차세대 반도체 디지털 혁신공유대학 사업단

POLARIS

Hardware Description Language Computing Units

2022.02.17 (Thu)

Road map

Review

Verilog HDL

Computing Unit

Appendix

Motivating example: Layer 0

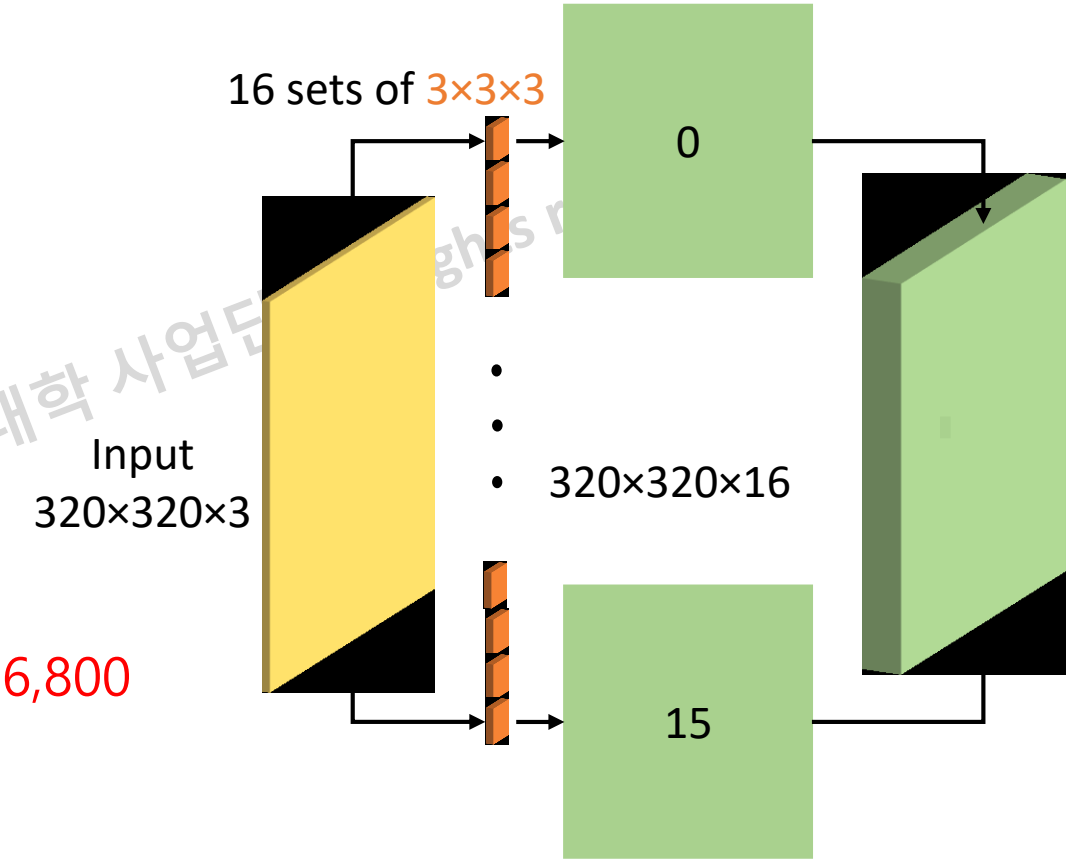
- The number of output pixels is 1,638,400 (=320×320×16)
  - Each output is calculated by

$$y = \sum_{i=0}^{3\times3\times3-1} W_i * x_i$$

Where W and x are weights and inputs

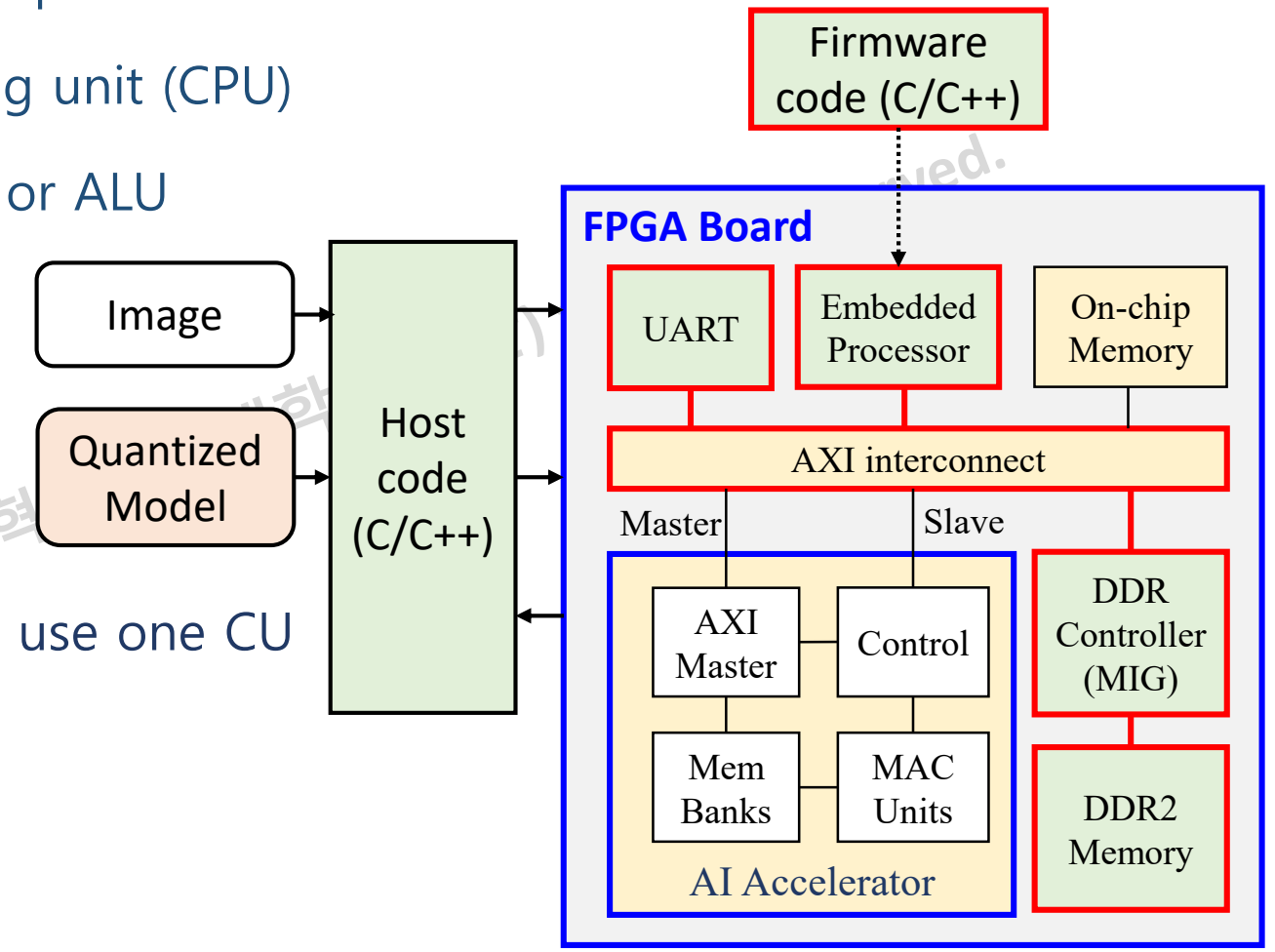
$\Rightarrow$ 27 multiplication operations

- The no. of multiplication operations is 44,236,800 (=320×320×16×27)



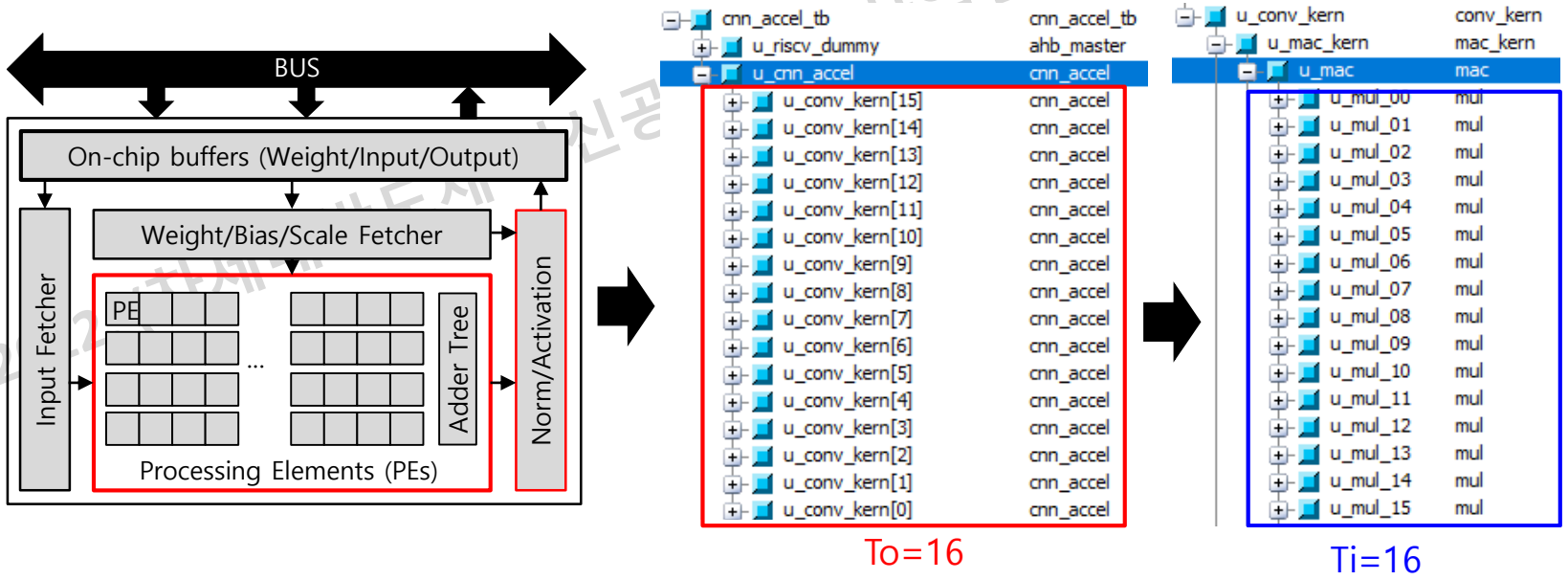
Accelerator

- Conventional general purpose computer
  - Processor or central processing unit (CPU)
    - One computing unit (CU) or ALU
  - Memory
  - Input/Output (IO)
- Performance issue:
  - Example: **44,236,800** times to use one CU
- How to boost the performance?



Accelerator

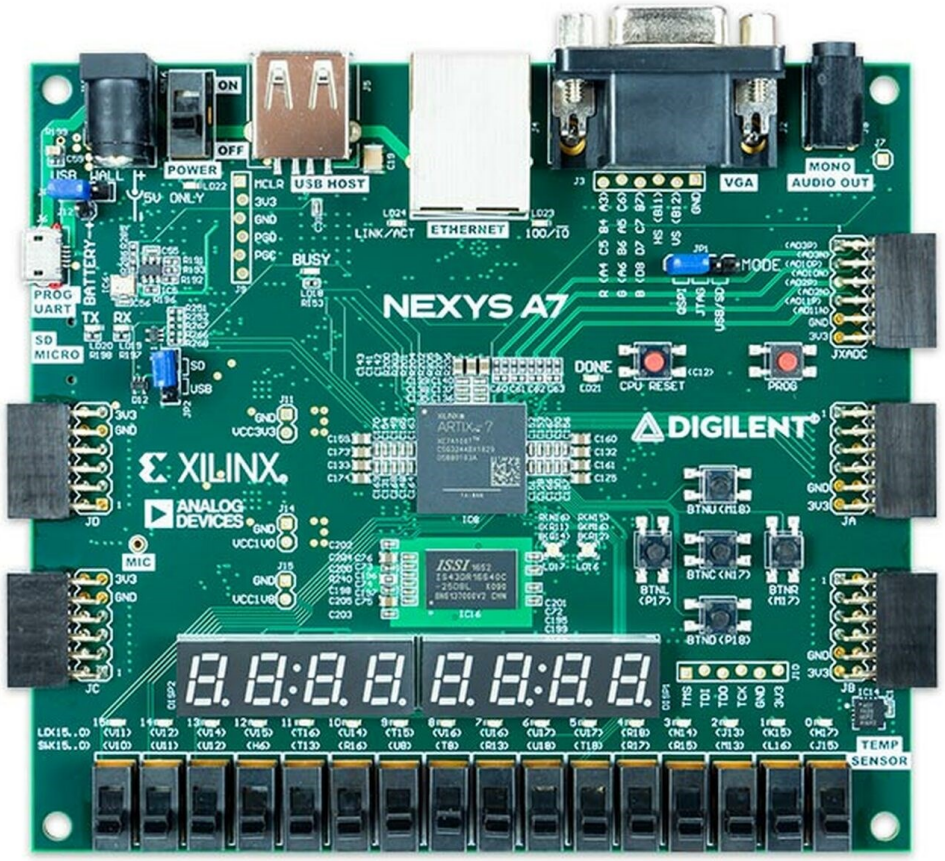
- Processing Element (PE) Array
  - Perform convolution/activation/quantization operations.
  - To: The number of convolutional kernels (output feature maps)
  - Ti: The number of multipliers in a kernel
- The number of multipliers is : To× Ti
  - **256 (=16×16) multipliers run in parallel**



Nexys A7 FPGA board

- Xilinx Artix-7 FPGA XC7A100T-1CSG324C
- 15,850 logic slices
  - Each with four 6-input LUTs and 8 FFs
- 4,860 Kbits of fast block RAM
- 240 DSP slices (constrained to To, Ti)
  - Dedicated to multiplication and accumulation (MAC)
- Internal clock speeds exceeding 450 MHz
- 128 MB DDR2 Memory
- USB-JTAG port for FPGA programming and communication

⇒ requires some techniques (e.g., quantization) to accelerate a network interface



Quantization

- A quantization scheme be an affine mapping of integers q to real numbers r, i.e. of the form
$$r = S(q - Z), \text{ For some constants } S \text{ and } Z$$
- For 8-bit quantization, q is quantized as an 8-bit integer
  - For B-bit quantization, q is quantized as an B-bit integer.
  - Some arrays, typically bias vectors, are quantized as 16/32-bit integers
- The constant S (for "scale") is an arbitrary positive real number.
- The constant Z (for "zero-point") is of the same type as quantized values q, and is in fact the quantized value q corresponding to the real value 0.

