

Lecture 10: MAC, CONV kernel

Xuan-Truong Nguyen



Road map

Review

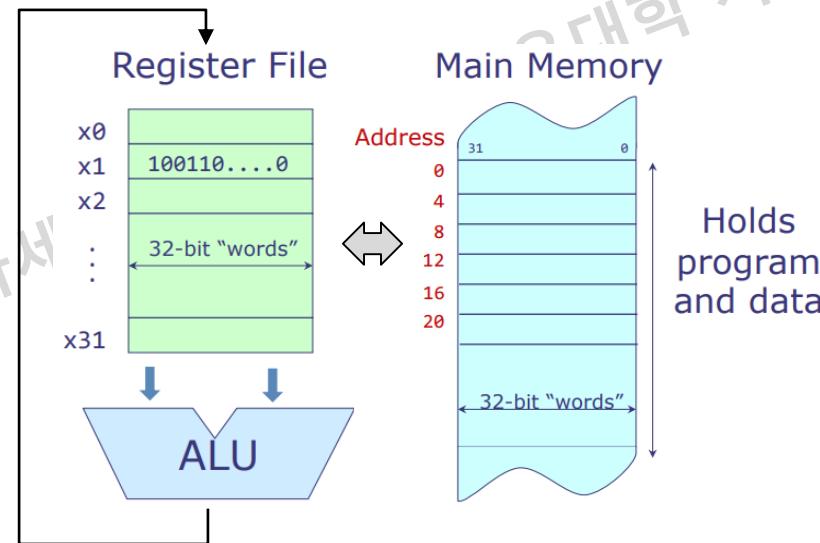
MAC

Mac kernel
(Accumulation)

Convolution kernel
(Normalization
& Activation)

Performance issue in a CPU

- A PC system consists of Processor (CPU), i.e. RISC-V, Memory, and IO
 - General-purpose: Execute any program
- Performance: Each single-core PC has only **one computation unit**, i.e. ALU, multiplier.
 - Example: bright adjustment
 - For each pixel in an image for display, increase or decrease its value by a number.
⇒ Complexity = total pixels in an image, i.e. 1,179,648 (= $768 \times 512 \times 3$)
 - How to **boost the performance?**

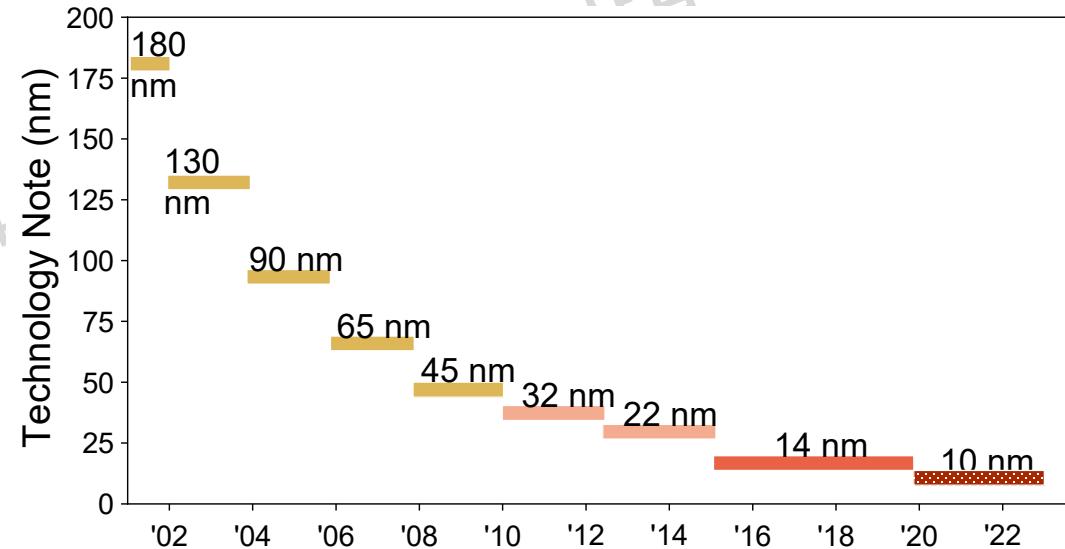


How to boost the performance?

- Moore's Law: Every two years, double the number of transistors
 - Build **higher performance general-purpose processors**
 - Make the transistors available to the masses
 - Increase performance ($1.8 \times \uparrow$)
 - Lower the cost of computing ($1.8 \times \downarrow$)

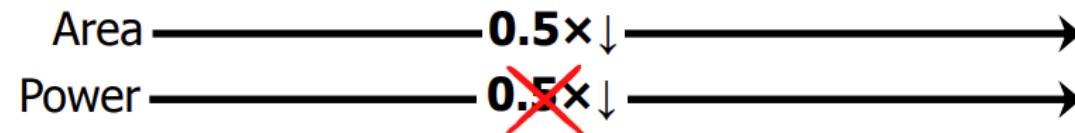


Gordon Moore

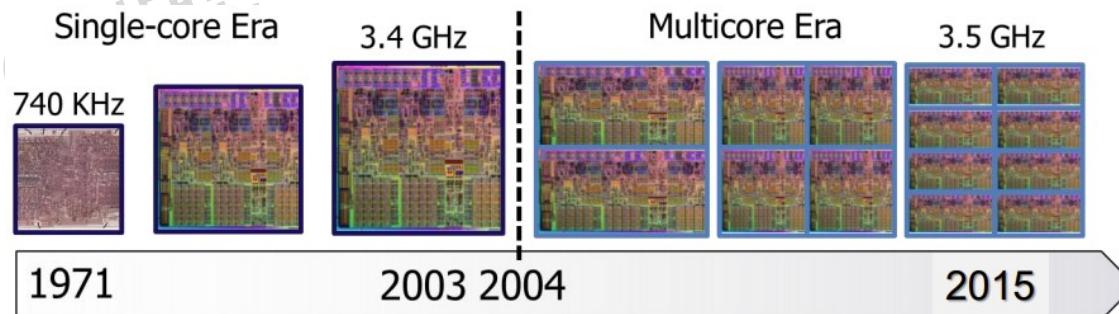


How to boost the performance?

- Power issue
 - Dark silicon – the fraction of transistors that need to be powered off at all times (due to power + thermal constraints)

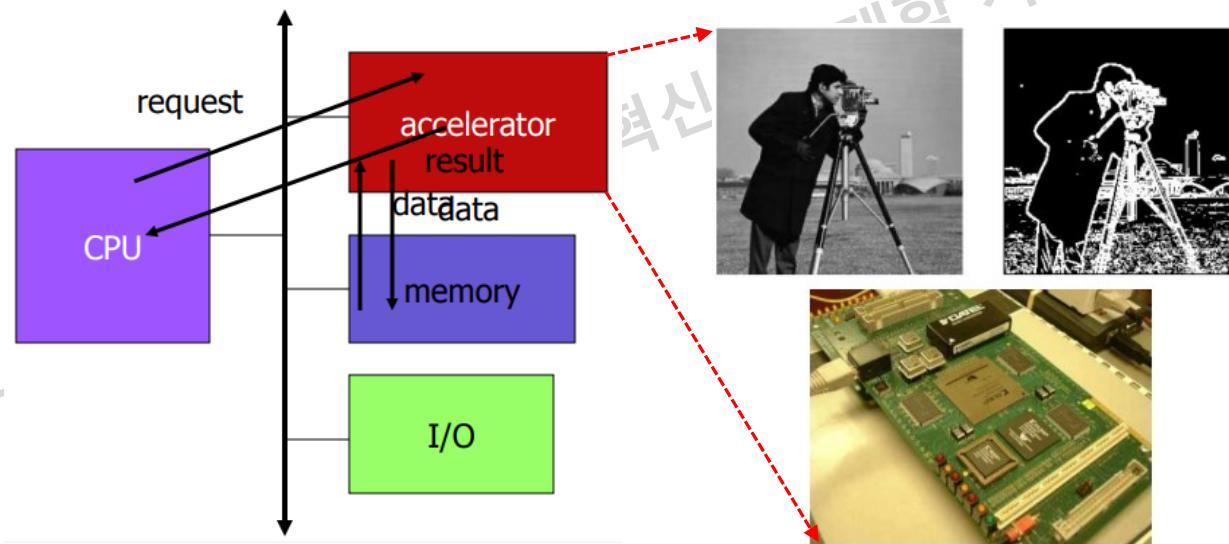


- Evolution of processors
 - From single core to multi-core
 - Post PC: Expected to continue evolution towards HW specialization/acceleration



Accelerators

- Accelerator: use additional computational units dedicated to some functionality
 - A H/W module for special applications
- Integration: An accelerator appears as a device on the bus
 - Typically controlled by registers (memory-mapped IO)
- Performance: When we talk about "**HW Accelerator**", we talk about **Efficiency**
 - Not just about "correctness", but how "GOOD" it is.



Deep Neural Networks

- Artificial neural networks try to mimic human brain's behaviors
- Deep learning starts to surpass human-level recognition on specific tasks.
- Deep neural networks:
 - Use multiple layers: Input neurons, weights, output neurons
 - Trainable/Learnable features and classifiers
 - Performance dramatically increases when the amount of data increases.

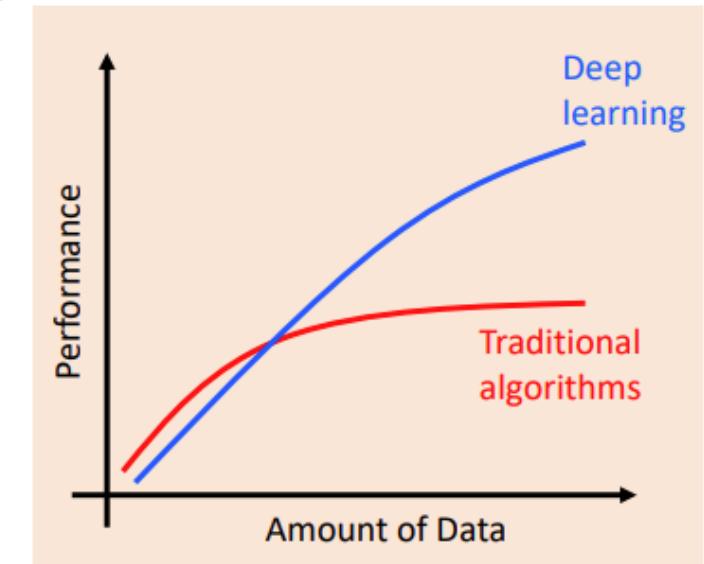
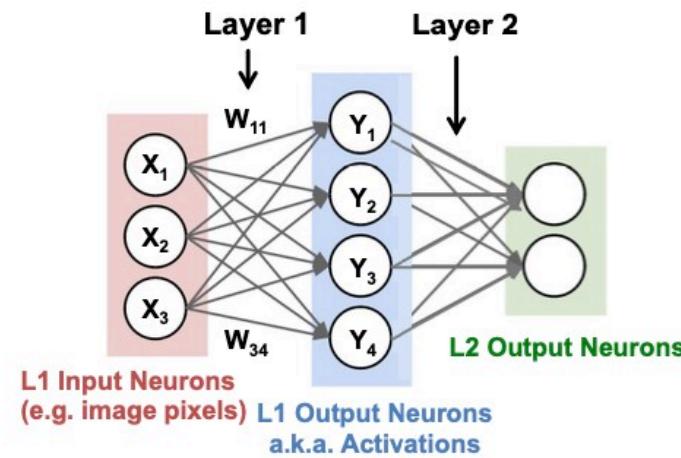
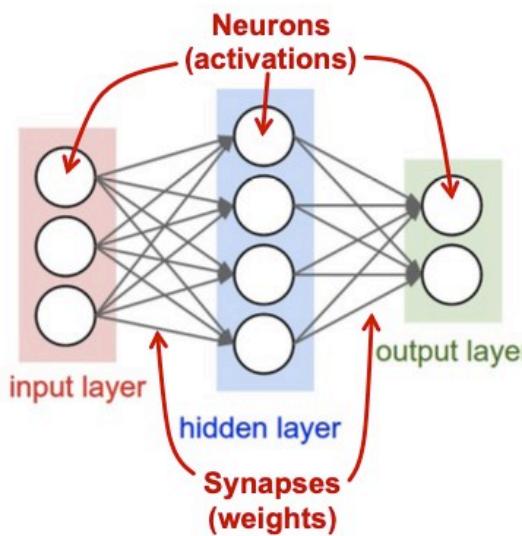
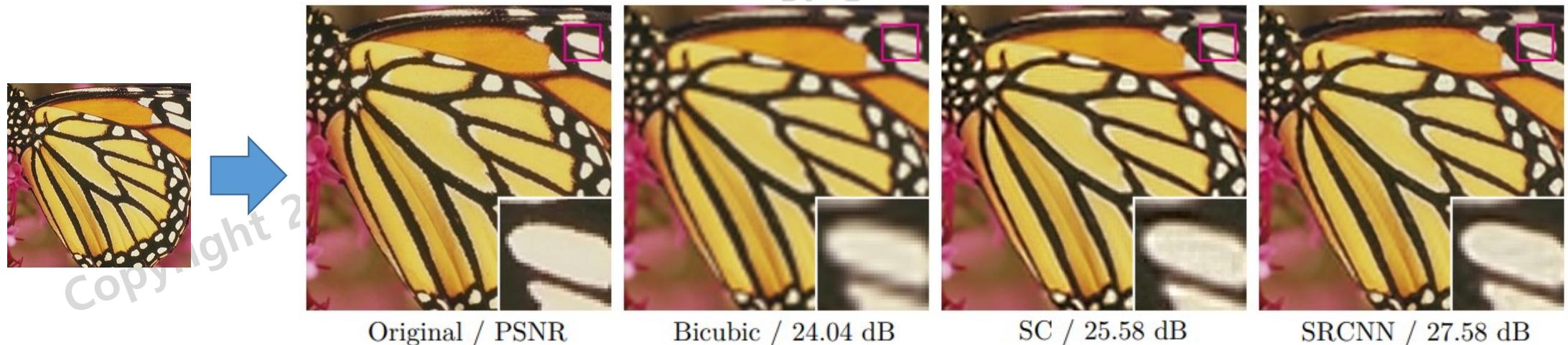


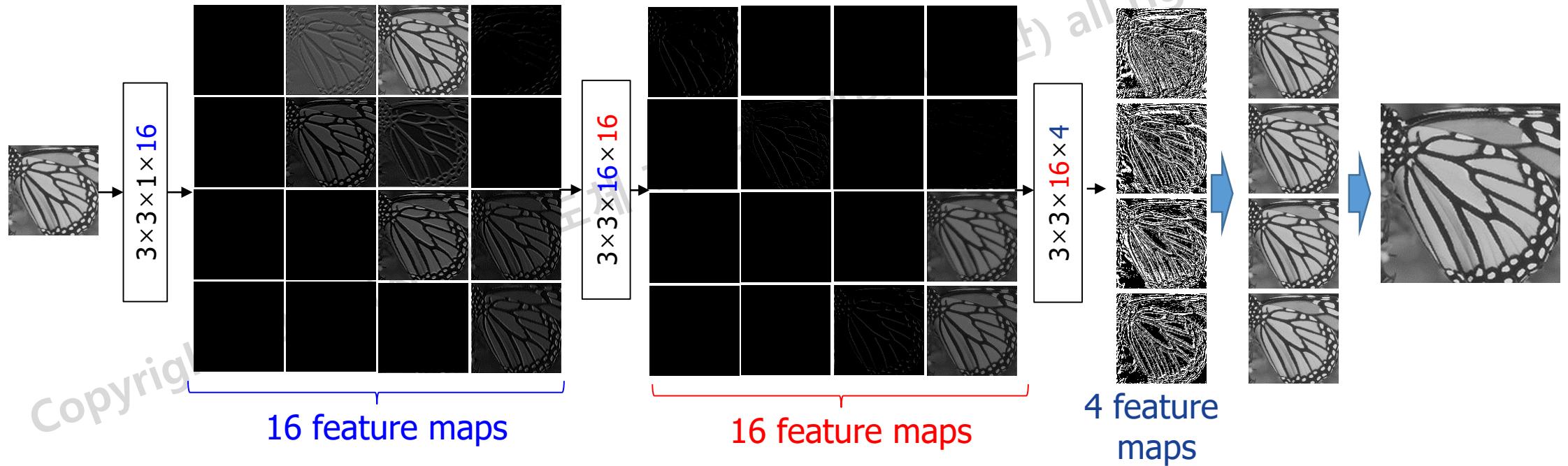
Image super-resolution (SR)

- Super-resolution (SR): generate a high-resolution image from a low-resolution one.
 - Estimating the inverse degradation function is an **ill-posed** problem → Many solutions
 - Methods
 - Interpolation-based methods: Simple, Fast, Low quality.
 - Reconstruction-based methods: Slow, moderate quality
 - (Deep) Learning-based methods: Complex, extremely high quality
- ⇒ Can be sped up by using H/W accelerators.



Sim-ESPCN (hw_uniform_architecture.m)

- Three CONV layers:
 - Layer 1: one input image → 16 output feature maps
 - Layer 2: 16 input feature maps → 16 output feature maps.
 - Layer 3: 16 input feature maps → 4 output feature maps.



Sim-ESPCN (hw_uniform_architecture.m)

- The number of multiplication operations (num_ops) to calculate **ONE output pixel**
 - Layer 1: 9 ($=3 \times 3 \times 1$), Layer 2 and 3: 144 ($= 3 \times 3 \times 16$).

