차세대 반도체 디지털 혁신공유대학 사업단

POLARIS

Hardware Description Language Computing Units

2024.03.15 (Friday)

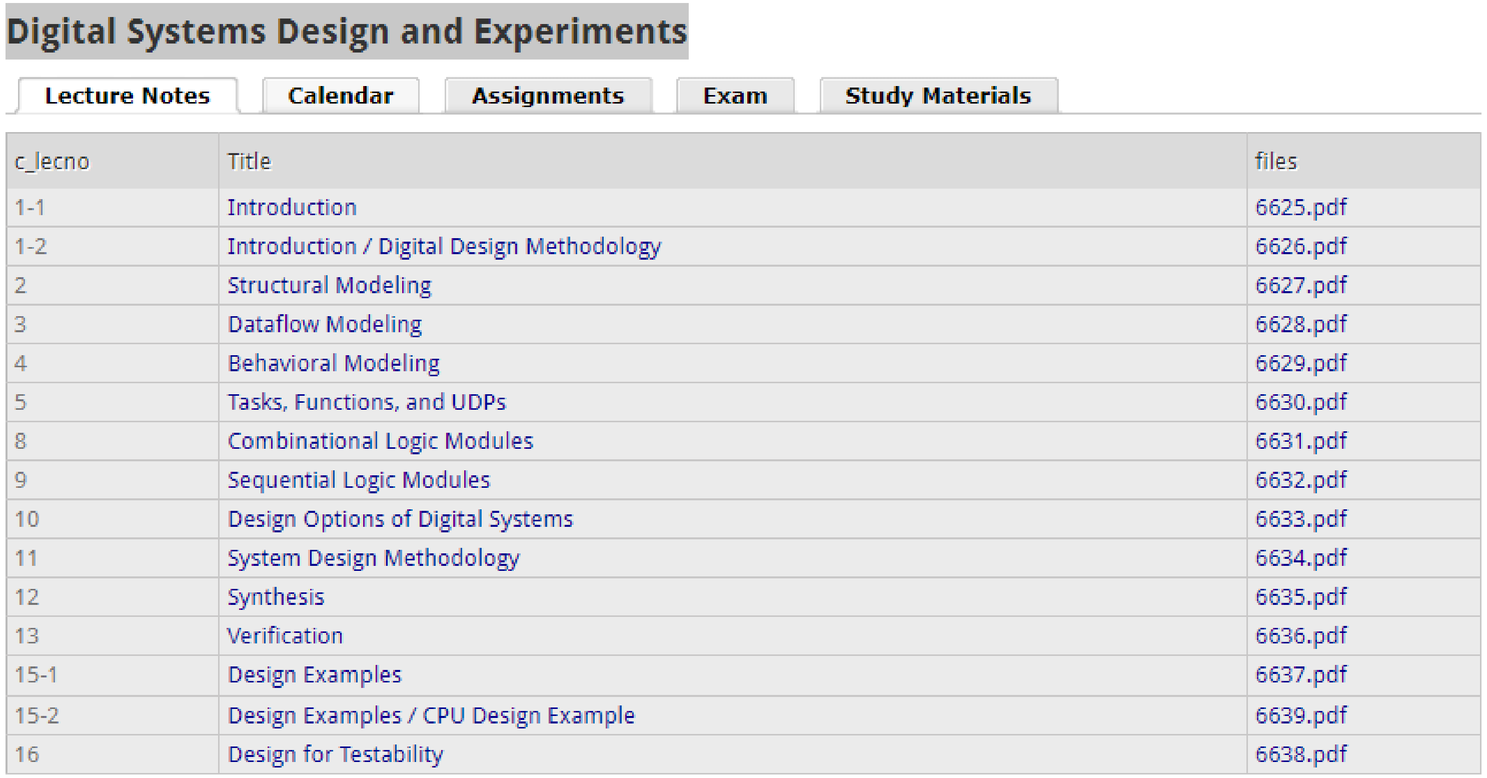
Outlines

- Verilog HDL

- Labs: Computing Units

References

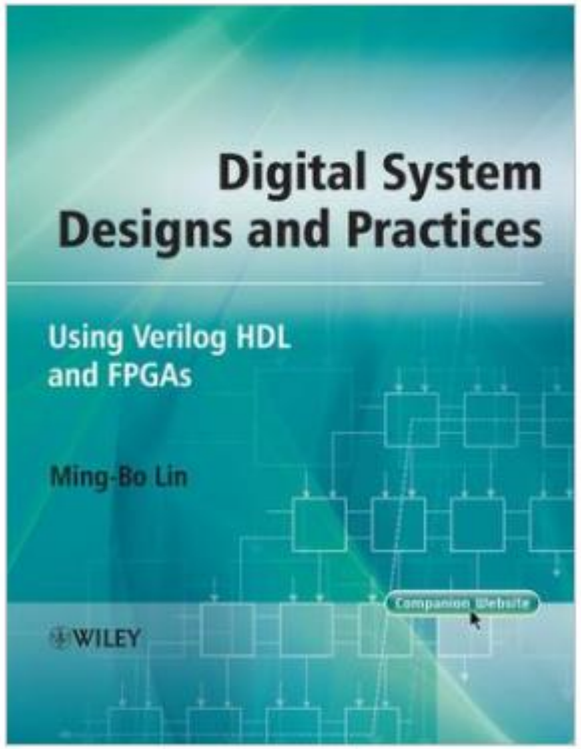
- Class: Digital Systems Design and Experiments (https://ocw.snu.ac.kr/node/2390)
- Books: Digital System Designs and Practices: Using Verilog HDL and FPGAs @Willy 2008



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

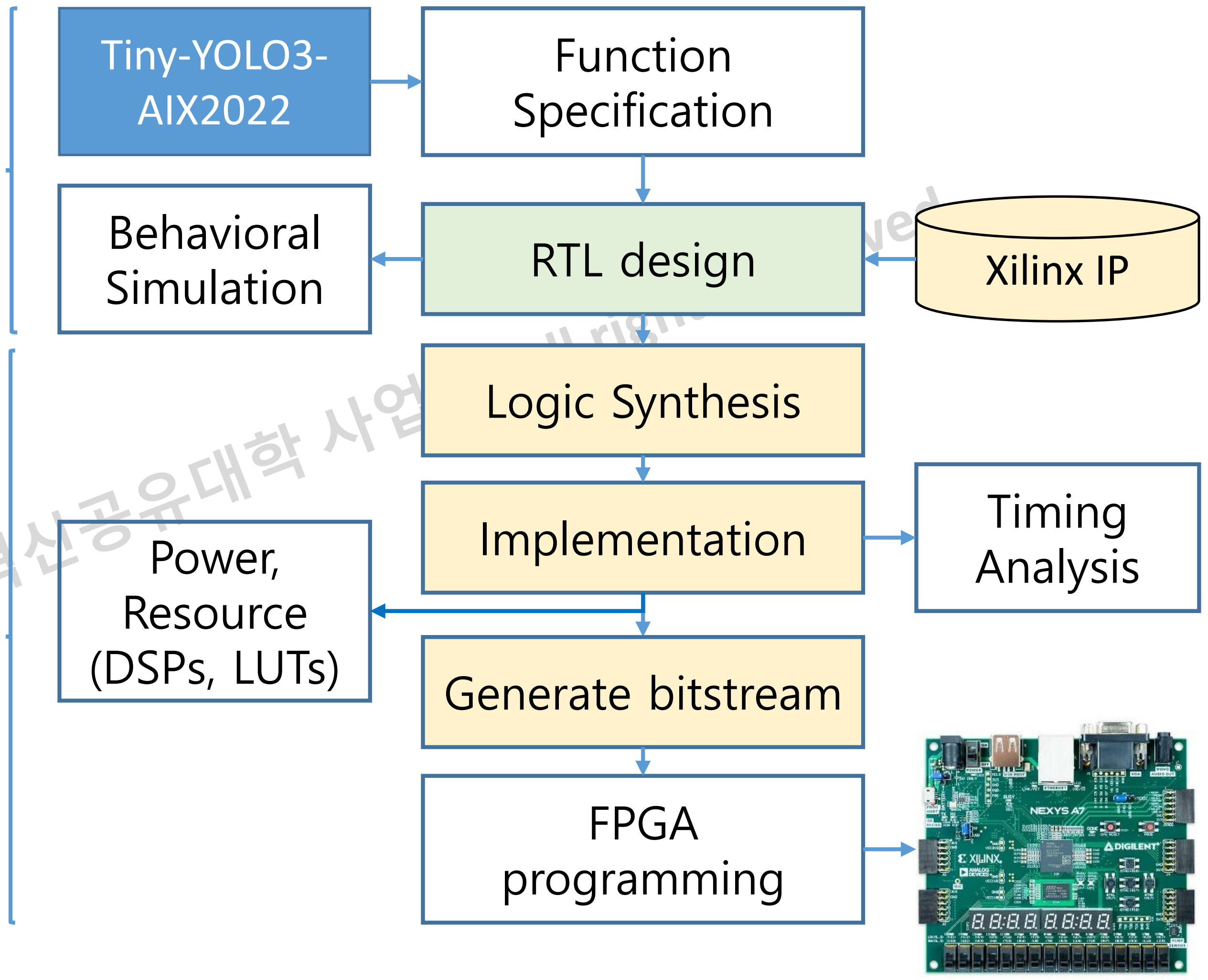

Digital System Designs and Practices
Using Verilog HDL and FPGAs
Ming-Bo Lin
WILEY
Companion Website

