]                 DOT_PRODUCT_1_inst_nxt,         DOT_PRODUCT_2_inst_nxt,
DOT_PRODUCT_3_inst_nxt, DOT_PRODUCT_4_inst_nxt, DONE_inst_nxt;

// Status
enum logic [2:0] {
    STATE_ID,
    STATE_OP126,
    STATE_DONE,
    STATE_OP5_DONE
} state_ff, state_nxt;

assign reset_ff = reset | manual_reset_ff;
assign o_done = o_done_ff | DONE_run_ff;
assign o_data = o_data_ff;
assign o_stall = o_stall_ff;

assign i_imem_wen = o_imem_wen;
assign i_imem_cen = o_imem_cen;
assign i_omem_wen = o_omem_wen;
assign i_omem_cen = o_omem_cen;
assign i_ovmem_wen = o_ovmem_wen;
assign i_ovmem_cen = o_ovmem_cen;
assign i_run = o_run;

/* TO DO: Write sequential code for your design here. */
always_ff @(posedge clk) begin
    if (~o_stall_ff) begin
        i_instruction_ff <= i_instruction_nxt;
        state_ff <= state_nxt;
        weight_vector_ff <= weight_vector_nxt;
        manual_reset_ff <= manual_reset_nxt;

        // Op1 pipeline register
        timer_ff <= timer_nxt;
        DOT_PRODUCT_1_run_ff <= DOT_PRODUCT_1_run_nxt;
        DOT_PRODUCT_1_inst_ff <= DOT_PRODUCT_1_inst_nxt;
    end else begin
        o_stall_count_ff <= o_stall_count_nxt;
        DOT_PRODUCT_1_run_ff <= DOT_PRODUCT_1_run_nxt;
        DOT_PRODUCT_1_inst_ff <= DOT_PRODUCT_1_inst_nxt;
    end
end

/* TO DO: Write combinational code for your design here. */
always_comb begin
    if (reset_ff) begin
        manual_reset_nxt = 0;
        i_instruction_nxt = 0;
        o_done_ff = 0;
        o_data_ff = 0;
        state_nxt = STATE_ID;
        timer_nxt = 0;
        DOT_PRODUCT_1_run_nxt = 0;
        DOT_PRODUCT_1_inst_nxt = 0;
        for (int i=0;i<VECTOR_LENGTH;i++) begin
            weight_vector_nxt[i] = 0;
        end
        o_stall_ff = 0;
    end else if (o_stall_ff) begin
        o_stall_count_nxt = o_stall_count_ff - 1;
        DOT_PRODUCT_1_run_nxt = 0;
        DOT_PRODUCT_1_inst_nxt = 0;
        ovmem_addr = i_instruction[15:8];
        if (o_stall_count_ff == 0) begin
            o_stall_ff = 0;
            timer_nxt = 5;
            DOT_PRODUCT_1_run_nxt = 1;
            DOT_PRODUCT_1_inst_nxt = i_instruction;
            ovmem_addr = i_instruction[15:8];
```

```
    end
end else begin
    i_instruction_nxt = i_instruction;
    manual_reset_nxt = 0;
    o_done_ff = 0;
    state_nxt = state_ff;
    weight_vector_nxt = weight_vector_ff;
    timer_nxt = 0;
    DOT_PRODUCT_1_run_nxt = 0;
    DOT_PRODUCT_1_inst_nxt = 0;

    case(state_ff)
        STATE_ID: begin
            if (i_instruction_ff != i_instruction) begin
                case(i_instruction[18:16])
                    // operation 0
                    3'b000: begin
                        state_nxt = STATE_DONE; // INST_DEC: state_nxt = STATE_ID
                        manual_reset_nxt = 1;
                    end
                    // operation 1
                    3'b001: begin
                        state_nxt = STATE_OP126;
                        timer_nxt = 5;
                        DOT_PRODUCT_1_run_nxt = 1;
                        DOT_PRODUCT_1_inst_nxt = i_instruction;
                        imem_addr = i_instruction[15:8];
                    end
                    // operation 2
                    3'b010: begin
                        state_nxt = STATE_OP126;
                        timer_nxt = 5;
                        DOT_PRODUCT_1_run_nxt = 1;
                        DOT_PRODUCT_1_inst_nxt = i_instruction;
                        ovmem_addr = i_instruction[15:8];
                    end
                    // operation 3
                    3'b011: begin
                        state_nxt = STATE_DONE;
                        imem_addr = i_instruction[7:0];
                        imem_wrdata = i_data;
                    end
                    // operation 4
                    3'b100: begin
                        state_nxt = STATE_DONE;
                        weight_vector_nxt[0] = i_data[15:0];
                        weight_vector_nxt[1] = i_data[31:16];
                        weight_vector_nxt[2] = i_data[47:32];
                        weight_vector_nxt[3] = i_data[63:48];
                        weight_vector_nxt[4] = i_data[79:64];
                        weight_vector_nxt[5] = i_data[95:80];
                        weight_vector_nxt[6] = i_data[111:96];
                        weight_vector_nxt[7] = i_data[127:112];
                        weight_vector_nxt[8] = i_data[143:128];
                        weight_vector_nxt[9] = i_data[159:144];
                        weight_vector_nxt[10] = i_data[175:160];
                        weight_vector_nxt[11] = i_data[191:176];
                        weight_vector_nxt[12] = i_data[207:192];
                        weight_vector_nxt[13] = i_data[223:208];
                        weight_vector_nxt[14] = i_data[239:224];
                        weight_vector_nxt[15] = i_data[255:240];
                    end
                    // operation 5
                    3'b101: begin
                        state_nxt = STATE_OP5_DONE;
                        omem_addr = i_instruction[15:8];
                    end
                    // operation 6
                    3'b110: begin
                        state_nxt = STATE_OP126;
                        timer_nxt = 5;
                        DOT_PRODUCT_1_run_nxt = 1;
                        DOT_PRODUCT_1_inst_nxt = i_instruction;
```

```
                    imem_addr = i_instruction[15:8];
                end
            endcase
        end
    end

    STATE_DONE: begin
        state_nxt = STATE_ID;
        o_done_ff = 1;
    end

    // Pipelined Op 1,2,6
    STATE_OP126: begin
        if (i_instruction_ff != i_instruction) begin
            case(i_instruction[18:16])
                // operation 1
                3'b001: begin
                    timer_nxt = 5;
                    DOT_PRODUCT_1_run_nxt = 1;
                    DOT_PRODUCT_1_inst_nxt = i_instruction;
                    imem_addr = i_instruction[15:8];
                end
                // operation 2
                3'b010: begin
                    if (DOT_PRODUCT_1_inst_ff[18:16] == 3'b110) begin
                        o_stall_ff = 1;
                        o_stall_count_nxt = 3;
                    end else if (DOT_PRODUCT_2_inst_ff[18:16] == 3'b110) begin
                        o_stall_ff = 1;
                        o_stall_count_nxt = 2;
                    end else if (DOT_PRODUCT_3_inst_ff[18:16] == 3'b110) begin
                        o_stall_ff = 1;
                        o_stall_count_nxt = 1;
                    end else if (DOT_PRODUCT_4_inst_ff[18:16] == 3'b110) begin
                        o_stall_ff = 1;
                        o_stall_count_nxt = 0;
                    end else begin
                        timer_nxt = 5;
                        DOT_PRODUCT_1_run_nxt = 1;
                        DOT_PRODUCT_1_inst_nxt = i_instruction;
                        ovmem_addr = i_instruction[15:8];
                    end
                end
                // operation 6
                3'b110: begin
                    timer_nxt = 5;
                    DOT_PRODUCT_1_run_nxt = 1;
                    DOT_PRODUCT_1_inst_nxt = i_instruction;
                    imem_addr = i_instruction[15:8];
                end
            endcase
        end else begin
            timer_nxt = timer_ff - 1;
            DOT_PRODUCT_1_run_nxt = 0;
            DOT_PRODUCT_1_inst_nxt = i_instruction;
        end
        if (timer_ff == 1) state_nxt = STATE_ID;
    end

    STATE_OP5_DONE: begin
        state_nxt = STATE_ID;
        o_data_ff = omem_rddata;
        o_done_ff = 1;
    end
    endcase
  end
end

// Pipelined Operation 1,2,6
always_ff @(posedge clk) begin
  if (reset_ff) begin
    DOT_PRODUCT_2_run_ff <= DOT_PRODUCT_2_run_nxt;
    DOT_PRODUCT_2_inst_ff <= DOT_PRODUCT_2_inst_nxt;
```

```
        DOT_PRODUCT_3_run_ff <= DOT_PRODUCT_3_run_nxt;
        DOT_PRODUCT_3_inst_ff <= DOT_PRODUCT_3_inst_nxt;
        DOT_PRODUCT_4_run_ff <= DOT_PRODUCT_4_run_nxt;
        DOT_PRODUCT_4_inst_ff <= DOT_PRODUCT_4_inst_nxt;
        DONE_run_ff <= DONE_run_nxt;
        DONE_inst_ff <= DONE_inst_nxt;
    end else begin
        if (state_ff == STATE_OP126) begin
            // cycle 2
            DOT_PRODUCT_2_run_ff <= DOT_PRODUCT_2_run_nxt;
            DOT_PRODUCT_2_inst_ff <= DOT_PRODUCT_2_inst_nxt;
            // cycle 3
            DOT_PRODUCT_3_run_ff <= DOT_PRODUCT_3_run_nxt;
            DOT_PRODUCT_3_inst_ff <= DOT_PRODUCT_3_inst_nxt;
            // cycle 4
            DOT_PRODUCT_4_run_ff <= DOT_PRODUCT_4_run_nxt;
            DOT_PRODUCT_4_inst_ff <= DOT_PRODUCT_4_inst_nxt;
            // cycle 5
            DONE_run_ff <= DONE_run_nxt;
            DONE_inst_ff <= DONE_inst_nxt;
            // cycle 6
        end
    end
end

always_comb begin
    if (reset_ff) begin
        DOT_PRODUCT_2_run_nxt = 0;
        DOT_PRODUCT_3_run_nxt = 0;
        DOT_PRODUCT_4_run_nxt = 0;
        DOT_PRODUCT_2_inst_nxt = 0;
        DOT_PRODUCT_3_inst_nxt = 0;
        DOT_PRODUCT_4_inst_nxt = 0;
        DONE_run_nxt = 0;
        DONE_inst_nxt = 0;
    end else begin
        // cycle 2
        DOT_PRODUCT_2_run_nxt = DOT_PRODUCT_1_run_ff;
        DOT_PRODUCT_2_inst_nxt = DOT_PRODUCT_1_inst_ff;
        if (DOT_PRODUCT_1_inst_ff[18:16] == 3'b001 || DOT_PRODUCT_1_inst_ff[18:16] == 3'b110) begin
            // Operation 1, 6
            dot_input[0] = imem_rddata[15:0];
            dot_input[1] = imem_rddata[31:16];
            dot_input[2] = imem_rddata[47:32];
            dot_input[3] = imem_rddata[63:48];
            dot_input[4] = imem_rddata[79:64];
            dot_input[5] = imem_rddata[95:80];
            dot_input[6] = imem_rddata[111:96];
            dot_input[7] = imem_rddata[127:112];
            dot_input[8] = imem_rddata[143:128];
            dot_input[9] = imem_rddata[159:144];
            dot_input[10] = imem_rddata[175:160];
            dot_input[11] = imem_rddata[191:176];
            dot_input[12] = imem_rddata[207:192];
            dot_input[13] = imem_rddata[223:208];
            dot_input[14] = imem_rddata[239:224];
            dot_input[15] = imem_rddata[255:240];
        end else if (DOT_PRODUCT_1_inst_ff[18:16] == 3'b010) begin
            // Operation 2
            dot_input[0] = ovmem_rddata[15:0];
            dot_input[1] = ovmem_rddata[31:16];
            dot_input[2] = ovmem_rddata[47:32];
            dot_input[3] = ovmem_rddata[63:48];
            dot_input[4] = ovmem_rddata[79:64];
            dot_input[5] = ovmem_rddata[95:80];
            dot_input[6] = ovmem_rddata[111:96];
            dot_input[7] = ovmem_rddata[127:112];
            dot_input[8] = ovmem_rddata[143:128];
            dot_input[9] = ovmem_rddata[159:144];
            dot_input[10] = ovmem_rddata[175:160];
            dot_input[11] = ovmem_rddata[191:176];
            dot_input[12] = ovmem_rddata[207:192];
            dot_input[13] = ovmem_rddata[223:208];
```

```
            dot_input[14] = ovmem_rddata[239:224];
            dot_input[15] = ovmem_rddata[255:240];
        end
        // cycle 3
        DOT_PRODUCT_3_run_nxt = DOT_PRODUCT_2_run_ff;
        DOT_PRODUCT_3_inst_nxt = DOT_PRODUCT_2_inst_ff;
        // cycle 4
        DOT_PRODUCT_4_run_nxt = DOT_PRODUCT_3_run_ff;
        DOT_PRODUCT_4_inst_nxt = DOT_PRODUCT_3_inst_ff;
        // cycle 5
        DONE_run_nxt = DOT_PRODUCT_4_run_ff;
        DONE_inst_nxt = DOT_PRODUCT_4_inst_ff;
        if (DOT_PRODUCT_4_inst_ff[18:16] == 3'b001 || DOT_PRODUCT_4_inst_ff[18:16] == 3'b010) begin
            // Operation 1,2
            omem_wrdata = result;
            omem_addr = DOT_PRODUCT_4_inst_ff[7:0];
        end else if (DOT_PRODUCT_4_inst_ff[18:16] == 3'b110) begin
            // Operation 6
            ovmem_wrdata = result;
            ovmem_addr = DOT_PRODUCT_4_inst_ff[7:0];
        end
        // cycle 6

    end
end


/* TO DO: Instantiate your design here. */
INST_DEC
#(

)
decoder
(
    .clk                                ( clk                                       ),
    .reset                              ( reset_ff                                          ),