Layer	Filter size	No. of input channels	No. of output channels	Input	Output	num_ops
1	3x3	1	16	128x128x1	128x128x16	9=(3x3x1)
2	3x3	16	16	128x128x16	128x128x16	144=(3x3x16)
3	3x3	16	4	128x128x16	128x128x4	144=(3x3x16)

- Verify those numbers by adding a debugging point in **convol2.m**
 - Size of kernel(:,:,k).

```
% Convolution
for k = 1:c_out % Number of output channels
    for i = 1:h_out % Row
        for j = 1:w_out % Column
            % Element-wise Multiplication => Products
            scalar = kernel(:,:,k).*...
                pad_img(1+(i-1)*s:1+(i-1)*s+f-1, 1+(j-1)*s:1+(j-1)*s+f-1, :);
            % Sum (Sum of Products)
            out(i,j,k) = sum(scalar(:));
        end
    end
end
```

Weight quantization

- Convert a FP tensor into a fixed-point format
- Example: 8-bit symmetric representation
 - 255/2, -254/2, ..., -1/2, 1/2, ..., 255/2

```
w1(:,:,1,1) =  
  
-0.3302   -0.3016   -0.1170  
0.3708   -0.2676   -0.1463  
-0.0760   -0.0007   0.3297
```

Floating point weight



```
wq1(:,:,1,1) =  
  
-169   -155   -59  
189   -137   -75  
-39    -1     169
```

quantized weight
ibits = 0, fbits = 8

```
wq1_store(:,:,1,1) =  
  
-85    -78    -30  
94     -69    -38  
-20     -1     84
```

stored weights
8 bits

energy reduction
for

- Memory load
- Calculations

```
% Function: Symetric Uniform quantization  
% Inputs  
% - x:      input tensor  
% - step: quantized step  
% - nbit: number of bits in fixed-point representation  
% Outputs  
% - output: Quantized value mapping to x  
% - output_store: Store value  
  
function [output, output_store] = uniform_quantize(x,step,nbit)  
% Initialization  
output = x;  
output_store = x;  
  
% Maximum and minimum ranges  
pos_end = 2 ^ nbit - 1;  
neg_end = -pos_end;  
  
% Quantized value  
output = 2 * round(x./step + 0.5) - 1;  
output(output > pos_end) = pos_end;  
output(output < neg_end) = neg_end;  
  
output_store = (output - 1) / 2;
```

Activation quantization (ReLU/Linear)

- Convert a FP tensor into a fixed-point format
 - Similar to weight/bias quantization
 - Normalize ranges of biases and convolution outputs

```
% Function: Activation quantization
% Inputs
% - x:      input tensor
% - step:   quantized step
% - nbit:   number of bits in fixed-point representation
% - biases_shift
% Outputs
% - activations: Quantized value mapping to x
% - activations_store: Store value

function [activations, activations_store] = hwu_relu_quantize(x, step, nbit, biases_shift)
% Initialization
activations = x;
activations_store = x;

% Maximum quantized range
pos_end = 2 ^ nbit - 1;

% Quantization
activations_store = x;
activations_store(x >= 0) = round(x(x >= 0)/(2^biases_shift*(step)));
activations_store(x < 0) = 0;

% ReLU
activations_store(activations_store > pos_end) = pos_end;
activations = activations_store;
```

```
% Function: Activation quantization
% Inputs
% - x:      input tensor
% - step:   quantized step
% - nbit:   number of bits in fixed-point representation
% - biases_shift
% Outputs
% - activations: Quantized value mapping to x
% - activations_store: Store value
function [activations, activations_store] = hwu_linear_quantize(x, step, nbit, biases_shift)
% Initialization
activations = x;
activations_store = x;

% Maximum and minimum ranges
pos_end = 2 ^ (nbit-1) - 1;
neg_end = -pos_end - 1;

% Quantization
activations_store = round(x/(2^biases_shift*(step)));

% Linear activation
activations_store(activations_store > pos_end) = pos_end;
activations_store(activations_store < neg_end) = neg_end;

% Outputs for buffering/storing into memory
activations = activations_store;
```

Lecture plan

- Lab 1: DSP and MAC
 - Multipliers and an adder tree
 - MAC
- Lab 2: MAC kernel
 - Accumulation
- Lab 3: Convolutional kernel
 - Scaling/normalization
 - Activation and quantization

Road map

Review

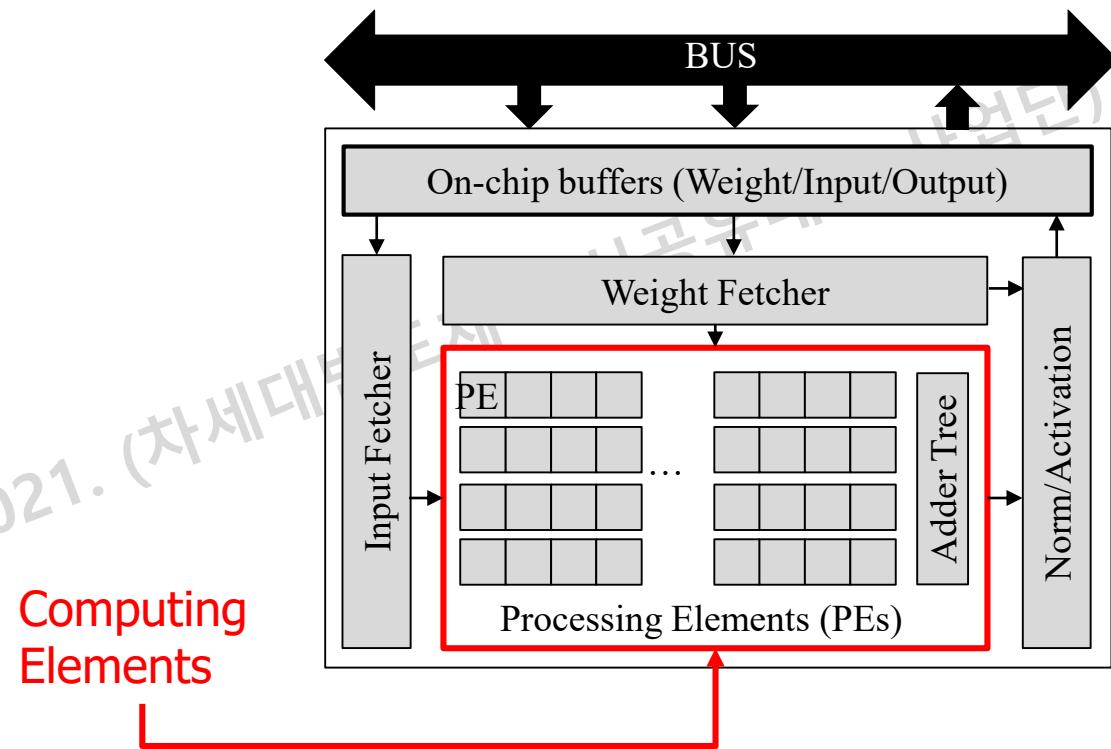
MAC

Mac kernel
(Accumulation)

Convolution kernel
(Normalization
& Activation)

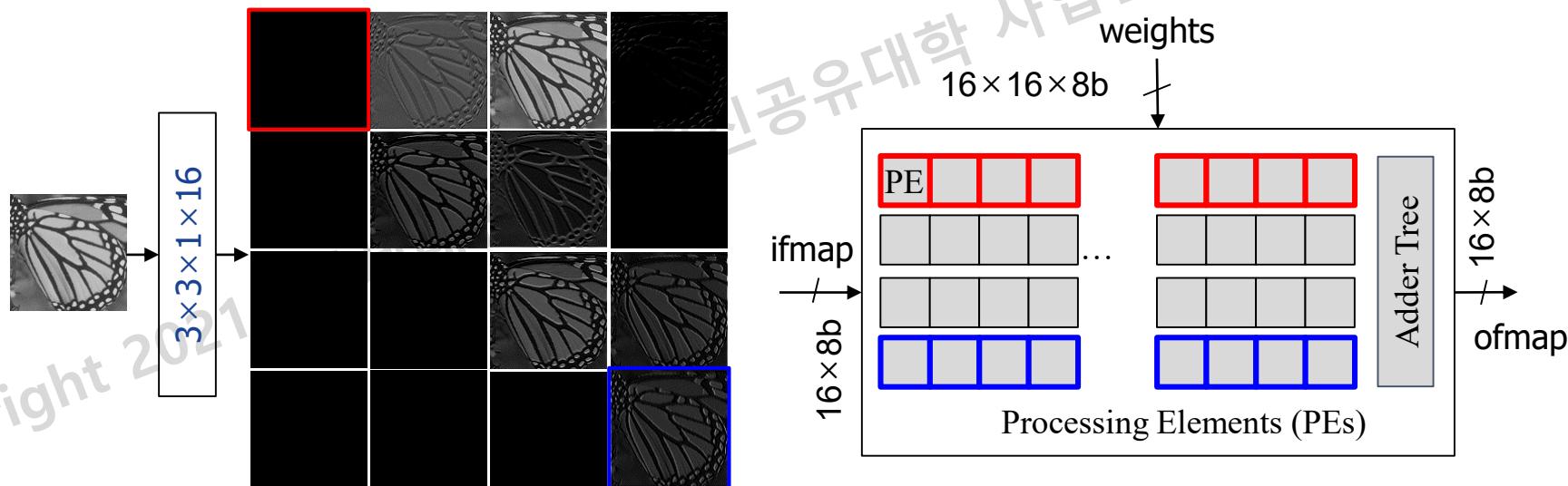
DNN accelerator

- Processing Element (PE) Array
 - An array of multiplication and accumulation (MAC).
 - Perform convolution operations.
 - Computing unit or "ALU" of a DNN accelerator



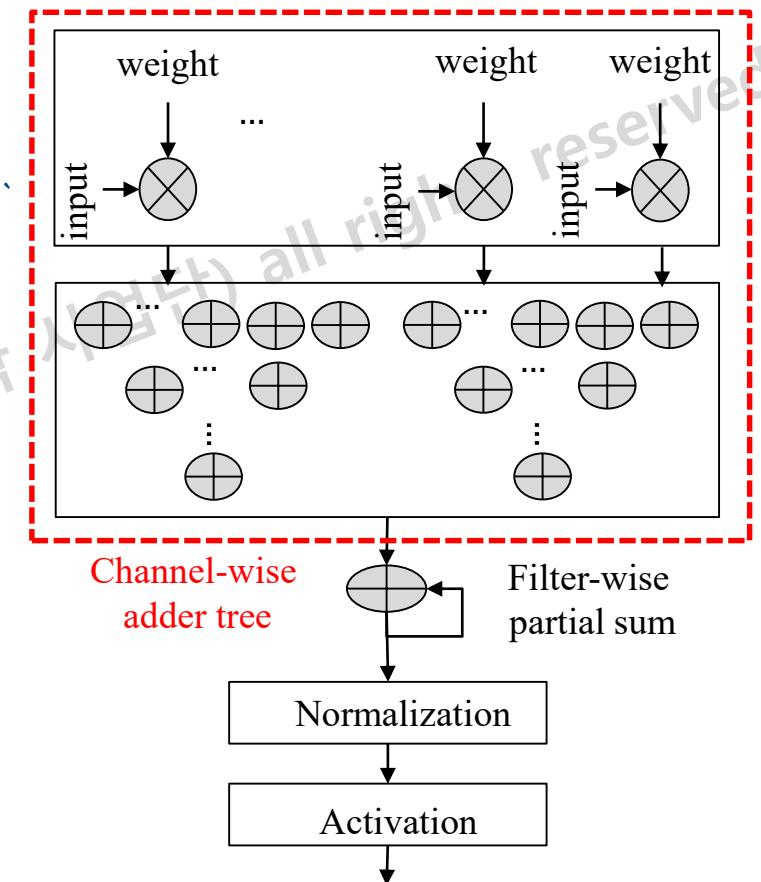
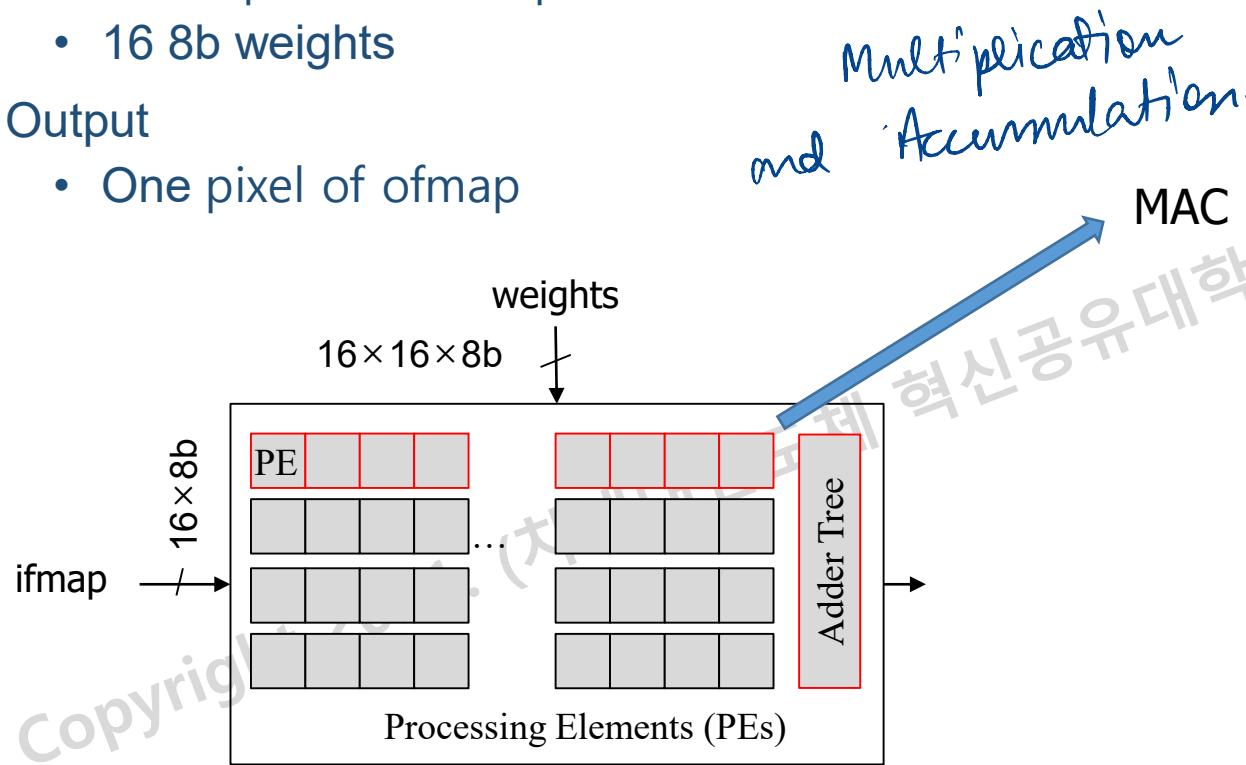
Mapping

- Convolution: $C = \# \text{ of input channels}$, $K = \# \text{ of output channels}$
- 2D Computing Unit
 - PEs: same inputs, different weights => one output pixel
 - Compute multiple pixels in output feature maps
 - RED** PEs → First output feature map
 - BLUE** PEs → Last output feature map



Processing Element

- Compute one output pixel
- Inputs
 - 16 8b pixels of ifmap
 - 16 8b weights
- Output
 - One pixel of ofmap



Some of Product (SOP)

- The sum of products is a **common operation** in DNNs.
 - Compute a sum of N products:

$$Y = \sum_{i=0}^{N-1} w_i * x_i$$

weights
Inputs.

$\Rightarrow N-1$ addition operations, N multiplication operations

- Example $N = 16$

Inputs	0	0	0	71	0	12	193	7	0	74	14	198	0	0	0	0
Weights	-35	-1	167	99	37	21	-7	-1	41	103	-11	85	-51	7	-9	57

Parallel Computing

- Compute a sum of N products

$$Y = \sum_{i=0}^{15} w_i * x_i$$

- Pseudo code

$$y_1^{(0)} = w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15}$$

// N multipliers

$$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \dots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$$

// N/2 adders

$$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \dots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$$

// N/4 adders

$$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \dots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$$

// N/4 adders

$$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$$

// N/4 adders

$$Y = y_1^{(4)}$$

// Output

Parallel Computing: Multipliers

- Compute a sum of N products

- Pseudo code

$$y_1^{(0)} = w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15}$$

$$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \dots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$$

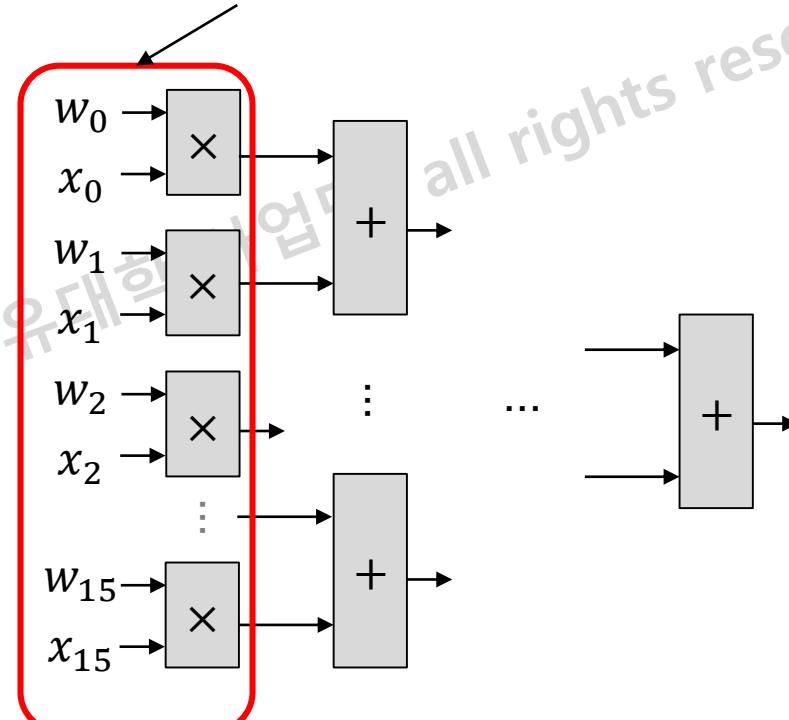
$$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \dots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$$

$$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \dots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$$

$$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$$

$$Y = y_1^{(4)}$$

$$Y = \sum_{i=0}^{15} w_i * x_i \quad N \text{ block/module of multipliers}$$



16 Multiplications

Parallel Computing: Adder tree (level=1)

- Compute a sum of N products

- Pseudo code

$$y_1^{(0)} = w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15}$$

$$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \dots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$$

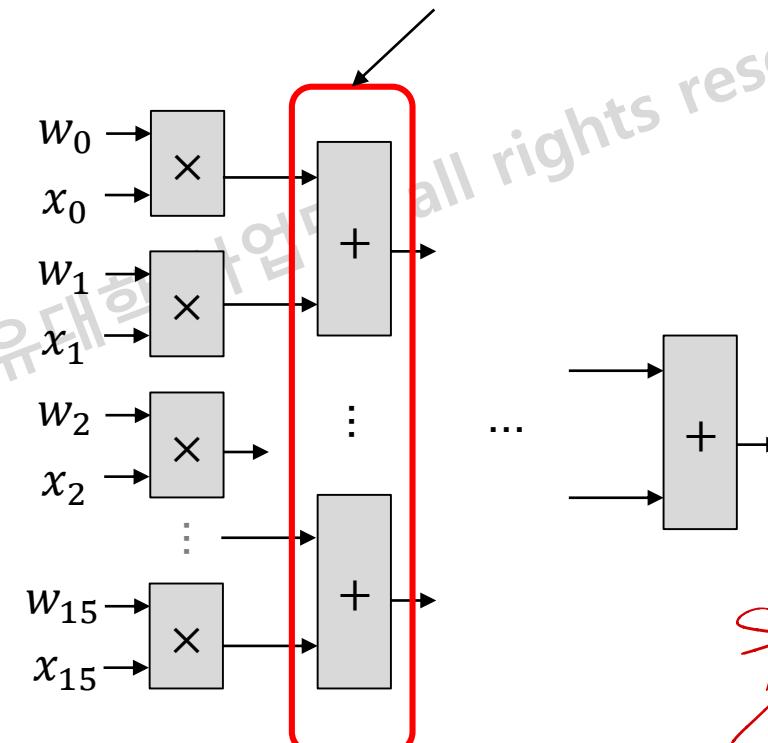
$$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \dots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$$

$$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \dots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$$

$$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$$

$$Y = y_1^{(4)}$$

$$Y = \sum_{i=0}^{15} w_i * x_i \quad N/2 \text{ block/module of adders}$$



$$S_2 = \frac{n}{2} - \frac{1}{2}$$

8 additions

$$S = n - 1$$

~~$$S = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + \frac{n}{2^k}$$~~

Parallel Computing: Adder tree (level=2)