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

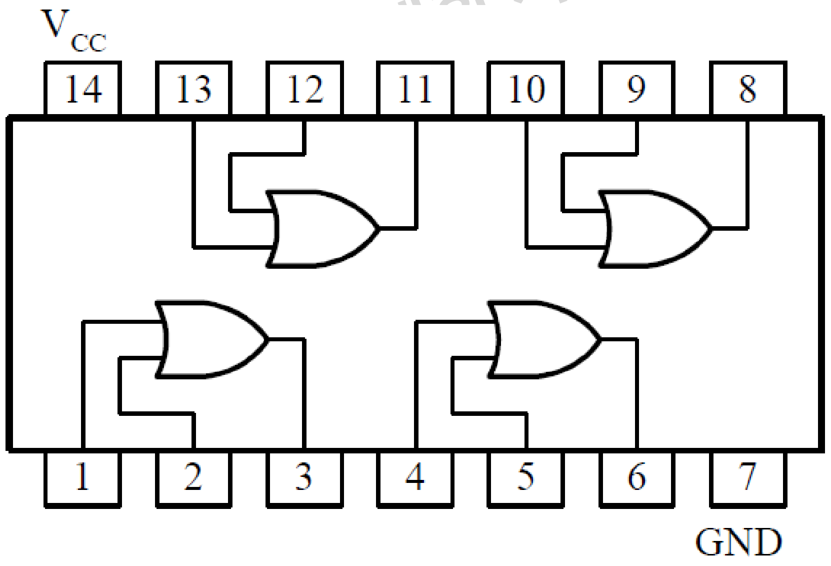


Hardware Description Language (HDL)

- 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.