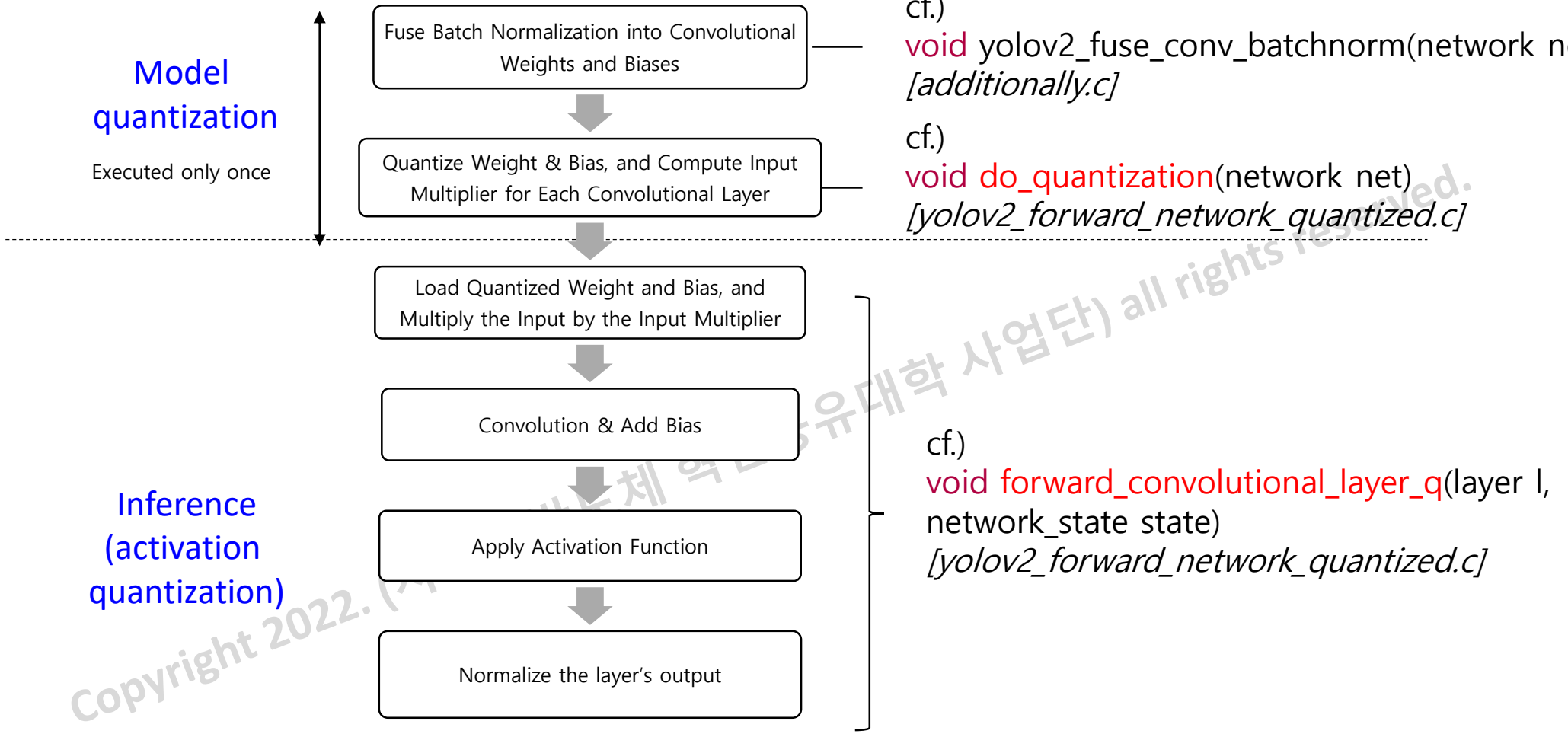
```
template<typename QType>  // e.g. QType=uint8
struct QuantizedBuffer {
  vector<QType> q;        // the quantized values
  float S;                // the scale
  QType Z;                // the zero-point
};
```


B. Jacob, et. al, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference", CVPR 2018.

Example

- Mapping values from fp32 to a 8-bit integer format
  - All quantized values are in {-128, -127, …, 127}

Convolutional layer in the Quantized Model



mAP comparison

```
class_id = 30, name = coffee_mate_french_vanilla,        ap = 97.68 %
class_id = 31, name = pepperidge_farm_milk_chocolate_macadamia_cookies,          ap = 71.63
class_id = 32, name = kitkat_king_size,          ap = 85.32 %
class_id = 33, name = snickers,          ap = 36.60 %
class_id = 34, name = toblerone_milk_chocolate,          ap = 97.04 %
class_id = 35, name = clif_z_bar_chocolate_chip,         ap = 98.70 %
class_id = 36, name = nature_valley_crunchy_oats_n_honey,        ap = 72.28 %
class_id = 37, name = ritz_crackers,     ap = 97.73 %
class_id = 38, name = palmolive_orange,          ap = 55.47 %
class_id = 39, name = crystal_hot_sauce,         ap = 100.00 %
class_id = 40, name = tapatio_hot_sauce,         ap = 0.00 %
class_id = 41, name = nabisco_nilla_wafers,      ap = 0.00 %
class_id = 42, name = pepperidge_farm_milano_cookies_double_chocolate,   ap = 0.00 %
class_id = 43, name = campbells_chicken_noodle_soup,     ap = 0.00 %
class_id = 44, name = frappuccino_coffee,        ap = 0.00 %
class_id = 45, name = chewy_dips_chocolate_chip,         ap = 34.73 %
class_id = 46, name = chewy_dips_peanut_butter,          ap = 0.00 %
class_id = 47, name = nature_vally_fruit_and_nut,        ap = 0.00 %
class_id = 48, name = cheerios,          ap = 0.00 %
class_id = 49, name = lindt_excellence_cocoa_dark_chocolate,     ap = 0.00 %
class_id = 50, name = hersheys_symphony,         ap = 0.00 %
class_id = 51, name = campbells_chunky_classic_chicken_noodle,   ap = 0.00 %
class_id = 52, name = martinellis_apple_juice,   ap = 0.00 %
class_id = 53, name = dove_pink,         ap = 0.00 %
class_id = 54, name = dove_white,        ap = 0.00 %
class_id = 55, name = david_sunflower_seeds,     ap = 0.00 %
class_id = 56, name = monster_energy,    ap = 0.00 %
class_id = 57, name = act_ii_butter_lovers_popcorn,      ap = 0.00 %
class_id = 58, name = coca_cola_glass_bottle,    ap = 0.00 %
class_id = 59, name = twix,      ap = 0.00 %
 for thresh = 0.24, precision = 0.83, recall = 0.71, F1-score = 0.76
 for thresh = 0.24, TP = 1362, FP = 282, FN = 562, average IoU = 62.79 %

 mean average precision (mAP) = 0.831957, or 83.20 %
Total Detection Time: 32.000000 Seconds
(base) truongnx@marlin:~/aix2022/skeleton-v1.1/bin$
```

```
class_id = 30, name = coffee_mate_french_vanilla,        ap = 55.10 %
class_id = 31, name = pepperidge_farm_milk_chocolate_macadamia_cookies,          ap = 0.26 %
class_id = 32, name = kitkat_king_size,          ap = 0.23 %
class_id = 33, name = snickers,          ap = 5.05 %
class_id = 34, name = toblerone_milk_chocolate,          ap = 20.10 %
class_id = 35, name = clif_z_bar_chocolate_chip,         ap = 59.39 %
class_id = 36, name = nature_valley_crunchy_oats_n_honey,        ap = 22.12 %
class_id = 37, name = ritz_crackers,     ap = 45.45 %
class_id = 38, name = palmolive_orange,          ap = 54.55 %
class_id = 39, name = crystal_hot_sauce,         ap = 0.00 %
class_id = 40, name = tapatio_hot_sauce,         ap = 0.00 %
class_id = 41, name = nabisco_nilla_wafers,      ap = 0.00 %
class_id = 42, name = pepperidge_farm_milano_cookies_double_chocolate,   ap = 0.00 %
class_id = 43, name = campbells_chicken_noodle_soup,     ap = 0.00 %
class_id = 44, name = frappuccino_coffee,        ap = 0.00 %
class_id = 45, name = chewy_dips_chocolate_chip,         ap = 0.00 %
class_id = 46, name = chewy_dips_peanut_butter,          ap = 0.00 %
class_id = 47, name = nature_vally_fruit_and_nut,        ap = 0.00 %
class_id = 48, name = cheerios,          ap = 0.00 %
class_id = 49, name = lindt_excellence_cocoa_dark_chocolate,     ap = 0.00 %
class_id = 50, name = hersheys_symphony,         ap = 0.00 %
class_id = 51, name = campbells_chunky_classic_chicken_noodle,   ap = 0.00 %
class_id = 52, name = martinellis_apple_juice,   ap = 0.00 %
class_id = 53, name = dove_pink,         ap = 0.00 %
class_id = 54, name = dove_white,        ap = 0.00 %
class_id = 55, name = david_sunflower_seeds,     ap = 0.00 %
class_id = 56, name = monster_energy,    ap = 0.00 %
class_id = 57, name = act_ii_butter_lovers_popcorn,      ap = 0.00 %
class_id = 58, name = coca_cola_glass_bottle,    ap = 0.00 %
class_id = 59, name = twix,      ap = 0.00 %
 for thresh = 0.24, precision = 0.93, recall = 0.03, F1-score = 0.05
 for thresh = 0.24, TP = 54, FP = 4, FN = 1870, average IoU = 57.32 %

 mean average precision (mAP) = 0.336020, or 33.60 %
Total Detection Time: 27.000000 Seconds
(base) truongnx@marlin:~/aix2022/skeleton-v1.1/bin$
```

FP model: 83.20%

Naïvely quantized INT8 model: 33.60%

Try to achieve higher mAP for the quantized model by implementing better quantization!

forward_convolutional_layer_q

- Processing steps
  - im2col_cpu_int8
  - Do convolution
  - Add a bias
  - Do activation
  - Normalization

```
136
137     // Use GEMM (as part of BLAS)
138     im2col_cpu_int8(state.input_int8, l.c, l.h, l.w, l.size, l.stride, l.pad, b);
139     int t;    // multi-thread gemm
140     #pragma omp parallel for
141     for (t = 0; t < m; ++t) {
142         gemm_nn_int8_int16(1, n, k, 1, a + t*k, k, b, n, c + t*n, n);
143     }
144     free(state.input_int8);
145
146     // Bias addition
147     int fil;
148     for (fil = 0; fil < l.n; ++fil) {
149         for (j = 0; j < out_size; ++j) {
150             output_q[fil*out_size + j] = output_q[fil*out_size + j] + l.biases_quant[fil];
151         }
152     }
153
154     // Activation
155     if (l.activation == LEAKY) {
156         for (i = 0; i < l.n*out_size; ++i) {
157             output_q[i] = (output_q[i] > 0) ? output_q[i] : output_q[i] / 10;
158         }
159     }
160
161     // De-scaling
162     float ALPHA1 = 1 / (l.input_quant_multiplier * l.weights_quant_multiplier);
163     for (i = 0; i < l.outputs; ++i) {
164         l.output[i] = output_q[i] * ALPHA1;
165     }
166
```

The accelerator should perform these operations

Data preparation

- Store weights and biases in hexadecimal files (32 bits per line)
  - RTL simulation
  - Read by the Host PC to send to the FPGA board
- You can adjust this code to save activations which are used for RTL verification

```
280  // Save quantized weights, bias, and scale
281  void save_quantized_model(network net) {
282      int j;
283      for (j = 0; j < net.n; ++j) {
284          layer *l = &net.layers[j];
285          if (l->type == CONVOLUTIONAL) {
286              size_t const weights_size = l->size*l->size*l->c*l->n;
287              size_t const filter_size = l->size*l->size*l->c;
288
289              printf(" Saving quantized weights, bias, and scale for CO
290
291              char weightfile[30];
292              char biasfile[30];
293              char scalefile[30];
294
295              sprintf(weightfile, "weights/CONV%d_W.txt", j);
296              sprintf(biasfile, "weights/CONV%d_B.txt", j);
297              sprintf(scalefile, "weights/CONV%d_S.txt", j);
298
```

```
FILE *fp_w = fopen(weightfile, "w");
for (k = 0; k < weights_size; k = k + 4) {
    uint8_t first = k < weights_size ? l->weights_int8[k] : 0;
    uint8_t second = k+1 < weights_size ? l->weights_int8[k+1] : 0;
    uint8_t third = k+2 < weights_size ? l->weights_int8[k+2] : 0;
    uint8_t fourth = k+3 < weights_size ? l->weights_int8[k+3] : 0;
    fprintf(fp_w, "%02x%02x%02x%02x\n", first, second, third, fourth);
}
fclose(fp_w);

FILE *fp_b = fopen(biasfile, "w");
for (k = 0; k < l->n; k = k + 4) {
    uint16_t first = k < l->n ? l->biases_quant[k] : 0;
    uint16_t second = k+1 < l->n ? l->biases_quant[k+1] : 0;
    fprintf(fp_b, "%04x%04x\n", first, second);
}
fclose(fp_b);
```

Road map

Review

Verilog HDL

Computing Unit

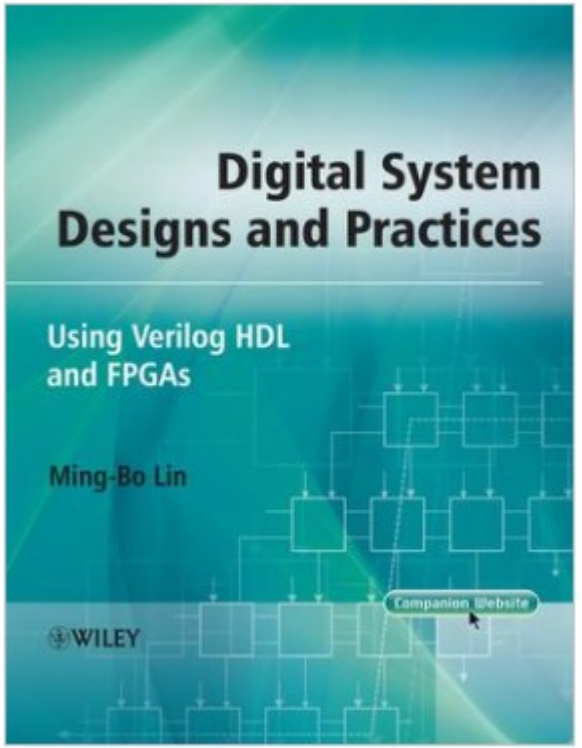
References

- Class: Digital Systems Design and Experiments (https://ocw.snu.ac.kr/node/2390)
- Books: Digital System Designs and Practices: Using Verilog HDL and FPGAs @Willy 2008