    .i_instruction                       ( i_instruction                                    ),

    .o_imem_wen                          (  o_imem_wen                                      ),
    .o_imem_cen                          (  o_imem_cen                               ),

    .o_omem_wen                          (  o_omem_wen                               ),
    .o_omem_cen                          (  o_omem_cen                               ),

    .o_ovmem_wen                          (  o_ovmem_wen                             ),
    .o_ovmem_cen                          (  o_ovmem_cen                             ),

    .o_run                              (  o_run                 ),
    .o_stall                            (                       )
);

// Memory
SRAM
#(
    .AWIDTH                             (ADDR_WIDTH              ),
    .DWIDTH                             (VECTOR_LENGTH * DATA_WIDTH )
)
IMEM
(
    .clk                                (  clk                                              ),

    .cen                                ( i_imem_cen                ),
    .wen                                ( i_imem_wen                ),
    .addr                               ( imem_addr                      ),
    .wrdata                             ( imem_wrdata             ),
    .rddata                             ( imem_rddata               )
);

SRAM
#(
    .AWIDTH                             (ADDR_WIDTH                                 ),
    .DWIDTH                             (VECTOR_LENGTH * DATA_WIDTH )
```

```
)
OMEM
(
    .clk                                ( clk                                           ),

    .cen                                ( i_omem_cen                ),
    .wen                                ( i_omem_wen               ),
    .addr                               ( omem_addr                     ),
    .wrdata                             (  omem_wrdata             ),
    .rddata                             (  omem_rddata            )
);

SRAM
#(
    .AWIDTH                             (ADDR_WIDTH                                 ),
    .DWIDTH                             (VECTOR_LENGTH * DATA_WIDTH )
)
OMEM_VECTOR
(
    .clk                                (  clk                                          ),

    .cen                                ( i_ovmem_cen                ),
    .wen                                ( i_ovmem_wen             ),
    .addr                               ( ovmem_addr              ),
    .wrdata                             ( ovmem_wrdata                ),
    .rddata                             ( ovmem_rddata            )
);


//SIMD UNIT
PIPE_SIMD
#(
    .DWIDTH                             (  DATA_WIDTH                    )
)
pipe_simd_unit1
(
    .clk                                ( clk                                       ),
    .reset                              ( reset_ff
         ),

    .i_run                              (  i_run                    ),

    .i_input                            ( dot_input            ),
    .i_weight                           ( weight_vector_ff            ),

    .o_result                           ( result              ),
    .o_pipe_status                       ( o_pipe_status               )

);
endmodule
```

B. Lab 5

i. vivado host code

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xil_io.h"
#include <time.h>
#include <stdlib.h>

void delay(int dl){
   for (int i=0;i<dl;){
      i = i + 1;
   }
}

#define CTRL_BASE_ADDR 0x00000000
#define IMEM_BASE_ADDR 0x00004000
#define WMEM_BASE_ADDR 0x00005000
```

```
#define OMEM_BASE_ADDR 0x00006000
#define ADDR_AP_START  0x00
#define ADDR_AP_DONE   0x04
#define ADDR_AP_IDLE   0x08
#define ADDR_DIM_L          0x10
#define ADDR_DIM_M          0x14
#define ADDR_DIM_N          0x18

#define GEMM_DIM_L    16
#define GEMM_DIM_M    16
#define GEMM_DIM_N    16

int main()
{
    init_platform();
    int error = 0;
    u32 rd_data;
    u32 ap_idle = 0x0;
    u32 i_mat[GEMM_DIM_N][GEMM_DIM_L];
    u32 w_mat[GEMM_DIM_L][GEMM_DIM_M];
    u32 o_mat[GEMM_DIM_N][GEMM_DIM_M];
    Xil_In32(XPAR_USR_RTL_APB_BASEADDR + ADDR_AP_DONE);

    // Make golden data
    for (int n=0;n<GEMM_DIM_N;n++){
         for (int l=0;l<GEMM_DIM_L;l++){
                i_mat[n][l] = ((rand() & 0x7fffu)<<17 | (rand() & 0x7fffu)<<2 ) | (rand() &
0x7fffu)>>13;
         }
    }
         for (int l=0;l<GEMM_DIM_L;l++){
                 for (int m=0;m<GEMM_DIM_M;m++){
                         w_mat[l][m] = ((rand() & 0x7fffu)<<17 | (rand() & 0x7fffu)<<2 ) | (rand()
& 0x7fffu)>>13;
                 }
         }
         for (int n=0;n<GEMM_DIM_N;n++){
                 for (int m=0;m<GEMM_DIM_M;m++){
                         o_mat[n][m] = 0;
                         for (int l=0;l<GEMM_DIM_L;l++){
                                 o_mat[n][m] = ((int) i_mat[n][l]) * ((int) w_mat[l][m]) + ((int)
o_mat[n][m]);
                         }
                 }
         }

         // Setup BRAM memory
         for (int n=0;n<GEMM_DIM_N;n++){
                 for (int l=0;l<GEMM_DIM_L;l++){
                         Xil_Out32(XPAR_USR_RTL_APB_BASEADDR + IMEM_BASE_ADDR + ((n*GEMM_DIM_L)
+ l)*4, i_mat[n][l]);
                 }
         }
         for (int l=0;l<GEMM_DIM_L;l++){
                 for (int m=0;m<GEMM_DIM_M;m++){
                         Xil_Out32(XPAR_USR_RTL_APB_BASEADDR + WMEM_BASE_ADDR + ((m*GEMM_DIM_L)
+ l)*4, w_mat[l][m]);
                 }
         }

         // Check whether the engine is idle
         ap_idle = 0x0;
         while (ap_idle == 0x0){
                 ap_idle = Xil_In32(XPAR_USR_RTL_APB_BASEADDR + ADDR_AP_IDLE);
                 delay(100000);
         }

         // Set control registers
         Xil_Out32(XPAR_USR_RTL_APB_BASEADDR + ADDR_DIM_L, GEMM_DIM_L);
         Xil_Out32(XPAR_USR_RTL_APB_BASEADDR + ADDR_DIM_M, GEMM_DIM_M);
         Xil_Out32(XPAR_USR_RTL_APB_BASEADDR + ADDR_DIM_N, GEMM_DIM_N);
         Xil_Out32(XPAR_USR_RTL_APB_BASEADDR + ADDR_AP_START, 0x1);
         delay(10000000);
```

```
        Xil_Out32(XPAR_USR_RTL_APB_BASEADDR + ADDR_AP_START, 0x0);

        // Check whether the engine is idle
        ap_idle = 0x0;
        while (ap_idle == 0x0){
                ap_idle = Xil_In32(XPAR_USR_RTL_APB_BASEADDR + ADDR_AP_IDLE);
                delay(100000);
        }

        // Read BRAM
        for (int n=0;n<GEMM_DIM_N;n++){
                for (int m=0;m<GEMM_DIM_M;m++){
                        rd_data   =   Xil_In32(XPAR_USR_RTL_APB_BASEADDR   +   OMEM_BASE_ADDR   +
((n*GEMM_DIM_M) + m)*4);
                        if (rd_data != o_mat[n][m]) {
                                error = error + 1;
                        }
                }
        }

        // Reset LED
        rd_data = Xil_In32(XPAR_USR_RTL_APB_BASEADDR + ADDR_AP_DONE);
        delay(100000);

   cleanup_platform();
   return 0;
}
```

ii. fpga_top

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename     : fpga_top.sv
// Author       : Castlab
//                  Junsoo Kim   < junsoo999@kaist.ac.kr   >
// ---------------------------------------------------------------------
// Description: FPGA Top module
// -FHDR----------------------------------------------------------------

module fpga_top
(
  input  logic               diff_clock_rtl_clk_n,
   input  logic               diff_clock_rtl_clk_p,
   input  logic [3:0]          axi_gpio_tri_i,
   input  logic               reset,
   input  logic               usb_uart_rxd,
   output logic               usb_uart_txd,
   output logic [3:0]          o_led
);

////////////////////////////////////////////////////////////////////////////

// APB interface
localparam APB_BASE_ADDR       = 0;
localparam APB_ADDR_WIDTH      = 32;
localparam APB_DATA_WIDTH      = 32;
localparam APB_PPROT_WIDTH     = 3;
localparam APB_PSTRB_WIDTH     = 4;

////////////////////////////////////////////////////////////////////////////

/* TODO: your code  */
logic [APB_ADDR_WIDTH-1:0]     apb_paddr;
logic                         apb_penable;
logic [APB_PPROT_WIDTH-1:0]    apb_pprot;
logic [APB_DATA_WIDTH-1:0]     apb_prdata;
logic                         apb_pready;
logic                         apb_psel;
logic                         apb_pslverr;
logic [APB_PSTRB_WIDTH-1:0]    apb_pstrb;
logic [APB_DATA_WIDTH-1:0]     apb_pwdata;
logic                         apb_pwrite;
```

```
/* TODO: end       */

////////////////////////////////////////////////////////////////////////////
// Shell Block Design
////////////////////////////////////////////////////////////////////////////

// Shell instantiation
// i for input port and o for output port;
design_1_wrapper
u_shell
(
   .axi_gpio_tri_i        ( axi_gpio_tri_i         ),   // io
   .diff_clock_rtl_clk_n   ( diff_clock_rtl_clk_n     ),   // i
   .diff_clock_rtl_clk_p   ( diff_clock_rtl_clk_p     ),   // i
   .reset                ( reset               ),   // i
   .usb_uart_rxd          ( usb_uart_rxd         ),   // i
   .usb_uart_txd          ( usb_uart_txd         ),   // o
   .usr_rtl_apb_paddr     ( apb_paddr           ),   // o
   .usr_rtl_apb_penable    ( apb_penable          ),   // o
   .usr_rtl_apb_prdata     ( apb_prdata           ),   // i
   .usr_rtl_apb_pready     ( apb_pready           ),   // i
   .usr_rtl_apb_psel      ( apb_psel            ),   // o
   .usr_rtl_apb_pslverr    ( apb_pslverr          ),   // i
   .usr_rtl_apb_pwdata     ( apb_pwdata           ),   // o
   .usr_rtl_apb_pwrite     ( apb_pwrite           ),   // o
   .usr_rtl_clk           ( CLK                 ),   // o
   .usr_rtl_rst           ( RESET                )    // o
);

////////////////////////////////////////////////////////////////////////////
// RTL Kernel
////////////////////////////////////////////////////////////////////////////

rtl_kernel
#(
  .APB_BASE_ADDR         ( APB_BASE_ADDR        ),
  .APB_ADDR_WIDTH        ( APB_ADDR_WIDTH        ),
  .APB_DATA_WIDTH        ( APB_DATA_WIDTH        ),
  .APB_PPROT_WIDTH       ( APB_PPROT_WIDTH       ),
  .APB_PSTRB_WIDTH       ( APB_PSTRB_WIDTH       )
)
u_rtl_kernel
(
  .clk                  ( CLK                ),
  .reset                ( ~RESET              ),

  // FPGA status
  .out_led              ( o_led               ),

  // APB
  .s_apb_paddr          ( apb_paddr           ),
  .s_apb_penable         ( apb_penable         ),
  .s_apb_pprot          ( apb_pprot         ),
  .s_apb_prdata         ( apb_prdata          ),
  .s_apb_pready         ( apb_pready          ),
  .s_apb_psel           ( apb_psel          ),
  .s_apb_pslverr        ( apb_pslverr         ),
  .s_apb_pstrb          ( apb_pstrb         ),
  .s_apb_pwdata         ( apb_pwdata          ),
  .s_apb_pwrite         ( apb_pwrite          )
);