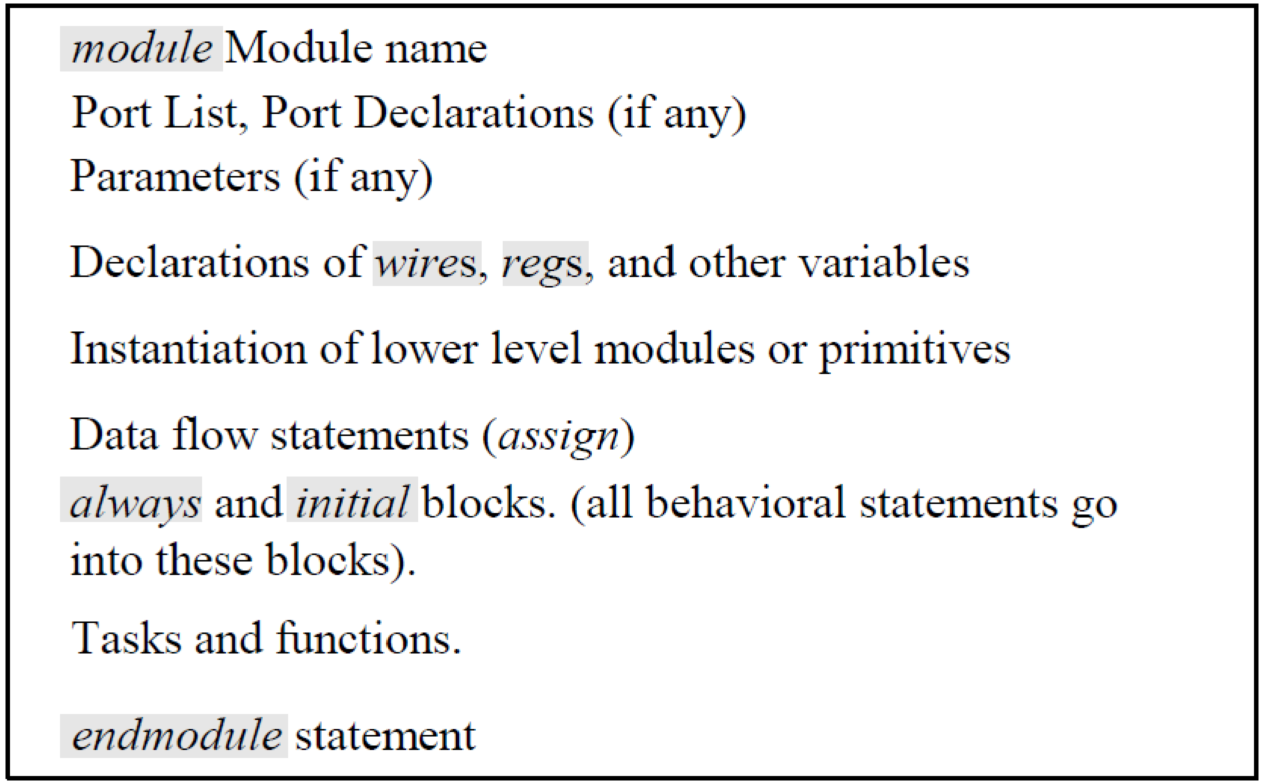
Modules – Hardware Module Concept

- 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

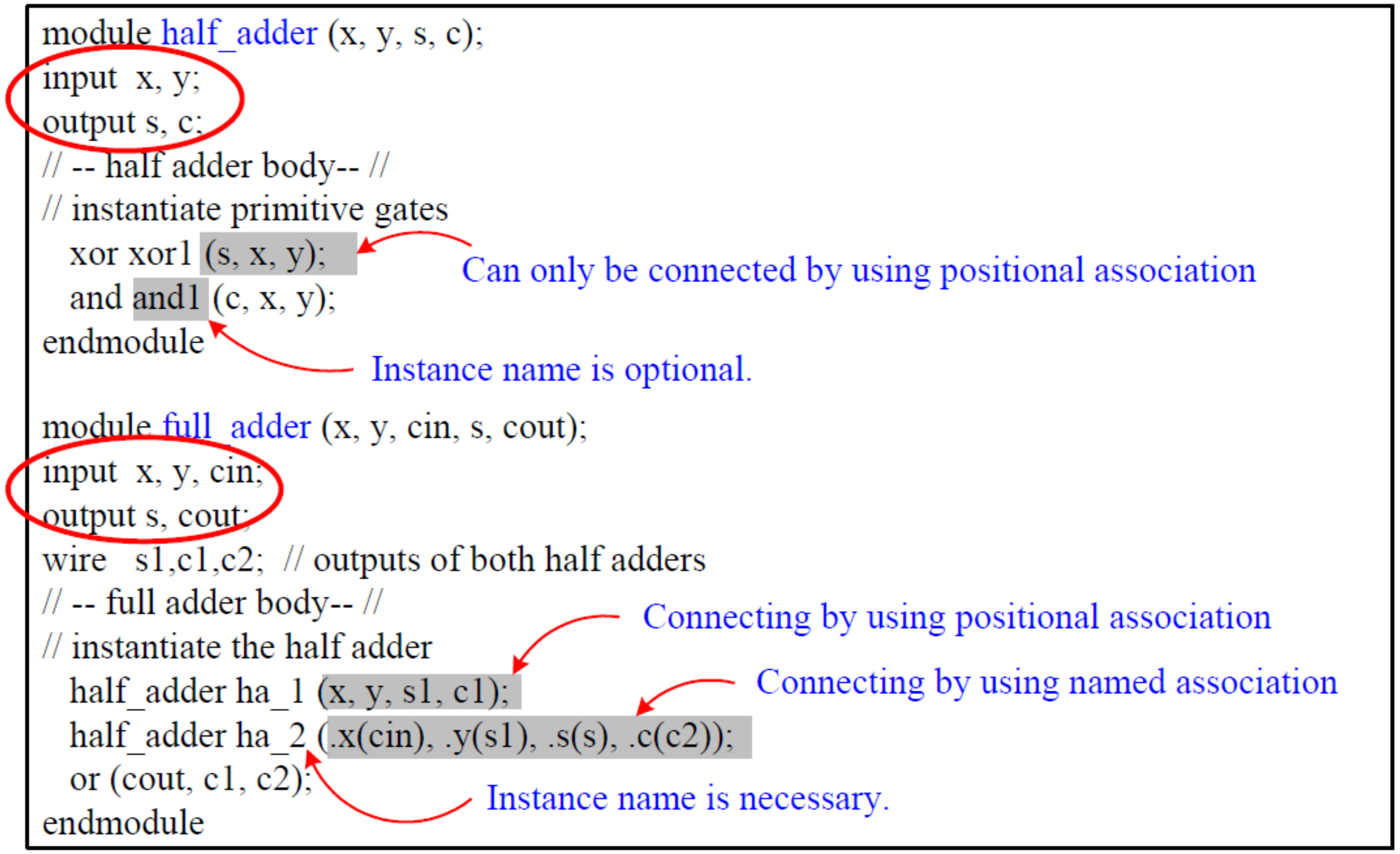
Data types

- 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
```

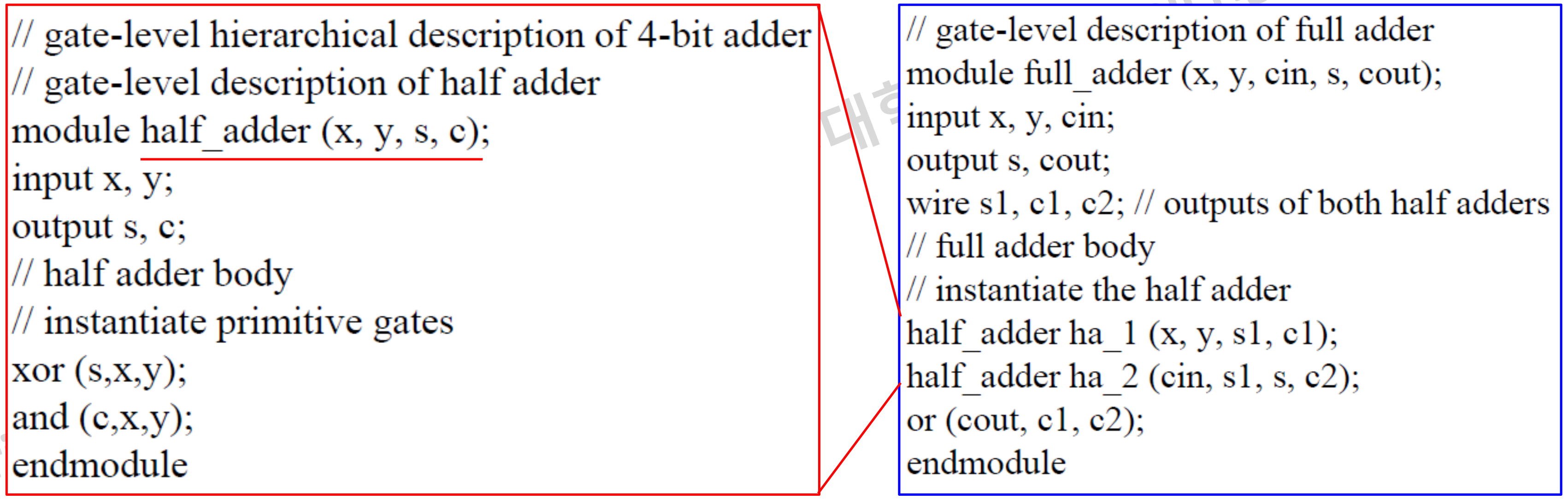
Can only be connected by using positional association
Instance name is optional.
Connecting by using positional association
Connecting by using named association
Instance name is necessary.

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.



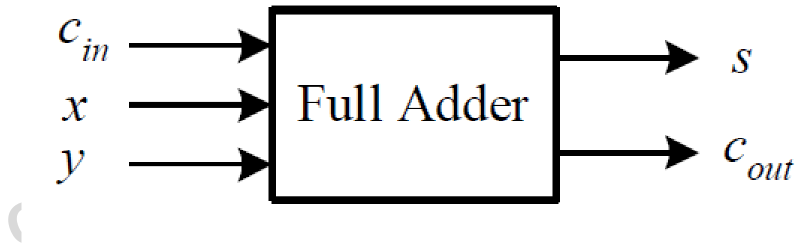
```
// gate-level hierarchical description of 4-bit adder
// gate-level description of half adder
module half_adder (x, y, s, c);
input x, y;
output s, c;
// half adder body
// instantiate primitive gates
xor (s,x,y);
and (c,x,y);
endmodule
```

```
// 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

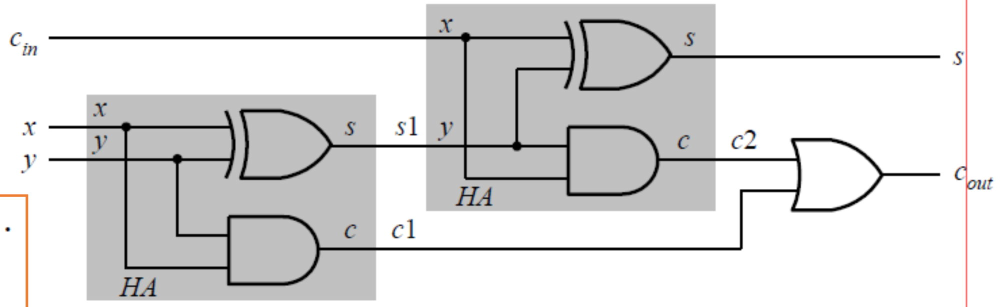
- Mixed style is the mixing use of above three modeling styles.
  - Commonly used in modeling large designs.

```
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
```

structural

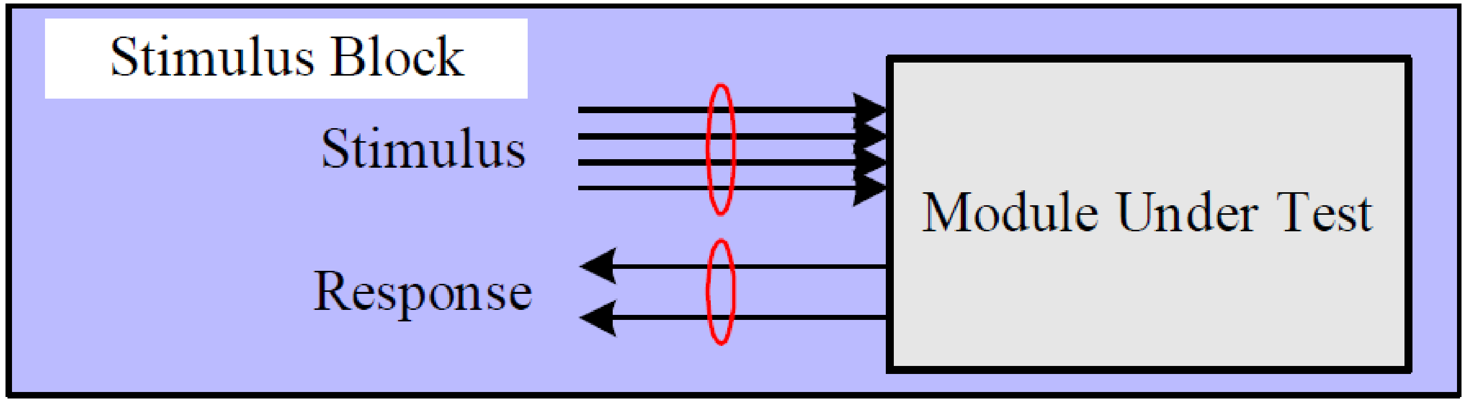
dataflow

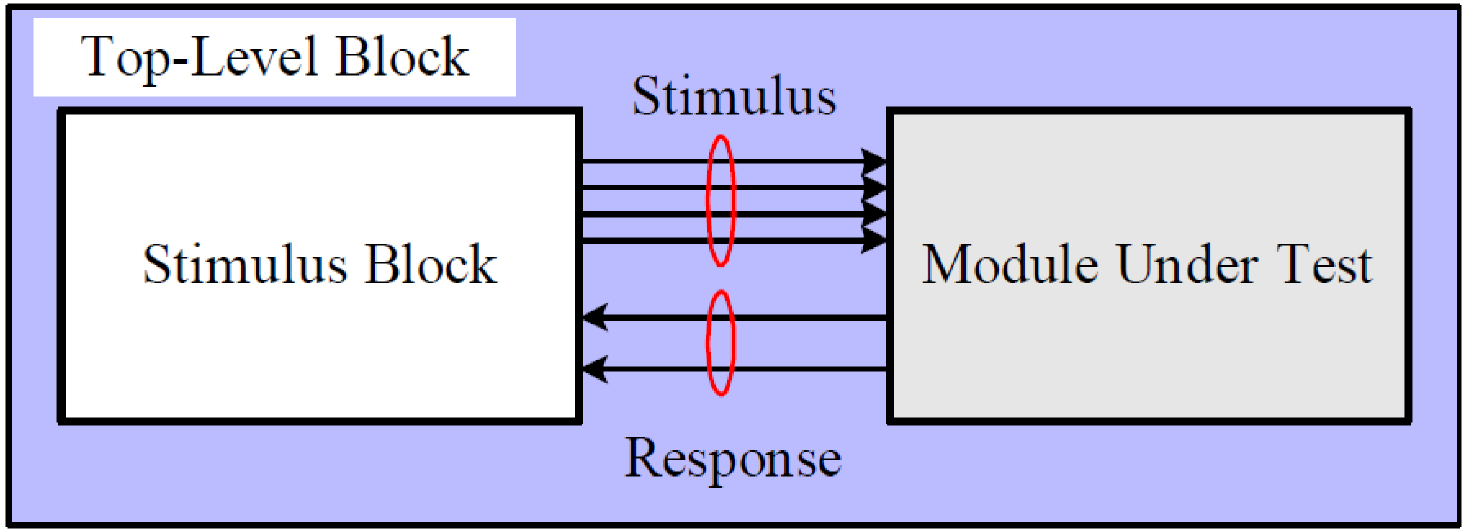
behavioral



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.