- Compute a sum of N products

- Pseudo code

$$y_1^{(0)} = w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15}$$

$$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \dots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$$

$$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \dots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$$

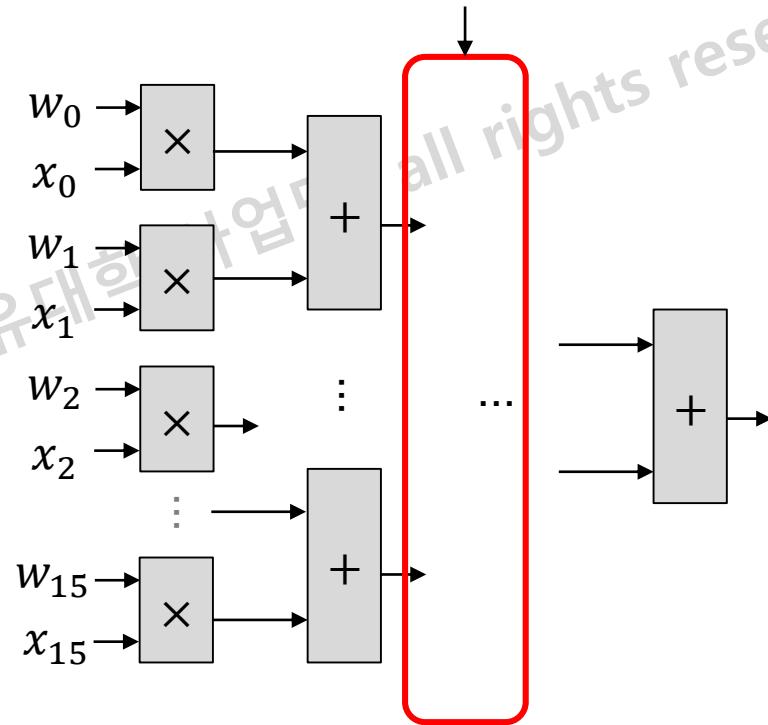
$$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \dots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$$

$$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$$

$$Y = y_1^{(4)}$$

$$Y = \sum_{i=0}^{15} w_i * x_i$$

N/4 block/module of adders



Parallel Computing: Adder tree (level=4)

- Compute a sum of N products

$$Y = \sum_{i=0}^{15} w_i * x_i$$

- Pseudo code

$$y_1^{(0)} = w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15}$$

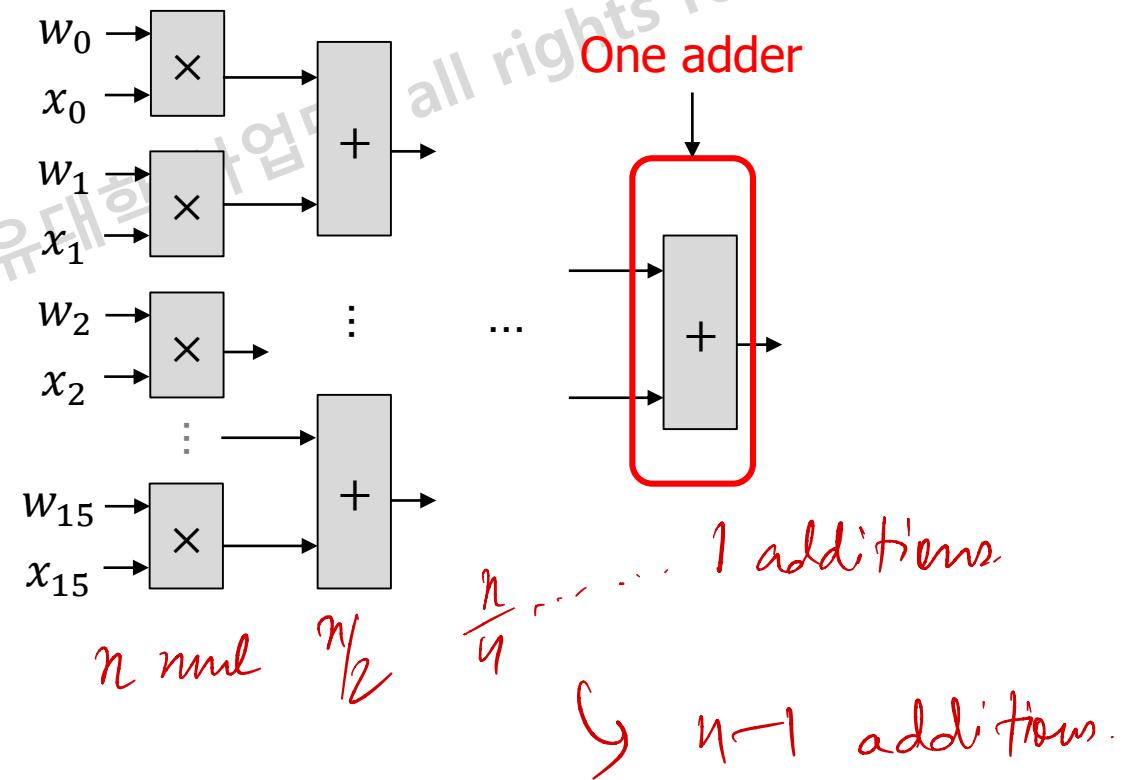
$$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \dots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$$

$$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \dots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$$

$$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \dots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$$

$$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$$

$$Y = y_1^{(4)}$$



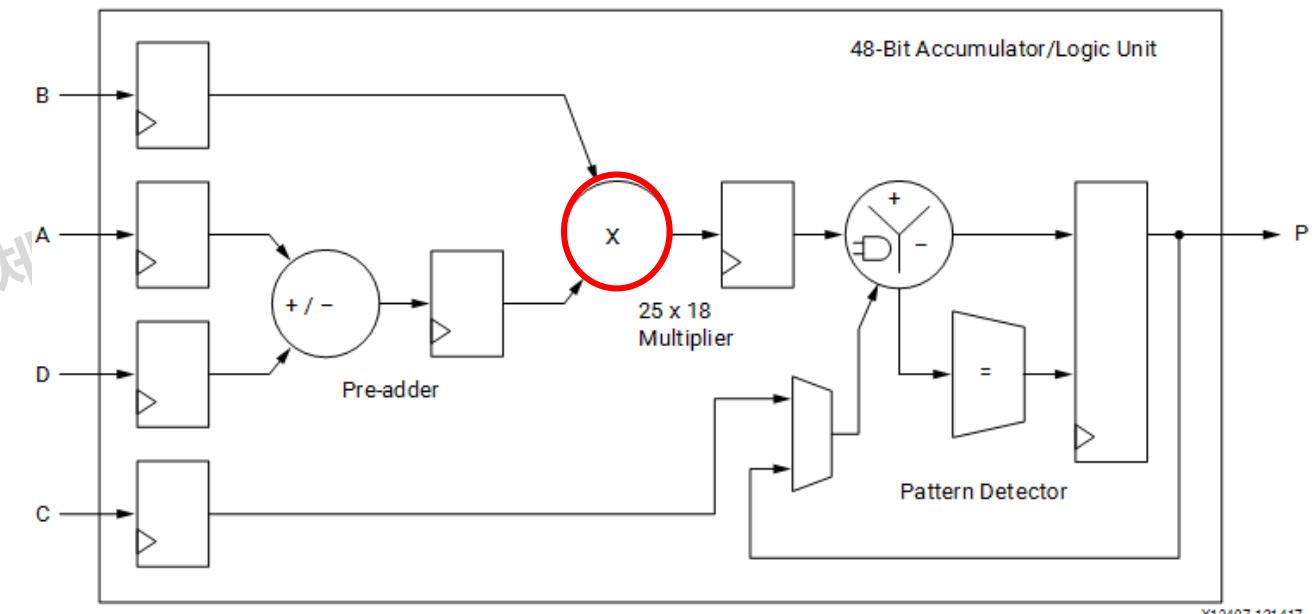
Lab 1: MAC

- Lab 1: Implement a multiplication and accumulation module (mac.v)
 - Complete the missing codes
 - An array of multipliers
 - Hierarchical adders
 - Delay
 - Output signals
 - Test the design
 - Do simulation
 - Capture the waveform

Multipliers (mul.v)

- A multiplier is used to implement a (signed) multiplication function: $y = w * x$
 - Default settings: y : 16 bits, w and x : 8 bits
- A multiplication can be mapped to a DSP block
 - For FPGA implementation, can use Xilinx IP catalog to generate DSP

```
1 module mul
2 #(parameter WI = 8,
3 parameter WO = 2*WI) (
4   input [WI-1:0] w,
5   input [WI-1:0] x,
6   output[WO-1:0] y
7 );
8
9   assign y = $signed(x) * $signed(w);
10
11 endmodule
```



Multiplier: Using DSPs or LUTs (FPGA)

different type of multiplication arch

```
1 `timescale 1ns / 1ps
2
3 module mul_DSP(
4   clk,
5   w_i,
6   x_i,
7   y_o
8 );
9 parameter DATA_WIDTH = 16;
10
11 input clk;
12 input signed [DATA_WIDTH-1:0] w_i;
13 input signed [DATA_WIDTH-1:0] x_i;
14 output signed [2*DATA_WIDTH-1:0] y_o;
15
16 // Combinational clk
17 assign y_o = x_i * w_i;
18
19 endmodule
```

Use DSP for a large bitwidth

Copy

```
1 `timescale 1ns / 1ps
2 (*use_DSP48 = "no")
3 module mul_LUT(
4   clk,
5   w_i,
6   x_i,
7   y_o
8 );
9 parameter DATA_WIDTH = 16;
10
11 input clk;
12 input signed [DATA_WIDTH-1:0] w_i;
13 input signed [DATA_WIDTH-1:0] x_i;
14 output signed [2*DATA_WIDTH-1:0] y_o;
15
16 // Combinational clk
17 assign y_o = x_i * w_i;
18
19 endmodule
```

Force to use LUT

DATA_WIDTH	mul_DSP		mul_LUT	
	LUT	DSP	LUT	DSP
4	23	0	23	0
6	36	0	36	0
8	61	0	61	0
9	88	0	88	0
10	102	0	102	0
11	0	1	132	0
12	0	1	148	0
16	0	1	265	0
24	0	2	595	0
32	47	4	1027	0

FPGA Resource by a multiplier

Multiplication and accumulation (mac.v)

- Parameters
 - **WI**: Bit width of input numbers (win, din)
 - **N**: Number of inputs,
 - **WN**: Depth of adder tree (depends on N)
 - **WO**: Bit width of output number (acc_o)
- Ports:
 - **win**, **din**: inputs, **vld_i**: Input valid signals
 - **acc_o**: output, **vld_o**: output valid signals
- Regs and Wires
 - **vld_d**: delayed valid signal
 - **y0**: 16 outputs of multipliers
 - **y1**: 8 outputs of adders (level 1)
 - **y2**: 4 outputs of adders (level 2)
 - **y3**: 2 outputs of adders (level 3)
 - **y4**: 1 outputs of adders (level 4)
 - **weight**: dequantized weight values
 - **activation**: activation values

```
1  module mac #(           Copyright 2022, 차세대반도체 혁신공유대학 사업단. All rights reserved.
2    parameter WI = 8,
3    parameter N  = 16,
4    parameter WN = $clog2(N),
5    parameter WO = 2*WI + WN) (
6      input  clk,
7      input  rstn,
8      input  vld_i,
9      input [N*WI-1:0] win,
10     input [N*WI-1:0] din,
11     output[WO+1:0] acc_o,
12     output  vld_o
13   );
14
15   reg [WN:0] vld_d;
16   wire[2*WI+1:0] y0[0:N-1]; // 16 outputs of multipliers
17   reg [2*WI+2:0] y1[0:N/2-1]; // 8 outputs of adders (level 1)
18   reg [2*WI+3:0] y2[0:N/4-1]; // 4 outputs of adders (level 2)
19   reg [2*WI+4:0] y3[0:N/8-1]; // 2 outputs of adders (level 3)
20   reg [2*WI+5:0] y4; // 1 output of adders (level 4)
21
22   reg [N*(WI+1)-1:0] weight;
23   reg [N*(WI+1)-1:0] activation;
```

Explanation: Pre-processing step

```
integer i;
//-----
// Components: Array of multipliers
//-----
always@(posedge clk, negedge rstn) begin
    if(~rstn) begin
        weight      <= 0;
        activation  <= 0;
    end
    else begin
        for(i = 0; i < N; i=i+1) begin
            weight[i*(WI+1)+:(WI+1)]      <= $signed($signed({win[( i*WI)+:WI],1'b0})+1);
            activation[i*(WI+1)+:(WI+1)]   <= $signed({1'b0,din[ i*WI+:WI]});
        end
    end
end
weight[i*(WI+1)+:(WI+1)]<=$signed($signed({win[( i*WI)+:WI],1'b0})+1);
```

$$2 * w + 1$$

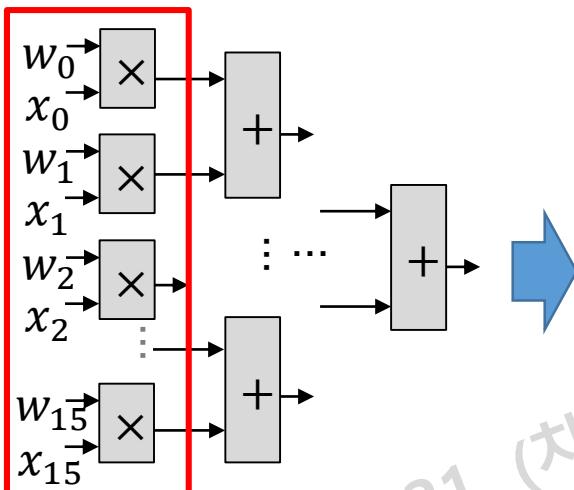
...converting to the representation (2's complement)
from the quantized value

```
activation[i*(WI+1)+:(WI+1)]<= $signed({1'b0,din[ i*WI+:WI]});
```

Explanation: Multiplication (mac.v)

- An array of multipliers:
 - Do 16 multiplication operations in parallel
 - $T_i = N = 16$

multipliers



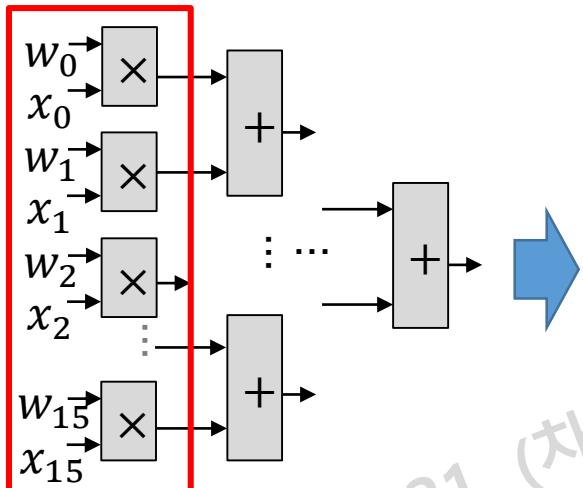
```
mul #(.WI(WI+1)) u_mul_00(.w(weight[( 0*(WI+1))+:(WI+1)]), .x(activation[( 0*(WI+1))+:(WI+1)]), .y(y0[ 0]));
mul #(.WI(WI+1)) u_mul_01(.w(weight[( 1*(WI+1))+:(WI+1)]), .x(activation[( 1*(WI+1))+:(WI+1)]), .y(y0[ 1]));
mul #(.WI(WI+1)) u_mul_02(.w(weight[( 2*(WI+1))+:(WI+1)]), .x(activation[( 2*(WI+1))+:(WI+1)]), .y(y0[ 2]));
mul #(.WI(WI+1)) u_mul_03(.w(weight[( 3*(WI+1))+:(WI+1)]), .x(activation[( 3*(WI+1))+:(WI+1)]), .y(y0[ 3]));
mul #(.WI(WI+1)) u_mul_04(.w(weight[( 4*(WI+1))+:(WI+1)]), .x(activation[( 4*(WI+1))+:(WI+1)]), .y(y0[ 4]));
mul #(.WI(WI+1)) u_mul_05(.w(weight[( 5*(WI+1))+:(WI+1)]), .x(activation[( 5*(WI+1))+:(WI+1)]), .y(y0[ 5]));
mul #(.WI(WI+1)) u_mul_06(.w(weight[( 6*(WI+1))+:(WI+1)]), .x(activation[( 6*(WI+1))+:(WI+1)]), .y(y0[ 6]));
mul #(.WI(WI+1)) u_mul_07(.w(weight[( 7*(WI+1))+:(WI+1)]), .x(activation[( 7*(WI+1))+:(WI+1)]), .y(y0[ 7]));
mul #(.WI(WI+1)) u_mul_08(.w(weight[( 8*(WI+1))+:(WI+1)]), .x(activation[( 8*(WI+1))+:(WI+1)]), .y(y0[ 8]));
mul #(.WI(WI+1)) u_mul_09(.w(weight[( 9*(WI+1))+:(WI+1)]), .x(activation[( 9*(WI+1))+:(WI+1)]), .y(y0[ 9]));
mul #(.WI(WI+1)) u_mul_10(.w(weight[(10*(WI+1))+:(WI+1)]), .x(activation[(10*(WI+1))+:(WI+1)]), .y(y0[10]));
mul #(.WI(WI+1)) u_mul_11(.w(weight[(11*(WI+1))+:(WI+1)]), .x(activation[(11*(WI+1))+:(WI+1)]), .y(y0[11]));
mul #(.WI(WI+1)) u_mul_12(.w(weight[(12*(WI+1))+:(WI+1)]), .x(activation[(12*(WI+1))+:(WI+1)]), .y(y0[12]));
mul #(.WI(WI+1)) u_mul_13(.w(weight[(13*(WI+1))+:(WI+1)]), .x(activation[(13*(WI+1))+:(WI+1)]), .y(y0[13]));
/* insert your code */
```

```
mul #(.WI(WI+1)) u_mul_00(
    .w(weight[( 0*(WI+1))+:(WI+1)]),
    .x(activation[( 0*(WI+1))+:(WI+1)]),
    .y(y0[ 0])); // Weight (w)
// Activation (x)
// Output (y)
```