////////////////////////////////////////////////////////////////////////////

endmodule
```

iii. rtl_kernel

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename     : rtl_kernel_control.sv
```

```
// Author       : Castlab
//                 Junsoo Kim   < junsoo999@kaist.ac.kr   >
// -----------------------------------------------------------------
// Description: RTL kernel top module
// -FHDR-----------------------------------------------------------

module rtl_kernel
  import kernel_pkg::*;
#(
  parameter APB_BASE_ADDR             = 0,
  parameter APB_ADDR_WIDTH            = 32,
  parameter APB_DATA_WIDTH            = 32,
  parameter APB_PPROT_WIDTH           = 3,
  parameter APB_PSTRB_WIDTH           = 4
)
(
  input  logic                       clk,
  input  logic                       reset,

  // FPGA status
  output logic [3:0]                  out_led,

  // APB
  input  logic [APB_ADDR_WIDTH-1:0]    s_apb_paddr,
  input  logic                       s_apb_penable,
  input  logic [APB_PPROT_WIDTH-1:0]   s_apb_pprot,
  output logic [APB_DATA_WIDTH-1:0]    s_apb_prdata,
  output logic                       s_apb_pready,
  input  logic                       s_apb_psel,
  output logic                       s_apb_pslverr,
  input  logic [APB_PSTRB_WIDTH-1:0]   s_apb_pstrb,
  input  logic [APB_DATA_WIDTH-1:0]    s_apb_pwdata,
  input  logic                       s_apb_pwrite
);

////////////////////////////////////////////////////////////////////////////

/* TODO: your code  */
logic                        ap_start;
logic                        ap_idle;
logic                        ap_done;

logic [DIM_L_WIDTH-1:0]         dim_l;
logic [DIM_M_WIDTH-1:0]         dim_m;
logic [DIM_N_WIDTH-1:0]         dim_n;

  // Memory interface
logic                        mem_cen;
logic                        mem_wen;
logic [BUFF_ADDR_WIDTH-1:0]     mem_addr;
logic [BUFF_DATA_WIDTH-1:0]     mem_din;
logic [BUFF_DATA_WIDTH-1:0]     mem_dout;
logic                        mem_valid;
/* TODO: end       */

////////////////////////////////////////////////////////////////////////////
// AP Control
////////////////////////////////////////////////////////////////////////////

apb_slave
#(
  .APB_BASE_ADDR     ( APB_BASE_ADDR    ),
  .APB_ADDR_WIDTH    ( APB_ADDR_WIDTH   ),
  .APB_DATA_WIDTH    ( APB_DATA_WIDTH   )
)
u_apb_slave
(
  .clk              ( clk              ),
  .reset            ( reset            ),

  // FPGA status
  .out_led          ( out_led          ),
```

```
  // Engine status
  .ap_start         ( ap_start   ),
  .ap_idle          ( ap_idle    ),
  .ap_done          ( ap_done    ),

  // GEMM dimension
  .dim_l            ( dim_l   ),
  .dim_m            ( dim_m   ),
  .dim_n            ( dim_n   ),

  // Memory interface
  .mem_cen          ( mem_cen   ),
  .mem_wen          ( mem_wen   ),
  .mem_addr         ( mem_addr  ),
  .mem_din          ( mem_din   ),
  .mem_dout         ( mem_dout  ),
  .mem_valid        ( mem_valid  ),

  // APB interface
  .s_apb_paddr       ( s_apb_paddr      ),
  .s_apb_penable     ( s_apb_penable    ),
  .s_apb_pprot       ( s_apb_pprot      ),
  .s_apb_prdata      ( s_apb_prdata     ),
  .s_apb_pready      ( s_apb_pready     ),
  .s_apb_psel        ( s_apb_psel       ),
  .s_apb_pslverr     ( s_apb_pslverr    ),
  .s_apb_pstrb       ( s_apb_pstrb      ),
  .s_apb_pwdata      ( s_apb_pwdata     ),
  .s_apb_pwrite      ( s_apb_pwrite     )
);

////////////////////////////////////////////////////////////////////////////
// GEMM Engine
////////////////////////////////////////////////////////////////////////////

gemm_engine
u_gemm_engine
(
  .clk              ( clk   ),
  .reset            ( reset  ),

  // Engine status
  .ap_start         ( ap_start   ),
  .ap_idle          ( ap_idle    ),
  .ap_done          ( ap_done    ),

  // GEMM dimension
  .dim_l            ( dim_l   ),
  .dim_m            ( dim_m   ),
  .dim_n            ( dim_n   ),

  // Memory interface
  .mem_cen          ( mem_cen    ),
  .mem_wen          ( mem_wen    ),
  .mem_addr         ( mem_addr   ),
  .mem_din          ( mem_din    ),
  .mem_dout         ( mem_dout   ),
  .mem_valid        ( mem_valid   )
);

////////////////////////////////////////////////////////////////////////////

endmodule
```

iv. apb_slave

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename     : apb_slave.sv
// Author       : Castlab
//                  Junsoo Kim   < junsoo999@kaist.ac.kr   >
// ---------------------------------------------------------------------
```

```
// Description: Core configuration with APB Slave
// -FHDR----------------------------------------------------------

module apb_slave
  import kernel_pkg::*;
#(
  parameter APB_BASE_ADDR             = 0,
  parameter APB_ADDR_WIDTH            = 32,
  parameter APB_DATA_WIDTH            = 32,
  parameter APB_PPROT_WIDTH           = 3,
  parameter APB_PSTRB_WIDTH           = 4
)
(
  input  logic                       clk,
  input  logic                       reset,

  // FPGA status
  output logic [3:0]                  out_led,

  // Engine status
  output logic                       ap_start,
  input  logic                       ap_idle,
  input  logic                       ap_done,

  // GEMM dimension
  output logic [DIM_L_WIDTH-1:0]       dim_l,
  output logic [DIM_M_WIDTH-1:0]       dim_m,
  output logic [DIM_N_WIDTH-1:0]       dim_n,

  // Memory buffers
  output logic                       mem_cen,
  output logic                       mem_wen,
  output logic [BUFF_ADDR_WIDTH-1:0]    mem_addr,
  output logic [BUFF_DATA_WIDTH-1:0]    mem_din,
  input  logic [BUFF_DATA_WIDTH-1:0]    mem_dout,
  input  logic                       mem_valid,

  // APB interface
  input  logic [APB_ADDR_WIDTH-1:0]    s_apb_paddr,
  input  logic                       s_apb_penable,
  input  logic [APB_PPROT_WIDTH-1:0]    s_apb_pprot,
  output logic [APB_DATA_WIDTH-1:0]    s_apb_prdata,
  output logic                       s_apb_pready,
  input  logic                       s_apb_psel,
  output logic                       s_apb_pslverr,
  input  logic [APB_PSTRB_WIDTH-1:0]    s_apb_pstrb,
  input  logic [APB_DATA_WIDTH-1:0]    s_apb_pwdata,
  input  logic                       s_apb_pwrite
);

////////////////////////////////////////////////////////////////////////

localparam ADDR_HEADER     = 2;
localparam CTRL_HEAD       = 0;
localparam DATA_HEAD       = 1;

localparam CTRL_ADDR_WIDTH  = 7;
localparam CTRL_DATA_WIDTH  = APB_DATA_WIDTH;   // 32
localparam DATA_ADDR_WIDTH  = BUFF_ADDR_WIDTH;  // 14
localparam DATA_DATA_WIDTH  = APB_DATA_WIDTH;   // 32

////////////////////////////////////////////////////////////////////////
// ------------------------- Address Information -------------------------
// 0x00 : ap_start                          ( write           )
// 0x04 : ap_done                           ( read            )
// 0x08 : ap_idle                           ( read            )
// 0x10 : dim_l                             ( write           )
// 0x14 : dim_m                             ( write           )
// 0x18 : dim_n                             ( write           )
// -----------------------------------------------------------------------
localparam ADDR_AP_START    = 7'h00;
localparam ADDR_AP_DONE     = 7'h04;
localparam ADDR_AP_IDLE     = 7'h08;
```

```
localparam ADDR_DIM_L      = 7'h10;
localparam ADDR_DIM_M      = 7'h14;
localparam ADDR_DIM_N      = 7'h18;

//////////////////////////////////////////////////////////////////////
// Status
typedef enum logic [1:0] {
   STATE_SETUP,
   STATE_READ_ACCESS,
   STATE_READING,
   STATE_WRITE_ACCESS
} StatusType;

// Contol registers
logic [CTRL_DATA_WIDTH-1:0] int_ap_start,int_ap_start_nxt;
logic [CTRL_DATA_WIDTH-1:0] int_ap_done,int_ap_done_nxt;
logic [CTRL_DATA_WIDTH-1:0] int_ap_idle,int_ap_idle_nxt;
logic [CTRL_DATA_WIDTH-1:0] int_dim_l,int_dim_l_nxt;
logic [CTRL_DATA_WIDTH-1:0] int_dim_m,int_dim_m_nxt;
logic [CTRL_DATA_WIDTH-1:0] int_dim_n,int_dim_n_nxt;

// Control signal
logic                       ctrl_wr_en;
logic [CTRL_ADDR_WIDTH-1:0] ctrl_wr_addr;
logic [CTRL_DATA_WIDTH-1:0] ctrl_wr_data;
logic                       ctrl_rd_en;
logic [CTRL_ADDR_WIDTH-1:0] ctrl_rd_addr;
logic [CTRL_DATA_WIDTH-1:0] ctrl_rd_data;

// Data signal
logic                       data_wr_en;
logic [DATA_ADDR_WIDTH-1:0] data_wr_addr;
logic [DATA_DATA_WIDTH-1:0] data_wr_data;
logic                       data_rd_en;
logic [DATA_ADDR_WIDTH-1:0] data_rd_addr;
logic [DATA_DATA_WIDTH-1:0] data_rd_data;

// To-do
// Status declaration
StatusType         state_ff, state_nxt;

// APB output declaration
logic
         s_apb_pready_ff;

// Output assignment using combinational logic
always_comb begin
    ap_start = int_ap_start;
    dim_l    = int_dim_l;
    dim_m    = int_dim_m;
    dim_n    = int_dim_n;
    s_apb_pslverr = 'b0;
end

// Sequential logic update
always_ff @(posedge clk) begin
    state_ff          <= reset ? STATE_SETUP : state_nxt;

    int_ap_start       <= int_ap_start_nxt;
    int_ap_idle       <= reset ? 'b0 : ap_idle;
    int_ap_done       <= reset ? 'b0 : ap_done;
    int_dim_l          <= int_dim_l_nxt;
    int_dim_m          <= int_dim_m_nxt;
    int_dim_n          <= int_dim_n_nxt;
end

always_comb begin
    state_nxt             = state_ff;
    int_ap_start_nxt       = int_ap_start;
    int_dim_l_nxt         = int_dim_l;
    int_dim_m_nxt         = int_dim_m;
    int_dim_n_nxt         = int_dim_n;
    mem_cen              = 'b0;
```

```
mem_wen              = 'b0;
mem_addr             = 'b0;
mem_din              = 'b0;
s_apb_prdata          = 'b0;
s_apb_pready          = 'b0;
ctrl_rd_en           = 'b0;

case(state_ff)
    STATE_SETUP: begin
        if (s_apb_psel == 'b1) begin
            case (s_apb_pwrite)
                1'b0: state_nxt = STATE_READ_ACCESS;
                1'b1: state_nxt = STATE_WRITE_ACCESS;
            endcase
        end
    end

    STATE_READ_ACCESS: begin
        // APB read
        if (s_apb_penable == 'b1) begin
            // BRAM
            if (s_apb_paddr[15:14] == DATA_HEAD) begin
                state_nxt = STATE_READING;
                mem_cen = 'b1;
                mem_wen = 'b0;
                mem_addr = s_apb_paddr[DATA_ADDR_WIDTH-1:0];
            end
            // Register
            else if (s_apb_paddr[15:14] == CTRL_HEAD) begin
                state_nxt = STATE_SETUP;
                ctrl_rd_en = 'b1;
                ctrl_rd_addr = s_apb_paddr[CTRL_ADDR_WIDTH-1:0];
                case(s_apb_paddr[CTRL_ADDR_WIDTH-1:0])
                    ADDR_AP_START: begin
                        s_apb_prdata = int_ap_start;
                        s_apb_pready = 'b1;
                    end
                    ADDR_AP_DONE: begin
                        s_apb_prdata = int_ap_done;
                        s_apb_pready = 'b1;
                    end
                    ADDR_AP_IDLE: begin
                        s_apb_prdata = int_ap_idle;
                        s_apb_pready = 'b1;
                    end
                    ADDR_DIM_L: begin
                        s_apb_prdata = int_dim_l;
                        s_apb_pready = 'b1;
                    end
                    ADDR_DIM_M: begin
                        s_apb_prdata = int_dim_m;
                        s_apb_pready = 'b1;
                    end
                    ADDR_DIM_N: begin
                        s_apb_prdata = int_dim_n;
                        s_apb_pready = 'b1;
                    end
                    default: begin
                        s_apb_prdata = {APB_DATA_WIDTH{1'bx}};
                    end
                endcase
            end
        end
    end

    STATE_READING: begin
        if(mem_valid) begin
            state_nxt = STATE_SETUP;
            s_apb_prdata = mem_dout;
            s_apb_pready = 'b1;
        end else begin
            state_nxt = STATE_READING;
        end
```

```
        end

        STATE_WRITE_ACCESS: begin
            // APB write
            if (s_apb_penable == 'b1) begin
                // BRAM
                if (s_apb_paddr[15:14] == DATA_HEAD) begin
                    mem_cen = 'b1;
                    mem_wen = 'b1;
                    mem_addr = s_apb_paddr[DATA_ADDR_WIDTH-1:0];
                    mem_din = s_apb_pwdata;
                end
                // Register
                else if (s_apb_paddr[15:14] == CTRL_HEAD) begin
                    case(s_apb_paddr[CTRL_ADDR_WIDTH-1:0])
                        ADDR_AP_START: begin
                            int_ap_start_nxt = s_apb_pwdata;
                        end
                        ADDR_DIM_L: begin
                            int_dim_l_nxt = s_apb_pwdata;
                        end
                        ADDR_DIM_M: begin
                            int_dim_m_nxt = s_apb_pwdata;
                        end
                        ADDR_DIM_N: begin
                            int_dim_n_nxt = s_apb_pwdata;
                        end
                    endcase
                end
                state_nxt = STATE_SETUP;
                s_apb_pready = 'b1;
            end
        end
    endcase
end