Outlines

- Verilog HDL

- Labs: Computing Units

Labs

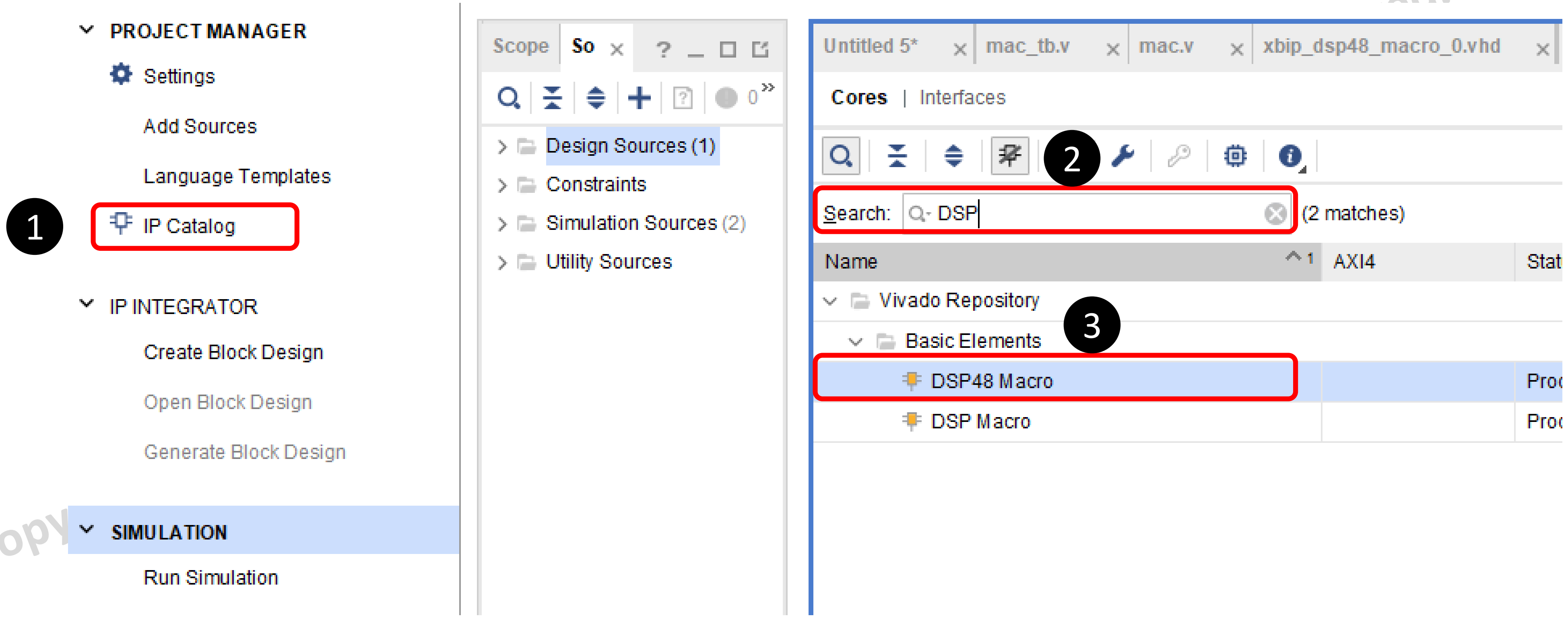
- Lab 1: DSP
    - How to use Vivado IP integrator to generate a DSP?
    - How to test a DSP unit?
- Lab 2: MAC
    - Multipliers and an adder tree
    - MAC
- Lab 3: Computing units

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

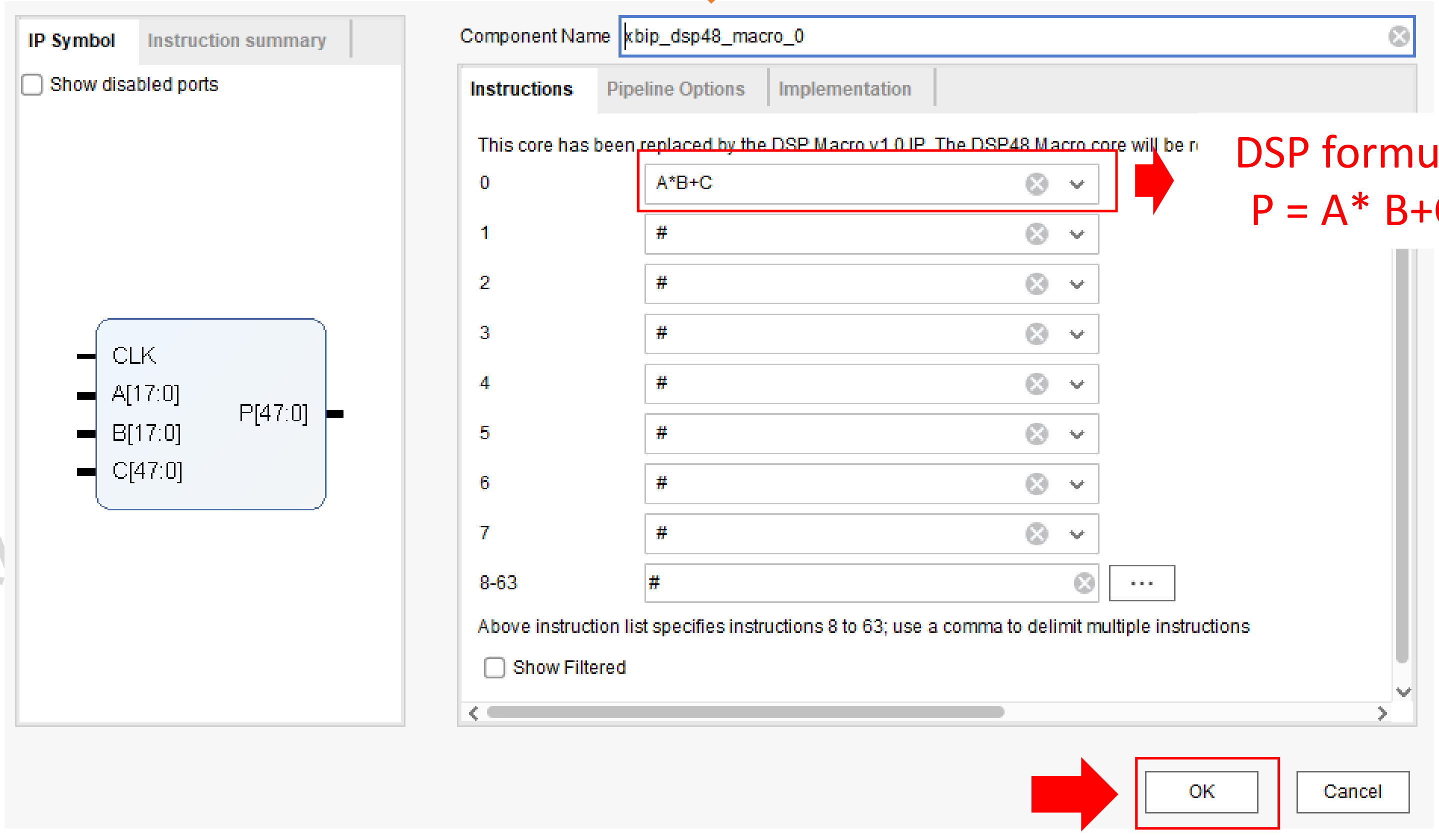
- Click "IP Catalog" on Vivado
- Search "DSP"
- Choose "DSP48 Marco" and double click on it



Creating a DSP using Xilinx IP generator

The name of the generated DSP

Ports of the DSP
Inputs:
CLK
A[17:0], B[17:0]
C[47:0]
Output:
P[47:0]



Component Name: xbip_dsp48_macro_0

This core has been replaced by the DSP Macro v1.0 IP. The DSP48 Macro core will be r