Digital Systems Design and Experiments

Lecture Notes | Calendar | Assignments | Exam | Study Materials

c_lecno	Title	files
1-1	Introduction	6625.pdf
1-2	Introduction / Digital Design Methodology	6626.pdf
2	Structural Modeling	6627.pdf
3	Dataflow Modeling	6628.pdf
4	Behavioral Modeling	6629.pdf
5	Tasks, Functions, and UDPs	6630.pdf
8	Combinational Logic Modules	6631.pdf
9	Sequential Logic Modules	6632.pdf
10	Design Options of Digital Systems	6633.pdf
11	System Design Methodology	6634.pdf
12	Synthesis	6635.pdf
13	Verification	6636.pdf
15-1	Design Examples	6637.pdf
15-2	Design Examples / CPU Design Example	6639.pdf
16	Design for Testability	6638.pdf

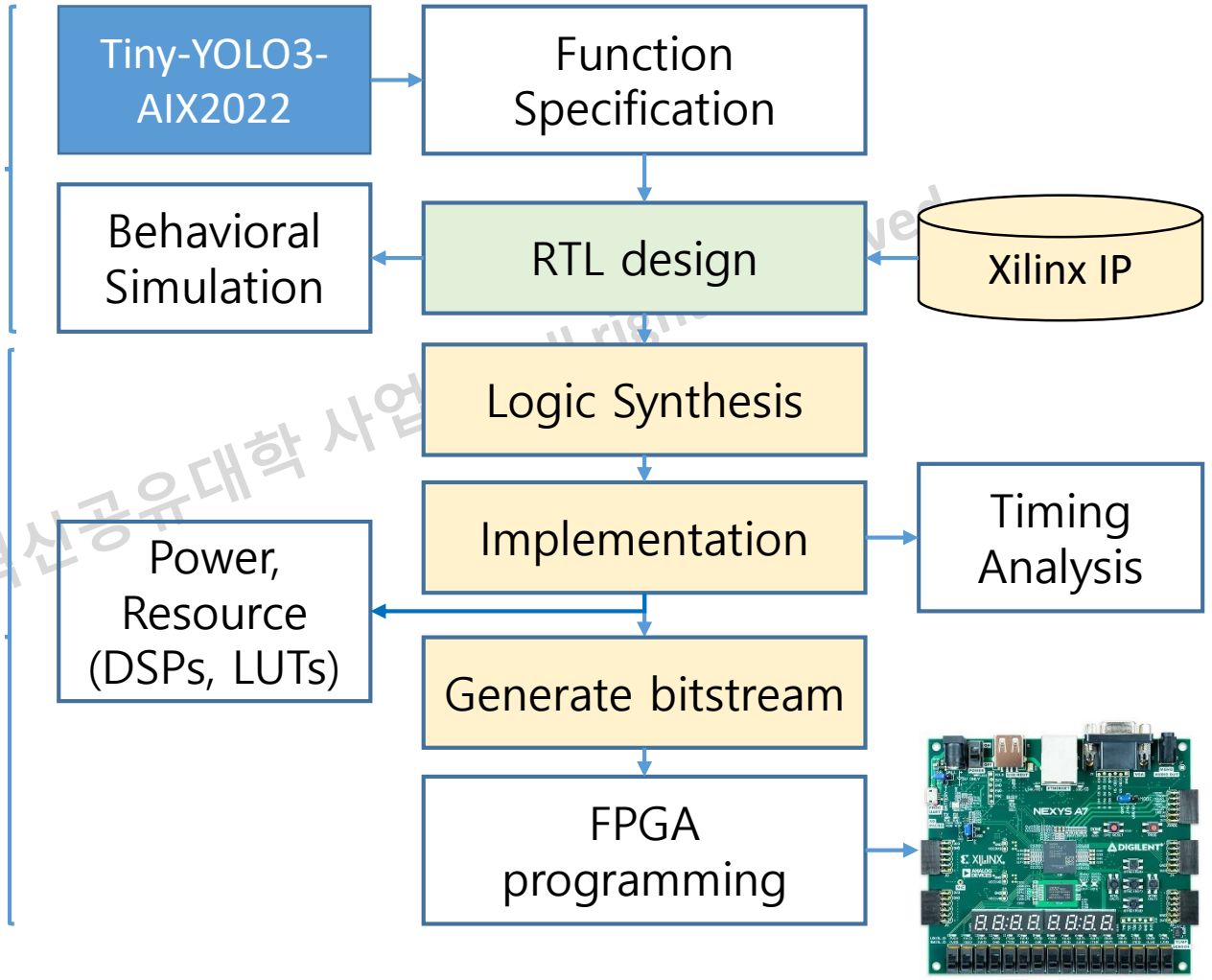


FPGA design flow

Function simulation
Text editor: *vim, notepad++, VS code*
RTL simulator: *ModelSim, ISE, Vivado*

FPGA Implementation (*ISE, Vivado*)
- Synthesize the design
- Implementation: mapping, placement and routing

Optimization
- Analyze timing, power, resources

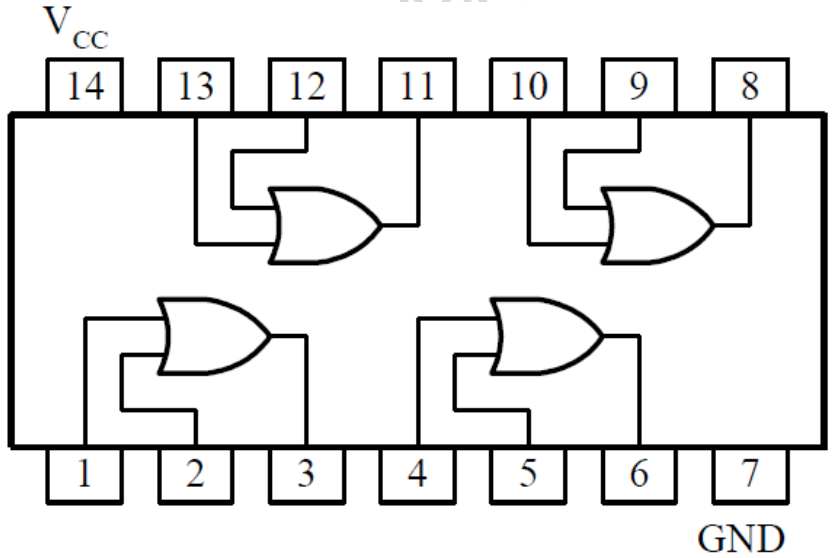


Hardware Description Language (HDL)

- HDL is an acronym of Hardware Description Language
- Two most commonly used HDLs:
  - Verilog HDL (also called Verilog for short)
  - VHDL (Very high-speed integrated circuits HDL)
- Features of HDLs:
  - Design can be described at a very abstract level.
  - Functional verification can be done early in the design cycle.
  - Designing with HDLs is analogous to computer programming.

Modules – Hardware Module Concept

- The basic unit of a digital system is a module.
- Each module consists of:
  - a core circuit (called internal or body) ---performs the required function
  - an interface (called ports) ---carries out the required communication between the core circuit and outside.



Modules –Verilog HDL modules

- module---The basic building block in Verilog HDL.
  - It can be an element or a collection of lower-level design blocks.

module Module name

Port List, Port Declarations (if any)

Parameters (if any)

Declarations of *wires*, *regs*, and other variables

Instantiation of lower level modules or primitives

Data flow statements (*assign*)

always and *initial* blocks. (all behavioral statements go into these blocks).

Tasks and functions.

endmodule statement

Lexical Conventions

- Sized number: <size>`<base format><number>
  - 4`b1001 ---a 4-bit binary number
  - 16`habcd ---a 16-bit hexadecimal number
- Unsized number: `<base format><number>
  - 2007 ---a 32-bit decimal number by default
  - `habc ---a 32-bit hexadecimal number
- x or z values: x denotes an unknown value; z denotes a high impedance value.
- Negative number: -<size>`<base format><number>
  - -4`b1001 ---a 4-bit binary number
  - -16`habcd ---a 16-bit hexadecimal number
- "_" and "?"
  - 16`b0101_1001_1110_0000
  - 8`b01??_11?? ---equivalent to a 8`b01zz_11zz

Data types

- A net variable
  - can be referenced anywhere in a module.
  - must be driven by a primitive, continuous assignment, force … release, or module port.

- A variable
  - can be referenced anywhere in a module.
  - can be assigned value only within a procedural statement, task, or function.
  - cannot be an input or inout port in a module.

Port Declaration

- Port Declaration
  - input: input ports.
  - output: output ports.
  - inout: bidirectional ports
- Port Connection Rules
  - Named association
  - Positional association

```
module half_adder (x, y, s, c);
input  x, y;
output s, c;
// -- half adder body-- //
// instantiate primitive gates
   xor xor1 (s, x, y);
   and and1 (c, x, y);
endmodule

module full_adder (x, y, cin, s, cout);
input  x, y, cin;
output s, cout;
wire   s1,c1,c2;  // outputs of both half adders
// -- full adder body-- //
// instantiate the half adder
   half_adder ha_1 (x, y, s1, c1);
   half_adder ha_2 (.x(cin), .y(s1), .s(s), .c(c2));
   or (cout, c1, c2);
endmodule
```

Can only be connected by using positional association

Instance name is optional.

Connecting by using positional association

Connecting by using named association

Instance name is necessary.

Module Modeling Styles

- Structural style
- Dataflow style
- Behavioral or algorithmic style
- Mixed style
- In industry, RTL (register-transfer level) means
  - RTL = synthesizable behavioral + dataflow constructs

Structural modeling

- Structural style
  - Gate level comprises a set of interconnected gate primitives.
  - Switch level consists of a set of interconnected switch primitives.

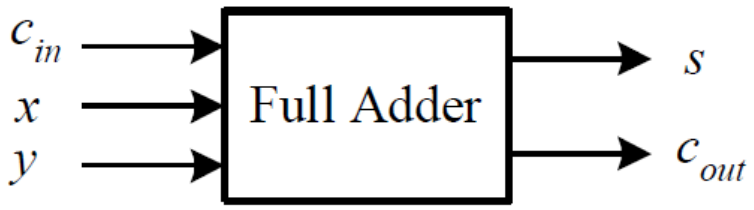
```
// gate-level hierarchical description of 4-bit adder
// gate-level description of half adder
module half_adder (x, y, s, c);
input x, y;
output s, c;
// half adder body
// instantiate primitive gates
xor (s,x,y);
and (c,x,y);
endmodule
```

```
// gate-level description of full adder
module full_adder (x, y, cin, s, cout);
input x, y, cin;
output s, cout;
wire s1, c1, c2; // outputs of both half adders
// full adder body
// instantiate the half adder
half_adder ha_1 (x, y, s1, c1);
half_adder ha_2 (cin, s1, s, c2);
or (cout, c1, c2);
endmodule
```

Hierarchical design

Dataflow modeling

- Dataflow style specifies the dataflow (i.e., data dependence) between registers.
- Use a set of continuous assignment statements
  - assign [delay] l_value = expression
  - delay: the amount of time between a change of operand used in expression and the assignment to l-value.
- **Continuous statement in a module execute concurrently regardless of the order they appear.**




```
module full_adder_dataflow(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
// specify the function of a full adder
assign #5 {c_out, sum} = x + y + c_in;
endmodule
```

Behavioral modeling

- Use two procedural constructs: initial and always
- initial statement
  - Executed only once at simulation time 0
  - Used to set up initial value of variable data types
- always statement
  - Executed repeatedly
- At simulation time 0, both initial and always statements are executed concurrently.

```
module full_adder_behavioral(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
reg sum, c_out; // sum and c_out need to be declared as reg types.
// specify the function of a full adder
always @(x, y, c_in)  //or always @(x or y or c_in)
#5 {c_out, sum} = x + y + c_in;
endmodule
```

```
module full_adder_behavioral(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
reg sum, c_out; // sum and c_out need to be declared as reg types.
// specify the function of a full adder
always @(*)
#5 {c_out, sum} = x + y + c_in;
endmodule
```

Mixed-Style Modeling

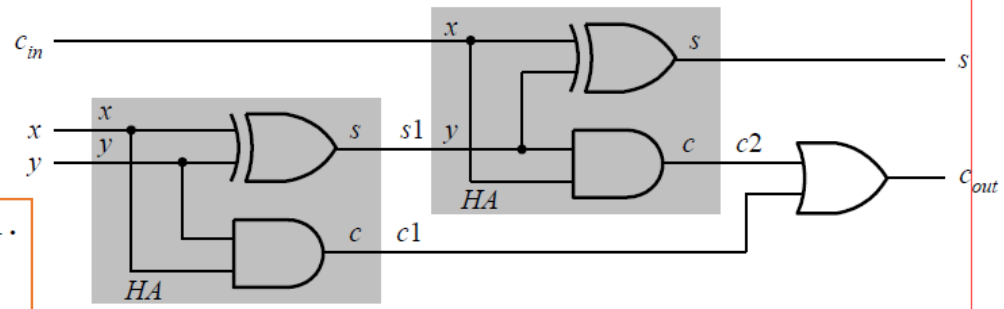
- Mixed style is the mixing use of above three modeling styles.
  - Commonly used in modeling large designs.