////////////////////////////////////////////////////////////////////////////
// Control Interface
////////////////////////////////////////////////////////////////////////////

/* TODO: your code  */

/* TODO: end       */

////////////////////////////////////////////////////////////////////////////
// Control Registers
////////////////////////////////////////////////////////////////////////////

/* TODO: your code  */

/* TODO: end       */

////////////////////////////////////////////////////////////////////////////
// Memory Interface
////////////////////////////////////////////////////////////////////////////

/* TODO: your code  */

/* TODO: end       */

////////////////////////////////////////////////////////////////////////////
// APB Protocol
////////////////////////////////////////////////////////////////////////////

/* TODO: your code  */

/* TODO: end       */

////////////////////////////////////////////////////////////////////////////
// LED : Do not modify this code!
////////////////////////////////////////////////////////////////////////////

// LED[0] : access input memory
```

```
always @(posedge clk) begin
  if (reset)
    out_led[0]  <= 0;

  else if (mem_cen && mem_addr[BUFF_ADDR_WIDTH-1 -: 2] == 0)
    out_led[0]  <= 1;

  else if (ctrl_rd_en && ctrl_rd_addr == ADDR_AP_DONE)
    out_led[0]  <= 0;
end

// LED[1] : access weight memory
always @(posedge clk) begin
  if (reset)
    out_led[1]  <= 0;

  else if (mem_cen && mem_addr[BUFF_ADDR_WIDTH-1 -: 2] == 1)
    out_led[1]  <= 1;

  else if (ctrl_rd_en && ctrl_rd_addr == ADDR_AP_DONE)
    out_led[1]  <= 0;
end

// LED[2] : access output memory
always @(posedge clk) begin
  if (reset)
    out_led[2]  <= 0;

  else if (mem_cen && mem_addr[BUFF_ADDR_WIDTH-1 -: 2] == 2)
    out_led[2]  <= 1;

  else if (ctrl_rd_en && ctrl_rd_addr == ADDR_AP_DONE)
    out_led[2]  <= 0;
end

// LED[3] : ap done signal
always @(posedge clk) begin
  if (reset)
    out_led[3]  <= 0;

  else if (ap_done)
    out_led[3]  <= 1;

  else if (ctrl_rd_en && ctrl_rd_addr == ADDR_AP_DONE)
    out_led[3]  <= 0;
end

////////////////////////////////////////////////////////////////////////////

endmodule
```

v. gemm_engine

```
// +FHDR-------------------------------------------------------------
//              Copyright (c) 2022 .
//                   ALL RIGHTS RESERVED
// ------------------------------------------------------------------
// Filename     : gemm_engine.sv
// Author       : Castlab
//                                      Junsoo Kim   < junsoo999@kaist.ac.kr   >
// ------------------------------------------------------------------
// Description: Engine module
// -FHDR-------------------------------------------------------------

module gemm_engine
  import kernel_pkg::*;
(
  input  logic                        clk,
  input  logic                        reset,

  // Engine status
  input  logic                        ap_start,
  output logic                        ap_idle,
  output logic                        ap_done,
```

```
  // GEMM dimension
  input  logic [DIM_L_WIDTH-1:0]        dim_l,
  input  logic [DIM_M_WIDTH-1:0]        dim_m,
  input  logic [DIM_N_WIDTH-1:0]        dim_n,

  // Memory interface
  input  logic                         mem_cen,
  input  logic                         mem_wen,
  input  logic [BUFF_ADDR_WIDTH-1:0]     mem_addr,
  input  logic [BUFF_DATA_WIDTH-1:0]     mem_din,
  output logic [BUFF_DATA_WIDTH-1:0]     mem_dout,
  output logic                         mem_valid
);

//////////////////////////////////////////////////////////////////////////

// Control memory
logic                                i_rd_en;
logic [IMEM_ADDR_WIDTH-1:0]             i_rd_addr;
logic [IMEM_DATA_WIDTH-1:0]             i_rd_data;
logic                                i_rd_valid;
logic                                w_rd_en;
logic [WMEM_ADDR_WIDTH-1:0]             w_rd_addr;
logic [WMEM_DATA_WIDTH-1:0]             w_rd_data;
logic                                w_rd_valid;
logic                                o_wr_en;
logic [OMEM_ADDR_WIDTH-1:0]             o_wr_addr;
logic [OMEM_DATA_WIDTH-1:0]             o_wr_data;

  // Control pipelinied adder-tree
logic                                at_status;
logic                                at_accum;
logic [VECTOR_LENGTH-1:0][DATA_WIDTH-1:0]  at_i_data;
logic [VECTOR_LENGTH-1:0]               at_i_valid;
logic [VECTOR_LENGTH-1:0][DATA_WIDTH-1:0]  at_w_data;
logic [VECTOR_LENGTH-1:0]               at_w_valid;
logic [DATA_WIDTH-1:0]                 at_o_data;
logic                                at_o_valid;

//////////////////////////////////////////////////////////////////////////
// Controller
//////////////////////////////////////////////////////////////////////////

controller
u_controller
(
  .clk          ( clk     ),
  .reset        ( reset   ),

  // Engine status
  .ap_start      ( ap_start   ),
  .ap_idle       ( ap_idle    ),
  .ap_done       ( ap_done    ),

  // GEMM dimension
  .dim_l         ( dim_l    ),
  .dim_m         ( dim_m    ),
  .dim_n         ( dim_n    ),

  // Control memory
  .i_rd_en       ( i_rd_en    ),
  .i_rd_addr     ( i_rd_addr    ),
  .i_rd_data     ( i_rd_data    ),
  .i_rd_valid    ( i_rd_valid    ),
  .w_rd_en       ( w_rd_en    ),
  .w_rd_addr     ( w_rd_addr    ),
  .w_rd_data     ( w_rd_data    ),
  .w_rd_valid    ( w_rd_valid    ),
  .o_wr_en       ( o_wr_en    ),
  .o_wr_addr     ( o_wr_addr    ),
  .o_wr_data     ( o_wr_data    ),
```

```
  // Control simd
  .at_status     ( at_status   ),
  .at_accum      ( at_accum    ),
  .at_i_data     ( at_i_data   ),
  .at_i_valid    ( at_i_valid  ),
  .at_w_data     ( at_w_data   ),
  .at_w_valid    ( at_w_valid  ),
  .at_o_data     ( at_o_data   ),
  .at_o_valid    ( at_o_valid  )
);

////////////////////////////////////////////////////////////////////////
// Data buffer
////////////////////////////////////////////////////////////////////////

data_buffer
u_data_buffer
(
  .clk          ( clk   ),
  .reset        ( reset   ),

  // Memory interface
  .mem_cen       ( mem_cen    ),
  .mem_wen       ( mem_wen    ),
  .mem_addr      ( mem_addr   ),
  .mem_din       ( mem_din    ),
  .mem_dout      ( mem_dout   ),
  .mem_valid     ( mem_valid   ),

  // Engine interface
  .i_rd_en       ( i_rd_en    ),
  .i_rd_addr     ( i_rd_addr   ),
  .i_rd_data     ( i_rd_data   ),
  .i_rd_valid    ( i_rd_valid  ),
  .w_rd_en       ( w_rd_en    ),
  .w_rd_addr     ( w_rd_addr   ),
  .w_rd_data     ( w_rd_data   ),
  .w_rd_valid    ( w_rd_valid  ),
  .o_wr_en       ( o_wr_en    ),
  .o_wr_addr     ( o_wr_addr   ),
  .o_wr_data     ( o_wr_data   )
);

////////////////////////////////////////////////////////////////////////
// Pipelined Adder-Tree
////////////////////////////////////////////////////////////////////////

pipe_addertree
u_pipe_addertree
(
  .clk          ( clk   ),
  .reset        ( reset   ),

  // Adder-tree status
  .at_status     ( at_status   ),
  .at_accum      ( at_accum    ),

  // Adder-tree data
  .at_i_data     ( at_i_data   ),
  .at_i_valid    ( at_i_valid  ),
  .at_w_data     ( at_w_data   ),
  .at_w_valid    ( at_w_valid  ),
  .at_o_data     ( at_o_data   ),
  .at_o_valid    ( at_o_valid  )
);

////////////////////////////////////////////////////////////////////////

endmodule
```

vi. controller

```
// +FHDR----------------------------------------------------------------
//              Copyright (c) 2022 .
```

```
//                   ALL RIGHTS RESERVED
// ----------------------------------------------------------------------
// Filename      : controller.sv
// Author        : Castlab
//                                        Junsoo Kim   < junsoo999@kaist.ac.kr   >
// ----------------------------------------------------------------------
// Description: Controller for pipelined adder-tree
// -FHDR----------------------------------------------------------------

module controller
  import kernel_pkg::*;
(
  input  logic                                  clk,
  input  logic                                  reset,

  // Engine status
  input  logic                                  ap_start,
  output logic                                  ap_idle,
  output logic                                  ap_done,

  // GEMM dimension
  input  logic [DIM_L_WIDTH-1:0]                  dim_l,
  input  logic [DIM_M_WIDTH-1:0]                  dim_m,
  input  logic [DIM_N_WIDTH-1:0]                  dim_n,

  // Control memory
  output logic                                  i_rd_en,
  output logic [IMEM_ADDR_WIDTH-1:0]              i_rd_addr,
  input  logic [IMEM_DATA_WIDTH-1:0]              i_rd_data,  // 32 * 16
  input  logic                                  i_rd_valid,
  output logic                                  w_rd_en,
  output logic [WMEM_ADDR_WIDTH-1:0]              w_rd_addr,
  input  logic [WMEM_DATA_WIDTH-1:0]              w_rd_data,  // 32 * 16
  input  logic                                  w_rd_valid,
  output logic                                  o_wr_en,
  output logic [OMEM_ADDR_WIDTH-1:0]              o_wr_addr,  // 32
  output logic [OMEM_DATA_WIDTH-1:0]              o_wr_data,

  // Control pipelinied adder-tree
  input  logic                                  at_status,  // 0 : idle, 1 : busy
  output logic                                  at_accum,
  output logic [VECTOR_LENGTH-1:0][DATA_WIDTH-1:0]  at_i_data,        // 32 * 16
  output logic [VECTOR_LENGTH-1:0]                  at_i_valid,
  output logic [VECTOR_LENGTH-1:0][DATA_WIDTH-1:0]  at_w_data,        // 32 * 16
  output logic [VECTOR_LENGTH-1:0]                  at_w_valid,
  input  logic [DATA_WIDTH-1:0]                   at_o_data,  // 32
  input  logic                                  at_o_valid
);

////////////////////////////////////////////////////////////////////////

localparam SIMD_LATENCY    = 6;

logic                   ap_idle_ff, ap_idle_nxt,at_accum_ff,at_accum_nxt;
logic [DIM_L_WIDTH-1:0]    dim_l_ff, dim_l_nxt;
logic [DIM_M_WIDTH-1:0]    dim_m_ff, dim_m_nxt;
logic [DIM_N_WIDTH-1:0]    dim_n_ff, dim_n_nxt;
logic [DIM_L_WIDTH-1:0]    dim_r_ff, dim_r_nxt;
logic [DIM_L_WIDTH-1:0]    r_ff,r_nxt;
logic [DIM_M_WIDTH-1:0]    in_m_ff,in_m_nxt;
logic [DIM_N_WIDTH-1:0]    in_n_ff,in_n_nxt;
logic [DIM_M_WIDTH-1:0]    out_m_ff,out_m_nxt;
logic [DIM_N_WIDTH-1:0]    out_n_ff,out_n_nxt;
logic [DIM_L_WIDTH-1:0]    out_r_ff,out_r_nxt;
logic [VECTOR_LENGTH-1:0]   at_i_valid_ff, at_i_valid_nxt;
logic [VECTOR_LENGTH-1:0]   at_w_valid_ff, at_w_valid_nxt;


// Status
enum logic [1:0] {
    STATE_IDLE,
    STATE_RUN,
    STATE_DONE
```

```
} state_ff, state_nxt;

assign ap_idle = ap_idle_ff;
assign at_accum = at_accum_ff;
assign at_i_valid = at_i_valid_ff;
assign at_w_valid = at_w_valid_ff;