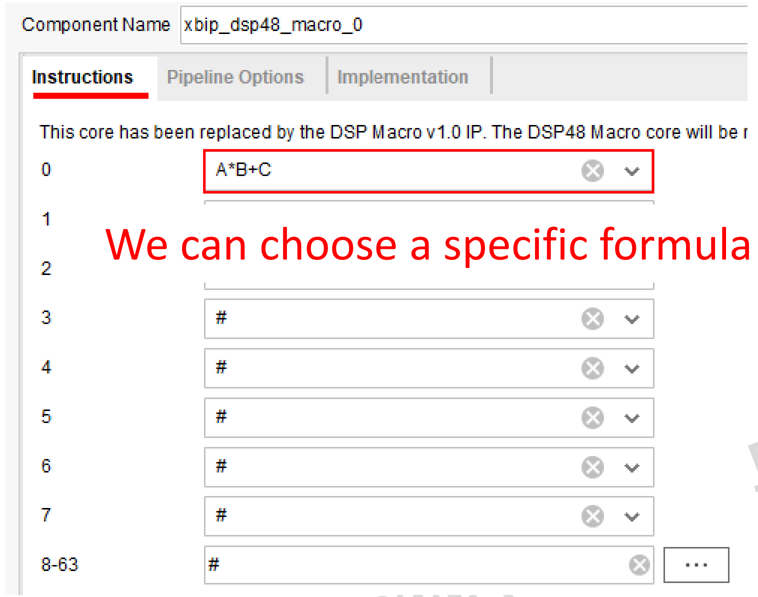
0	A*B+C
1	#
2	#
3	#
4	#
5	#
6	#
7	#
8-63	#

Above instruction list specifies instructions 8 to 63; use a comma to delimit multiple instructions

Show Filtered

DSP formula
P = A* B+C

DSP configuration



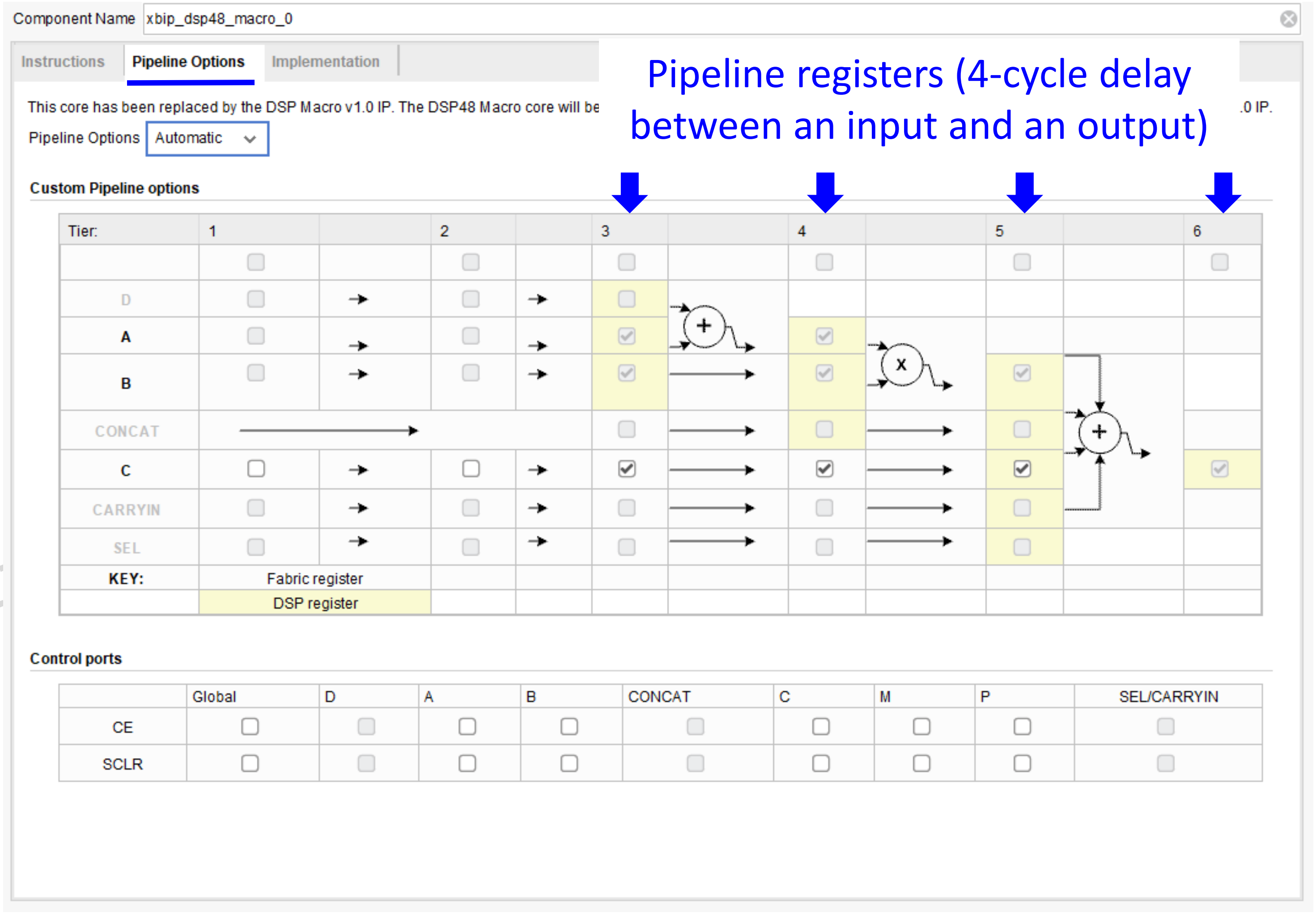
Component Name xbip_dsp48_macro_0

Instructions | Pipeline Options | Implementation

This core has been replaced by the DSP Macro v1.0 IP. The DSP48 Macro core will be r

0	A*B+C
1	
2	
3	#
4	#
5	#
6	#
7	#
8-63	#

We can choose a specific formula



Component Name xbip_dsp48_macro_0

Instructions | Pipeline Options | Implementation

This core has been replaced by the DSP Macro v1.0 IP. The DSP48 Macro core will be … .0 IP.

Pipeline Options: Automatic

Pipeline registers (4-cycle delay between an input and an output)

Custom Pipeline options

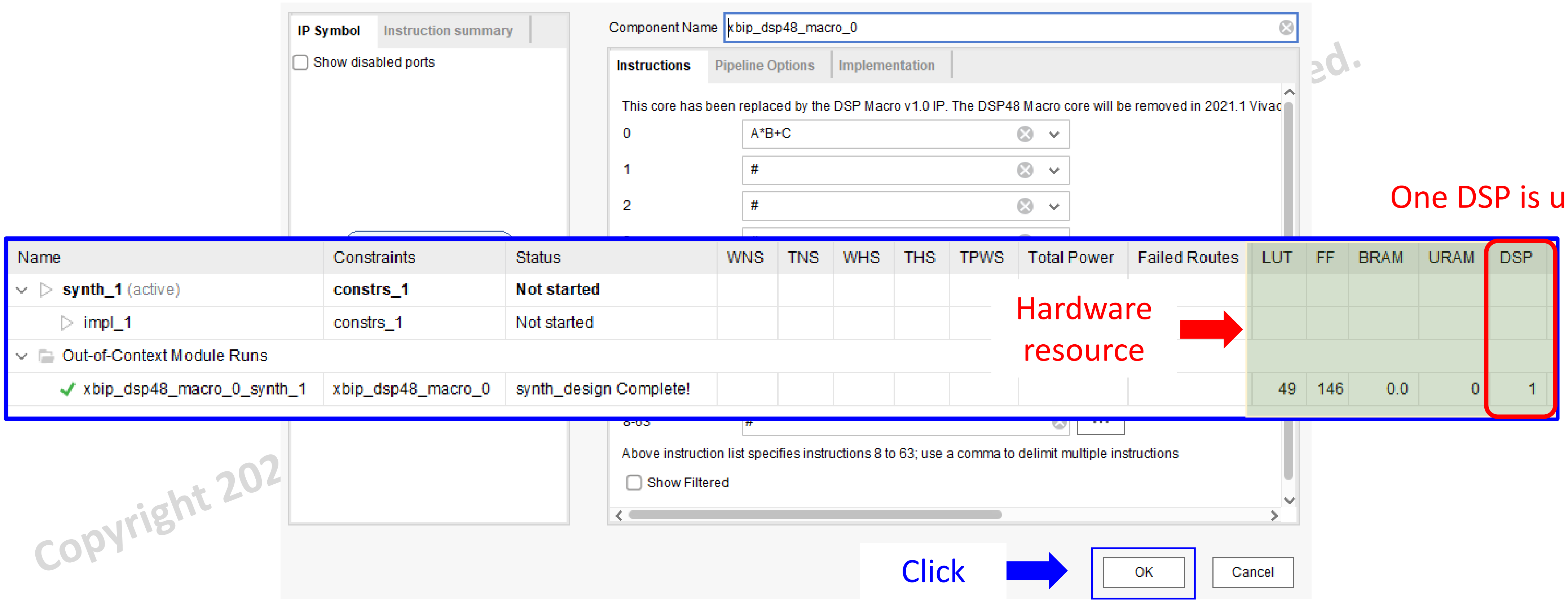
Tier:	1	2	3	4	5	6
	☐	☐	☐	☐	☐	☐
D	☐	☐	☐			
A	☐	☐	✓	✓		
B	☐	☐	✓	✓	✓	
CONCAT			☐	☐	☐	
C	☐	☐	✓	✓	✓	✓
CARRYIN	☐	☐	☐	☐	☐	
SEL	☐	☐	☐	☐	☐	
KEY:	Fabric register					
	DSP register					