structural

dataflow

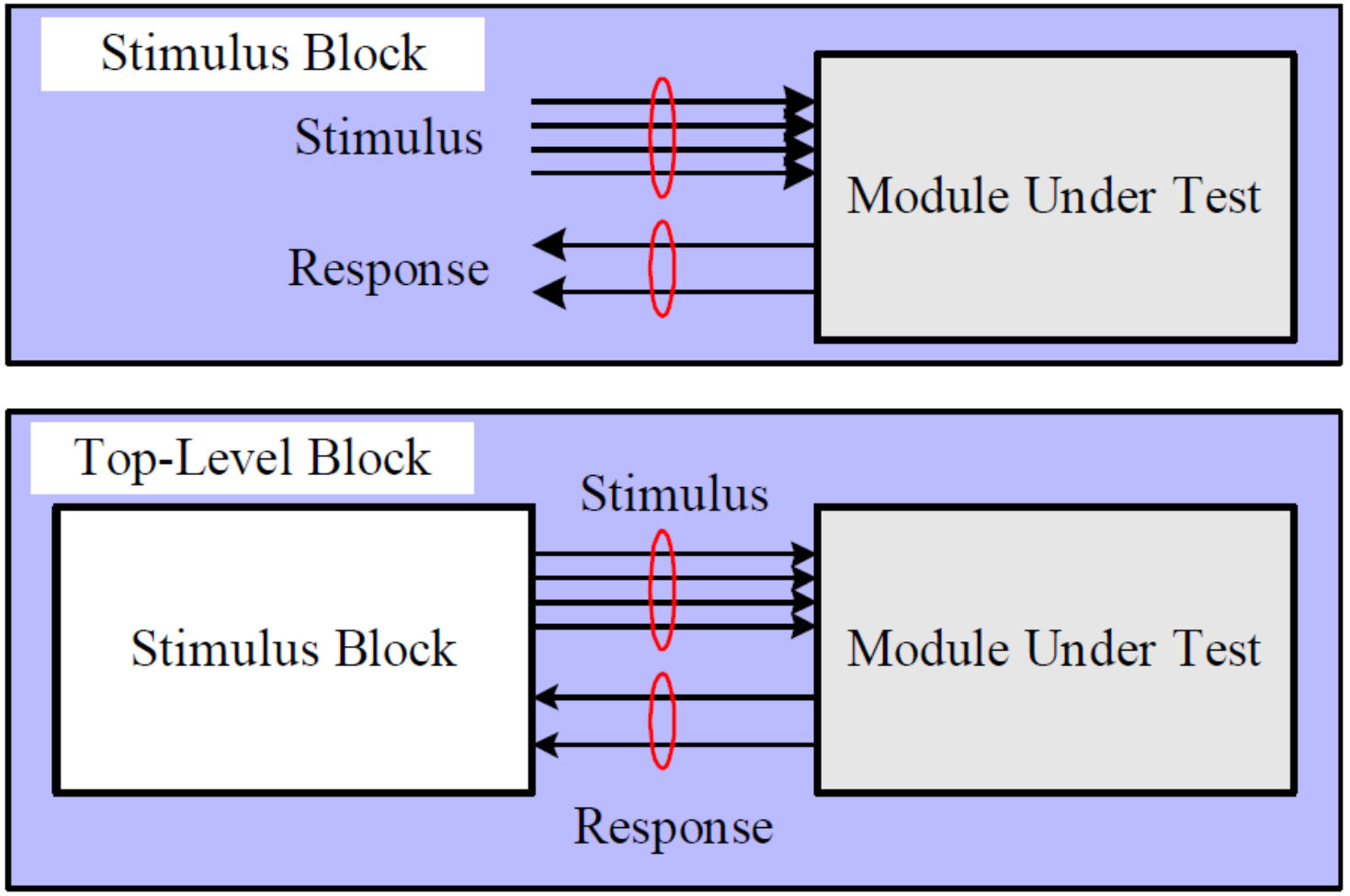
behavioral

```
module full_adder_mixed_style(x, y, c_in, s, c_out);
// I/O port declarations
input x, y, c_in;
output s, c_out;
reg c_out;
wire s1, c1, c2;
// structural modeling of HA 1.
xor xor_ha1 (s1, x, y);
and and_ha1(c1, x, y);
// dataflow modeling of HA 2.
assign s = c_in ^ s1;
assign c2 = c_in & s1;
// behavioral modeling of output OR gate.
always @(c1, c2) // can also use always @(*)
c_out = c1 | c2;
endmodule
```



Simulation

- Design
- Simulation
- Verification
- Stimulus block: testbench
- Unit under test (UUT)
- Design under test (DUT)



System Tasks for Simulation

- $display displays values of variables, string, or expressions
  - $display(*ep*1, *ep*2, …, *epn*);
  - *ep*1, *ep*2, …, *epn*: quoted strings, variables, expressions.
- $monitor monitors a signal when its value changes.
  - $monitor(*ep*1, *ep*2, …, *epn*);
- $monitoton enables monitoring operation.
- $monitotoff disables monitoring operation.
- $stop suspends the simulation.
- $finish terminates the simulation.

Time Scale for Simulations

- Time scale compiler directive
  - `timescale time_unit / time_precision
- The time_precision must not exceed the time_unit.
- Example:
  - with a timescale 1 ns/1 ps, the delay specification #15 corresponds to 15 ns.
- It uses the same time unit in both behavioral and gate-level modeling.
- For FPGA designs, it is suggested to use ns as the time unit.

Road map

Review

Verilog HDL

Computing Unit

Labs

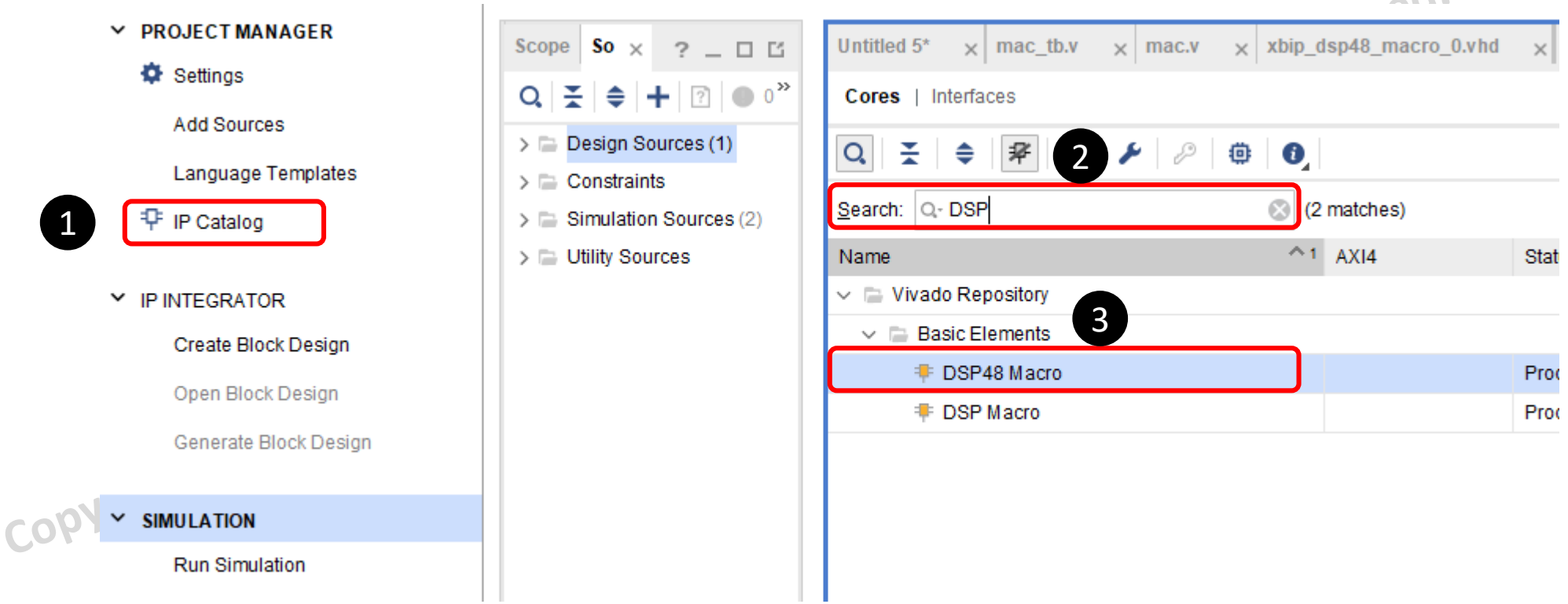
- Lab 1: DSP
  - How to use Vivado IP integrator to generate a DSP?
  - How to test a DSP unit?
- Lab 2: MAC
  - Multipliers and an adder tree
  - MAC
- Lab 3: Computing units

Lab 1: DSP

- Build a multiplier based on a DSP generated by Vivado IP generator
  - Generate a DSP
  - Use it to build a multiplier
- Make a testbench
  - Test the multiplier

Creating a DSP using Xilinx IP generator

- Click "IP Catalog" on Vivado
- Search "DSP"
- Choose "DSP48 Marco" and double click on it



Creating a DSP using Xilinx IP generator

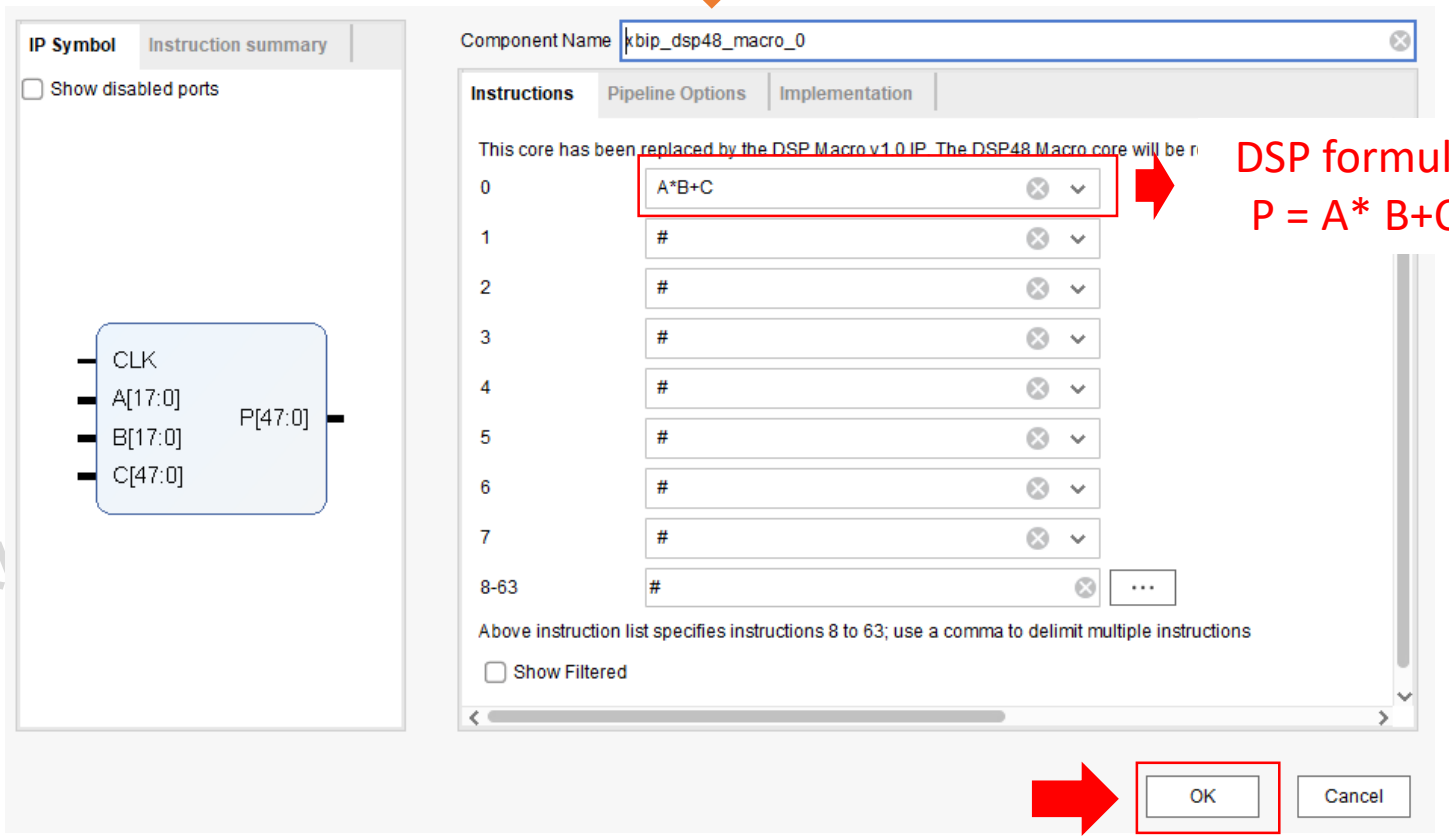
Ports of the DSP

Inputs:

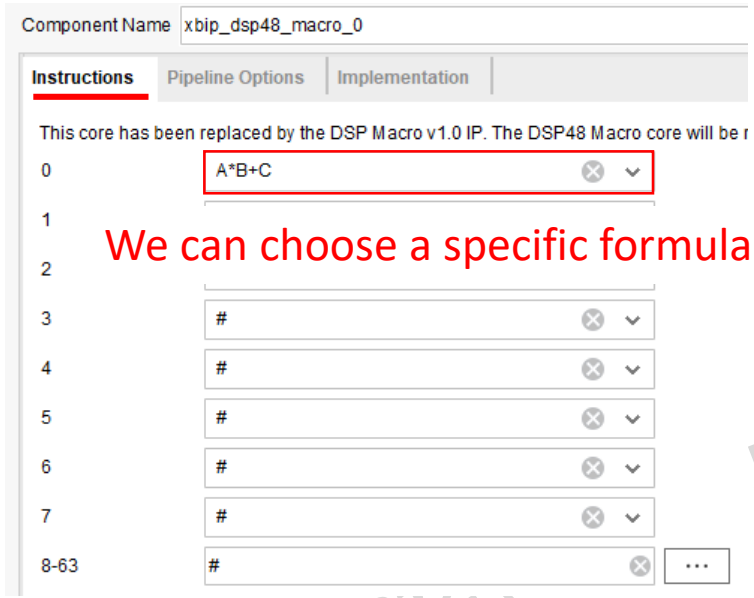
- CLK
- A[17:0], B[17:0]
- C[47:0]