To do: Multipliers

- Complete the missing codes of multipliers

multipliers

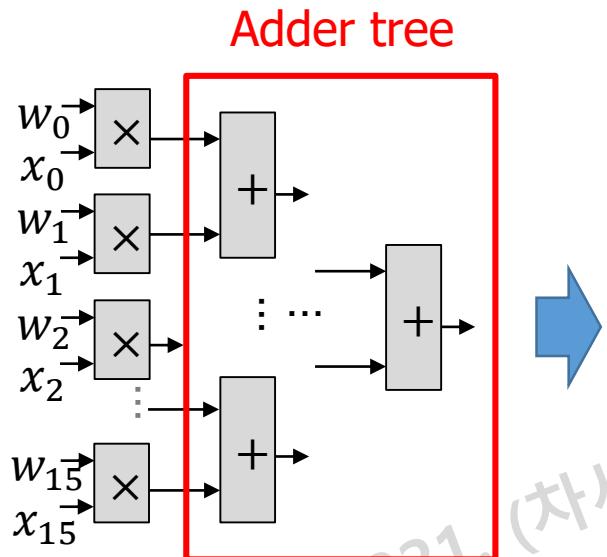


```
integer i;
//-
// Components: Array of multipliers
//-
always@(posedge clk, negedge rstn) begin
    if(~rstn) begin
        weight      <= 0;
        activation  <= 0;
    end
    else begin
        for(i = 0; i < N; i=i+1) begin
            weight[i*(WI+1)+(WI+1)]      <= $signed($signed({win[({ i*WI}+:WI],1'b0})+1);
            activation[i*(WI+1)+(WI+1)]   <= $signed({1'b0,din[ i*WI+:WI]}));
        end
    end
end
mul #(WI(WI+1)) u_mul_00(.w(weight[{ 0*(WI+1)}:(WI+1)]),.x(activation[{ 0*(WI+1)}:(WI+1)]),.y(y0[ 0]));
mul #(WI(WI+1)) u_mul_01(.w(weight[{ 1*(WI+1)}:(WI+1)]),.x(activation[{ 1*(WI+1)}:(WI+1)]),.y(y0[ 1]));
mul #(WI(WI+1)) u_mul_02(.w(weight[{ 2*(WI+1)}:(WI+1)]),.x(activation[{ 2*(WI+1)}:(WI+1)]),.y(y0[ 2]));
mul #(WI(WI+1)) u_mul_03(.w(weight[{ 3*(WI+1)}:(WI+1)]),.x(activation[{ 3*(WI+1)}:(WI+1)]),.y(y0[ 3]));
mul #(WI(WI+1)) u_mul_04(.w(weight[{ 4*(WI+1)}:(WI+1)]),.x(activation[{ 4*(WI+1)}:(WI+1)]),.y(y0[ 4]));
mul #(WI(WI+1)) u_mul_05(.w(weight[{ 5*(WI+1)}:(WI+1)]),.x(activation[{ 5*(WI+1)}:(WI+1)]),.y(y0[ 5]));
mul #(WI(WI+1)) u_mul_06(.w(weight[{ 6*(WI+1)}:(WI+1)]),.x(activation[{ 6*(WI+1)}:(WI+1)]),.y(y0[ 6]));
mul #(WI(WI+1)) u_mul_07(.w(weight[{ 7*(WI+1)}:(WI+1)]),.x(activation[{ 7*(WI+1)}:(WI+1)]),.y(y0[ 7]));
mul #(WI(WI+1)) u_mul_08(.w(weight[{ 8*(WI+1)}:(WI+1)]),.x(activation[{ 8*(WI+1)}:(WI+1)]),.y(y0[ 8]));
mul #(WI(WI+1)) u_mul_09(.w(weight[{ 9*(WI+1)}:(WI+1)]),.x(activation[{ 9*(WI+1)}:(WI+1)]),.y(y0[ 9]));
mul #(WI(WI+1)) u_mul_10(.w(weight[{10*(WI+1)}:(WI+1)]),.x(activation[{10*(WI+1)}:(WI+1)]),.y(y0[10]));
mul #(WI(WI+1)) u_mul_11(.w(weight[{11*(WI+1)}:(WI+1)]),.x(activation[{11*(WI+1)}:(WI+1)]),.y(y0[11]));
mul #(WI(WI+1)) u_mul_12(.w(weight[{12*(WI+1)}:(WI+1)]),.x(activation[{12*(WI+1)}:(WI+1)]),.y(y0[12]));
mul #(WI(WI+1)) u_mul_13(.w(weight[{13*(WI+1)}:(WI+1)]),.x(activation[{13*(WI+1)}:(WI+1)]),.y(y0[13]));
/* insert your code */
```

TODO: insert remaining multipliers

Explanation: Adder Tree (mac.v)

- Adder tree: sum up all multipliers' results.



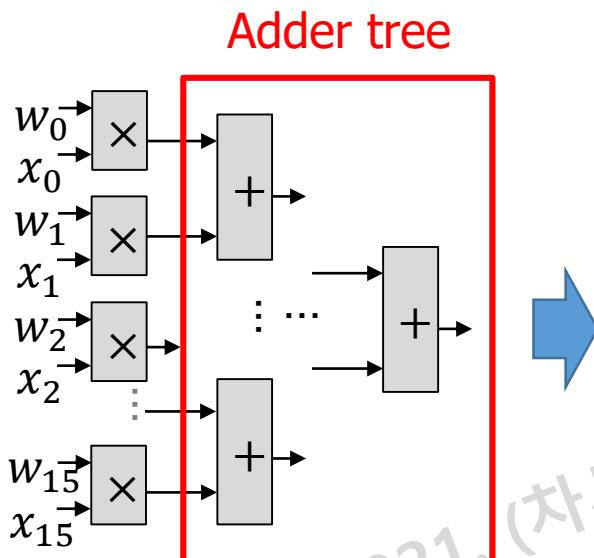
Pseudo code

$$\begin{aligned}y_1^{(0)} &= w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15} \\y_1^{(1)} &= y_0^{(0)} + y_1^{(0)}, \dots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)} \\y_1^{(2)} &= y_0^{(1)} + y_1^{(1)}, \dots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)} \\y_1^{(3)} &= y_0^{(2)} + y_1^{(2)}, \dots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)} \\y_1^{(4)} &= y_0^{(3)} + y_1^{(3)} \\Y &= y_1^{(4)}\end{aligned}$$

```
always@(posedge clk, negedge rstn) begin
    if(~rstn) begin
        for(i = 0; i < N/2; i=i+1) begin
            y1[i] <= 0;
        end
        for( i = 0; i < N/4; i=i+1) begin
            y2[i] <= 0;
        end
        for( i = 0; i < N/8; i=i+1) begin
            y3[i] <= 0;
        end
        y4 <=0;
    end
    else begin
        // Level 1
        y1[0] <= $signed(y0[ 0]) + $signed(y0[ 1]);
        y1[1] <= $signed(y0[ 2]) + $signed(y0[ 3]);
        y1[2] <= $signed(y0[ 4]) + $signed(y0[ 5]);
        y1[3] <= $signed(y0[ 6]) + $signed(y0[ 7]);
        y1[4] <= $signed(y0[ 8]) + $signed(y0[ 9]);
        y1[5] <= $signed(y0[10]) + $signed(y0[11]);
        y1[6] <= $signed(y0[12]) + $signed(y0[13]);
        y1[7] <= $signed(y0[14]) + $signed(y0[15]);
        // Level 2
        y2[0] <= $signed(y1[0]) + $signed(y1[ 1]);
        y2[1] <= $signed(y1[2]) + $signed(y1[ 3]);
        y2[2] <= $signed(y1[4]) + $signed(y1[ 5]);
        y2[3] <= $signed(y1[6]) + $signed(y1[ 7]);
        /* insert your code */
        // Level 3
        /* insert your code */
    end
end
```

To do ...: Adder tree

- Complete the missing code



```
else begin
    // Level 1
    yl[0] <= $signed(y0[ 0]) + $signed(y0[ 1]);
    yl[1] <= $signed(y0[ 2]) + $signed(y0[ 3]);
    yl[2] <= $signed(y0[ 4]) + $signed(y0[ 5]);
    yl[3] <= $signed(y0[ 6]) + $signed(y0[ 7]);
    yl[4] <= $signed(y0[ 8]) + $signed(y0[ 9]);
    yl[5] <= $signed(y0[10]) + $signed(y0[11]);
    yl[6] <= $signed(y0[12]) + $signed(y0[13]);
    yl[7] <= $signed(y0[14]) + $signed(y0[15]);
    // Level 2
    y2[0] <= $signed(yl[0]) + $signed(yl[ 1]);
    y2[1] <= $signed(yl[2]) + $signed(yl[ 3]);
    y2[2] <= $signed(yl[4]) + $signed(yl[ 5]);
    y2[3] <= $signed(yl[6]) + $signed(yl[ 7]);
    /* insert your code */
    // Level 3
    ...
    // Level 4
    /* insert your code */
end
```

Insert your code for levels 3 and 4

To do: Delay signals

- Delays (vld_d) are used to determine the output valid signal (vld_o) from the input valid (vld_i).

```
//-----  
//Output and Delay signals  
//-----  
always@(posedge clk, negedge rstn) begin  
    if(~rstn) begin  
        vld_d <= 0;  
    end  
    else begin  
        /* insert your code */  
    end  
end  
assign vld_o = vld_d[WN];  
assign acc_o = $signed(y4);
```

Test bench (mac_tb.v)

- Test bench
 - Internal signals
 - Design under test
 - Clock
 - Test vector

```
module mac_tb;
parameter WI = 8;
parameter N = 16;
parameter WN = $clog2(N);
parameter WO = 2*(WI+1) + WN;
reg clk;
reg rstn;
reg vld_i;
reg [WI*N-1:0] win;
reg [WI*N-1:0] din;
wire[WO-1:0] acc_o;
wire vld_o;

reg [WI-1:0] weight_block [0:N-1];
reg [WI-1:0] activation_block [0:N-1];
// DUT
]mac u_mac(
/*input */clk(clk),
/*input */rstn(rstn),
/*input */vld_i(vld_i),
/*input [N*WI-1:0] */win(win),
/*input [N*WI-1:0] */din(din),
/*output[WO-1:0】 */acc_o(acc_o),
/*output reg */vld_o(vld_o)
);

// Clock
parameter CLK_PERIOD = 10; //100MHz
]initial begin
    clk = 1'bl;
    forever #(CLK_PERIOD/2) clk = ~clk;
end
```

Copyright 2021. (차세대반도체)

Test bench (mac_tb.v)

- Test bench
 - Internal signals
 - Design under test
 - Clock
 - Test vector

```
// Data preparation
initial begin
    for(i = 0; i < N; i = i +1) begin
        activation_block[i] = i;
        weight_block[i] = 1;
    end
end

// Test cases
initial begin
    rstn = 1'b0;                      // Reset, low active
    vld_i = 1'b0;
    win = 0;
    din = 0;

    #(4*CLK_PERIOD) rstn = 1'bl;

    //
    #(4*CLK_PERIOD)
    @(posedge clk)      // Fixing the timing issue in waveform
    for(i = 0; i < N; i = i+1) begin
        //@(posedge clk)      // Fixing the timing issue in waveform
        vld_i = 1'bl;
        win[(i*WI)+:WI] = weight_block[i];
        din[(i*WI)+:WI] = activation_block[i];
    end

    #(CLK_PERIOD)
    @(posedge clk) // Fixing the timing issue in waveform
    vld_i = 1'b0;
    win = 0;
    din = 0;
end
```

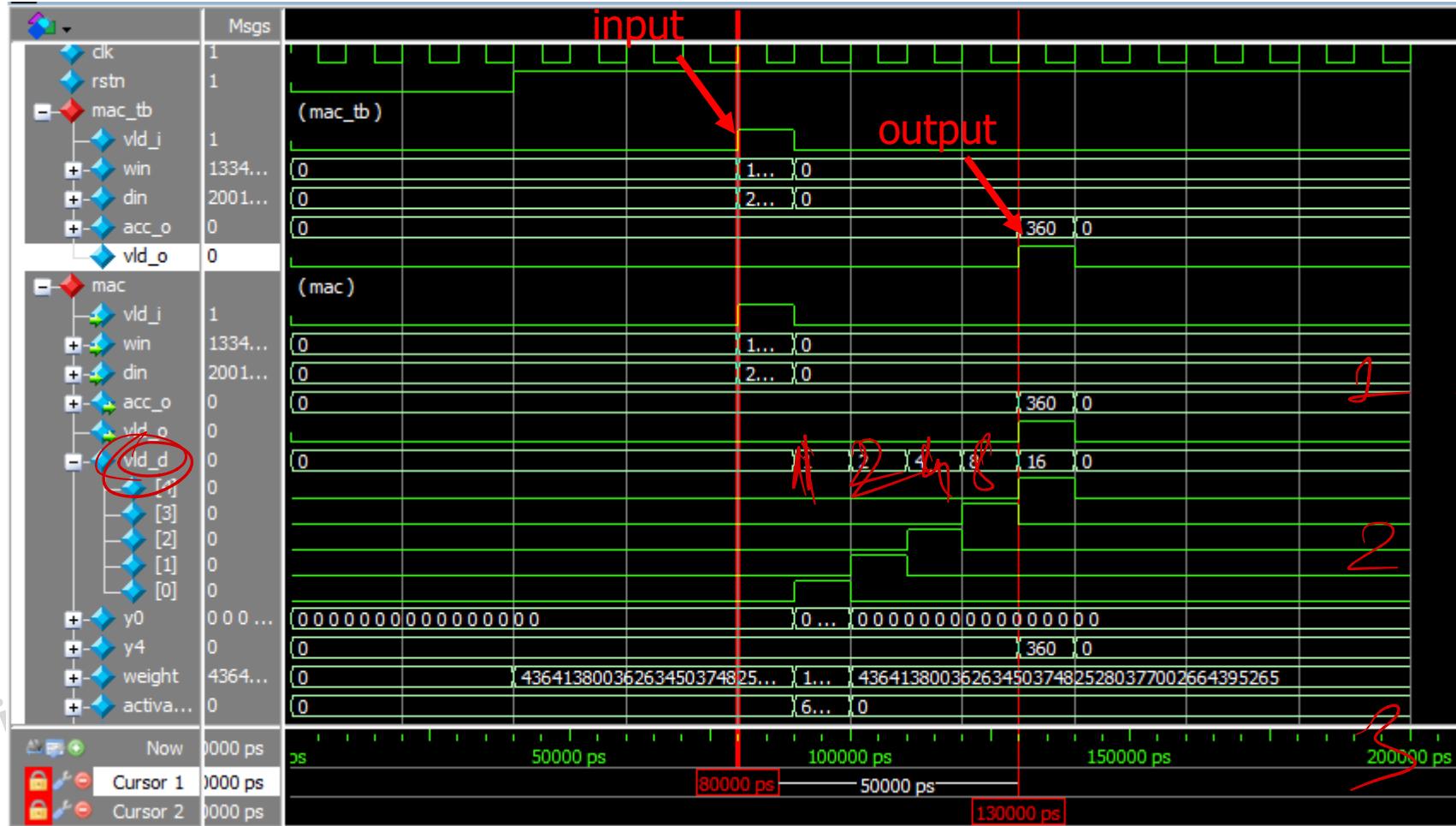
ved.

Copyright 2021. (차세대반도

Waveform

11111

32101



To do ...

- Implement mac.v
- Complete the missing codes
 - An array of multipliers
 - Hierarchical adders
 - Delay
 - Output signals
- Do a simulation with time = 200 ns
- Show the waveform

Road map

Review

MAC

Mac kernel
(Accumulation)

Convolution kernel
(Normalization
& Activation)

MAC (mac.v)

- Compute a sum of N products

- Pseudo code

$$y_1^{(0)} = w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15}$$

$$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \dots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$$

$$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \dots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$$

$$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \dots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$$