Control ports

	Global	D	A	B	CONCAT	C	M	P	SEL/CARRYIN
CE	☐	☐	☐	☐	☐	☐	☐	☐	☐
SCLR	☐	☐	☐	☐	☐	☐	☐	☐	☐

Creating a DSP using Xilinx IP generator

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



IP Symbol | Instruction summary

Show disabled ports

Component Name: xbip_dsp48_macro_0

Instructions | Pipeline Options | Implementation

This core has been replaced by the DSP Macro v1.0 IP. The DSP48 Macro core will be removed in 2021.1 Vivac

0	A*B+C
1	#
2	#
8-63	#

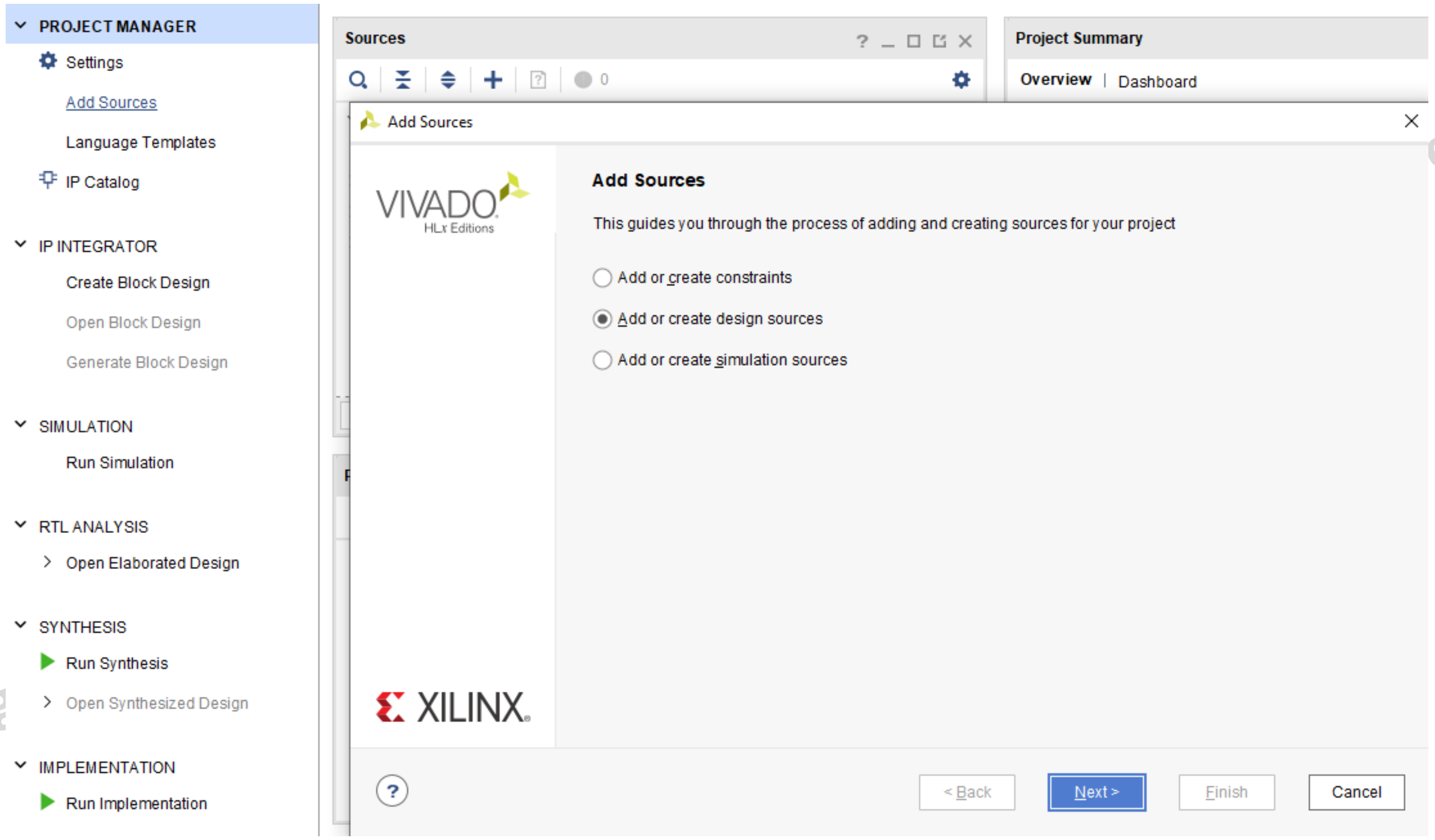
Above instruction list specifies instructions 8 to 63; use a comma to delimit multiple instructions

Show Filtered

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP
synth_1 (active)	constrs_1	Not started												
impl_1	constrs_1	Not started												
Out-of-Context Module Runs														
✓ xbip_dsp48_macro_0_synth_1	xbip_dsp48_macro_0	synth_design Complete!								49	146	0.0	0	1

Multiplier (mul.v)

- Choose “Add sources” >> “Add or create design sources” >> Click “Next”



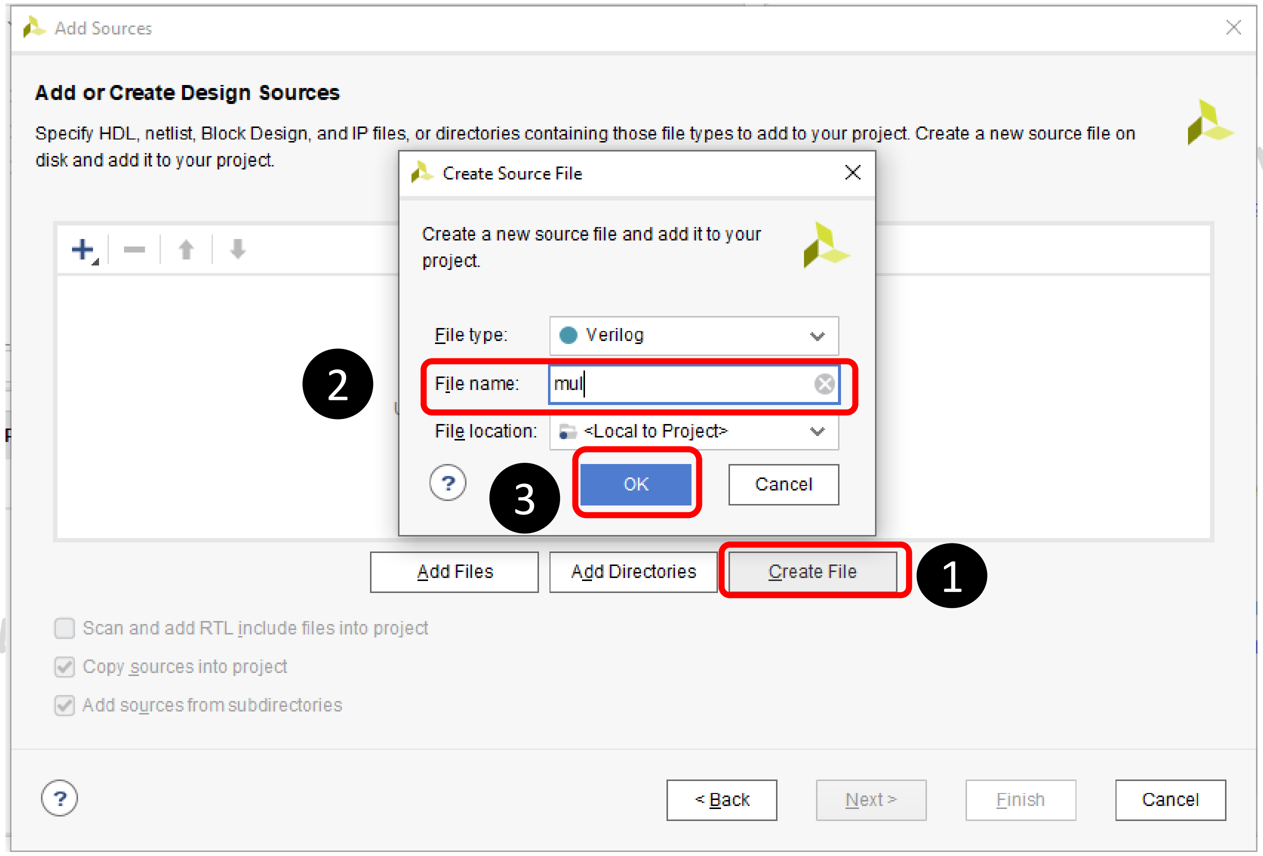
Add Sources

This guides you through the process of adding and creating sources for your project

( ) Add or create constraints
[✓ Add or create design sources]
( ) Add or create simulation sources

Multiplier (mul.v)

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



Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

Create a new source file and add it to your project.