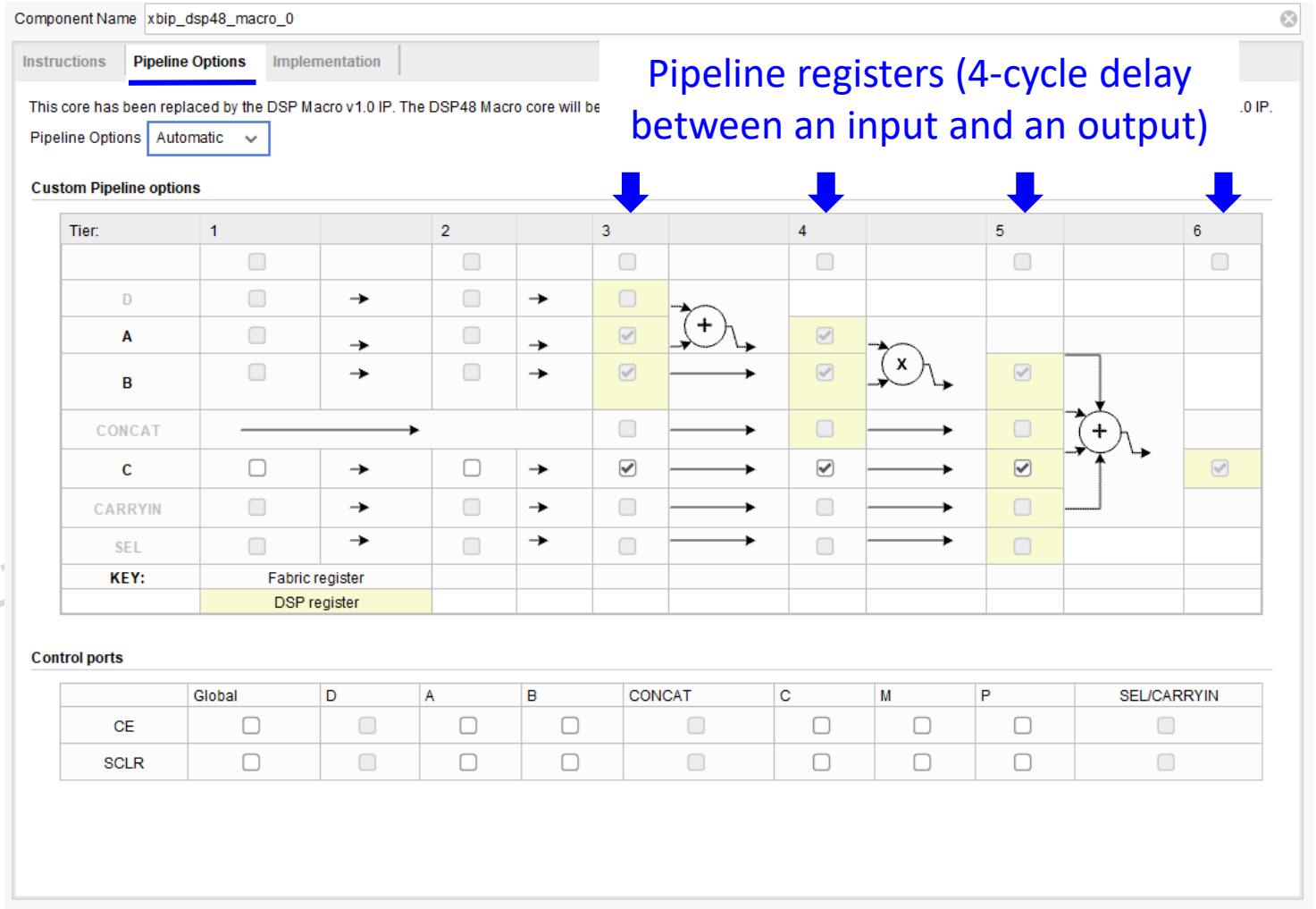
Output:

- P[47:0]



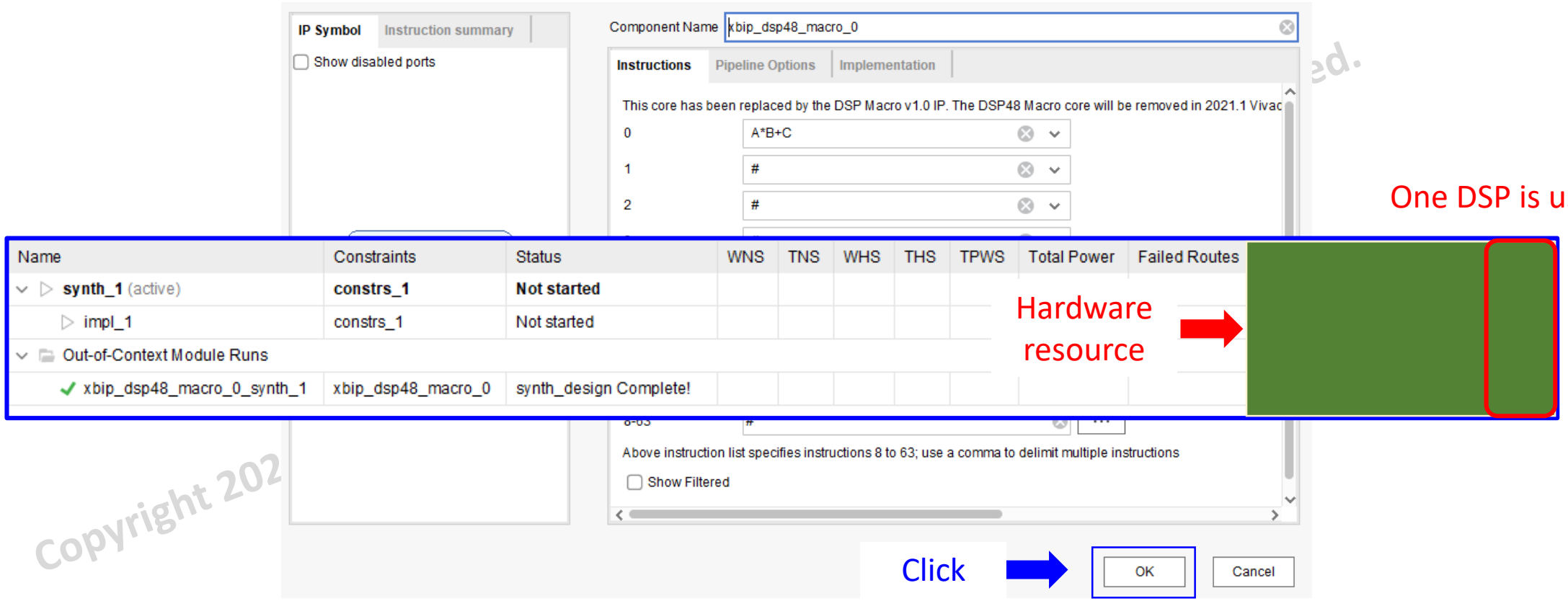
DSP configuration





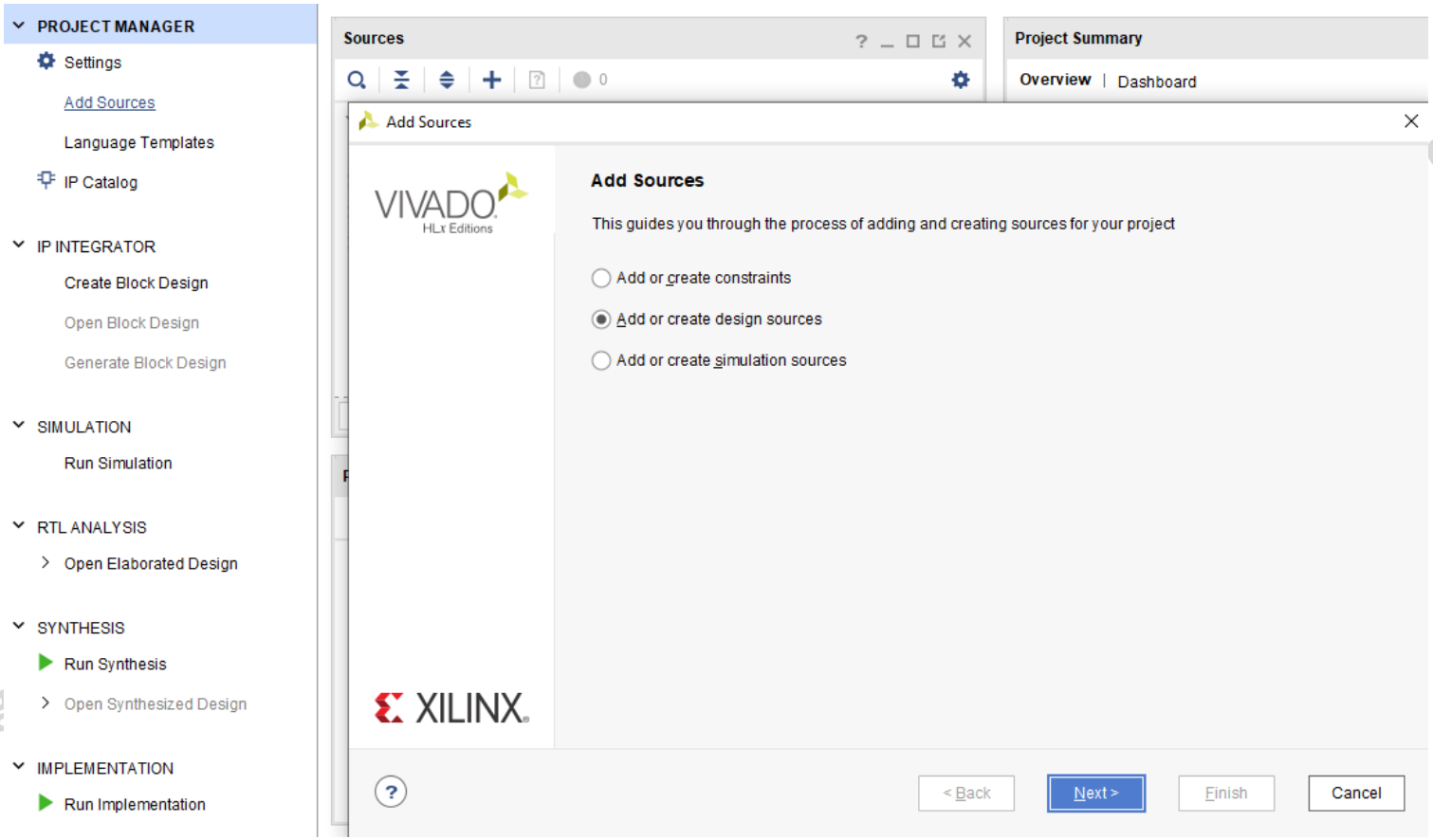
Creating a DSP using Xilinx IP generator

- Click "OK" and then "Generate" on the popup window
  - A DSP is generated



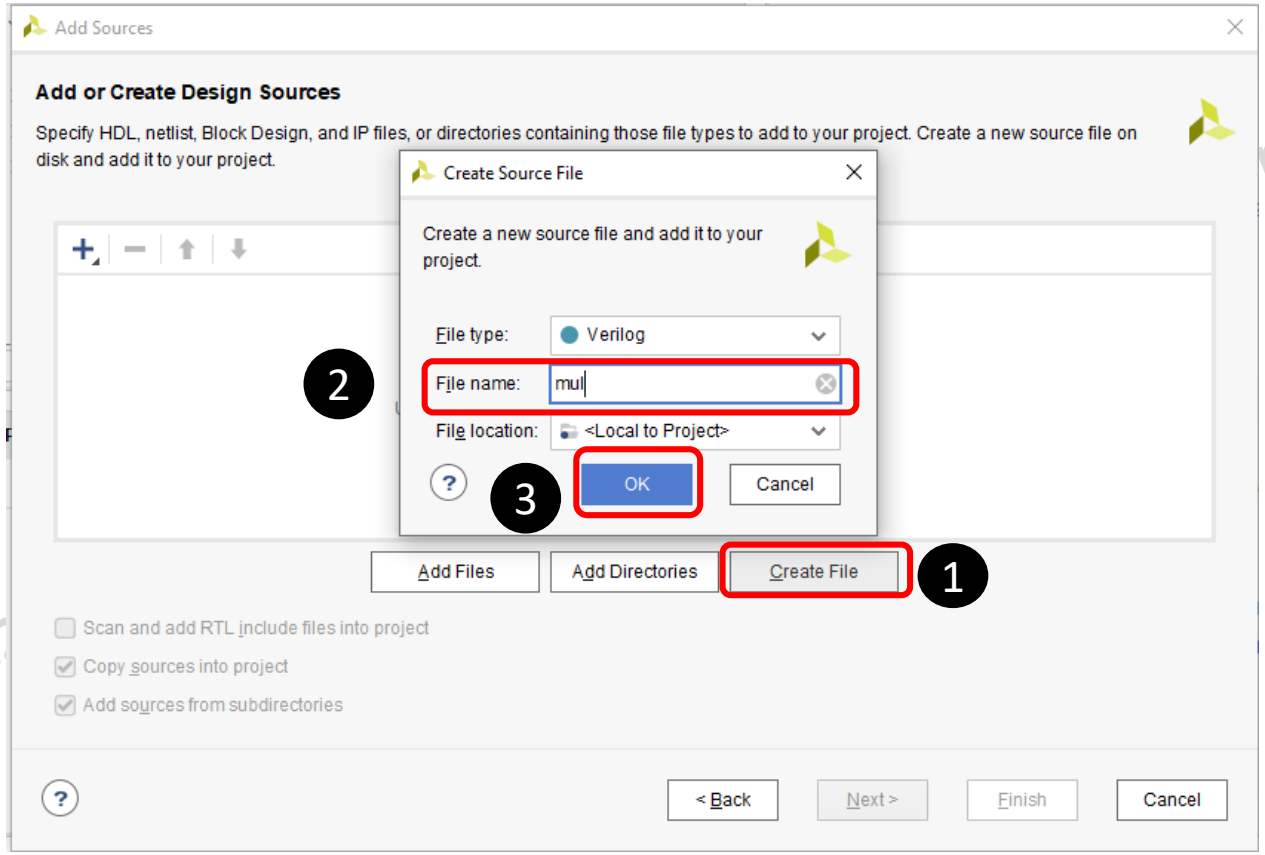
Multiplier (mul.v)

- Choose "Add sources" >> "Add or create design sources" >> Click "Next"



Multiplier (mul.v)

- Click "Create File" >> Make a file name "mul" in "Verilog" >> click "OK"
- Note: the "mul" code is given.