////////////////////////////////////////////////////////////////////////

always_ff @ (posedge clk) begin
    state_ff <= state_nxt;
    ap_idle_ff <= ap_idle_nxt;
    at_accum_ff <= at_accum_nxt;
    dim_l_ff <= dim_l_nxt;
    dim_m_ff <= dim_m_nxt;
    dim_n_ff <= dim_n_nxt;
    dim_r_ff <= dim_r_nxt;
    r_ff <= r_nxt;
    in_m_ff <= in_m_nxt;
    in_n_ff <= in_n_nxt;
    out_m_ff <= out_m_nxt;
    out_n_ff <= out_n_nxt;
    out_r_ff <= out_r_nxt;
    at_i_valid_ff <= at_i_valid_nxt;
    at_w_valid_ff <= at_w_valid_nxt;
end

always_comb begin
    if (reset) begin
        state_nxt = STATE_IDLE;
        ap_idle_nxt = 1;
        at_accum_nxt = 0;
        ap_done = 0;
        dim_l_nxt = 0;
        dim_m_nxt = 0;
        dim_n_nxt = 0;
        dim_r_nxt = 0;
        r_nxt = 0;
        in_m_nxt = 0;
        in_n_nxt = 0;
        out_m_nxt = 0;
        out_n_nxt = 0;
        out_r_nxt = 0;
        i_rd_en = 0;
        i_rd_addr = 0;
        w_rd_en = 0;
        w_rd_addr = 0;
        o_wr_en = 0;
        o_wr_addr = 0;
        o_wr_data = 0;
        at_i_data = 0;
        at_i_valid_nxt = 0;
        at_w_data = 0;
        at_w_valid_nxt = 0;
    end else begin
        state_nxt = state_ff;
        ap_idle_nxt = 1;
        at_accum_nxt = 0;
        ap_done = 0;
        dim_l_nxt = dim_l_ff;
        dim_m_nxt = dim_m_ff;
        dim_n_nxt = dim_n_ff;
        dim_r_nxt = dim_r_ff;
        r_nxt = r_ff;
        in_m_nxt = in_m_ff;
        in_n_nxt = in_n_ff;
        out_m_nxt = out_m_ff;
        out_n_nxt = out_n_ff;
        out_r_nxt = out_r_ff;

        i_rd_en = 0;
        i_rd_addr = 0;
        w_rd_en = 0;
```

```
w_rd_addr = 0;
o_wr_en = 0;
o_wr_addr = 0;
o_wr_data = 0;

at_i_data = 0;
at_i_valid_nxt = 0;
at_w_data = 0;
at_w_valid_nxt = 0;

case (state_ff)
    STATE_IDLE: begin
        if (ap_start) begin
            state_nxt = STATE_RUN;
            ap_idle_nxt = 0;
            dim_l_nxt = dim_l;
            dim_m_nxt = dim_m;
            dim_n_nxt = dim_n;
            dim_r_nxt = dim_l / VECTOR_LENGTH;
            r_nxt = 0;
            in_m_nxt = 0;
            in_n_nxt = 0;
            out_m_nxt = 0;
            out_n_nxt = 0;
            out_r_nxt = 0;
        end
    end

    STATE_RUN: begin
        ap_idle_nxt = 0;
        // Increase input matrix indeces
        if (r_ff < dim_r_ff-1) begin
            r_nxt = r_ff + 1;
        end else begin
            if (in_m_ff < dim_m_ff-1) begin
                in_m_nxt = in_m_ff + 1;
                r_nxt = 0;
            end else begin
                if (in_n_ff < dim_n_ff-1) begin
                    in_n_nxt = in_n_ff + 1;
                    in_m_nxt = 0;
                    r_nxt = 0;
                end else begin
                    in_n_nxt = dim_n_ff;
                    in_m_nxt = dim_m_ff;
                    r_nxt = dim_r_ff;
                end
            end
        end
        // Read input values
        if (r_ff < dim_r_ff && in_m_ff < dim_m_ff && in_n_ff < dim_n_ff) begin
            i_rd_en = 1;
            i_rd_addr = in_n_ff*dim_r_ff + r_ff;
            w_rd_en = 1;
            w_rd_addr = in_m_ff*dim_r_ff + r_ff;
            if (r_ff > 0) at_accum_nxt = 1;
            for (int i=0;i<VECTOR_LENGTH;i++) begin
                at_i_valid_nxt[i] = 1;
            end
            for (int i=0;i<VECTOR_LENGTH;i++) begin
                at_w_valid_nxt[i] = 1;
            end
        end
        // Insert to pipe_addertree
        if (i_rd_valid) begin
            at_i_data[0] = i_rd_data[31:0];
            at_i_data[1] = i_rd_data[63:32];
            at_i_data[2] = i_rd_data[95:64];
            at_i_data[3] = i_rd_data[127:96];
            at_i_data[4] = i_rd_data[159:128];
            at_i_data[5] = i_rd_data[191:160];
            at_i_data[6] = i_rd_data[223:192];
            at_i_data[7] = i_rd_data[255:224];
```

```
                at_i_data[8] = i_rd_data[287:256];
                at_i_data[9] = i_rd_data[319:288];
                at_i_data[10] = i_rd_data[351:320];
                at_i_data[11] = i_rd_data[383:352];
                at_i_data[12] = i_rd_data[415:384];
                at_i_data[13] = i_rd_data[447:416];
                at_i_data[14] = i_rd_data[479:448];
                at_i_data[15] = i_rd_data[511:480];
            end
            if (w_rd_valid) begin
                at_w_data[0] = w_rd_data[31:0];
                at_w_data[1] = w_rd_data[63:32];
                at_w_data[2] = w_rd_data[95:64];
                at_w_data[3] = w_rd_data[127:96];
                at_w_data[4] = w_rd_data[159:128];
                at_w_data[5] = w_rd_data[191:160];
                at_w_data[6] = w_rd_data[223:192];
                at_w_data[7] = w_rd_data[255:224];
                at_w_data[8] = w_rd_data[287:256];
                at_w_data[9] = w_rd_data[319:288];
                at_w_data[10] = w_rd_data[351:320];
                at_w_data[11] = w_rd_data[383:352];
                at_w_data[12] = w_rd_data[415:384];
                at_w_data[13] = w_rd_data[447:416];
                at_w_data[14] = w_rd_data[479:448];
                at_w_data[15] = w_rd_data[511:480];
            end
            // Save calculation result
            if (at_o_valid) begin
                // Increase output matrix indeces
                if (out_r_ff < dim_r_ff-1) begin
                    out_r_nxt = out_r_ff + 1;
                end else begin
                    if (out_m_ff < dim_m_ff-1) begin
                        out_m_nxt = out_m_ff + 1;
                        out_r_nxt = 0;
                    end else begin
                        if (out_n_ff < dim_n_ff -1) begin
                            out_n_nxt = out_n_ff + 1;
                            out_m_nxt = 0;
                            out_r_nxt = 0;
                        end else begin
                            state_nxt = STATE_DONE;
                        end
                    end
                end
                if (out_r_ff == dim_r_ff-1) begin
                    o_wr_en = 1;
                    o_wr_addr = (out_n_ff*dim_m_ff)+out_m_ff;
                    o_wr_data = at_o_data;
                end
            end
        end

        STATE_DONE: begin
            state_nxt = STATE_IDLE;
            ap_idle_nxt = 0;
            dim_l_nxt = 0;
            dim_m_nxt = 0;
            dim_n_nxt = 0;
            dim_r_nxt = 0;
            r_nxt = 0;
            in_m_nxt = 0;
            in_n_nxt = 0;
            out_m_nxt = 0;
            out_n_nxt = 0;
            out_r_nxt = 0;
            ap_done = 1;
        end
    endcase
  end
end
```

```
endmodule
```

vii. pipe_addertree

```
// +FHDR-------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ------------------------------------------------------------------
// Filename      : pipe_addertree.sv
// Author        : Castlab
//                                      Junsoo Kim   < junsoo999@kaist.ac.kr   >
// ------------------------------------------------------------------
// Description: Pipelined Adder-Tree module
// -FHDR-------------------------------------------------------------

module pipe_addertree
  import kernel_pkg::*;
(
  input  logic                                  clk,
  input  logic                                  reset,

  // Adder-tree status
  output logic                                  at_status,    // 0 : idle, 1 : busy
  input  logic                                  at_accum,

  // Adder-tree data
  input   logic [VECTOR_LENGTH-1:0][DATA_WIDTH-1:0]   at_i_data,         // vector_length = 16,
data_width = 32
  input  logic [VECTOR_LENGTH-1:0]                at_i_valid,
  input  logic [VECTOR_LENGTH-1:0][DATA_WIDTH-1:0]  at_w_data,
  input  logic [VECTOR_LENGTH-1:0]                at_w_valid,
  output logic [DATA_WIDTH-1:0]                 at_o_data,
  output logic                                  at_o_valid
);

////////////////////////////////////////////////////////////////////////

localparam SIMD_LATENCY    = 6;

logic stage1_run_ff, stage2_run_ff, stage3_run_ff, stage4_run_ff, stage5_run_ff, stage6_run_ff;
logic stage2_run_nxt, stage3_run_nxt, stage4_run_nxt, stage5_run_nxt, stage6_run_nxt;
logic stage12_accum_ff, stage12_accum_nxt;
logic stage23_accum_ff, stage23_accum_nxt;
logic stage34_accum_ff, stage34_accum_nxt;
logic stage45_accum_ff, stage45_accum_nxt;
logic stage56_accum_ff, stage56_accum_nxt;
logic [VECTOR_LENGTH-1:0][DATA_WIDTH-1:0]      stage12_ff, stage12_nxt;    // 16 * 32
logic [VECTOR_LENGTH/2-1:0][DATA_WIDTH-1:0]    stage23_ff, stage23_nxt;    // 8 * 32
logic [VECTOR_LENGTH/4-1:0][DATA_WIDTH-1:0]    stage34_ff, stage34_nxt;    // 4 * 32
logic [VECTOR_LENGTH/8-1:0][DATA_WIDTH-1:0]    stage45_ff, stage45_nxt;    // 2 * 32
logic [VECTOR_LENGTH/16-1:0][DATA_WIDTH-1:0]   stage56_ff, stage56_nxt;    // 1 * 32
logic [DATA_WIDTH-1:0]                      prev_result_ff, prev_result_nxt;
logic [VECTOR_LENGTH-1:0]                valid;

////////////////////////////////////////////////////////////////////////
// SIMD status
////////////////////////////////////////////////////////////////////////

assign  at_status  =  (stage1_run_ff  ||  stage2_run_ff  ||  stage3_run_ff  ||  stage4_run_ff  ||
stage5_run_ff || stage6_run_ff) ? 1 : 0;

////////////////////////////////////////////////////////////////////////
// Control signal
////////////////////////////////////////////////////////////////////////

always_ff @ (posedge clk) begin
    // Stage 1
    stage12_accum_ff <= stage12_accum_nxt;
    stage12_ff <= stage12_nxt;
    // Stage 2
    stage2_run_ff <= stage2_run_nxt;
    stage23_accum_ff <= stage23_accum_nxt;
    stage23_ff <= stage23_nxt;
    // Stage 3
```

```
    stage3_run_ff <= stage3_run_nxt;
    stage34_accum_ff <= stage34_accum_nxt;
    stage34_ff <= stage34_nxt;
    // Stage 4
    stage4_run_ff <= stage4_run_nxt;
    stage45_accum_ff <= stage45_accum_nxt;
    stage45_ff <= stage45_nxt;
    // Stage 5
    stage5_run_ff <= stage5_run_nxt;
    stage56_accum_ff <= stage56_accum_nxt;
    stage56_ff <= stage56_nxt;
    // Stage 6
    stage6_run_ff <= stage6_run_nxt;
    prev_result_ff <= prev_result_nxt;
end

////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////

always_comb begin
    if (reset) begin
        stage2_run_nxt = 0;
        stage3_run_nxt = 0;
        stage4_run_nxt = 0;
        stage5_run_nxt = 0;
        stage6_run_nxt = 0;
        stage12_accum_nxt = 0;
        stage23_accum_nxt = 0;
        stage34_accum_nxt = 0;
        stage45_accum_nxt = 0;
        stage56_accum_nxt = 0;
        for (int i=0; i<VECTOR_LENGTH; i++) begin
            stage12_nxt[i] = 0;
            if (i<VECTOR_LENGTH/2) stage23_nxt[i] = 0;
            if (i<VECTOR_LENGTH/4) stage34_nxt[i] = 0;
            if (i<VECTOR_LENGTH/8) stage45_nxt[i] = 0;
            if (i<VECTOR_LENGTH/16) stage56_nxt[i] = 0;
        end
        prev_result_nxt = 0;
        valid = 0;
    end else begin
        // Stage1 : Multiplier //
        // check the input is valid
        valid = 1;
        for (int i=0; i<VECTOR_LENGTH; i++) begin
            if (at_i_valid[i] == 0) valid = 0;
            if (at_w_valid[i] == 0) valid = 0;
        end
        stage1_run_ff = (valid == 1) ? 1 : 0;
        stage2_run_nxt = stage1_run_ff;
        // Calculation
        stage12_accum_nxt = at_accum;
        for (int i=0; i<VECTOR_LENGTH; i++) begin
            stage12_nxt[i] = at_i_data[i] * at_w_data[i];
        end

        // Stage2 : Level1 Adder tree //
        stage3_run_nxt = stage2_run_ff;
        stage23_accum_nxt = stage12_accum_ff;
        for (int i=0; i<VECTOR_LENGTH/2; i++) begin
            stage23_nxt[i] = stage12_ff[i] + stage12_ff[VECTOR_LENGTH - i - 1];
        end

        // Stage3 : Level2 Adder tree //
        stage4_run_nxt = stage3_run_ff;
        stage34_accum_nxt = stage23_accum_ff;
        for (int i=0; i<VECTOR_LENGTH/4; i++) begin
            stage34_nxt[i] = stage23_ff[i] + stage23_ff[VECTOR_LENGTH/2 - i - 1];
        end

        // Stage4 : Level3 Adder tree //
        stage5_run_nxt = stage4_run_ff;
```

```
        stage45_accum_nxt = stage34_accum_ff;
        for (int i=0; i<VECTOR_LENGTH/8; i++) begin
            stage45_nxt[i] = stage34_ff[i] + stage34_ff[VECTOR_LENGTH/4 - i - 1];
        end