$$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$$

$$Y = y_1^{(4)}$$

$$Y = \sum_{i=0}^{15} w_i * x_i$$

// N multipliers

// N/2 adders

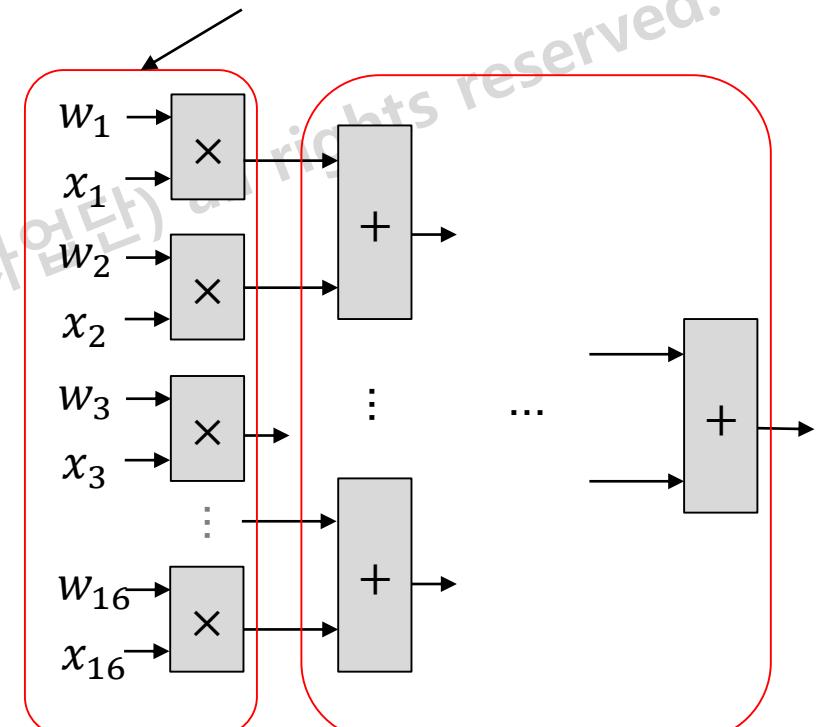
// N/4 adders

// N/4 adders

// N/4 adders

// Output

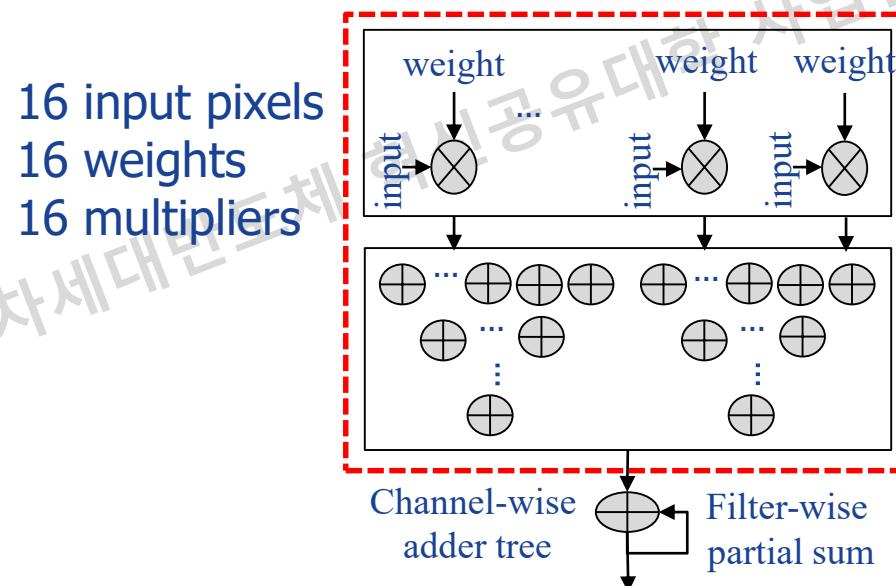
N block/module of multipliers



Adder Tree

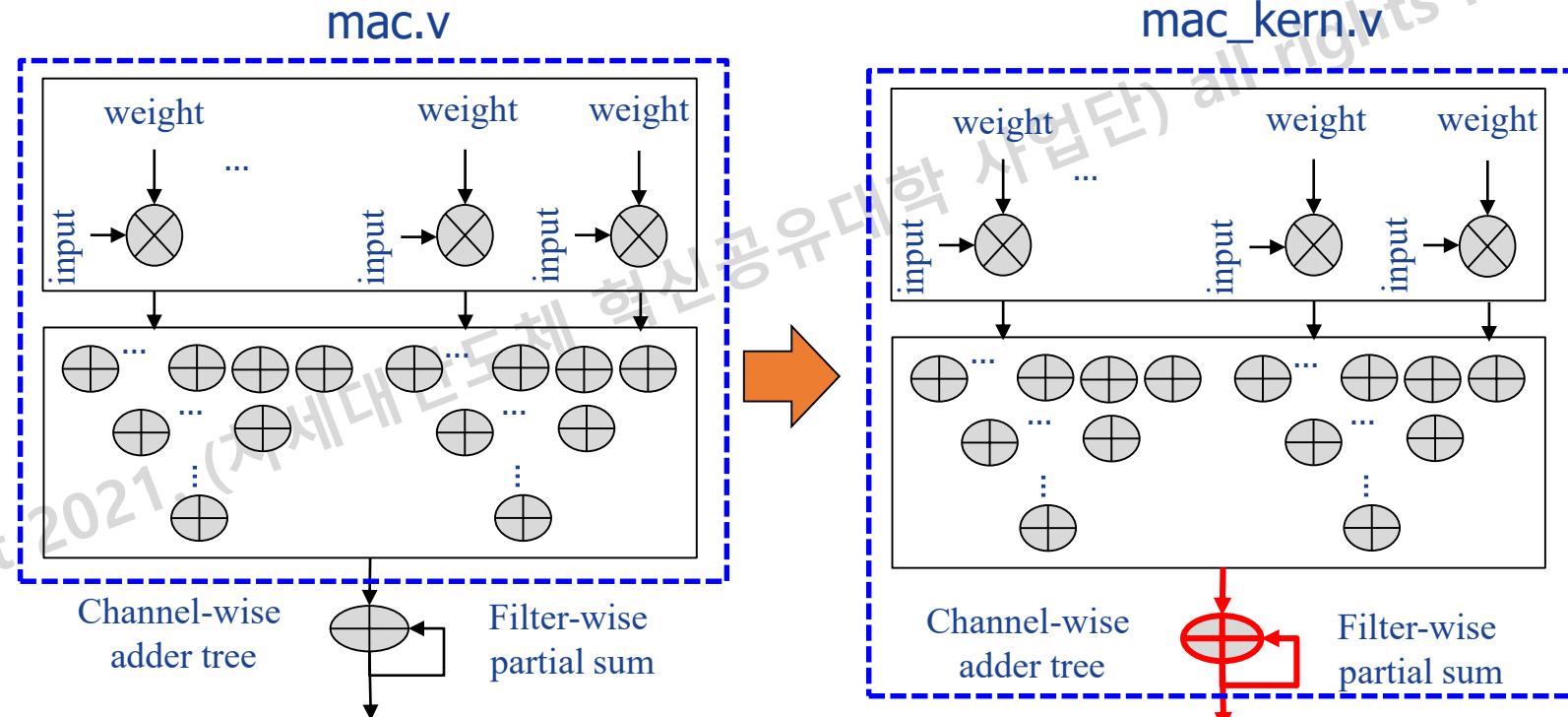
Mapping

- The required number of multiplication operations (num_ops) to calculate **ONE output pixel**
 - Layer 1: num_ops = 9 ($=3 \times 3 \times 1$)
 - Layer 2 and 3: num_ops = 144 ($= 3 \times 3 \times 16$).
- H/W module: N = 16
 - Layer 1: num_ops < N \Rightarrow calculating an output pixel needs 1 cycles
 - Layer 2, 3: num_ops > N \Rightarrow calculating an output pixel needs 9 cycles



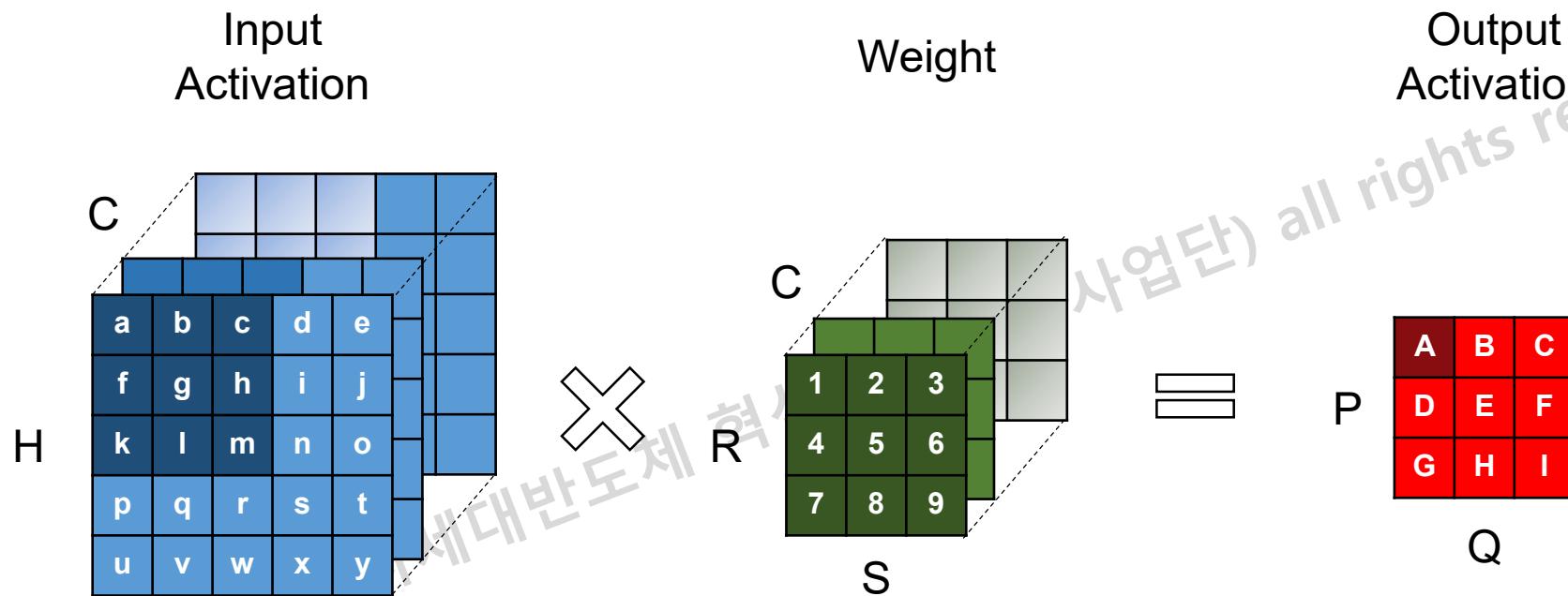
To do ...

- Implement mac_kern.v
 - Include an instance of mac
 - Add an accumulated sum
- Mode: 0: conv 1x1, 1: conv 3x3.



Convolution: 3x3

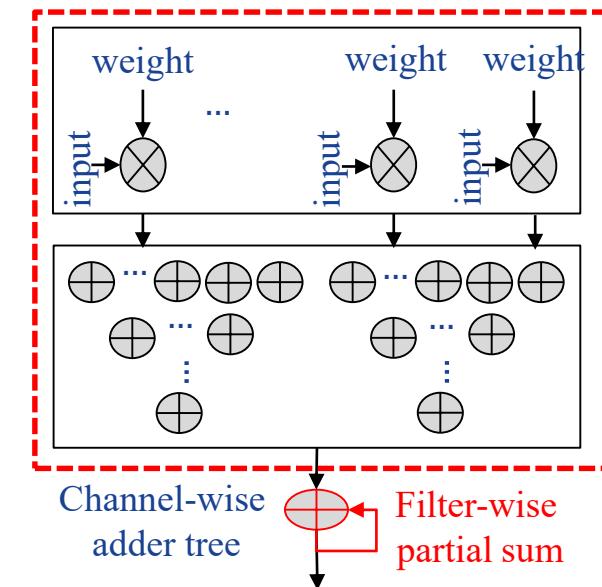
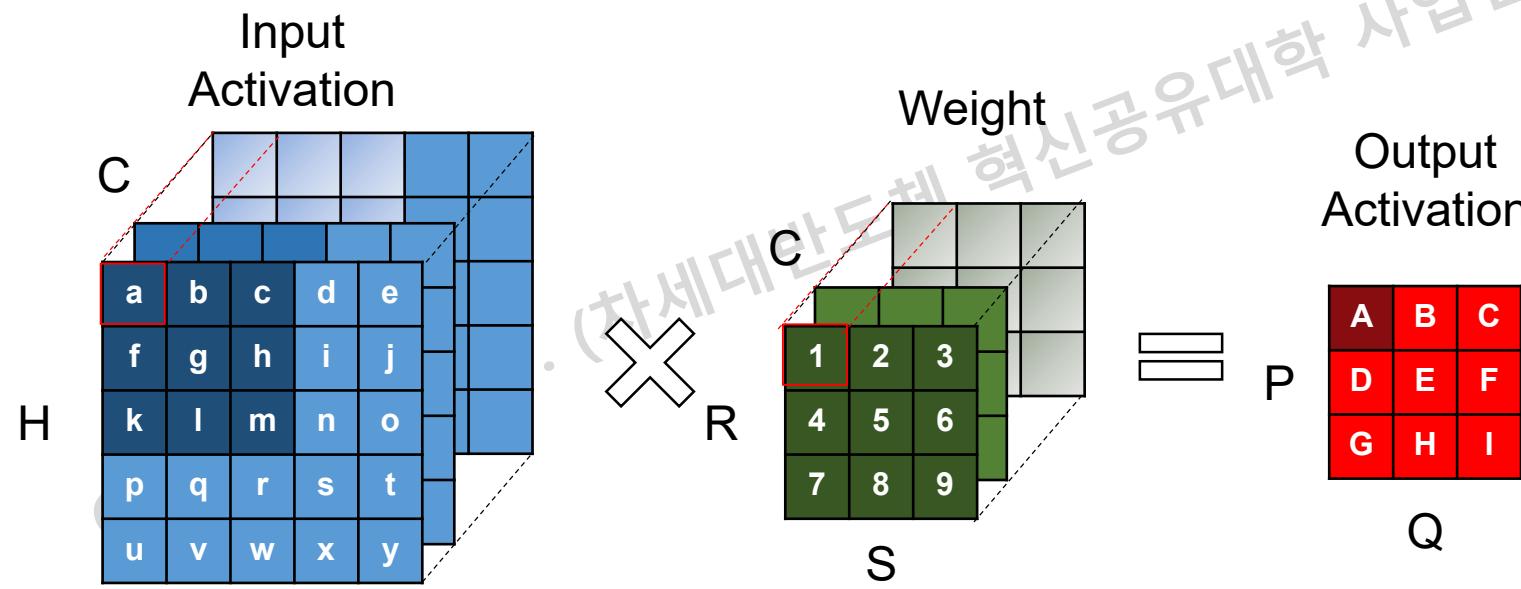
- **C:** # of Input Channels



Mapping: conv3x3

- Cycle 1:
 - Input activation "a": 1x1x16
 - Weight "1": 1x1x16
 - mac_kern.v:

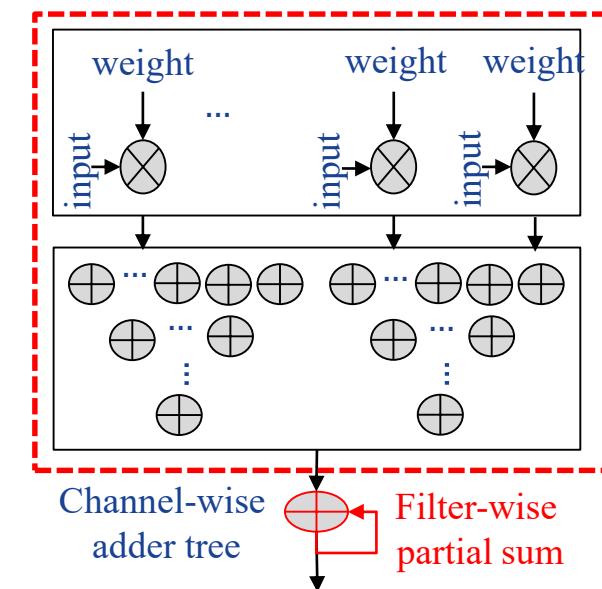
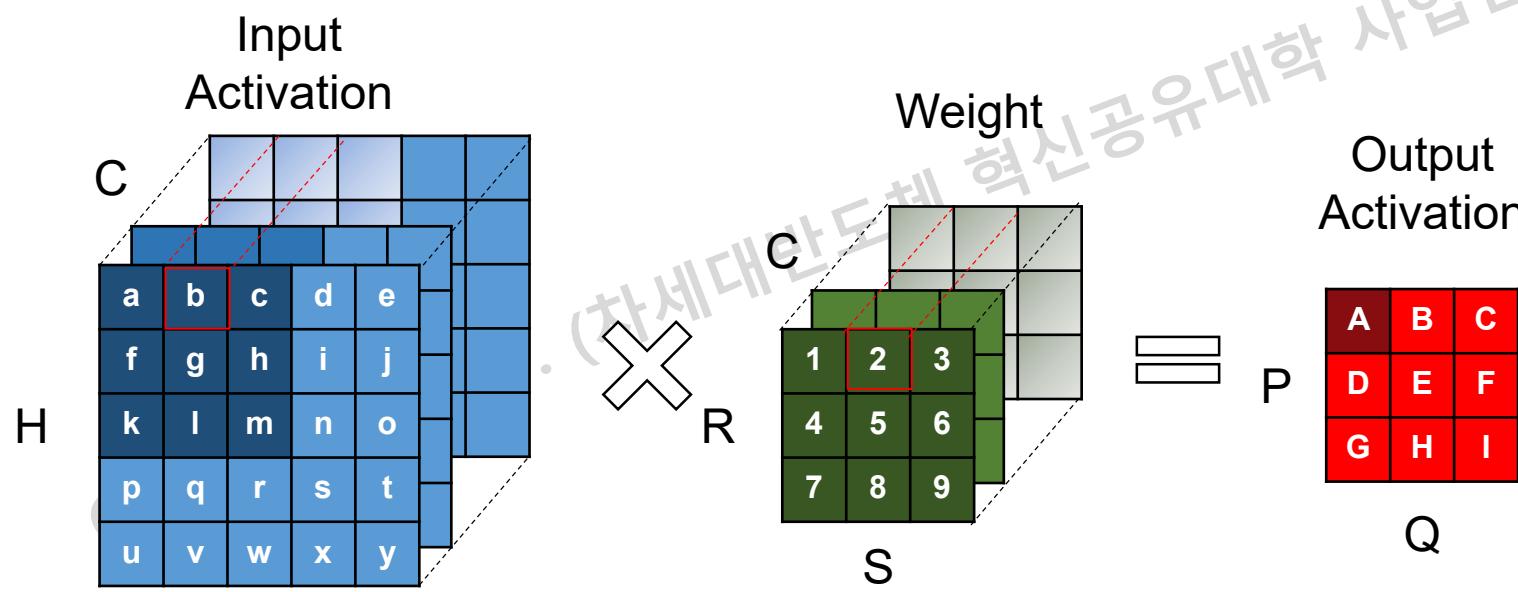
$$Y \leftarrow \sum_{i=0}^{15} w_i * x_i$$



Mapping: conv3x3

- Cycle 2:
 - Input activation "b": 1x1x16
 - Weight "2": 1x1x16
 - mac_kern.v:

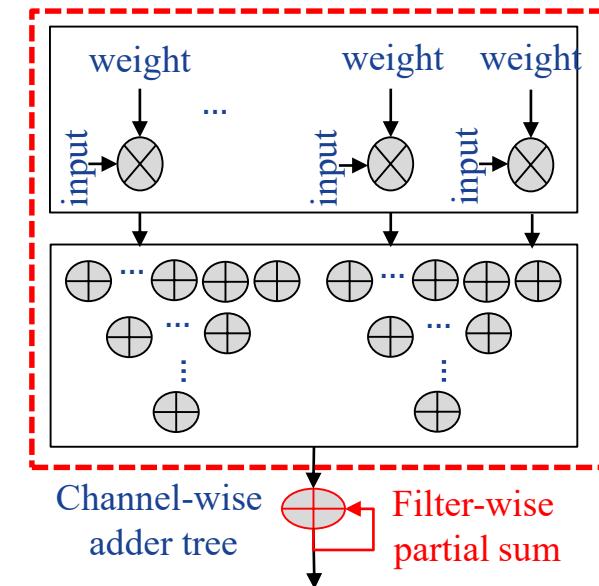
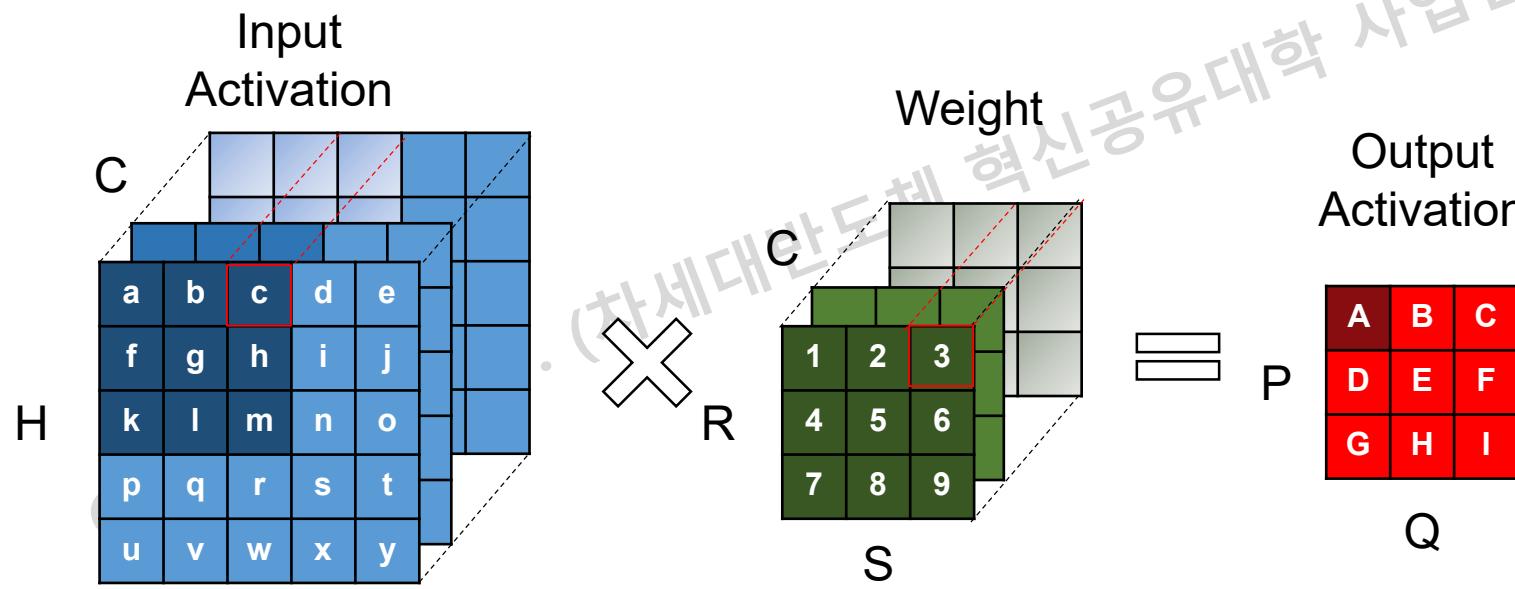
$$Y \leftarrow Y + \sum_{i=16}^{31} w_i * x_i$$