Multiplier (mul.v)

Inputs:
- clk // Clock
- w[7:0] // Weight
- x[7:0] // Input pixel

Output:
- y[15:0] // Product

mul.v

C:/Users/User/aix2022/aix2022.srcs/sources_1/new/mul.v

```
1   `timescale 1ns / 1ps
2   module mul(
3   input clk,
4   input [7:0] w,
5   input [7:0] x,
6   output[15:0] y
7   );
8
9   wire [17:0] dsp_A, dsp_B;
10  wire [47:0] dsp_P;
11
12  assign dsp_A = w[7]? {10'b11_1111_1111, w} : {10'b00_0000_0000, w};
13  assign dsp_B = x[7]? {10'b11_1111_1111, x} : {10'b00_0000_0000, x};
14  assign y = dsp_P[15:0];
15
16  xbip_dsp48_macro_0 u_dsp(.CLK(clk), .A(dsp_A), .B(dsp_B), .C(48'b0), .P(dsp_P));
17  endmodule
18
```

Make inputs and outputs for the DSP

Negative Positive

2's complement for a negative number

A DSP instance

Testbench (mul_tb.v)

Time scale: 1ns

```
1  `timescale 1ns / 1ps
2  module mul_tb;
3  reg clk;
4  reg rstn;
5  reg [7:0] w, x;
6  wire[15:0] y;
7  //-----------------------------------------
8  // DUT: multiplier
9  //-----------------------------------------
10 mul u_mul(
11 .*input        */clk(clk),
12 .*input [ 7:0] */w(w),
13 .*input [ 7:0] */x(x),
14 .*output[15:0] */y(y)
15 );
16
17 // Clock
18 parameter CLK_PERIOD = 10; //100MHz
19 initial begin
20     clk = 1'b1;
21     forever #(CLK_PERIOD/2) clk = ~clk;
22 end
```

Registers (e.g. clk, w, x) are connected to the inputs
A wire (e.g., y) is connected to the output

Design under test:
multiplier (mul)

Clock: 100MHz
(Period = 10ns)

A clock signal is defined in an “initial” block
- Initialized at 1
- State is changed at every 5ns

Testbench (mul_tb.v)

Initialized states
Set all registers to default values

Generate test cases in 16 cycles
w: 4
x: 0 -> 1 -> 2 -> … -> 15

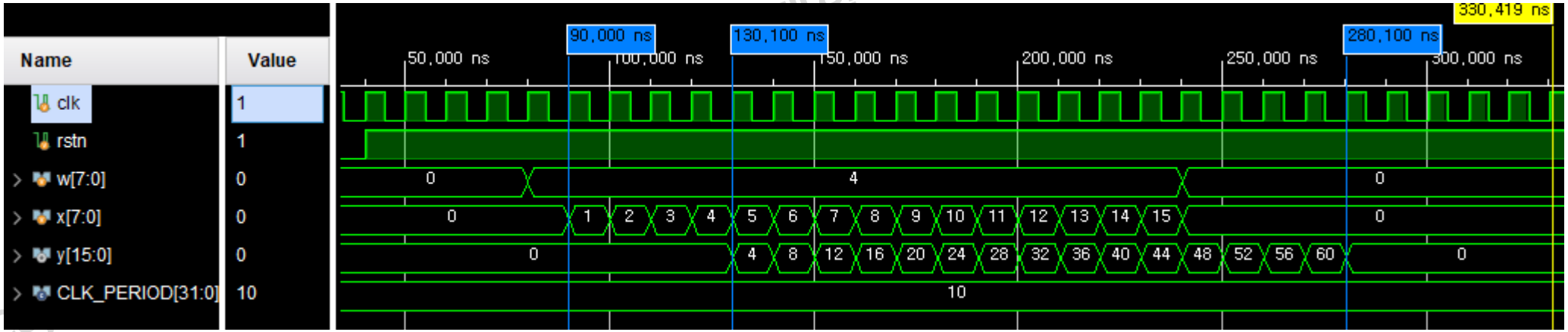
```
23  integer i;
24  // Test cases
25  initial begin
26      rstn = 1'b0;            // Reset, low active
27      w = 0;
28      x = 0;
29      i = 0;
30      #(4*CLK_PERIOD) rstn = 1'b1;
31
32      #(4*CLK_PERIOD)
33      for(i = 0; i<16; i=i+1) begin
34          @(posedge clk)
35              w = 8'd4;
36              x = i;
37      end
38
39      #(CLK_PERIOD)
40      @(posedge clk)
41          w = 8'd0;
42          x = 8'd0;
43  end
```

Waveform

- In Tab "Simulation", click on "run simulation" >> "Run behavioral simulation"
- Visualize the waveform
  - Weight (w) is set to 4 in 16 cycles (e.g., 70~230ns)
  - During the 16 cycles, x is set from 0 to 15

  ⇒ The result (y) is 0, 4, …, 60

  ⇒ For a given input, its output comes out after 4 cycles (e.g., DSP's pipelined registers)

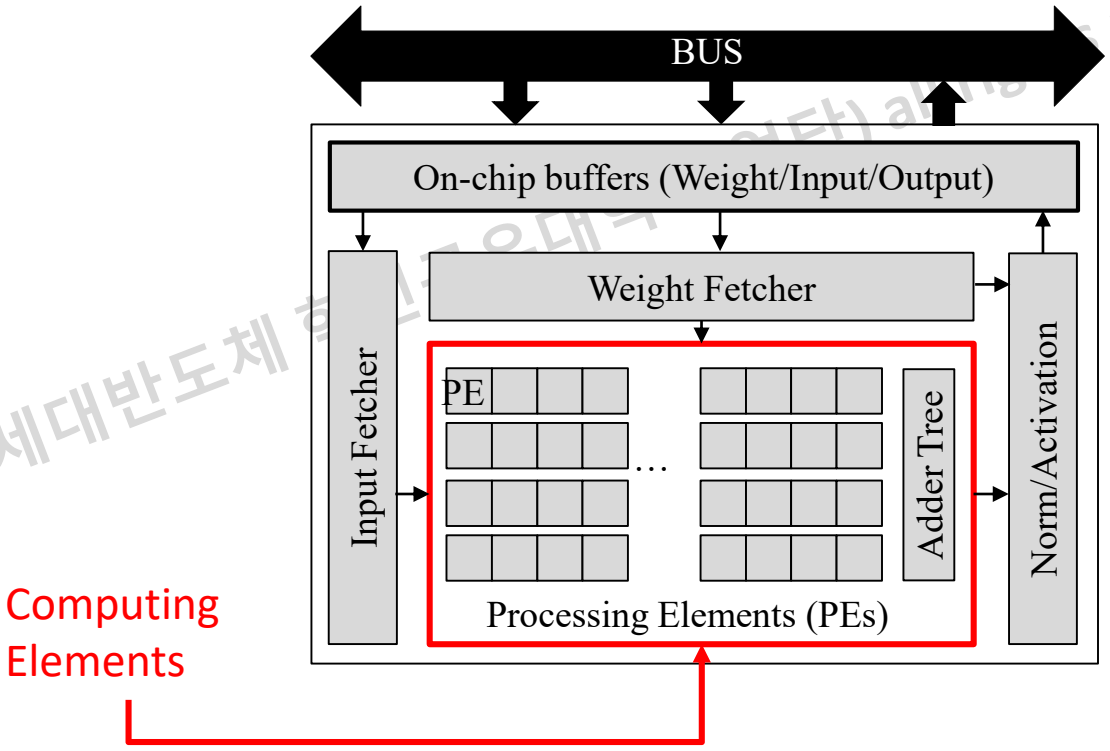


Lab 2: MAC

- Build a MAC
  - Use multiple multipliers
  - Adder tree
- Make a testbench
  - Test the MAC

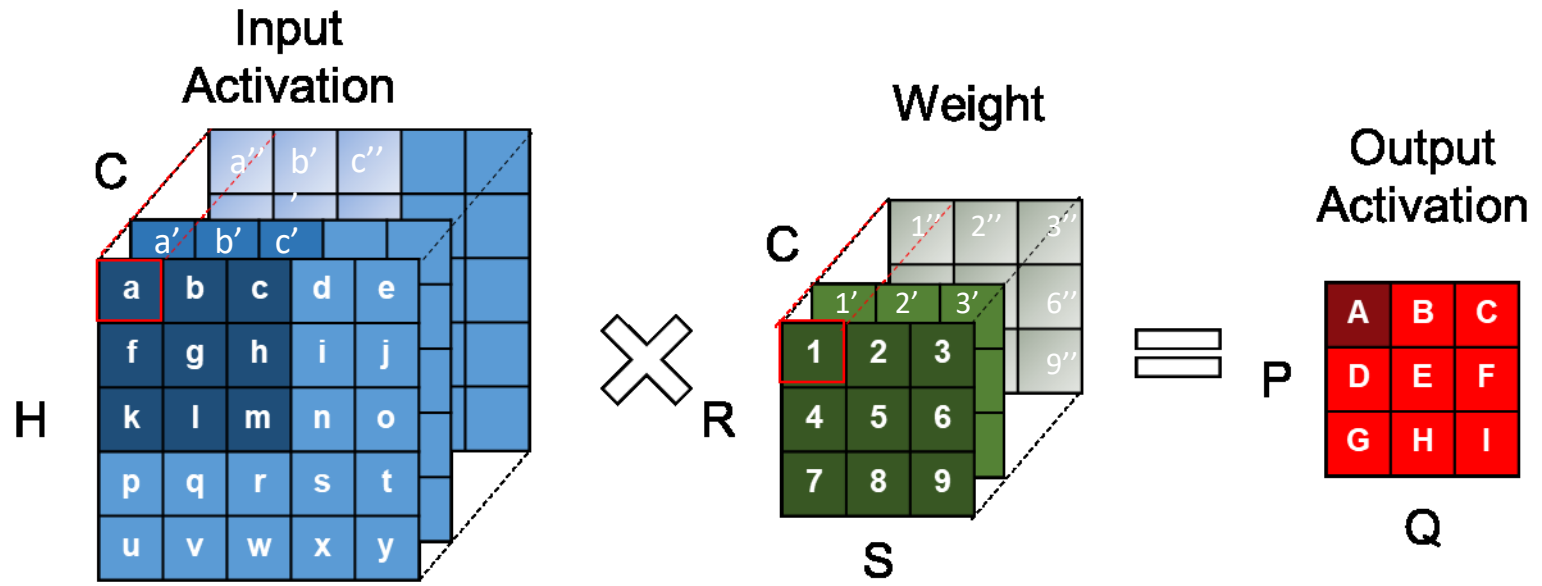
DNN accelerator

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



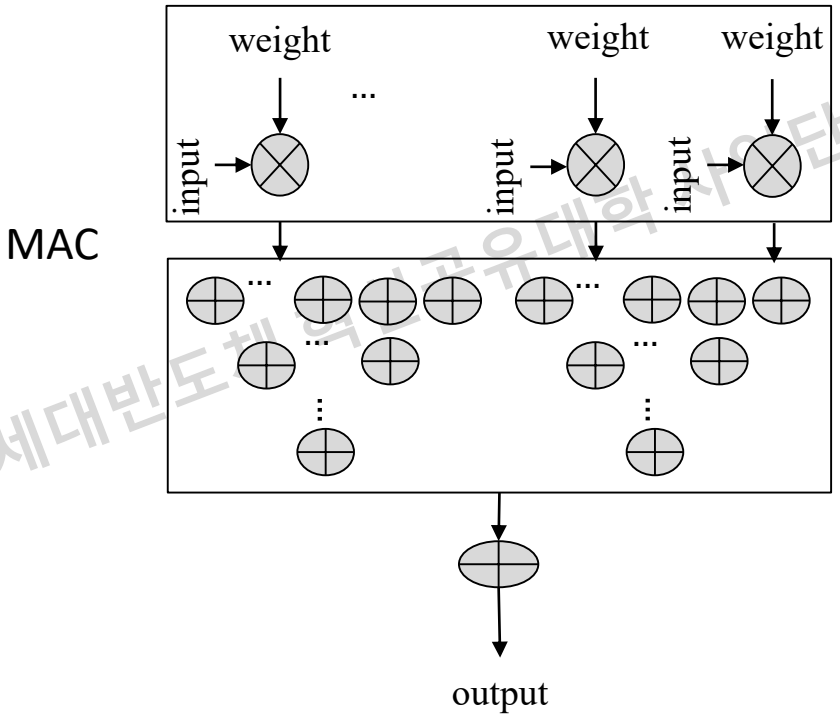
MAC

- MAC is a module that performs many multiplications and accumulations in parallel.
- Review: convolution
  - From this example, to calculate output pixel A, we need to calculate:
  - (a*1 + b*2 + … + m*9) + (a′*1′ + b′*2′ + … + m′*9′) + … + (a″*1″ + b″*2″ + … + m″*9″)
  - lots of multiplication / accumulation!



A simple MAC

- In this tutorial, we will give and explain an example MAC module that calculates 16 multiplications and accumulations every cycle.
- Inputs are 8 bit 2's complement numbers
- and the output is a 20 bit 2's complement number



A simple MAC

- Compute a sum of 16 products

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

- Pseudo code

$y_0^{(0)} = w_0 * x_0, \ldots, y_{15}^{(0)} = w_{15} * x_{15}$ // N multipliers

$y_0^{(1)} = y_0^{(0)} + y_1^{(0)}, \ldots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$ // N/2 adders

$y_0^{(2)} = y_0^{(1)} + y_1^{(1)}, \ldots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$ // N/4 adders

$y_0^{(3)} = y_0^{(2)} + y_1^{(2)}, \ldots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$ // N/4 adders

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

$Y = y_0^{(4)}$ // Output

MAC (mac.v)

- Compute a sum of 16 products

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

- Ports
  - clk, rstn: clock and reset signals
  - vld_i: a valid signal for inputs (e.g., win, din)
  - Inputs win, din
    - win[7:0] = w_0, din[7:0] = x_0
    - win[15:8] = w_0, din[15:0] = x_0
    - …
  - Outputs
    - Accumulated result (acc_o)
    - A valid signal (vld_o)

```
1   `timescale 1ns / 1ps
2
3   module mac(
4   input clk,
5   input rstn,
6   input vld_i,
7   input [127:0] win,
8   input [127:0] din,
9   output[ 19:0] acc_o,
10  output        vld_o
11  );
12
```

Multiplication (mac.v)

- Compute a sum of 16 products

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

```
13  //------------------------------------------------------------------------
14  // Internal signals
15  //------------------------------------------------------------------------
16  wire[15:0] y00;
17  wire[15:0] y01;
18  wire[15:0] y02;
19  wire[15:0] y03;
20  wire[15:0] y04;
21  wire[15:0] y05;
22  wire[15:0] y06;
23  wire[15:0] y07;
24  wire[15:0] y08;
25  wire[15:0] y09;
26  wire[15:0] y10;
27  wire[15:0] y11;
28  wire[15:0] y12;
29  wire[15:0] y13;
30  wire[15:0] y14;
31  wire[15:0] y15;
```

```
34  //------------------------------------------------------------------------
35  // 16 multipliers running in parallel
36  //------------------------------------------------------------------------
37  mul u_mul_00(.clk(clk), .w(win[  7:  0]),.x(din[  7:  0]),.y(y00));
38  mul u_mul_01(.clk(clk), .w(win[ 15:  8]),.x(din[ 15:  8]),.y(y01));
39  mul u_mul_02(.clk(clk), .w(win[ 23: 16]),.x(din[ 23: 16]),.y(y02));
40  mul u_mul_03(.clk(clk), .w(win[ 31: 24]),.x(din[ 31: 24]),.y(y03));
41  mul u_mul_04(.clk(clk), .w(win[ 39: 32]),.x(din[ 39: 32]),.y(y04));
42  mul u_mul_05(.clk(clk), .w(win[ 47: 40]),.x(din[ 47: 40]),.y(y05));
43  mul u_mul_06(.clk(clk), .w(win[ 55: 48]),.x(din[ 55: 48]),.y(y06));
44  mul u_mul_07(.clk(clk), .w(win[ 63: 56]),.x(din[ 63: 56]),.y(y07));
45  mul u_mul_08(.clk(clk), .w(win[ 71: 64]),.x(din[ 71: 64]),.y(y08));
46  mul u_mul_09(.clk(clk), .w(win[ 79: 72]),.x(din[ 79: 72]),.y(y09));
47  mul u_mul_10(.clk(clk), .w(win[ 87: 80]),.x(din[ 87: 80]),.y(y10));
48  mul u_mul_11(.clk(clk), .w(win[ 95: 88]),.x(din[ 95: 88]),.y(y11));
49  mul u_mul_12(.clk(clk), .w(win[103: 96]),.x(din[103: 96]),.y(y12));
50  mul u_mul_13(.clk(clk), .w(win[111:104]),.x(din[111:104]),.y(y13));
51  mul u_mul_14(.clk(clk), .w(win[119:112]),.x(din[119:112]),.y(y14));
52  mul u_mul_15(.clk(clk), .w(win[127:120]),.x(din[127:120]),.y(y15));
```