        // Stage5 : Level4 Adder tree //
        stage6_run_nxt = stage5_run_ff;
        stage56_accum_nxt = stage45_accum_ff;
        for (int i=0; i<VECTOR_LENGTH/16; i++) begin
            stage56_nxt[i] = stage45_ff[i] + stage45_ff[VECTOR_LENGTH/8 - i - 1];
        end

        // Stage6 : Accumulator //
        // Save Result
        if (stage56_accum_ff) prev_result_nxt = prev_result_ff + stage56_ff[0];
        else prev_result_nxt = stage56_ff[0];
        // Calculation
        if (stage56_accum_ff) begin
            at_o_data = stage56_ff[0] + prev_result_ff;
        end else begin
            at_o_data = stage56_ff[0];
        end
        at_o_valid = (stage6_run_ff) ? 1 : 0;
    end
end
endmodule
```

viii. data_buffer

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename      : data_buffer.sv
// Author        : Castlab
//                                        Junsoo Kim   < junsoo999@kaist.ac.kr   >
// ---------------------------------------------------------------------
// Description: Data buffers for core
// -FHDR----------------------------------------------------------------

module data_buffer
  import kernel_pkg::*;
(
  input  logic                        clk,
  input  logic                        reset,

  // Memory interface
  input  logic                        mem_cen,
  input  logic                        mem_wen,
  input  logic [BUFF_ADDR_WIDTH-1:0]    mem_addr,
  input  logic [BUFF_DATA_WIDTH-1:0]    mem_din,
  output logic [BUFF_DATA_WIDTH-1:0]    mem_dout,
  output logic                        mem_valid,

  // Engine interface
  input  logic                        i_rd_en,
  input  logic [IMEM_ADDR_WIDTH-1:0]    i_rd_addr,
  output logic [IMEM_DATA_WIDTH-1:0]    i_rd_data,    // 32 * 16
  output logic                        i_rd_valid,
  input  logic                        w_rd_en,
  input  logic [WMEM_ADDR_WIDTH-1:0]    w_rd_addr,
  output logic [WMEM_DATA_WIDTH-1:0]    w_rd_data,    // 32 * 16
  output logic                        w_rd_valid,
  input  logic                        o_wr_en,
  input  logic [OMEM_ADDR_WIDTH-1:0]    o_wr_addr,
  input  logic [OMEM_DATA_WIDTH-1:0]    o_wr_data     // 32
);

////////////////////////////////////////////////////////////////////////

logic [BUFF_DATA_WIDTH-1:0]    imem_dout,wmem_dout,omem_dout;
logic                       imem_valid,wmem_valid,omem_valid;
logic [BUFF_ADDR_WIDTH-1:0]    mem_addr_ff;
```

```
////////////////////////////////////////////////////////////////////////
// Input Buffer
////////////////////////////////////////////////////////////////////////

imem
#(
  .IMEM_ADDR_WIDTH  ( IMEM_ADDR_WIDTH   ),
  .IMEM_DATA_WIDTH  ( IMEM_DATA_WIDTH   )
)
u_imem
(
  .clk            ( clk    ),
  .reset          ( reset    ),

  // Memory interface
  .mem_cen         ( mem_cen    ),
  .mem_wen         ( mem_wen    ),
  .mem_addr        ( mem_addr    ),
  .mem_din         ( mem_din    ),
  .mem_dout        ( imem_dout    ),
  .mem_valid       ( imem_valid    ),

  // Engine interface
  .i_rd_en         ( i_rd_en    ),
  .i_rd_addr       ( i_rd_addr    ),
  .i_rd_data       ( i_rd_data    ),
  .i_rd_valid      ( i_rd_valid    )
);

/* TODO: your code  */

/* TODO: end       */

////////////////////////////////////////////////////////////////////////
// Weight Buffer
////////////////////////////////////////////////////////////////////////

wmem
#(
  .WMEM_ADDR_WIDTH  ( WMEM_ADDR_WIDTH   ),
  .WMEM_DATA_WIDTH  ( WMEM_DATA_WIDTH   )
)
u_wmem
(
  .clk            ( clk    ),
  .reset          ( reset    ),

  // Memory interface
  .mem_cen         ( mem_cen    ),
  .mem_wen         ( mem_wen    ),
  .mem_addr        ( mem_addr    ),
  .mem_din         ( mem_din    ),
  .mem_dout        ( wmem_dout    ),
  .mem_valid       ( wmem_valid    ),

  // Engine interface
  .w_rd_en         ( w_rd_en    ),
  .w_rd_addr       ( w_rd_addr    ),
  .w_rd_data       ( w_rd_data    ),
  .w_rd_valid      ( w_rd_valid    )
);

/* TODO: your code  */

/* TODO: end       */

////////////////////////////////////////////////////////////////////////
// Output Buffer
////////////////////////////////////////////////////////////////////////

omem
#(
  .OMEM_ADDR_WIDTH  ( OMEM_ADDR_WIDTH   ),
```

```
  .OMEM_DATA_WIDTH  ( OMEM_DATA_WIDTH   )
)
u_omem
(
  .clk            ( clk    ),
  .reset          ( reset   ),

  // Memory interface
  .mem_cen         ( mem_cen    ),
  .mem_wen         ( mem_wen    ),
  .mem_addr        ( mem_addr    ),
  .mem_din         ( mem_din    ),
  .mem_dout        ( omem_dout    ),
  .mem_valid       ( omem_valid    ),

  // Engine interface
  .o_wr_en         ( o_wr_en    ),
  .o_wr_addr       ( o_wr_addr    ),
  .o_wr_data       ( o_wr_data    )
);

/* TODO: your code  */

/* TODO: end       */

////////////////////////////////////////////////////////////////////////////
// Arbitration
////////////////////////////////////////////////////////////////////////////
always_ff @ (posedge clk) begin
    if (mem_cen) mem_addr_ff <= mem_addr;
end
always_comb begin
    case(mem_addr_ff[13:12])      // memory type
        0: begin
            // imem
            mem_dout = imem_dout;
            mem_valid = imem_valid;
        end
        1: begin
            // wmem
            mem_dout = wmem_dout;
            mem_valid = wmem_valid;
        end
        2: begin
            // omem
            mem_dout = omem_dout;
            mem_valid = omem_valid;
        end
    endcase
end

////////////////////////////////////////////////////////////////////////////

endmodule
```