File type: Verilog

File name: mul

File location: <Local to Project>

☐ Scan and add RTL include files into project

☑ Copy sources into project

☑ Add sources from subdirectories

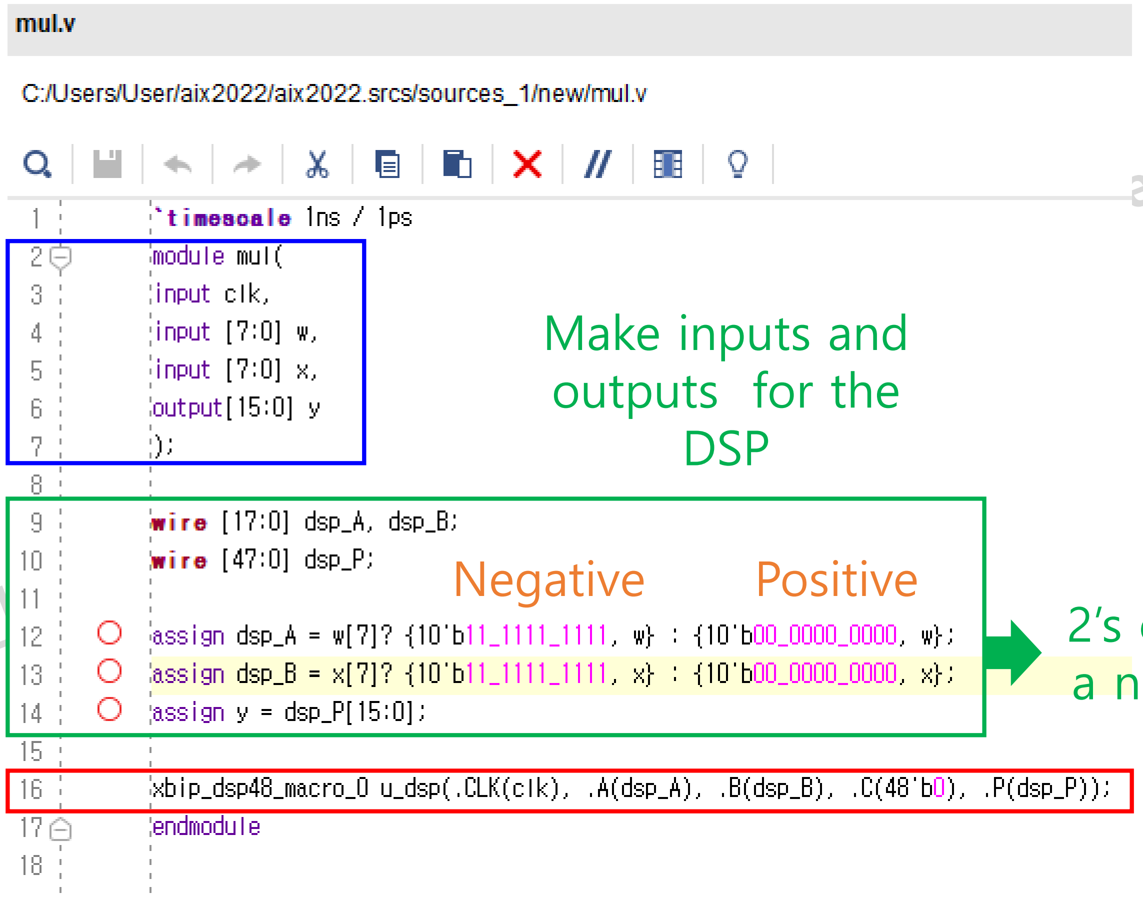
Multiplier (mul.v)

Inputs:

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

Output:

y[15:0]	// Product



```verilog
`timescale 1ns / 1ps
module mul(
input clk,
input [7:0] w,
input [7:0] x,
output[15:0] y
);

wire [17:0] dsp_A, dsp_B;
wire [47:0] dsp_P;

assign dsp_A = w[7]? {10'b11_1111_1111, w} : {10'b00_0000_0000, w};
assign dsp_B = x[7]? {10'b11_1111_1111, x} : {10'b00_0000_0000, x};
assign y = dsp_P[15:0];

xbip_dsp48_macro_0 u_dsp(.CLK(clk), .A(dsp_A), .B(dsp_B), .C(48'b0), .P(dsp_P));
endmodule
```

Make inputs and outputs for the DSP

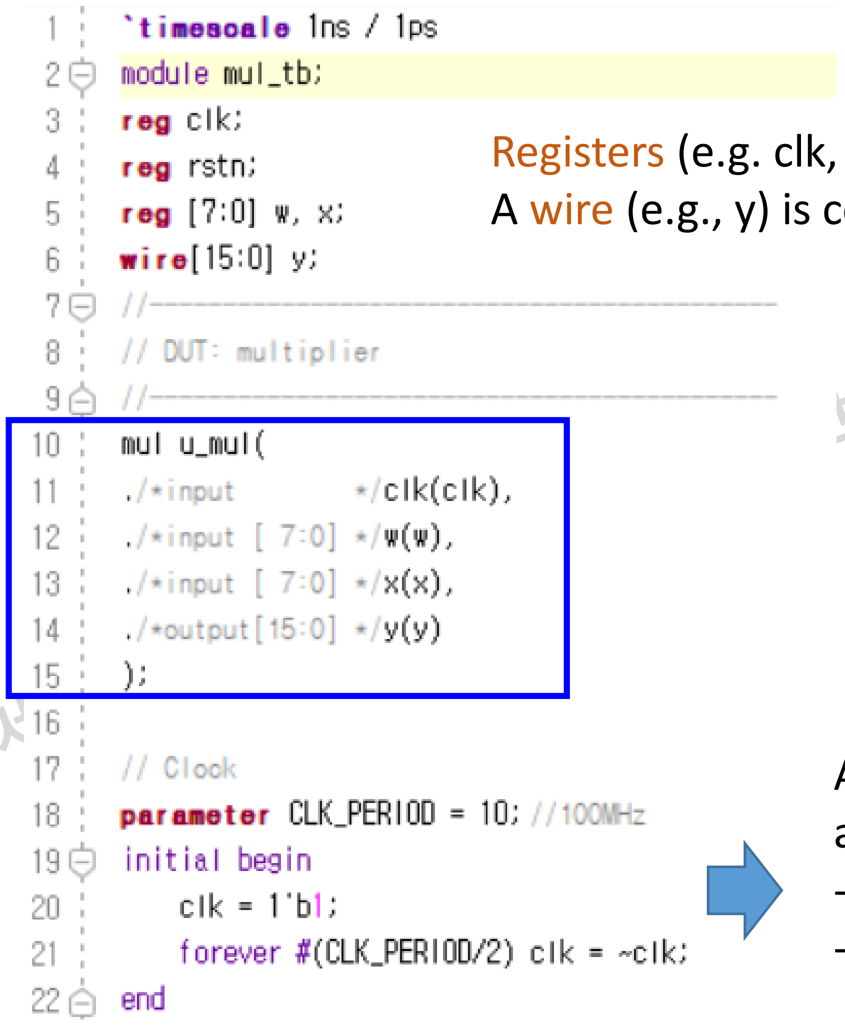
Negative Positive

2's complement for a negative number

A DSP instance

Testbench (mul_tb.v)