$y00 = w_0 * x_0$
win[7:0] = w_0
din[7:0] = x_0

$y15 = w_{15} * x_{15}$
win[127:120] = w_{15}
din[127:120] = x_{15}

Accumulation (mac.v)

- Compute a sum of 16 products

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

- Pseudo code

$$y_0^{(0)} = w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15}$$

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

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

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

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

$$Y = y_0^{(4)}$$

```
13
14
15
16  wire[15:0] y00;
17  wire[15:0] y01;
18  wire[15:0] y02;
19  wire[15:0] y03;
20  wire[15:0] y04;
21  wire[15:0] y05;
22  wire[15:0] y06;
23  wire[15:0] y07;
24  wire[15:0] y08;
25  wire[15:0] y09;
26  wire[15:0] y10;
27  wire[15:0] y11;
28  wire[15:0] y12;
29  wire[15:0] y13;
30  wire[15:0] y14;
31  wire[15:0] y15;
```

Internal signals

```
adder_tree u_adder_tree(
./*input         */clk(clk),
./*input         */rstn(rstn),
./*input         */vld_i(vld_i_d4),
./*input [15:0] */mul_00(y00),
./*input [15:0] */mul_01(y01),
./*input [15:0] */mul_02(y02),
./*input [15:0] */mul_03(y03),
./*input [15:0] */mul_04(y04),
./*input [15:0] */mul_05(y05),
./*input [15:0] */mul_06(y06),
./*input [15:0] */mul_07(y07),
./*input [15:0] */mul_08(y08),
./*input [15:0] */mul_09(y09),
./*input [15:0] */mul_10(y10),
./*input [15:0] */mul_11(y11),
./*input [15:0] */mul_12(y12),
./*input [15:0] */mul_13(y13),
./*input [15:0] */mul_14(y14),
./*input [15:0] */mul_15(y15),
./*output[19:0] */acc_o(acc_o),
./*output        */vld_o(vld_o)
);
```

Accumulation (adder_tree.v)

- Compute a sum of 16 products

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

```
// Level 1
always@(posedge clk, negedge rstn) begin
    if(!rstn) begin
        y1_0 <= 17'd0;
        y1_1 <= 17'd0;
        y1_2 <= 17'd0;
        y1_3 <= 17'd0;
        y1_4 <= 17'd0;
        y1_5 <= 17'd0;
        y1_6 <= 17'd0;
        y1_7 <= 17'd0;
    end
    else begin
        y1_0 <= $signed(mul_00) + $signed(mul_01);
        y1_1 <= $signed(mul_02) + $signed(mul_03);
        y1_2 <= $signed(mul_04) + $signed(mul_05);
        y1_3 <= $signed(mul_06) + $signed(mul_07);
        y1_4 <= $signed(mul_08) + $signed(mul_09);
        y1_5 <= $signed(mul_10) + $signed(mul_11);
        y1_6 <= $signed(mul_12) + $signed(mul_13);
        y1_7 <= $signed(mul_14) + $signed(mul_15);
    end
end
```

```
// Level 2
always@(posedge clk, negedge rstn) begin
    if(!rstn) begin
        y2_0 <= 18'd0;
        y2_1 <= 18'd0;
        y2_2 <= 18'd0;
        y2_3 <= 18'd0;
    end
    else begin
        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);
    end
end
```

```
// Level 3
always@(posedge clk, negedge rstn) begin
    if(!rstn) begin
        y3_0 <= 19'd0;
        y3_1 <= 19'd0;
    end
    else begin
        y3_0 <= $signed(y2_0) + $signed(y2_1);
        y3_1 <= $signed(y2_2) + $signed(y2_3);
    end
end
```

```
// Level 4
always@(posedge clk, negedge rstn) begin
    if(!rstn)
        y4 <= 20'd0;
    else
        y4 <= $signed(y3_0) + $signed(y3_1);
end
```

Delays

- Calculating delayed valid signals
- vld_i: when high, indicates that win and din are valid inputs, and must be calculated
- Delays in the adder tree (adder_tree.v)
  - vld_d: an internal variable to calculate the timing for vld_o to be high
  - vld_o: when high, indicates that output acc_o is a valid calculation result

```
113     //------------------------------------------------------------
114     // Valid signal
115     //------------------------------------------------------------
116     always@(posedge clk, negedge rstn) begin
117         if(!rstn) begin
118             vld_i_d1 <= 0;
119             vld_i_d2 <= 0;
120             vld_i_d3 <= 0;
121             vld_i_d4 <= 0;
122         end
123         else begin
124             vld_i_d1 <= vld_i   ;
125             vld_i_d2 <= vld_i_d1;
126             vld_i_d3 <= vld_i_d2;
127             vld_i_d4 <= vld_i_d3;
128         end
129     end
130     //Output
131     assign vld_o = vld_i_d4;
132     assign acc_o = $signed(y4);
```

Test bench (mac_tb.v)

Time scale: 1ns

```
1   `timescale 1ns / 1ps
2
3   module mac_tb;
4   reg clk;
5   reg rstn;
6   reg vld_i;
7   reg [127:0] win, din;
8   wire[19:0] acc_o;
9   wire       vld_o;
10
11  //------------------------------------------
12  // DUT: multiplier
13  //------------------------------------------
14  mac u_mac(
15  .clk(clk),
16  .rstn(rstn),
17  .vld_i(vld_i),
18  .win(win),
19  .din(din),
20  .acc_o(acc_o),
21  .vld_o(vld_o)
22  );
```

Registers (e.g. clk, win, din) are connected to the inputs
Wires (e.g., acc_o, vld_o) are connected to the output

Design under test: mac

Clock: 100MHz
A clock signal is defined in an “initial” block
- Initialized at 1
- State is changed at every 5ns

```
24      // Clock
25      parameter CLK_PERIOD = 10; //100MHz
26      initial begin
27          clk = 1'b1;
28          forever #(CLK_PERIOD/2) clk = ~clk;
29      end
```

Test bench (mac_tb.v)

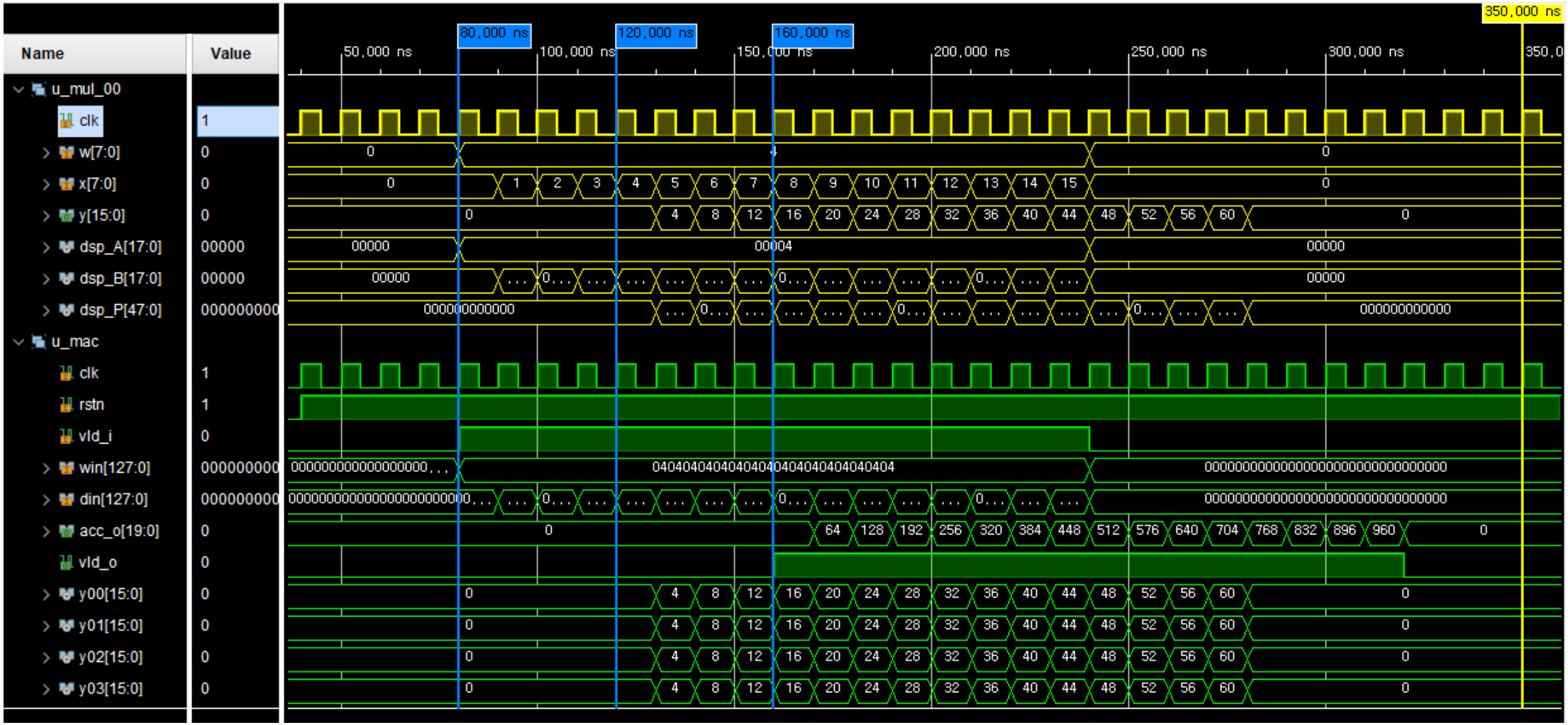
Initialized states
Set all registers to default values

```
31      // Test cases
32      initial begin
33          rstn = 1'b0;            // Reset, low active
34          vld_i= 0;
35          win = 0;
36          din = 0;
37          i = 0;
38          #(4*CLK_PERIOD) rstn = 1'b1;
39
40          #(4*CLK_PERIOD)
41          for(i = 0; i<16; i=i+1) begin
42              @(posedge clk)
43                  vld_i = 1'b1;
44                  win = {16{8'd4}};
45                  din[  7:  0] = i;
46                  din[ 15:  8] = i;
47                  din[ 23: 16] = i;
48                  din[ 31: 24] = i;
49                  din[ 39: 32] = i;
50                  din[ 47: 40] = i;
51                  din[ 55: 48] = i;
52                  din[ 63: 56] = i;
53                  din[ 71: 64] = i;
54                  din[ 79: 72] = i;
55                  din[ 87: 80] = i;
56                  din[ 95: 88] = i;
57                  din[103: 96] = i;
58                  din[111:104] = i;
59                  din[119:112] = i;
60                  din[127:120] = i;
61      end
```

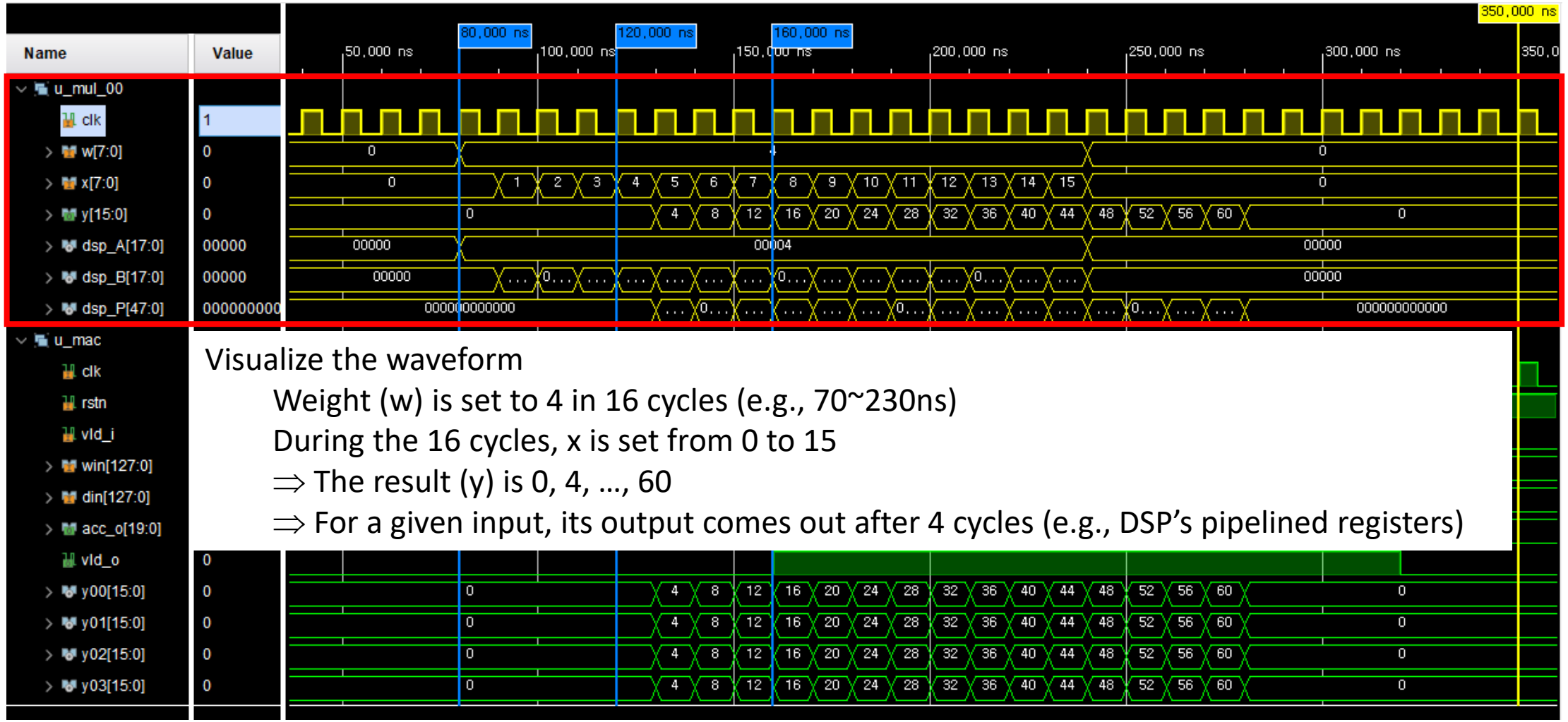
Generate test cases in 16 cycles
w: 4
x: 0 -> 1 -> 2 -> … -> 15

Note: we have 16 weights and 16 inputs now.