Mapping: conv3x3

- Cycle 3:
 - Input activation "c": 1x1x16
 - Weight "3": 1x1x16
 - mac_kern.v:

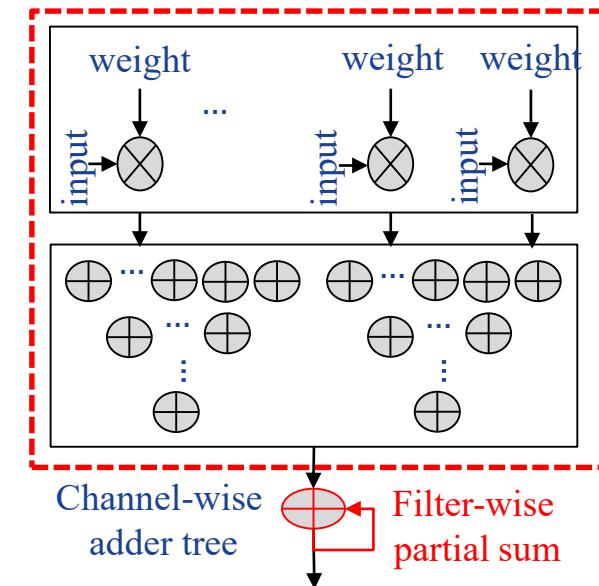
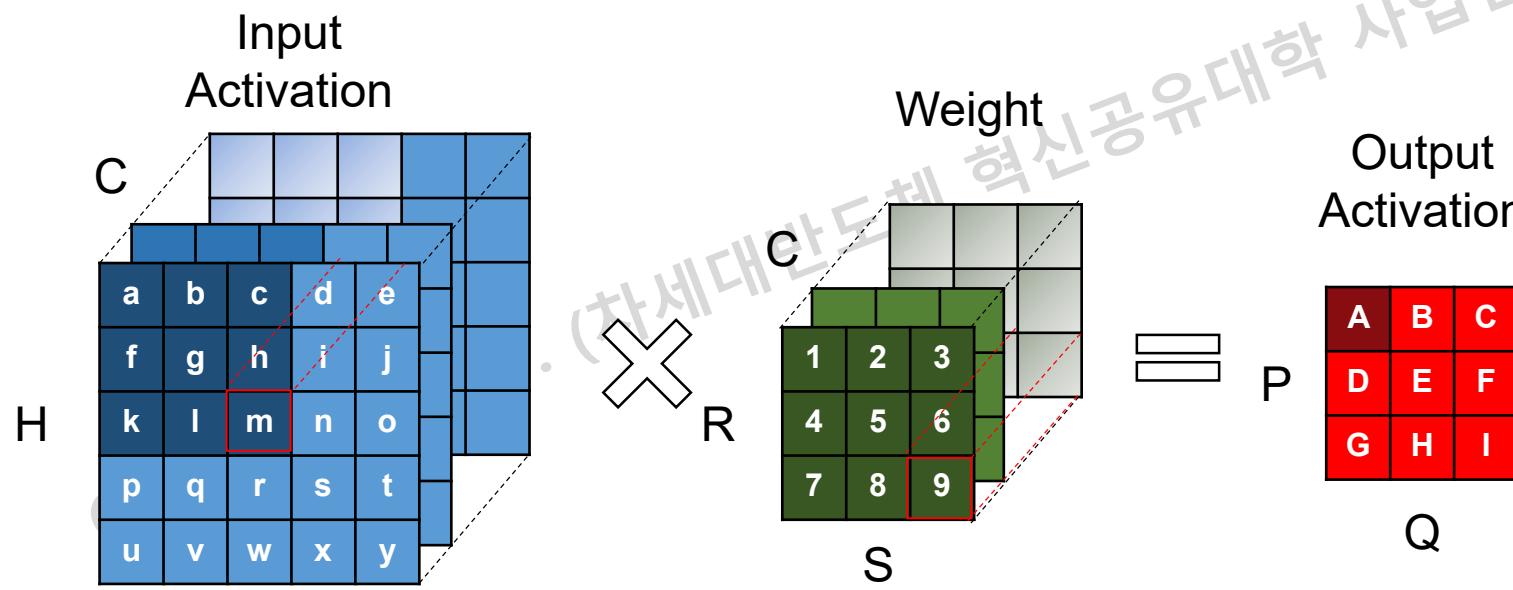
$$Y \leftarrow Y + \sum_{i=32}^{47} w_i * x_i$$



Mapping: conv3x3

- Cycle 9:
 - Input activation "m": 1x1x16
 - Weight "9": 1x1x16
 - mac_kern.v:

$$Y \leftarrow Y + \sum_{i=128}^{143} w_i * x_i$$



mac_kern.v

- Ports
 - **win, din**: input / **acc_o**: output
 - **vld_i, vld_o**: valid signals

- Parameters
 - **WI**: Bit width of input numbers (win, din)
 - **N**: Number of inputs to calculate
 - **WN**: Depth of adder tree (depends on N)
 - **WO**: Bit width of output number (acc_o)

```
module mac_kern #(  
    parameter WI = 8,  
    parameter N  = 16,  
    parameter WN = $clog2(N),  
    parameter WO = 2*(WI+1) + WN)(  
    clk,  
    rstn,  
    is_conv3x3,           //0: 1x1, 1:3x3  
    vld_i,  
    win,  
    din,  
    acc_o,  
    vld_o  
);  
  
localparam CONV3x3_DELAY      = 9;  
localparam CONV3x3_DELAY_W    = 4;  
// Ports  
input clk;  
input rstn;  
input is_conv3x3;           //0: 1x1, 1:3x3  
input vld_i;  
input [N*WI-1:0] win;  
input [N*WI-1:0] din;  
output [WO+CONV3x3_DELAY_W:0] acc_o;  
output reg vld_o;
```

mac_kern.v

- Internal registers and wires
 - **sub_acc_o**: mac accumulation result
 - **sub_vld_o**: vld_o signal of mac
 - **is_conv3x3_d**: stores delayed values of is_conv3x3
 - **sub_vld_o_d**: stores delayed values of sub_vld_o
 - **psum**: temporary register to sum the accumulated values of 3x3 kernel
 - **pix_idx**: counter for conv3x3
 - $0 \rightarrow 1 \rightarrow \dots \rightarrow 8 \rightarrow 0$

```
// Incoming signals from MACs
wire[WO-1:0] sub_acc_o;
wire sub_vld_o;
// Delay signals
reg [WN+1:0] is_conv3x3_d;
reg [CONV3x3_DELAY:0] sub_vld_o_d;
reg [WO+CONV3x3_DELAY_W:0] psum;
reg [3:0] pix_idx;
//-----
// Component: MAC
//-----
// DUT
mac u_mac(
    /*input */clk(clk),
    /*input */rstn(rstn),
    /*input */vld_i(vld_i),
    /*input */[N*WI-1:0] win(win),
    /*input */[N*WI-1:0] din(din),
    /*output*/[WO-1:0] acc_o(sub_acc_o),
    /*output*/reg vld_o(sub_vld_o)
);
```

Wi

Counter

- **pix_idx**: counter for conv3x3
 - 0→1→...→8→0

```
// Counter for CONV3x3
//{{{
always@ (posedge clk, negedge rstn) begin
    if(~rstn) begin
        pix_idx <= 0;
    end
    else begin
        if(is_conv3x3_d[WN] && sub_vld_o) begin
            if(pix_idx == 8)
                pix_idx <= 0;
            else
                pix_idx <= pix_idx + 1;
        end
    end
end
//}}}
```

Copyright 202

To do ...

- Complete the missing codes
 - Calculate an accumulated/partial sum (psum)
 - Generate the valid output signal (vld_o)
- Hint: Use pix_idx

```
-----  
// Accumulation  
-----  
//{{{  
always@(posedge clk, negedge rstn) begin  
    if(~rstn) begin  
        psum <= 0;  
    end  
    else begin  
        if(sub_vld_o) begin  
            /* insert your code */  
        end  
    end  
end  
}}}}
```

Y = Yt E wini

```
-----  
//Output and Delay signals  
-----  
//{{{  
always@(posedge clk, negedge rstn) begin  
    if(~rstn) begin  
        is_conv3x3_d <= 0;  
        sub_vld_o_d <= 0;  
    end  
    else begin  
        is_conv3x3_d <= {is_conv3x3_d[WN:0],is_conv3x3};  
        sub_vld_o_d <= {sub_vld_o_d[CONV3x3_DELAY-1:0],sub_vld_o};  
    end  
end  
  
assign acc_o = $signed(psum);  
  
always@(posedge clk, negedge rstn) begin  
    if(~rstn) begin  
        vld_o <= 1'b0;  
    end  
    else begin  
        if(~is_conv3x3_d[WN]) // conv1x1  
            vld_o <= sub_vld_o;  
        else begin // conv3x3  
            /* insert your code */  
        end  
    end  
end
```

= 9

Test bench (mac_kern_tb.v)

- Test bench (mac_kern_tb.v)
- Signals and parameters
- Design under test (DUT) is mac_kern

```
module mac_kern_tb;
parameter WI = 8;
parameter N = 16;
parameter WN = $clog2(N);
parameter WO = 2*(WI+1) + WN;
localparam CONV3x3_DELAY = 9;
localparam CONV3x3_DELAY_W = 4;
parameter W_DATA = 128;

parameter N_DELAY = 1;

reg clk;
reg rstn;
reg vld_i;
reg [WI*N-1:0] win;
reg [WI*N-1:0] din;
wire[WO+CONV3x3_DELAY_W:0] acc_o;
wire vld_o;

reg is_conv3x3;
```

```
// DUT
mac_kern u_mac_kern(
/*input*/clk(clk),
/*input*/rstn(rstn),
/*input*/is_conv3x3(is_conv3x3),
/*input*/vld_i(vld_i),
/*input*/[N*WI-1:0]win(win),
/*input*/[N*WI-1:0]din(din),
/*output*/[WO-1:0]acc_o(acc_o),
/*output*/vld_o(vld_o)
);

// Clock
parameter CLK_PERIOD = 10; //100MHz
initial begin
    clk = 1'b0;
    forever #(CLK_PERIOD/2) clk = ~clk;
end
```

Test bench

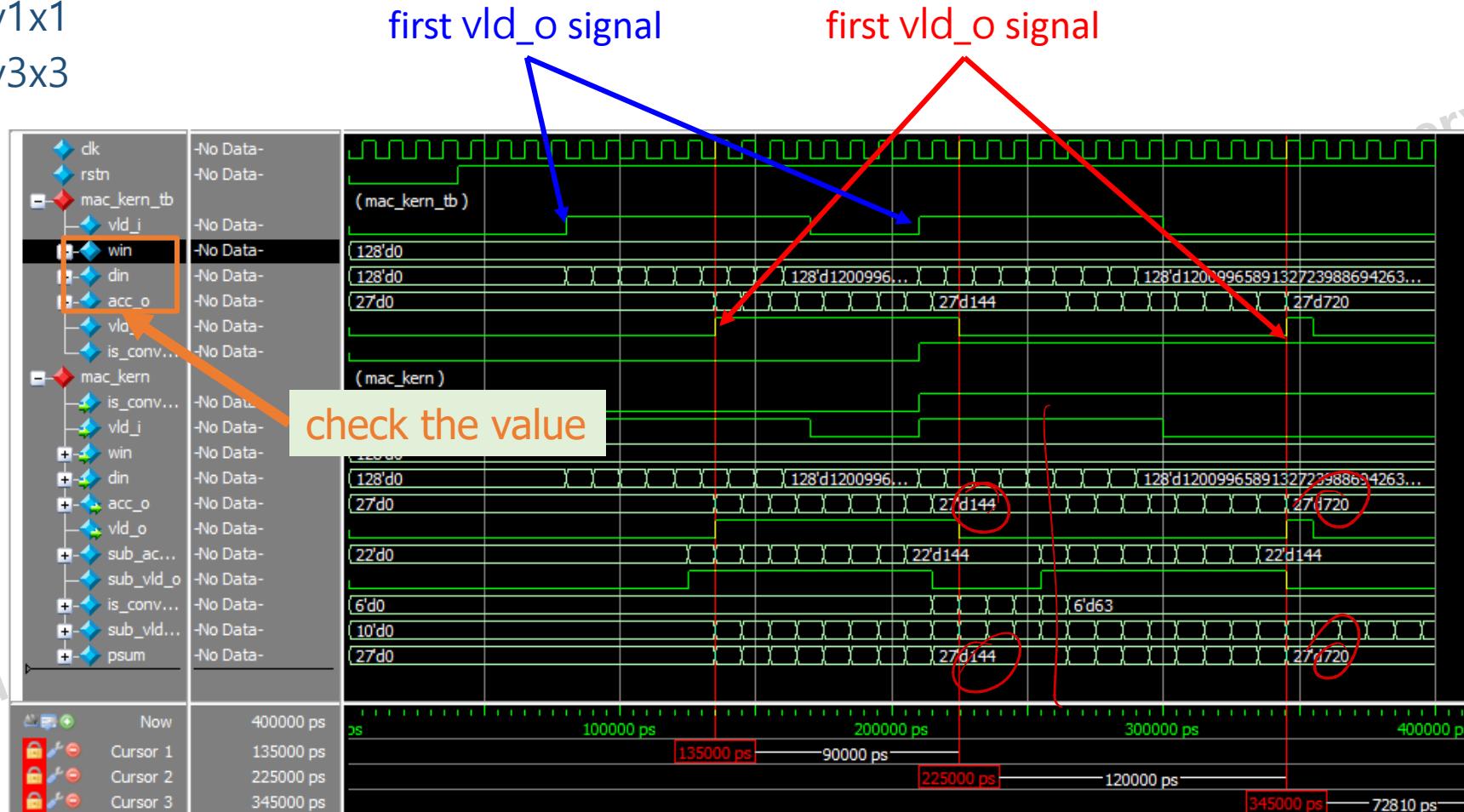
- Test conv1x1
- Test conv3x3

```
// Test case 1: test conv1x1
is_conv3x3 = 1'b0;
#(4*CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd1}}; //1
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd2}}; //2
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd3}}; //3
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd4}}; //4
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd5}}; //5
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd6}}; //6
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd7}}; //7
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd8}}; //8
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd9}}; //9
#(CLK_PERIOD) vld_i = 1'b0;
#(CLK_PERIOD)

// Test case 2: test conv3x3
#(4*CLK_PERIOD) is_conv3x3 = 1'b1;
vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd1}}; //1
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd2}}; //2
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd3}}; //3
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd4}}; //4
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd5}}; //5
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd6}}; //6
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd7}}; //7
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd8}}; //8
#(CLK_PERIOD) vld_i = 1'b1;
win = {16{8'd0 }}; //1
din = {16{8'd9}}; //9
#(CLK_PERIOD) vld_i = 1'b0;
#(CLK_PERIOD)
```