ix. imem

```
// +FHDR-------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ------------------------------------------------------------------
// Filename     : imem.sv
// Author       : Castlab
//                                      Junsoo Kim   < junsoo999@kaist.ac.kr   >
// ------------------------------------------------------------------
// Description: Input data buffers for core
// -FHDR-------------------------------------------------------------

// check memory address width

module imem
  import kernel_pkg::*;
#(
  parameter IMEM_ADDR_WIDTH           = 10,
```

```
  parameter IMEM_DATA_WIDTH           = DATA_WIDTH * VECTOR_LENGTH
)
(
  input  logic                       clk,
  input  logic                       reset,

  // Memory interface
  input  logic                       mem_cen,
  input  logic                       mem_wen,
  input  logic [BUFF_ADDR_WIDTH-1:0]    mem_addr,   // 14
  input  logic [BUFF_DATA_WIDTH-1:0]    mem_din,    // 32
  output logic [BUFF_DATA_WIDTH-1:0]    mem_dout,   // 32
  output logic                       mem_valid,

  // Engine interface
  input  logic                       i_rd_en,
  input  logic [IMEM_ADDR_WIDTH-1:0]    i_rd_addr,  // 6
  output logic [IMEM_DATA_WIDTH-1:0]    i_rd_data,  // 32 * 16
  output logic                       i_rd_valid
);

////////////////////////////////////////////////////////////////////////
// mem_addr (14bit) = mem type (2bit) + real address (6 bit) + bank # (4bit) + byte (2bit)

localparam IMEM_BANK       = IMEM_DATA_WIDTH / BUFF_DATA_WIDTH;    // 16
localparam HIGH_ADDR       = IMEM_ADDR_WIDTH;                   // 6
localparam LOW_ADDR        = $clog2(IMEM_BANK);                 // 4
localparam BYTE_ADDR       = $clog2(BUFF_DATA_WIDTH / 8);         // 2

localparam BRAM_ADDR_WIDTH  = IMEM_ADDR_WIDTH;                   // 6
localparam BRAM_DATA_WIDTH  = BUFF_DATA_WIDTH;                   // 32

////////////////////////////////////////////////////////////////////////

logic
mem_cen0,mem_cen1,mem_cen2,mem_cen3,mem_cen4,mem_cen5,mem_cen6,mem_cen7,mem_cen8,mem_cen9,mem_ce
n10,mem_cen11,mem_cen12,mem_cen13,mem_cen14,mem_cen15;
logic
mem_wen0,mem_wen1,mem_wen2,mem_wen3,mem_wen4,mem_wen5,mem_wen6,mem_wen7,mem_wen8,mem_wen9,mem_we
n10,mem_wen11,mem_wen12,mem_wen13,mem_wen14,mem_wen15;
logic                                                              [BUFF_ADDR_WIDTH-1:0]
mem_addr0,mem_addr1,mem_addr2,mem_addr3,mem_addr4,mem_addr5,mem_addr6,mem_addr7,mem_addr8,mem_ad
dr9,mem_addr10,mem_addr11,mem_addr12,mem_addr13,mem_addr14,mem_addr15;
logic                                                              [BUFF_DATA_WIDTH-1:0]
mem_dout0,mem_dout1,mem_dout2,mem_dout3,mem_dout4,mem_dout5,mem_dout6,mem_dout7,mem_dout8,mem_do
ut9,mem_dout10,mem_dout11,mem_dout12,mem_dout13,mem_dout14,mem_dout15;
logic [BUFF_ADDR_WIDTH-1:0]    mem_addr_ff;
logic                       mem_cen_ff, mem_wen_ff;

assign                 i_rd_data                 =                 (i_rd_valid)                ?
{mem_dout0,mem_dout1,mem_dout2,mem_dout3,mem_dout4,mem_dout5,mem_dout6,mem_dout7,mem_dout8,mem_d
out9,mem_dout10,mem_dout11,mem_dout12,mem_dout13,mem_dout14,mem_dout15} : 0;

////////////////////////////////////////////////////////////////////////
// Single Port SRAM
////////////////////////////////////////////////////////////////////////

// Host interface
always_ff @(posedge clk) begin
    mem_cen_ff <= mem_cen;
    mem_wen_ff <= mem_wen;
    mem_addr_ff <= mem_addr;
end
// Engine interface
always_ff @(posedge clk) begin
    if (i_rd_en) i_rd_valid <= 1;
    else i_rd_valid <= 0;
end

always_comb begin
    mem_dout = 0;
    mem_valid = 0;
    mem_cen0 = 0;
```

```
mem_cen1 = 0;
mem_cen2 = 0;
mem_cen3 = 0;
mem_cen4 = 0;
mem_cen5 = 0;
mem_cen6 = 0;
mem_cen7 = 0;
mem_cen8 = 0;
mem_cen9 = 0;
mem_cen10 = 0;
mem_cen11 = 0;
mem_cen12 = 0;
mem_cen13 = 0;
mem_cen14 = 0;
mem_cen15 = 0;
mem_wen0 = 0;
mem_wen1 = 0;
mem_wen2 = 0;
mem_wen3 = 0;
mem_wen4 = 0;
mem_wen5 = 0;
mem_wen6 = 0;
mem_wen7 = 0;
mem_wen8 = 0;
mem_wen9 = 0;
mem_wen10 = 0;
mem_wen11 = 0;
mem_wen12 = 0;
mem_wen13 = 0;
mem_wen14 = 0;
mem_wen15 = 0;
mem_addr0 = 0;
mem_addr1 = 0;
mem_addr2 = 0;
mem_addr3 = 0;
mem_addr4 = 0;
mem_addr5 = 0;
mem_addr6 = 0;
mem_addr7 = 0;
mem_addr8 = 0;
mem_addr9 = 0;
mem_addr10 = 0;
mem_addr11 = 0;
mem_addr12 = 0;
mem_addr13 = 0;
mem_addr14 = 0;
mem_addr15 = 0;

// mem_cen, mem_wen connection //
// Write in BRAM
if (mem_cen) begin                    // memory enable
    if (mem_addr[13:12] == 0) begin    // imem selected
        case (mem_addr[5:2])           // bank number
            0: begin
                mem_cen0 = 1;
                mem_wen0 = mem_wen;
                mem_addr0 = mem_addr[11:6];
            end
            1: begin
                mem_cen1 = 1;
                mem_wen1 = mem_wen;
                mem_addr1 = mem_addr[11:6];
            end
            2: begin
                mem_cen2 = 1;
                mem_wen2 = mem_wen;
                mem_addr2 = mem_addr[11:6];
            end
            3: begin
                mem_cen3 = 1;
                mem_wen3 = mem_wen;
                mem_addr3 = mem_addr[11:6];
            end
```

```
            4: begin
                mem_cen4 = 1;
                mem_wen4 = mem_wen;
                mem_addr4 = mem_addr[11:6];
            end
            5: begin
                mem_cen5 = 1;
                mem_wen5 = mem_wen;
                mem_addr5 = mem_addr[11:6];
            end
            6: begin
                mem_cen6 = 1;
                mem_wen6 = mem_wen;
                mem_addr6 = mem_addr[11:6];
            end
            7: begin
                mem_cen7 = 1;
                mem_wen7 = mem_wen;
                mem_addr7 = mem_addr[11:6];
            end
            8: begin
                mem_cen8 = 1;
                mem_wen8 = mem_wen;
                mem_addr8 = mem_addr[11:6];
            end
            9: begin
                mem_cen9 = 1;
                mem_wen9 = mem_wen;
                mem_addr9 = mem_addr[11:6];
            end
            10: begin
                mem_cen10 = 1;
                mem_wen10 = mem_wen;
                mem_addr10 = mem_addr[11:6];
            end
            11: begin
                mem_cen11 = 1;
                mem_wen11 = mem_wen;
                mem_addr11 = mem_addr[11:6];
            end
            12: begin
                mem_cen12 = 1;
                mem_wen12 = mem_wen;
                mem_addr12 = mem_addr[11:6];
            end
            13: begin
                mem_cen13 = 1;
                mem_wen13 = mem_wen;
                mem_addr13 = mem_addr[11:6];
            end
            14: begin
                mem_cen14 = 1;
                mem_wen14 = mem_wen;
                mem_addr14 = mem_addr[11:6];
            end
            15: begin
                mem_cen15 = 1;
                mem_wen15 = mem_wen;
                mem_addr15 = mem_addr[11:6];
            end
        endcase
    end
// Engine Interface: Read 16 vectors from imem
end else if (i_rd_en) begin
    mem_cen0 = 1;
    mem_cen1 = 1;
    mem_cen2 = 1;
    mem_cen3 = 1;
    mem_cen4 = 1;
    mem_cen5 = 1;
    mem_cen6 = 1;
    mem_cen7 = 1;
    mem_cen8 = 1;
```

```
    mem_cen9 = 1;
    mem_cen10 = 1;
    mem_cen11 = 1;
    mem_cen12 = 1;
    mem_cen13 = 1;
    mem_cen14 = 1;
    mem_cen15 = 1;
    mem_addr0 = i_rd_addr;
    mem_addr1 = i_rd_addr;
    mem_addr2 = i_rd_addr;
    mem_addr3 = i_rd_addr;
    mem_addr4 = i_rd_addr;
    mem_addr5 = i_rd_addr;
    mem_addr6 = i_rd_addr;
    mem_addr7 = i_rd_addr;
    mem_addr8 = i_rd_addr;
    mem_addr9 = i_rd_addr;
    mem_addr10 = i_rd_addr;
    mem_addr11 = i_rd_addr;
    mem_addr12 = i_rd_addr;
    mem_addr13 = i_rd_addr;
    mem_addr14 = i_rd_addr;
    mem_addr15 = i_rd_addr;
end

// mem_dout connection //
if (mem_cen_ff) begin                    // memory enable
    if (mem_addr_ff[13:12] == 0) begin     // imem selected
        if (mem_wen_ff == 0) begin         // read
            mem_valid = 1;
            case (mem_addr_ff[5:2])        // bank number
                0: begin
                    mem_dout = mem_dout0;
                end
                1: begin
                    mem_dout = mem_dout1;
                end
                2: begin
                    mem_dout = mem_dout2;
                end
                3: begin
                    mem_dout = mem_dout3;
                end
                4: begin
                    mem_dout = mem_dout4;
                end
                5: begin
                    mem_dout = mem_dout5;
                end
                6: begin
                    mem_dout = mem_dout6;
                end
                7: begin
                    mem_dout = mem_dout7;
                end
                8: begin
                    mem_dout = mem_dout8;
                end
                9: begin
                    mem_dout = mem_dout9;
                end
                10: begin
                    mem_dout = mem_dout10;
                end
                11: begin
                    mem_dout = mem_dout11;
                end
                12: begin
                    mem_dout = mem_dout12;
                end
                13: begin
                    mem_dout = mem_dout13;
                end
```

```
                14: begin
                    mem_dout = mem_dout14;
                end
                15: begin
                    mem_dout = mem_dout15;
                end
            endcase
        end
    end
end

// Instantiate BRAMs
blk_mem_gen_0
u_mem0_0
(
    .clka   ( clk    ),
    .ena    ( mem_cen0    ),
    .wea    ( mem_wen0    ),
    .addra  ( mem_addr0    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout0    )
);

blk_mem_gen_0
u_mem0_1
(
    .clka   ( clk    ),
    .ena    ( mem_cen1    ),
    .wea    ( mem_wen1    ),
    .addra  ( mem_addr1    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout1    )
);

blk_mem_gen_0
u_mem0_2
(
    .clka   ( clk    ),
    .ena    ( mem_cen2    ),
    .wea    ( mem_wen2    ),
    .addra  ( mem_addr2    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout2    )
);

blk_mem_gen_0
u_mem0_3
(
    .clka   ( clk    ),
    .ena    ( mem_cen3    ),
    .wea    ( mem_wen3    ),
    .addra  ( mem_addr3    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout3    )
);

blk_mem_gen_0
u_mem0_4
(
    .clka   ( clk    ),
    .ena    ( mem_cen4    ),
    .wea    ( mem_wen4    ),
    .addra  ( mem_addr4    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout4    )
);

blk_mem_gen_0
u_mem0_5
(
    .clka   ( clk    ),
    .ena    ( mem_cen5    ),
```

```
    .wea   ( mem_wen5   ),
    .addra  ( mem_addr5   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout5   )
);

blk_mem_gen_0
u_mem0_6
(
    .clka   ( clk   ),
    .ena    ( mem_cen6   ),
    .wea    ( mem_wen6   ),
    .addra  ( mem_addr6   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout6   )
);

blk_mem_gen_0
u_mem0_7
(
    .clka   ( clk   ),
    .ena    ( mem_cen7   ),
    .wea    ( mem_wen7   ),
    .addra  ( mem_addr7   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout7   )
);

blk_mem_gen_0
u_mem0_8
(
    .clka   ( clk   ),
    .ena    ( mem_cen8   ),
    .wea    ( mem_wen8   ),
    .addra  ( mem_addr8   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout8   )
);

blk_mem_gen_0
u_mem0_9
(
    .clka   ( clk   ),
    .ena    ( mem_cen9   ),
    .wea    ( mem_wen9   ),
    .addra  ( mem_addr9   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout9   )
);

blk_mem_gen_0
u_mem0_10
(
    .clka   ( clk   ),
    .ena    ( mem_cen10   ),
    .wea    ( mem_wen10   ),
    .addra  ( mem_addr10   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout10   )
);

blk_mem_gen_0
u_mem0_11
(
    .clka   ( clk   ),
    .ena    ( mem_cen11   ),
    .wea    ( mem_wen11   ),
    .addra  ( mem_addr11   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout11   )
);

blk_mem_gen_0
```

```
u_mem0_12
(
   .clka   ( clk    ),
   .ena    ( mem_cen12    ),
   .wea    ( mem_wen12    ),
   .addra  ( mem_addr12    ),
   .dina   ( mem_din    ),
   .douta  ( mem_dout12    )
);

blk_mem_gen_0
u_mem0_13
(
   .clka   ( clk    ),
   .ena    ( mem_cen13    ),
   .wea    ( mem_wen13    ),
   .addra  ( mem_addr13    ),
   .dina   ( mem_din    ),
   .douta  ( mem_dout13    )
);

blk_mem_gen_0
u_mem0_14
(
   .clka   ( clk    ),
   .ena    ( mem_cen14    ),
   .wea    ( mem_wen14    ),
   .addra  ( mem_addr14    ),
   .dina   ( mem_din    ),
   .douta  ( mem_dout14    )
);

blk_mem_gen_0
u_mem0_15
(
   .clka   ( clk    ),
   .ena    ( mem_cen15    ),
   .wea    ( mem_wen15    ),
   .addra  ( mem_addr15    ),
   .dina   ( mem_din    ),
   .douta  ( mem_dout15    )
);

endmodule
```

x. wmem

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename     : wmem.sv
// Author       : Castlab
//                                      Junsoo Kim   < junsoo999@kaist.ac.kr   >
// ---------------------------------------------------------------------
// Description: Weight data buffers for core
// -FHDR----------------------------------------------------------------

// mem_dout? 16 cycles or 1 cycle to read/write?
module wmem
  import kernel_pkg::*;
#(
  parameter WMEM_ADDR_WIDTH           = 10,
  parameter WMEM_DATA_WIDTH           = DATA_WIDTH * VECTOR_LENGTH
)
(
  input  logic                      clk,
  input  logic                      reset,

  // Memory interface
  input  logic                      mem_cen,
  input  logic                      mem_wen,
  input  logic [BUFF_ADDR_WIDTH-1:0]    mem_addr,
  input  logic [BUFF_DATA_WIDTH-1:0]    mem_din,
```

```
  output logic [BUFF_DATA_WIDTH-1:0]   mem_dout,
  output logic                         mem_valid,

  // Engine interface
  input  logic                         w_rd_en,
  input  logic [WMEM_ADDR_WIDTH-1:0]   w_rd_addr,
  output logic [WMEM_DATA_WIDTH-1:0]   w_rd_data,
  output logic                         w_rd_valid
);

////////////////////////////////////////////////////////////////////////
// mem_addr (14bit) = mem type (2bit) + real address (6 bit) + bank # (4bit) + byte (2bit)

localparam WMEM_BANK       = WMEM_DATA_WIDTH / BUFF_DATA_WIDTH;
localparam HIGH_ADDR       = WMEM_ADDR_WIDTH;
localparam LOW_ADDR        = $clog2(WMEM_BANK);
localparam BYTE_ADDR       = $clog2(BUFF_DATA_WIDTH / 8);

localparam BRAM_ADDR_WIDTH  = WMEM_ADDR_WIDTH;
localparam BRAM_DATA_WIDTH  = BUFF_DATA_WIDTH;

////////////////////////////////////////////////////////////////////////

logic
mem_cen0,mem_cen1,mem_cen2,mem_cen3,mem_cen4,mem_cen5,mem_cen6,mem_cen7,mem_cen8,mem_cen9,mem_ce
n10,mem_cen11,mem_cen12,mem_cen13,mem_cen14,mem_cen15;
logic
mem_wen0,mem_wen1,mem_wen2,mem_wen3,mem_wen4,mem_wen5,mem_wen6,mem_wen7,mem_wen8,mem_wen9,mem_we
n10,mem_wen11,mem_wen12,mem_wen13,mem_wen14,mem_wen15;
logic                                                                [BUFF_ADDR_WIDTH-1:0]
mem_addr0,mem_addr1,mem_addr2,mem_addr3,mem_addr4,mem_addr5,mem_addr6,mem_addr7,mem_addr8,mem_ad
dr9,mem_addr10,mem_addr11,mem_addr12,mem_addr13,mem_addr14,mem_addr15;
logic                                                                [BUFF_DATA_WIDTH-1:0]
mem_dout0,mem_dout1,mem_dout2,mem_dout3,mem_dout4,mem_dout5,mem_dout6,mem_dout7,mem_dout8,mem_do
ut9,mem_dout10,mem_dout11,mem_dout12,mem_dout13,mem_dout14,mem_dout15;
logic [BUFF_ADDR_WIDTH-1:0]   mem_addr_ff;
logic                       mem_cen_ff, mem_wen_ff;

assign                 w_rd_data                =                 (w_rd_valid)                ?
{mem_dout0,mem_dout1,mem_dout2,mem_dout3,mem_dout4,mem_dout5,mem_dout6,mem_dout7,mem_dout8,mem_d
out9,mem_dout10,mem_dout11,mem_dout12,mem_dout13,mem_dout14,mem_dout15} : 0;

////////////////////////////////////////////////////////////////////////
// Single Port SRAM
////////////////////////////////////////////////////////////////////////

// Host interface
always_ff @(posedge clk) begin
   mem_cen_ff <= mem_cen;
   mem_wen_ff <= mem_wen;
   mem_addr_ff <= mem_addr;
end
// Engine interface
always_ff @(posedge clk) begin
   if (w_rd_en) w_rd_valid <= 1;
   else w_rd_valid <= 0;
end

always_comb begin
   mem_dout = 0;
   mem_valid = 0;
   mem_cen0 = 0;
   mem_cen1 = 0;
   mem_cen2 = 0;
   mem_cen3 = 0;
   mem_cen4 = 0;
   mem_cen5 = 0;
   mem_cen6 = 0;
   mem_cen7 = 0;
   mem_cen8 = 0;
   mem_cen9 = 0;
   mem_cen10 = 0;
   mem_cen11 = 0;
```

```
mem_cen12 = 0;
mem_cen13 = 0;
mem_cen14 = 0;
mem_cen15 = 0;
mem_wen0 = 0;
mem_wen1 = 0;
mem_wen2 = 0;
mem_wen3 = 0;
mem_wen4 = 0;
mem_wen5 = 0;
mem_wen6 = 0;
mem_wen7 = 0;
mem_wen8 = 0;
mem_wen9 = 0;
mem_wen10 = 0;
mem_wen11 = 0;
mem_wen12 = 0;
mem_wen13 = 0;
mem_wen14 = 0;
mem_wen15 = 0;
mem_addr0 = 0;
mem_addr1 = 0;
mem_addr2 = 0;
mem_addr3 = 0;
mem_addr4 = 0;
mem_addr5 = 0;
mem_addr6 = 0;
mem_addr7 = 0;
mem_addr8 = 0;
mem_addr9 = 0;
mem_addr10 = 0;
mem_addr11 = 0;
mem_addr12 = 0;
mem_addr13 = 0;
mem_addr14 = 0;
mem_addr15 = 0;