Waveform



Waveform



Visualize the waveform

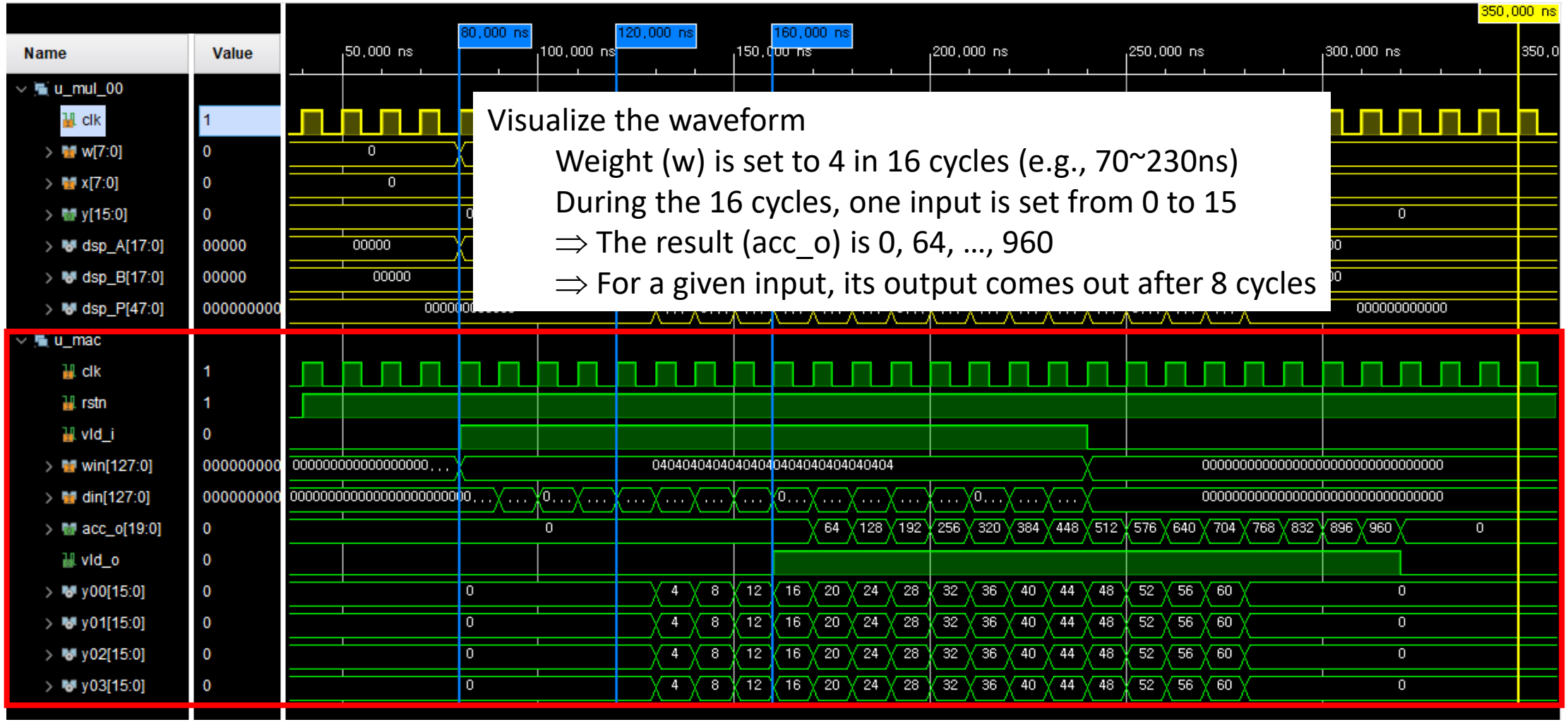
Weight (w) is set to 4 in 16 cycles (e.g., 70~230ns)

During the 16 cycles, x is set from 0 to 15

⇒ The result (y) is 0, 4, …, 60

⇒ For a given input, its output comes out after 4 cycles (e.g., DSP's pipelined registers)

Waveform



Lab 3: Convolutional layer (Practice)

- Convolutional layer
  - Use four MAC modules
  - Initialize four sets of filters
  - Read an image from file
  - Do convolution

Convolutional layer (cnv_tb.v)

- Parameters for an image file, e.g., WIDTH, HEIGHT, file location (INFILE), frame/image size
- A buffer to store an image (in_img)
- Internal signals
  - clk, rstn,
  - vld_i, din[127:0]
  - Four sets of filters win[0:3][127:0]
  - acc_o[0:3][19:0]
  - vld_o[0:3]

```
1   `timescale 1ns / 1ps
2
3   module cnv_tb;
4   parameter WIDTH    = 128;
5   parameter HEIGHT   = 128;
6   parameter INFILE    = "./hex/butterfly_08bit.hex";
7   localparam FRAME_SIZE = WIDTH * HEIGHT;
8   localparam FRAME_SIZE_W = $clog2(FRAME_SIZE);
9   reg [7:0] in_img [0:FRAME_SIZE-1]; // Input image
10  reg clk;
11  reg rstn;
12  reg vld_i;
13  reg [127:0] win[0:3];
14  reg [127:0] din;
15  wire[ 19:0] acc_o[0:3];
16  wire        vld_o[0:3];
```

Convolutional layer (cnv_tb.v)

- There are four MAC instances
- Each module:
  - Uses one set of convolutional filters, e.g., win
  - Outputs a specific acc_o

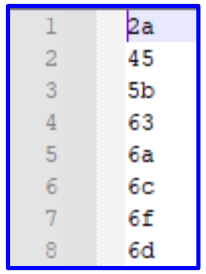
```
20      mac u_mac_00(
21      .clk(clk),
22      .rstn(rstn),
23      .vld_i(vld_i),
24      .win(win[0]),
25      .din(din),
26      .acc_o(acc_o[0]),
27      .vld_o(vld_o[0])
28      );
29      mac u_mac_01(
30      .clk(clk),
31      .rstn(rstn),
32      .vld_i(vld_i),
33      .win(win[1]),
34      .din(din),
35      .acc_o(acc_o[1]),
36      .vld_o(vld_o[1])
37      );
```

```
38      mac u_mac_02(
39      .clk(clk),
40      .rstn(rstn),
41      .vld_i(vld_i),
42      .win(win[2]),
43      .din(din),
44      .acc_o(acc_o[2]),
45      .vld_o(vld_o[2])
46      );
47      mac u_mac_03(
48      .clk(clk),
49      .rstn(rstn),
50      .vld_i(vld_i),
51      .win(win[3]),
52      .din(din),
53      .acc_o(acc_o[3]),
54      .vld_o(vld_o[3])
55      );
```

Convolutional layer (cnv_tb.v)

Read the hex file into a buffer

```
// Read the input file to memory
initial begin
    $readmemh(INFILE, in_img ,0,FRAME_SIZE-1);
end
initial begin
    rstn = 1'b0;            // Reset, low active
    vld_i= 0;
    din = 0;
    i = 0;
```



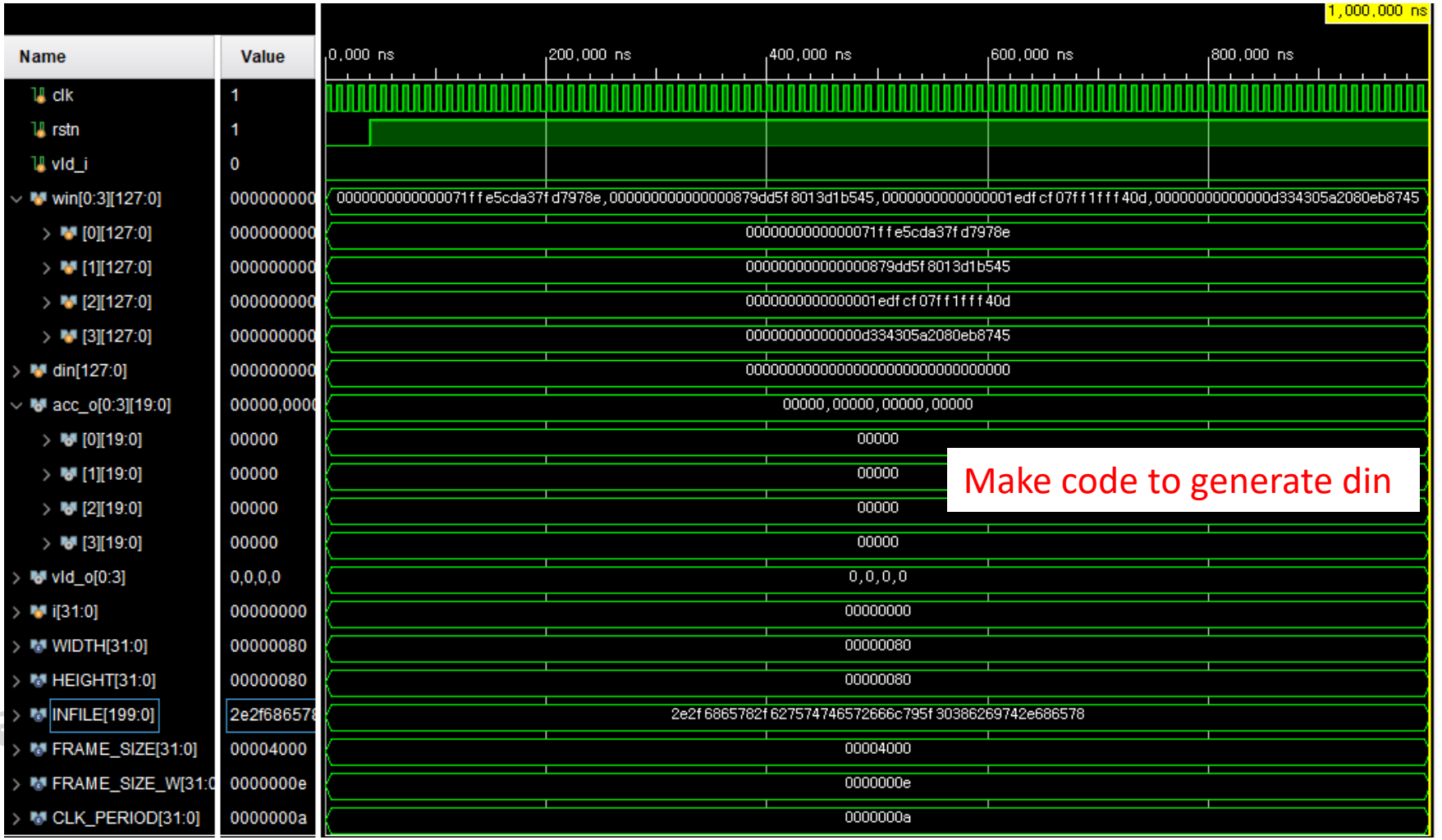
```
2a
45
5b
63
6a
6c
6f
6d
```

Initialize four set of convolutional filters

```
// CNN filters of four output channels
win[0][  7:  0] = 8'd142; win[1][  7:  0] = 8'd69 ; win[2][  7:  0] = 8'd13 ; win[3][  7:  0] = 8'd69 ;
win[0][ 15:  8] = 8'd151; win[1][ 15:  8] = 8'd181; win[2][ 15:  8] = 8'd244; win[3][ 15:  8] = 8'd135;
win[0][ 23: 16] = 8'd215; win[1][ 23: 16] = 8'd209; win[2][ 23: 16] = 8'd255; win[3][ 23: 16] = 8'd235;
win[0][ 31: 24] = 8'd127; win[1][ 31: 24] = 8'd19 ; win[2][ 31: 24] = 8'd241; win[3][ 31: 24] = 8'd128;
win[0][ 39: 32] = 8'd163; win[1][ 39: 32] = 8'd128; win[2][ 39: 32] = 8'd127; win[3][ 39: 32] = 8'd32 ;
win[0][ 47: 40] = 8'd205; win[1][ 47: 40] = 8'd95 ; win[2][ 47: 40] = 8'd240; win[3][ 47: 40] = 8'd90 ;
win[0][ 55: 48] = 8'd229; win[1][ 55: 48] = 8'd221; win[2][ 55: 48] = 8'd252; win[3][ 55: 48] = 8'd48 ;
win[0][ 63: 56] = 8'd255; win[1][ 63: 56] = 8'd121; win[2][ 63: 56] = 8'd237; win[3][ 63: 56] = 8'd52 ;
win[0][ 71: 64] = 8'd113; win[1][ 71: 64] = 8'd8  ; win[2][ 71: 64] = 8'd1  ; win[3][ 71: 64] = 8'd211;
win[0][ 79: 72] = 8'd0  ; win[1][ 79: 72] = 8'd0  ; win[2][ 79: 72] = 8'd0  ; win[3][ 79: 72] = 8'd0  ;
win[0][ 87: 80] = 8'd0  ; win[1][ 87: 80] = 8'd0  ; win[2][ 87: 80] = 8'd0  ; win[3][ 87: 80] = 8'd0  ;
win[0][ 95: 88] = 8'd0  ; win[1][ 95: 88] = 8'd0  ; win[2][ 95: 88] = 8'd0  ; win[3][ 95: 88] = 8'd0  ;
win[0][103: 96] = 8'd0  ; win[1][103: 96] = 8'd0  ; win[2][103: 96] = 8'd0  ; win[3][103: 96] = 8'd0  ;
win[0][111:104] = 8'd0  ; win[1][111:104] = 8'd0  ; win[2][111:104] = 8'd0  ; win[3][111:104] = 8'd0  ;
win[0][119:112] = 8'd0  ; win[1][119:112] = 8'd0  ; win[2][119:112] = 8'd0  ; win[3][119:112] = 8'd0  ;
win[0][127:120] = 8'd0  ; win[1][127:120] = 8'd0  ; win[2][127:120] = 8'd0  ; win[3][127:120] = 8'd0  ;
```

⇒ Come from a quantized model

Waveform



Incoming lecture …

- Memory
  - SPRAM
  - DPRAM