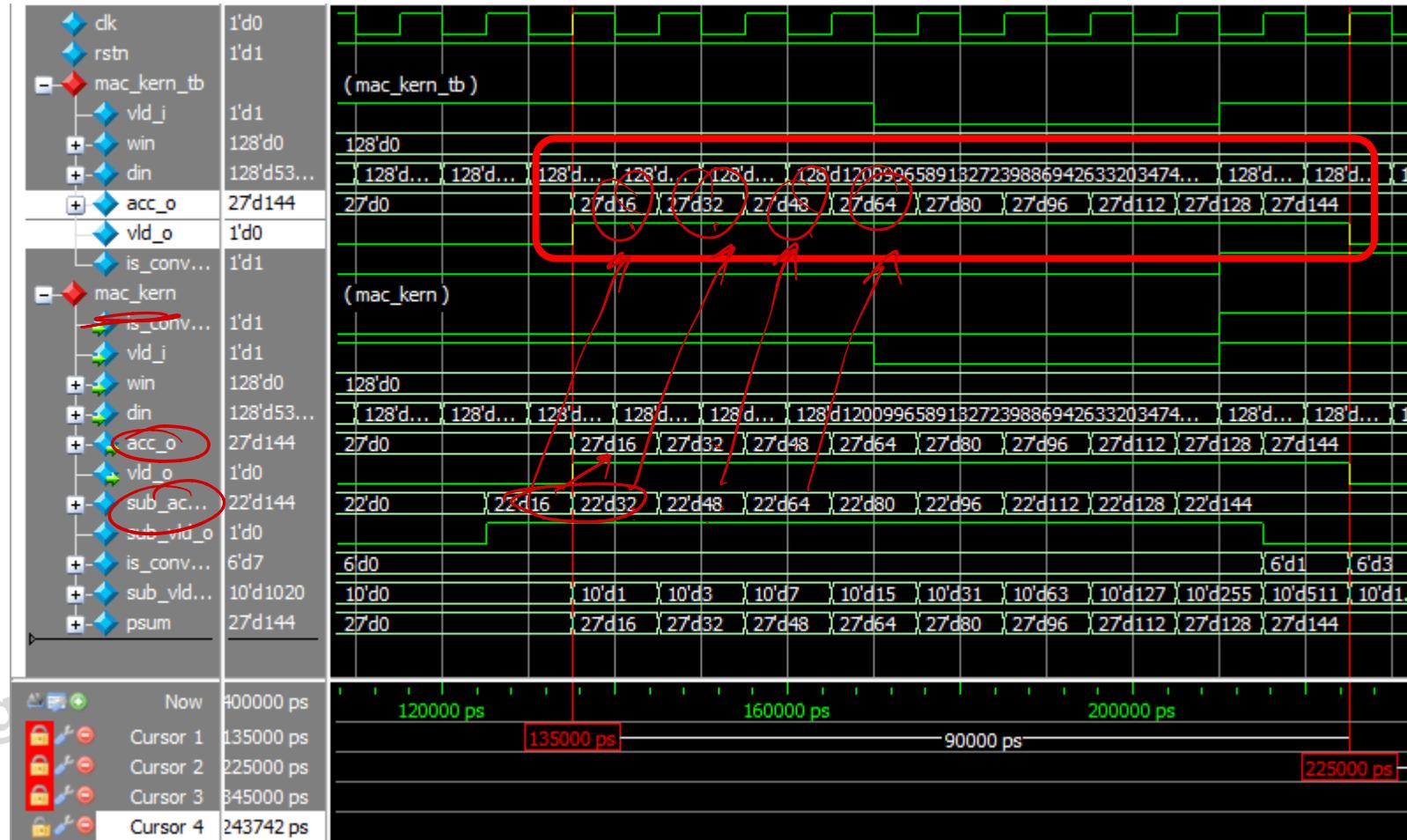
Waveform

- Test cases:
 - conv1x1
 - conv3x3



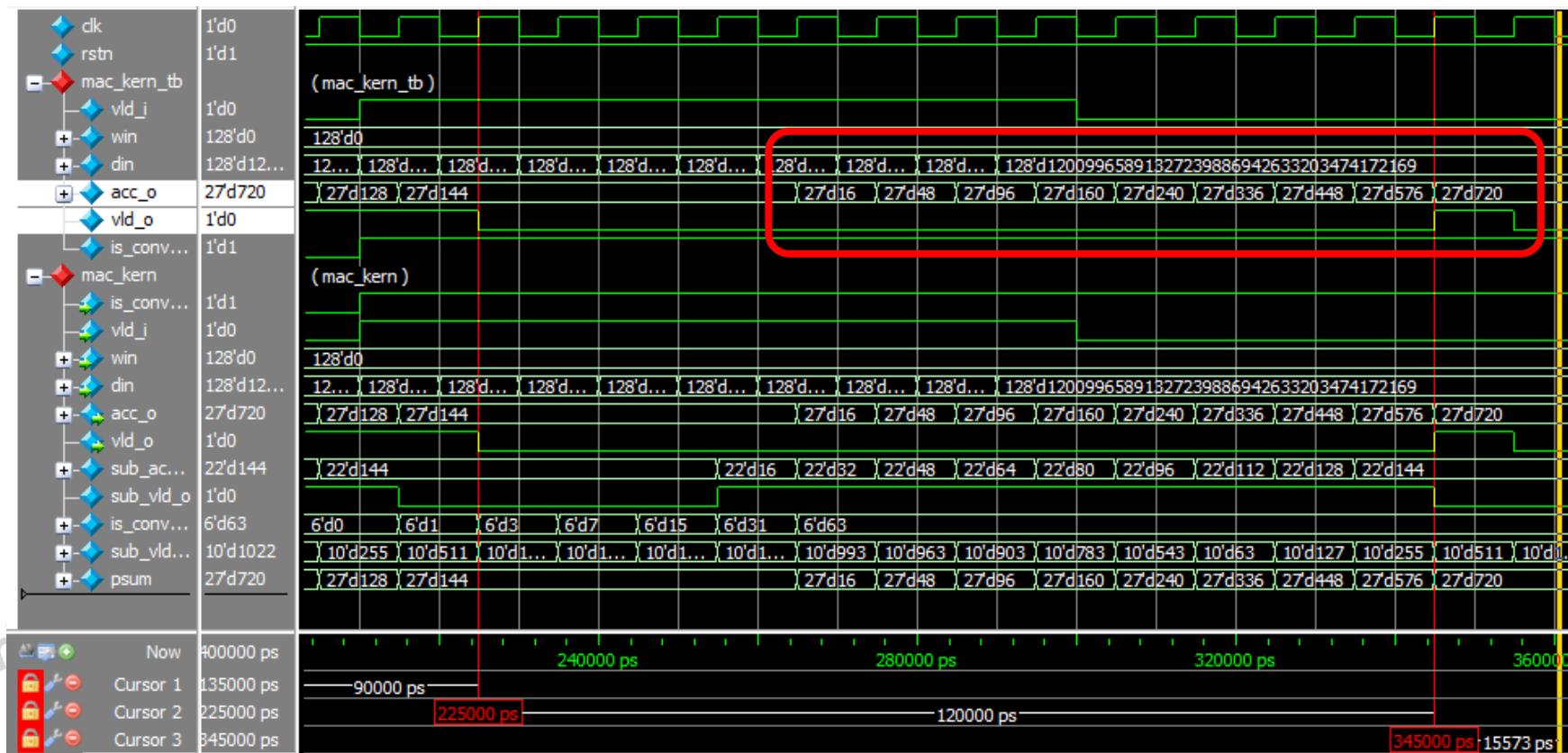
Waveform

- Conv1x1: we can output a pixel **every cycle**



Waveform

- Conv3x3: we can output a pixel **every 9 cycles**
 - It takes 9 cycles to give an output.



To do ...

- Use mac.v in the previous exercise
- Complete the missing codes in mac_kern.v
- Do a simulation with time = 400 ns
- Show the waveform
- Explain the output result

Road map

Review

MAC

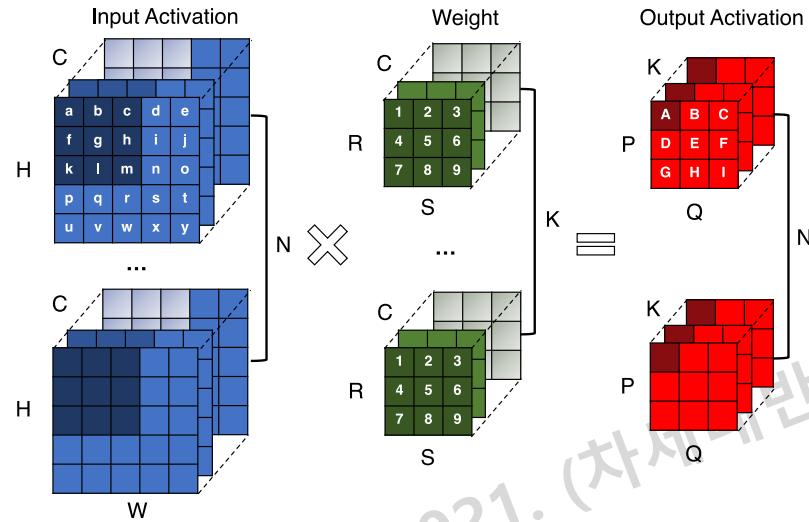
Mac kernel
(Accumulation)

Convolution kernel
(Normalization
& Activation)

Lab 3: Convolution kernel

- Lab 3: Implement a convolution kernel
 - Use mac.v, mac_kern.v in the previous exercises
 - Complete the missing codes in bias_shifter, act_shifter.v and conv_kern_tb.v
 - Do simulation with time = 250ns
 - Show the waveform

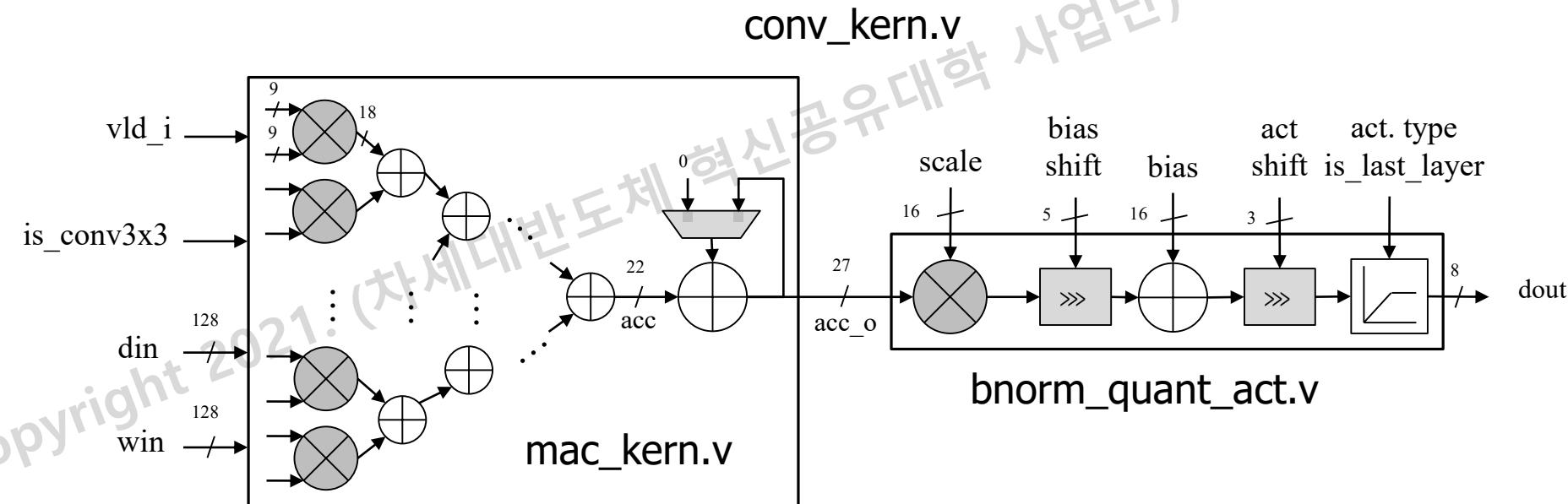
CONV loop



```
for (n=0; n<N; n++) {  
    for (k=0; k<K; k++) {  
        for (p=0; p<P; p++) {  
            for (q=0; q<Q; q++) {  
                OA[n][k][p][q] = 0;  
                for (r=0; r<R; r++) {  
                    for (s=0; s<S; s++) {  
                        for (c=0; c<C; c++) {  
                            h = p * stride - pad + r;  
                            w = q * stride - pad + s;  
                            OA[n][k][p][q] += IA[n][c][h][w] * W[k][c][r][s];  
                        }  
                    }  
                }  
                OA[n][k][p][q] = Activation(OA[n][k][p][q]);  
            }  
        }  
    }  
}
```

Convolutional kernel (conv_kern.v)

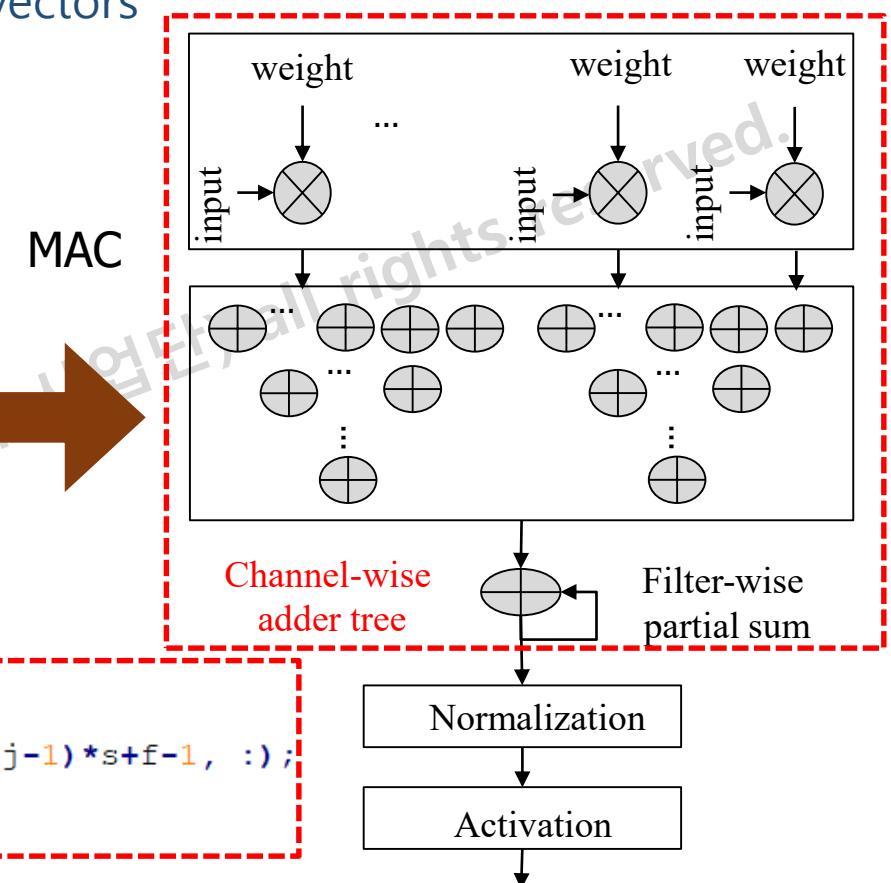
- Convolution kernel
 - Do multiplication and accumulation (mac_kern.v)
 - Do normalization/scaling, then add a bias
 - Do activation quantization



Convolution

- The operation is to do an element-wise product of two vectors
 - Layer 1: $N = 9$, layer 2, and 3: $N = 144$.
 - Multiplication: $y_1=w_1*x_1, y_2=w_2*x_2, \dots, y_N=w_N*x_N$
 - Accumulation: $y = y_1+y_2+\dots+y_N$

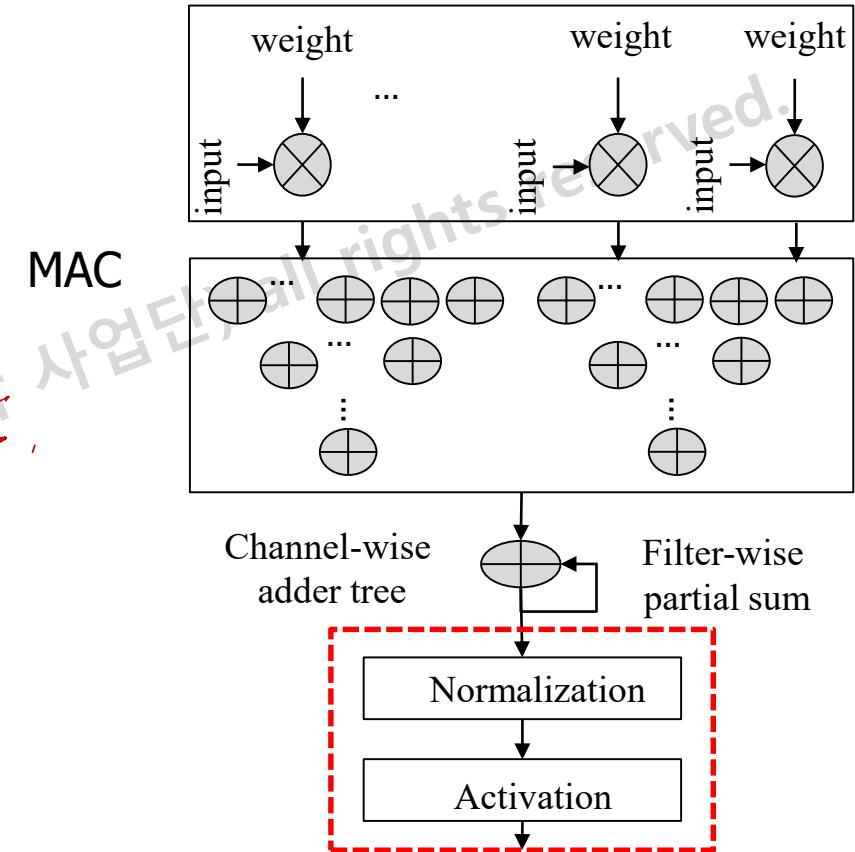
```
% Convolution
for k = 1:c_out
    for i = 1:h_out
        for j = 1:w_out
            % Element-wise Multiplication => Products
            scalar = kernel(:,:, :, k) .* ...
                pad_img(1+(i-1)*s:1+(i-1)*s+f-1, 1+(j-1)*s:1+(j-1)*s+f-1, :);
            % Sum (Sum of Products)
            out(i,j,k) = sum(scalar(:));
        end
    end
end
```



Normalization and quantization

- Activation quantization

- Multiply by a scale: $ys = y * \text{scale}$
- Adding a bias: $z = (ys >>> \text{bias_shift}) + b$
 - bias_shift is used to make a valid addition, i.e. same binary point
- Activation: $\text{out} = f(z) >>> \text{act_shift}$
 - act_shift is used to quantize the final output, i.e. 8 bits.



Convolutional kernel (conv_kern.v)

- Inputs
 - Ports are connected to the MAC kernel
 - is_conv3x3, vld_i, win, din
 - Ports are connected batch-norm, activation and quantization module
 - is_last_layer (RELU/Linear activation)
 - scale, bias, bias_shift, act_shift
- Outputs
 - acc_o and vld_o

```
module conv_kern(  
    clk,  
    rstn,  
    is_last_layer,  
    scale,  
    bias,  
    act_shift,  
    bias_shift,  
    is_conv3x3,           //0: 1x1, 1:3x3  
    vld_i,  
    win,  
    din,  
    acc_o,  
    vld_o  
);
```

Convolutional kernel (conv_kern.v)

- Two modules
- mac_kern.v
 - Do convolution
- bnorm_quant_act.v
 - Do batch normalization.
 - Adding a bias.
 - Do activation quantization
- An accumulated result of mac_kern is an input of bnorm_quant_act.v

```
//-----  
// Component: MAC  
//-----  
// DUT  
mac_kern u_mac_kern(  
    /*input*/          clk(clk),  
    /*input*/          rstn(rstn),  
    /*input*/          is_conv3x3(is_conv3x3),  
    /*input*/          vld_i(vld_i),  
    /*input [N*WI-1:0]*/ win(win),  
    /*input [N*WI-1:0]*/ din(din),  
    /*output [WO-1:0] */ acc_o(mac_kern_acc_o),  
    /*output reg */   vld_o(mac_kern_acc_vld_o)  
);  
//-----  
// Component: Batch-normalization, Activation quantization  
//-----  
bnorm_quant_act #( .DATA_BITS(DATA_BITS) )  
u_bnorm_quant_act  
(  
    /*input*/          clk(clk),  
    /*input*/          resetn(rstn),  
    /*input*/          is_last_layer(is_last_layer),  
    /*input [PARAM_BITS-1:0] */ scale(scale),  
    /*input [PARAM_BITS-1:0] */ bias(bias),  
    /*input [2:0] */      act_shift(act_shift),  
    /*input [4:0] */      bias_shift(bias_shift),  
    /*input [DATA_BITS-1:0] */ accum_in(mac_kern_acc_o),  
    /*input */          accum_vld_in(mac_kern_acc_vld_o),  
    /*output [ACT_BITS-1:0] */ accum_out(acc_o),  
    /*output */          accum_vld_out(vld_o)  
);
```

bnorm_quant_act.v

- Do batch normalization.
- Adding a bias.
- Do activation quantization

```
module bnorm_quant_act(
    clk,
    resetn,
    is_last_layer,
    scale,
    bias,
    act_shift,
    bias_shift,
    accum_in,
    accum_vld_in,
    accum_out,
    accum_vld_out
);
```

Copyright 2021. (차세대반도체 혁신공유대학)

Batch normalization/Scaling

- bnorm_quant_act.v
 - Do batch normalization/scaling: An accumulated result from MAC is multiplied by a scale
 - Adding a bias.
 - Do activation quantization

```
//-----
// Batch normalization
//-----
always @ (posedge clk)
begin
    accum_in_r <= accum_in;
    act_shift_r <= act_shift;
    bias_shift_r <= bias_shift;
    scale_r <= scale;
    bias_r <= bias;
end

always @ (posedge clk)
begin
    acc_mult <= $signed(accum_in_r) * $signed(scale_r);
end
```

Copyright 2021.