// mem_cen, mem_wen connection //
// Write in BRAM
if (mem_cen) begin                    // memory enable
    if (mem_addr[13:12] == 1) begin     // wmem selected
        case (mem_addr[5:2])          // bank number
            0: begin
                mem_cen0 = 1;
                mem_wen0 = mem_wen;
                mem_addr0 = mem_addr[11:6];
            end
            1: begin
                mem_cen1 = 1;
                mem_wen1 = mem_wen;
                mem_addr1 = mem_addr[11:6];
            end
            2: begin
                mem_cen2 = 1;
                mem_wen2 = mem_wen;
                mem_addr2 = mem_addr[11:6];
            end
            3: begin
                mem_cen3 = 1;
                mem_wen3 = mem_wen;
                mem_addr3 = mem_addr[11:6];
            end
            4: begin
                mem_cen4 = 1;
                mem_wen4 = mem_wen;
                mem_addr4 = mem_addr[11:6];
            end
            5: begin
                mem_cen5 = 1;
                mem_wen5 = mem_wen;
                mem_addr5 = mem_addr[11:6];
            end
            6: begin
```

```
                mem_cen6 = 1;
                mem_wen6 = mem_wen;
                mem_addr6 = mem_addr[11:6];
            end
            7: begin
                mem_cen7 = 1;
                mem_wen7 = mem_wen;
                mem_addr7 = mem_addr[11:6];
            end
            8: begin
                mem_cen8 = 1;
                mem_wen8 = mem_wen;
                mem_addr8 = mem_addr[11:6];
            end
            9: begin
                mem_cen9 = 1;
                mem_wen9 = mem_wen;
                mem_addr9 = mem_addr[11:6];
            end
            10: begin
                mem_cen10 = 1;
                mem_wen10 = mem_wen;
                mem_addr10 = mem_addr[11:6];
            end
            11: begin
                mem_cen11 = 1;
                mem_wen11 = mem_wen;
                mem_addr11 = mem_addr[11:6];
            end
            12: begin
                mem_cen12 = 1;
                mem_wen12 = mem_wen;
                mem_addr12 = mem_addr[11:6];
            end
            13: begin
                mem_cen13 = 1;
                mem_wen13 = mem_wen;
                mem_addr13 = mem_addr[11:6];
            end
            14: begin
                mem_cen14 = 1;
                mem_wen14 = mem_wen;
                mem_addr14 = mem_addr[11:6];
            end
            15: begin
                mem_cen15 = 1;
                mem_wen15 = mem_wen;
                mem_addr15 = mem_addr[11:6];
            end
        endcase
    end
// Engine Interface: Read 16 vectors from wmem
end else if (w_rd_en) begin
    mem_cen0 = 1;
    mem_cen1 = 1;
    mem_cen2 = 1;
    mem_cen3 = 1;
    mem_cen4 = 1;
    mem_cen5 = 1;
    mem_cen6 = 1;
    mem_cen7 = 1;
    mem_cen8 = 1;
    mem_cen9 = 1;
    mem_cen10 = 1;
    mem_cen11 = 1;
    mem_cen12 = 1;
    mem_cen13 = 1;
    mem_cen14 = 1;
    mem_cen15 = 1;
    mem_addr0 = w_rd_addr;
    mem_addr1 = w_rd_addr;
    mem_addr2 = w_rd_addr;
    mem_addr3 = w_rd_addr;
```

```
        mem_addr4 = w_rd_addr;
        mem_addr5 = w_rd_addr;
        mem_addr6 = w_rd_addr;
        mem_addr7 = w_rd_addr;
        mem_addr8 = w_rd_addr;
        mem_addr9 = w_rd_addr;
        mem_addr10 = w_rd_addr;
        mem_addr11 = w_rd_addr;
        mem_addr12 = w_rd_addr;
        mem_addr13 = w_rd_addr;
        mem_addr14 = w_rd_addr;
        mem_addr15 = w_rd_addr;
    end

    // mem_dout connection //
    if (mem_cen_ff) begin                    // memory enable
        if (mem_addr_ff[13:12] == 1) begin     // wmem selected
            if (mem_wen_ff == 0) begin         // read
                mem_valid = 1;
                case (mem_addr_ff[5:2])        // bank number
                    0: begin
                        mem_dout = mem_dout0;
                    end
                    1: begin
                        mem_dout = mem_dout1;
                    end
                    2: begin
                        mem_dout = mem_dout2;
                    end
                    3: begin
                        mem_dout = mem_dout3;
                    end
                    4: begin
                        mem_dout = mem_dout4;
                    end
                    5: begin
                        mem_dout = mem_dout5;
                    end
                    6: begin
                        mem_dout = mem_dout6;
                    end
                    7: begin
                        mem_dout = mem_dout7;
                    end
                    8: begin
                        mem_dout = mem_dout8;
                    end
                    9: begin
                        mem_dout = mem_dout9;
                    end
                    10: begin
                        mem_dout = mem_dout10;
                    end
                    11: begin
                        mem_dout = mem_dout11;
                    end
                    12: begin
                        mem_dout = mem_dout12;
                    end
                    13: begin
                        mem_dout = mem_dout13;
                    end
                    14: begin
                        mem_dout = mem_dout14;
                    end
                    15: begin
                        mem_dout = mem_dout15;
                    end
                endcase
            end
        end
    end
end
```

```
// Instantiate BRAMs
blk_mem_gen_1
u_mem1_0
(
    .clka   ( clk    ),
    .ena    ( mem_cen0    ),
    .wea    ( mem_wen0    ),
    .addra  ( mem_addr0    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout0    )
);

blk_mem_gen_1
u_mem1_1
(
    .clka   ( clk    ),
    .ena    ( mem_cen1    ),
    .wea    ( mem_wen1    ),
    .addra  ( mem_addr1    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout1    )
);

blk_mem_gen_1
u_mem1_2
(
    .clka   ( clk    ),
    .ena    ( mem_cen2    ),
    .wea    ( mem_wen2    ),
    .addra  ( mem_addr2    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout2    )
);

blk_mem_gen_1
u_mem1_3
(
    .clka   ( clk    ),
    .ena    ( mem_cen3    ),
    .wea    ( mem_wen3    ),
    .addra  ( mem_addr3    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout3    )
);

blk_mem_gen_1
u_mem1_4
(
    .clka   ( clk    ),
    .ena    ( mem_cen4    ),
    .wea    ( mem_wen4    ),
    .addra  ( mem_addr4    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout4    )
);

blk_mem_gen_1
u_mem1_5
(
    .clka   ( clk    ),
    .ena    ( mem_cen5    ),
    .wea    ( mem_wen5    ),
    .addra  ( mem_addr5    ),
    .dina   ( mem_din    ),
    .douta  ( mem_dout5    )
);

blk_mem_gen_1
u_mem1_6
(
    .clka   ( clk    ),
    .ena    ( mem_cen6    ),
```

```
    .wea   ( mem_wen6   ),
    .addra  ( mem_addr6   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout6   )
);

blk_mem_gen_1
u_mem1_7
(
    .clka   ( clk   ),
    .ena    ( mem_cen7   ),
    .wea    ( mem_wen7   ),
    .addra  ( mem_addr7   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout7   )
);

blk_mem_gen_1
u_mem1_8
(
    .clka   ( clk   ),
    .ena    ( mem_cen8   ),
    .wea    ( mem_wen8   ),
    .addra  ( mem_addr8   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout8   )
);

blk_mem_gen_1
u_mem1_9
(
    .clka   ( clk   ),
    .ena    ( mem_cen9   ),
    .wea    ( mem_wen9   ),
    .addra  ( mem_addr9   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout9   )
);

blk_mem_gen_1
u_mem1_10
(
    .clka   ( clk   ),
    .ena    ( mem_cen10   ),
    .wea    ( mem_wen10   ),
    .addra  ( mem_addr10   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout10   )
);

blk_mem_gen_1
u_mem1_11
(
    .clka   ( clk   ),
    .ena    ( mem_cen11   ),
    .wea    ( mem_wen11   ),
    .addra  ( mem_addr11   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout11   )
);

blk_mem_gen_1
u_mem1_12
(
    .clka   ( clk   ),
    .ena    ( mem_cen12   ),
    .wea    ( mem_wen12   ),
    .addra  ( mem_addr12   ),
    .dina   ( mem_din   ),
    .douta  ( mem_dout12   )
);

blk_mem_gen_1
```

```
u_mem1_13
(
   .clka   ( clk    ),
   .ena    ( mem_cen13    ),
   .wea    ( mem_wen13    ),
   .addra  ( mem_addr13    ),
   .dina   ( mem_din    ),
   .douta  ( mem_dout13    )
);

blk_mem_gen_1
u_mem1_14
(
   .clka   ( clk    ),
   .ena    ( mem_cen14    ),
   .wea    ( mem_wen14    ),
   .addra  ( mem_addr14    ),
   .dina   ( mem_din    ),
   .douta  ( mem_dout14    )
);

blk_mem_gen_1
u_mem1_15
(
   .clka   ( clk    ),
   .ena    ( mem_cen15    ),
   .wea    ( mem_wen15    ),
   .addra  ( mem_addr15    ),
   .dina   ( mem_din    ),
   .douta  ( mem_dout15    )
);

endmodule
```

xi. omem

```
// +FHDR----------------------------------------------------------------
//               Copyright (c) 2022 .
//                    ALL RIGHTS RESERVED
// ---------------------------------------------------------------------
// Filename     : omem.sv
// Author       : Castlab
//                                      Junsoo Kim   < junsoo999@kaist.ac.kr   >
// ---------------------------------------------------------------------
// Description: Output data buffers for core
// -FHDR----------------------------------------------------------------

module omem
  import kernel_pkg::*;
#(
  parameter OMEM_ADDR_WIDTH           = 10,
  parameter OMEM_DATA_WIDTH           = DATA_WIDTH
)
(
  input  logic                       clk,
  input  logic                       reset,

  // Memory interface
  input  logic                       mem_cen,
  input  logic                       mem_wen,
  input  logic [BUFF_ADDR_WIDTH-1:0]    mem_addr,   // 14
  input  logic [BUFF_DATA_WIDTH-1:0]    mem_din,    // 32
  output logic [BUFF_DATA_WIDTH-1:0]    mem_dout,   // 32
  output logic                       mem_valid,

  // Core interface
  input  logic                       o_wr_en,
  input  logic [OMEM_ADDR_WIDTH-1:0]    o_wr_addr,  // 10
  input  logic [OMEM_DATA_WIDTH-1:0]    o_wr_data   // 32
);

////////////////////////////////////////////////////////////////////////
// mem_addr (14bit) = mem type (2bit) + real address (10 bit) + byte (2bit)
```

```
localparam OMEM_BANK       = OMEM_DATA_WIDTH / BUFF_DATA_WIDTH;    // 1
localparam HIGH_ADDR       = OMEM_ADDR_WIDTH;                    // 10
localparam LOW_ADDR        = $clog2(OMEM_BANK);                  // 0
localparam BYTE_ADDR       = $clog2(BUFF_DATA_WIDTH / 8);         // 2

localparam BRAM_ADDR_WIDTH  = OMEM_ADDR_WIDTH;                    // 10
localparam BRAM_DATA_WIDTH  = BUFF_DATA_WIDTH;                    // 32

////////////////////////////////////////////////////////////////////////

logic                       mem_cen0;
logic                       mem_wen0;
logic [OMEM_ADDR_WIDTH-1:0]    mem_addr0;
logic [OMEM_DATA_WIDTH-1:0]    mem_din0;

////////////////////////////////////////////////////////////////////////
// Single Port SRAM
////////////////////////////////////////////////////////////////////////

always_ff @(posedge clk) begin
    if (mem_cen) begin
        if (mem_addr[13:12] == 2) begin
            if(|mem_wen) begin
                mem_valid <= 0;
            end
            else begin
                mem_valid <= 1;
            end
        end
    end else begin
        mem_valid <= 0;
    end
end

always_comb begin
    mem_cen0 = 0;
    mem_wen0 = 0;
    mem_addr0 = 0;
    mem_din0 = 0;

    // Memory interface
    if (mem_cen) begin                    // memory enable
        if (mem_addr[13:12] == 2) begin     // omem selected
            mem_cen0 = 1;
            mem_wen0 = mem_wen;
            mem_addr0 = mem_addr[11:2];
            mem_din0 = mem_din;
        end
    // Engine Interface: Write operation result to omem
    end else if (o_wr_en) begin
        mem_cen0 = 1;
        mem_wen0 = 1;
        mem_addr0 = o_wr_addr;
        mem_din0 = o_wr_data;
    end
end

// Instantiate BRAM
blk_mem_gen_2
u_mem2
(
    .clka   ( clk    ),
    .ena    ( mem_cen0    ),
    .wea    ( mem_wen0    ),
    .addra  ( mem_addr0    ),
    .dina   ( mem_din0    ),
    .douta  ( mem_dout    )
);
////////////////////////////////////////////////////////////////////////

endmodule
```