Adding bias

- bnorm_quant_act.v
 - Do batch normalization.
 - Adding a bias.
 - bias_shift is used to make a valid addition, i.e. same binary point
 - Shift right is used in bias_shifter.v
 - Do activation quantization

```
//-----
// Bias shift
//-----
bias_shifter #(
    .DATA_BITS (DATA_BITS+PARAM_BITS),
    .SHIFT_W (5)
)
u_bias_shift (
    .d_in (acc_mult),
    .n_shift (bias_shift_r),
    .d_out (acc_mult_shift)
);

always @(posedge clk) begin
    acc_mean <= $signed(acc_mult_shift) + $signed(bias_r);
end
```

Activation

- Do activation and quantization
 - act_shift is used to quantize the final output, i.e. 8 bits.

```
//-----  
// Activation Shift  
//-----  
  
act_shifter #(  
    .DATA_BITS (DATA_BITS),  
    .SHIFT_W (3)  
)  
u_act_shift (  
    .d_in (acc_mean),  
    .n_shift(act_shift_r),  
    .d_out (acc_mean_shift)  
)  
  
// RELU  
assign acc_relu = $signed(~acc_mean_shift[DATA_BITS-1] ? acc_mean_shift : 'h0);  
assign acc_quant = (~is_last_layer) ? acc_relu : acc_mean_shift; //linear  
always @(posedge clk or negedge resetn)  
    if(!resetn)  
        accum_relu <= 'h0;  
    else begin  
        // Clipping  
        accum_relu <= (acc_quant > 255) ? 255 : acc_quant[ACT_BITS-1:0];  
    end  
    .....  
// Linear (Last Layer)  
always @(posedge clk or negedge resetn)  
    if(!resetn) accum_final <= 'h0;  
    else      accum_final <= acc_quant;  
.....
```

Copyright 2021. (차세대반

Outputs

- Generate outputs
 - A quantized output (accum_out)
 - A valid signal (accum_vld_out)

```
//-----
// Delays and valid signals
//-----
assign accum_out = is_last_layer ? accum_final : accum_relu;
always @(posedge clk or negedge resetn)
begin
    if(!resetn) accum_vld <= 'h0;
    else        accum_vld <= {accum_vld[N_DELAY-2:0], accum_vld_in};
end
assign accum_vld_out = accum_vld[N_DELAY-1];
```

To do ...

- Complete the missing codes of bias_shifter.v

```
-----  
// Main body  
//-----  
// NOTE: Use a barrel shifter for further optimization  
    always @*  
begin  
    // d_out_r <= 'h0;  
    case (n_shift)  
        'd5 : d_out_r = {{ 5{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:5]};  
        'd6 : d_out_r = {{ 6{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:6]};  
        'd7 : d_out_r = {{ 7{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:7]};  
        //'d8 : d_out_r = /*Insert your code*/;  
        //'d9 : d_out_r = /*Insert your code*/;  
        //'d10: d_out_r = /*Insert your code*/;  
        //'d11: d_out_r = /*Insert your code*/;  
        'd12: d_out_r = {{12{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:12]};  
        'd13: d_out_r = {{13{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:13]};  
        'd14: d_out_r = {{14{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:14]};  
        'd15: d_out_r = {{15{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:15]};  
        'd16: d_out_r = {{16{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:16]};  
        'd17: d_out_r = {{17{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:17]};  
        'd18: d_out_r = {{18{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:18]};  
        'd19: d_out_r = {{19{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:19]};  
        'd20: d_out_r = {{20{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:20]};  
        'd21: d_out_r = {{21{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:21]};  
        'd22: d_out_r = {{22{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:22]};  
        'd23: d_out_r = {{23{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:23]};  
        'd24: d_out_r = {{24{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:24]};  
        'd25: d_out_r = {{25{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:25]};  
    default: d_out_r = 'h0;  
    endcase  
end
```

To do ...

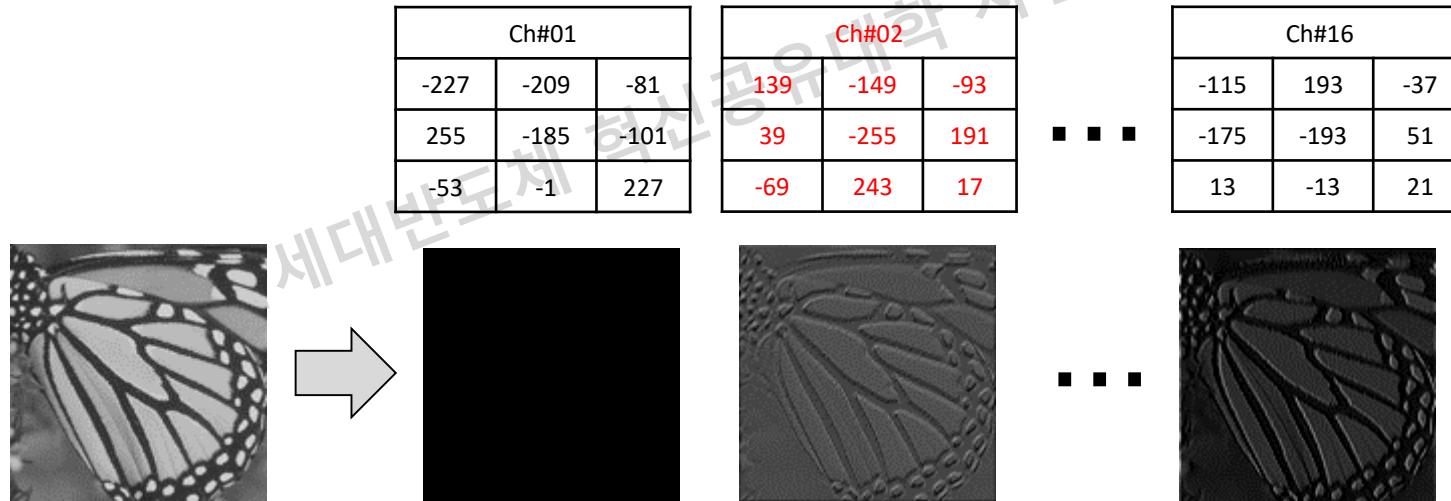
- Complete the missing codes of `act_shifter.v`

```
//-----
// Main body
//-----
// NOTE: Use a barrel shifter for further optimization
always @*
begin
    case (n_shift)
        'd0: d_out_r = d_in;
        'd1: d_out_r = {{1{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:1]};
        'd2: d_out_r = {{2{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:2]};
        'd3: d_out_r = {{3{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:3]};
        'd4: d_out_r = {{4{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:4]};
        'd5: d_out_r = {{5{d_in[DATA_BITS-1]}}, d_in[DATA_BITS-1:5]};
        //'d6: d_out_r = /*Insert your code*/;
        //'d7: d_out_r = /*Insert your code*/;
        default: d_out_r = 'h0;
    endcase
end
```

Copyright 2

Test bench (see reference S/W)

- Build a CNN computing unit and its test bench
- Example: test the first layer
 - Input image
 - Output a feature map, i.e., channel 2.



Test bench (see reference S/W)

- Input pixels in the top-left corner.



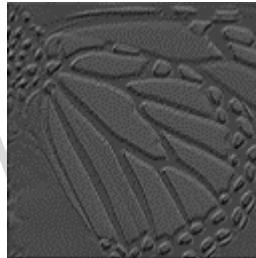
42	69	91	99	106	108	111
105	42	56	84	106	113	112
72	43	42	68	109	112	104

- Filter of channel 2

Ch#02		
139	-149	-93
39	-255	191
-69	243	17

Ch#02 (stored)		
69	181	209
19	128	95
221	121	8

- Output feature map in the top-left corner



108	71	79	89	93	94
42	58	60	70	77	73
33	81	65	69	66	58

Test bench (conv_kern_tb.v)

- Test conv_kern.v
- Define signals and parameters
- Add a DUT (conv_kern)

```
// DUT
conv_kern u_conv_kern(
    /*input          */clk(clk),
    /*input          */rstn(rstn),
    /*input          */is_last_layer(is_last_layer),
    /*input [PARAM_BITS-1:0] */scale(scale),
    /*input [PARAM_BITS-1:0] */bias(bias),
    /*input [2:0]        */act_shift(act_shift),
    /*input [4:0]        */bias_shift(bias_shift),
    /*input            */is_conv3x3(is_conv3x3),
    /*input            */vld_i(vld_i),
    /*input [N*WI-1:0]   */win(win),
    /*input [N*WI-1:0]   */din(din),
    /*output [ACT_BITS-1:0] */acc_o(acc_o),
    /*output           */vld_o(vld_o)
);
);
```

```
module conv_kern_tb;
parameter WI = 8;
parameter N = 16;
parameter WN = $clog2(N);
parameter WO = 2*(WI+1) + WN;
parameter PARAM_BITS = 16;
parameter WEIGHT_BITS = 8;
parameter ACT_BITS = 8;
parameter DATA_BITS = WO;

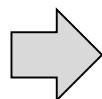
localparam CONV3x3_DELAY = 9;
localparam CONV3x3_DELAY_W = 4;

reg clk;
reg rstn;
reg is_last_layer;
reg [PARAM_BITS-1:0] scale;
reg [PARAM_BITS-1:0] bias;
reg [2:0] act_shift;
reg [4:0] bias_shift;
reg is_conv3x3; //0: 1x1, 1:3x3
reg vld_i;
reg [N*WI-1:0] win;
reg [N*WI-1:0] din;
wire [ACT_BITS-1:0] acc_o;
wire vld_o;
```

Test case

- Test the filter channel 2 of the first layer
 - Bias 8066, scale 103
 - bias_shift = 9
 - act_shift = 7

Ch#02 (stored)		
69	181	209
19	128	95
221	121	8



```
// Test cases
initial begin
    rstn = 1'b0;                                // Reset, low active
    vld_i = 1'b0;
    win = 0;
    din = 0;
    is_conv3x3 = 1'b0;
    is_last_layer = 1'b0;
    scale = 16'd103;
    bias = 16'd8066;
    bias_shift = 9;
    act_shift = 7;
    #(4*CLK_PERIOD) rstn = 1'b1;

    // First layer, channel 2:
    win[0*WI+:WI] = 8'd69;
    win[1*WI+:WI] = 8'd181;
    win[2*WI+:WI] = 8'd209;
    win[3*WI+:WI] = 8'd19;
    win[4*WI+:WI] = 8'd128;
    win[5*WI+:WI] = 8'd95;
    win[6*WI+:WI] = 8'd221;
    win[7*WI+:WI] = 8'd121;                      // This line is highlighted in blue
    win[8*WI+:WI] = 8'd8;
    // Test case 1: test convlxl
    is_conv3x3 = 1'b0;
```

Test case

- A 3x3 window

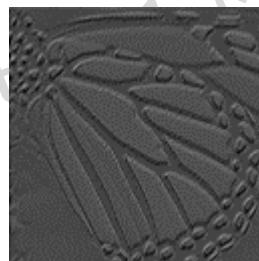
Input



0	0	0	91	99	106	108	111
0	42	69	56	84	106	113	112
0	105	42	42	68	109	112	104
72	43	42					

```
#(4*CLK_PERIOD) vld_i = 1'b1;  
din[0*WI+:WI] = 8'd0;  
din[1*WI+:WI] = 8'd0;  
din[2*WI+:WI] = 8'd0;  
din[3*WI+:WI] = 8'd0;  
din[4*WI+:WI] = 8'd42;  
din[5*WI+:WI] = 8'd69;  
din[6*WI+:WI] = 8'd0;  
din[7*WI+:WI] = 8'd105;  
din[8*WI+:WI] = 8'd42;
```

Output



108	71	79	89	93	94
42	58	60	70	77	73
33	81	65	69	66	58

Test case

- A 3x3 window

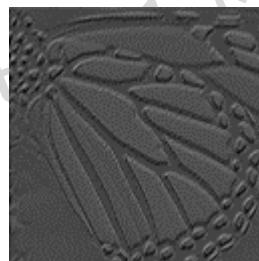
Input



0	0	0	99	106	108	111
42	69	91	84	106	113	112
105	42	56	68	109	112	104
72	43	42				

```
#(CLK_PERIOD) vld_i = 1'b1;  
din[0*WI+:WI] = 8'd0;  
din[1*WI+:WI] = 8'd0;  
din[2*WI+:WI] = 8'd0;  
din[3*WI+:WI] = 8'd42;  
din[4*WI+:WI] = 8'd69;  
din[5*WI+:WI] = 8'd91;  
din[6*WI+:WI] = 8'd105;  
din[7*WI+:WI] = 8'd42;  
din[8*WI+:WI] = 8'd56;
```

Output



108	71	79	89	93	94
42	58	60	70	77	73
33	81	65	69	66	58

Test case

- A 3x3 window

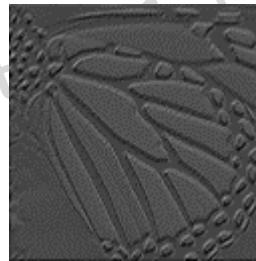
Input



		0	0	0		
42	69	91	99		106	108
105	42	56	84		106	113
72	43	42	68		109	112

```
#(CLK_PERIOD) vld_i = 1'b1;  
din[0*WI+:WI] = 8'd0;  
din[1*WI+:WI] = 8'd0;  
din[2*WI+:WI] = 8'd0;  
din[3*WI+:WI] = 8'd69;  
din[4*WI+:WI] = 8'd91;  
din[5*WI+:WI] = 8'd99;  
din[6*WI+:WI] = 8'd42;  
din[7*WI+:WI] = 8'd56;  
din[8*WI+:WI] = 8'd84;
```

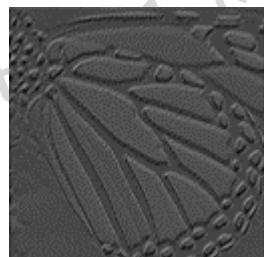
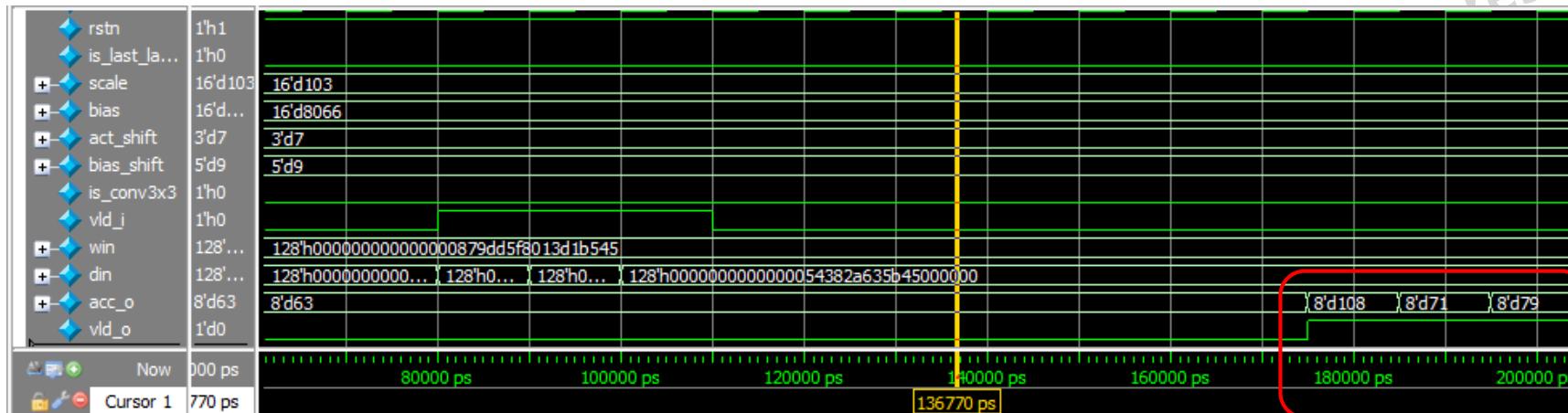
Output



108	71	79	89	93	94
42	58	60	70	77	73
33	81	65	69	66	58

Waveform

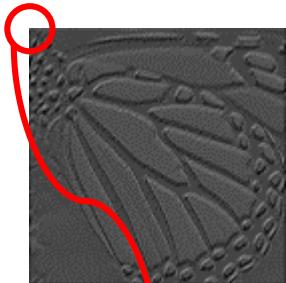
- Do a simulation with time = 250 ns
- Show the waveform



108	71	79	89	93	94
42	58	60	70	77	73
33	81	65	69	66	58

To do ...

- Complete the missing codes in conv_kern_tb.v to compute three output pixels



108	71	79	89	93	94
42	58	60	70	77	73
33	81	65	69	66	58

```
#(CLK_PERIOD) vld_i = 1'b1;  
//din[0*WI+:WI] = /* Insert your code*/;  
//din[1*WI+:WI] = /* Insert your code*/;  
//din[2*WI+:WI] = /* Insert your code*/;  
//din[3*WI+:WI] = /* Insert your code*/;  
//din[4*WI+:WI] = /* Insert your code*/;  
//din[5*WI+:WI] = /* Insert your code*/;  
//din[6*WI+:WI] = /* Insert your code*/;  
//din[7*WI+:WI] = /* Insert your code*/;  
//din[8*WI+:WI] = /* Insert your code*/;  
vld_i = 1'b1;  
//din[0*WI+:WI] = /* Insert your code*/;  
//din[1*WI+:WI] = /* Insert your code*/;  
//din[2*WI+:WI] = /* Insert your code*/;  
//din[3*WI+:WI] = /* Insert your code*/;  
//din[4*WI+:WI] = /* Insert your code*/;  
//din[5*WI+:WI] = /* Insert your code*/;  
//din[6*WI+:WI] = /* Insert your code*/;  
//din[7*WI+:WI] = /* Insert your code*/;  
//din[8*WI+:WI] = /* Insert your code*/;  
vld_i = 1'b1;  
//din[0*WI+:WI] = /* Insert your code*/;  
//din[1*WI+:WI] = /* Insert your code*/;  
//din[2*WI+:WI] = /* Insert your code*/;  
//din[3*WI+:WI] = /* Insert your code*/;  
//din[4*WI+:WI] = /* Insert your code*/;  
//din[5*WI+:WI] = /* Insert your code*/;  
//din[6*WI+:WI] = /* Insert your code*/;  
//din[7*WI+:WI] = /* Insert your code*/;  
//din[8*WI+:WI] = /* Insert your code*/;
```

To do ...

- Use mac.v, mac_kern.v in the previous exercises
- Complete the missing codes in bias_shifter, act_shifter.v and conv_kern_tb.v
- Do a simulation with time = 250 ns
- Show the waveform

Copyright 2021. (차세대반도체 혁신공유대학 사업단) all rights reserved.