Time scale: 1ns



```
`timescale 1ns / 1ps
module mul_tb;
reg clk;
reg rstn;
reg [7:0] w, x;
wire[15:0] y;
//------------------------------------------------
// DUT: multiplier
//------------------------------------------------
mul u_mul(
./*input        */clk(clk),
./*input [ 7:0] */w(w),
./*input [ 7:0] */x(x),
./*output[15:0] */y(y)
);

// Clock
parameter CLK_PERIOD = 10; //100MHz
initial begin
    clk = 1'b1;
    forever #(CLK_PERIOD/2) clk = ~clk;
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)

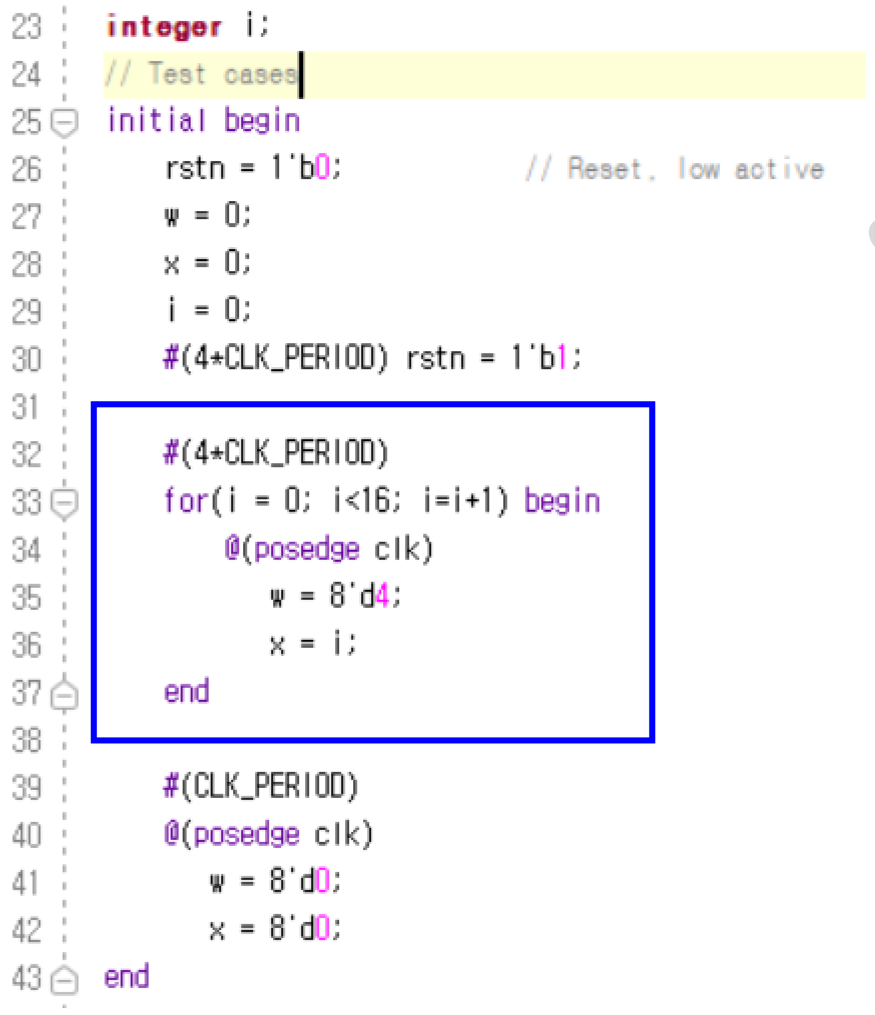
A clock signal is defined in an "initial" block

- 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



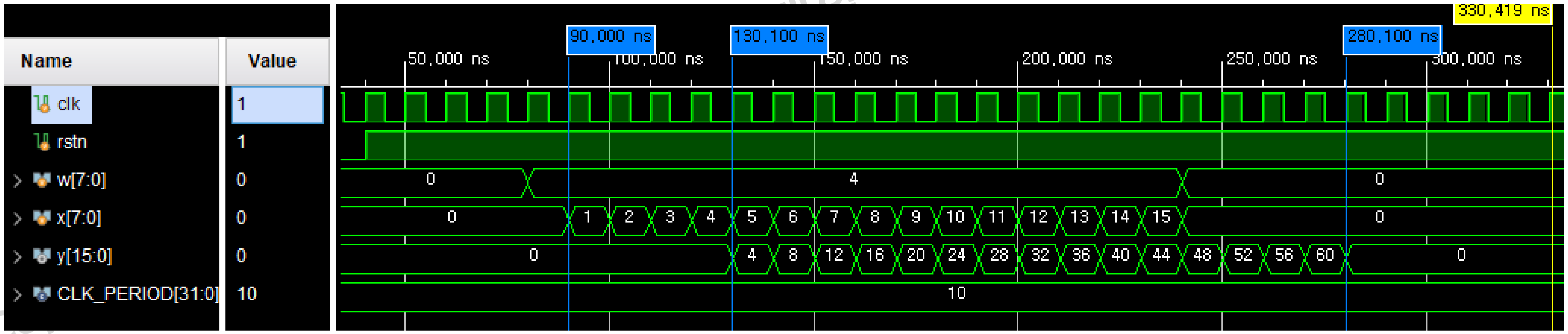
```
integer i;
// Test cases
initial begin
    rstn = 1'b0;            // Reset, low active
    w = 0;
    x = 0;
    i = 0;
    #(4*CLK_PERIOD) rstn = 1'b1;

    #(4*CLK_PERIOD)
    for(i = 0; i<16; i=i+1) begin
        @(posedge clk)
            w = 8'd4;
            x = i;
    end

    #(CLK_PERIOD)
    @(posedge clk)
        w = 8'd0;
        x = 8'd0;
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)

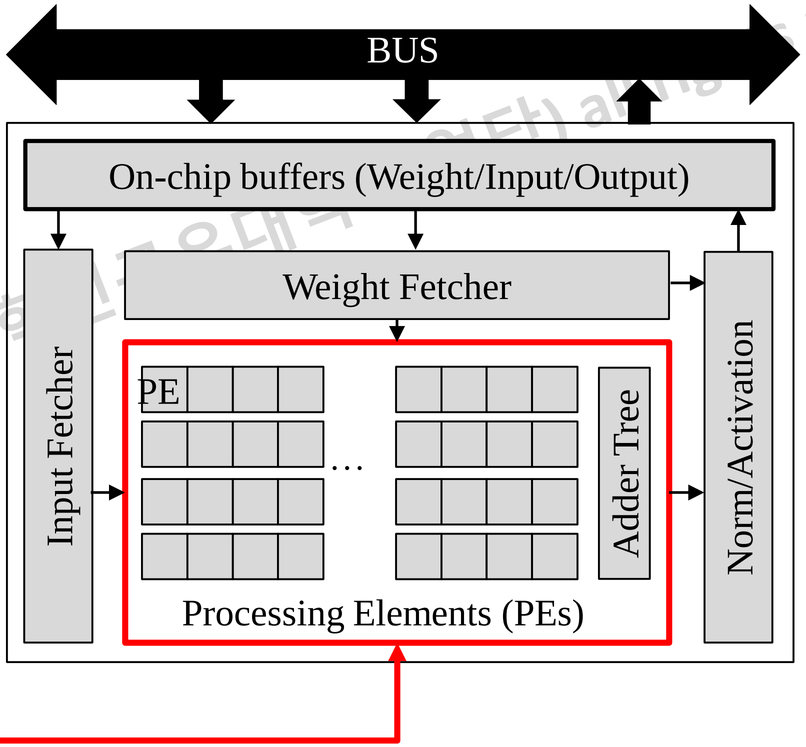


Lab 2: MAC

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

DNN accelerator

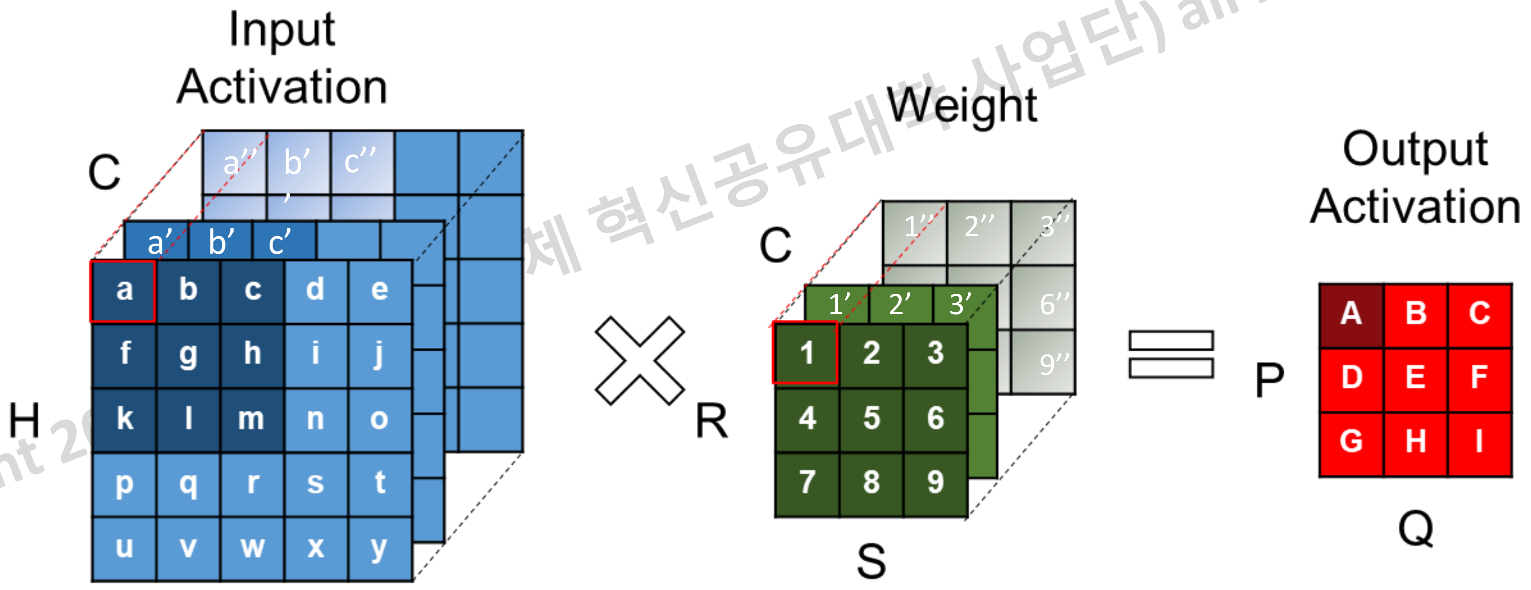
- Processing Element (PE) Array
  - An array of multiplication and accumulation (MAC).
  - Perform convolution operations.
  - Computing unit or "ALU" of a DNN accelerator


BUS
On-chip buffers (Weight/Input/Output)
Weight Fetcher
Input Fetcher
PE
…
Adder Tree
Norm/Activation
Processing Elements (PEs)

Computing Elements

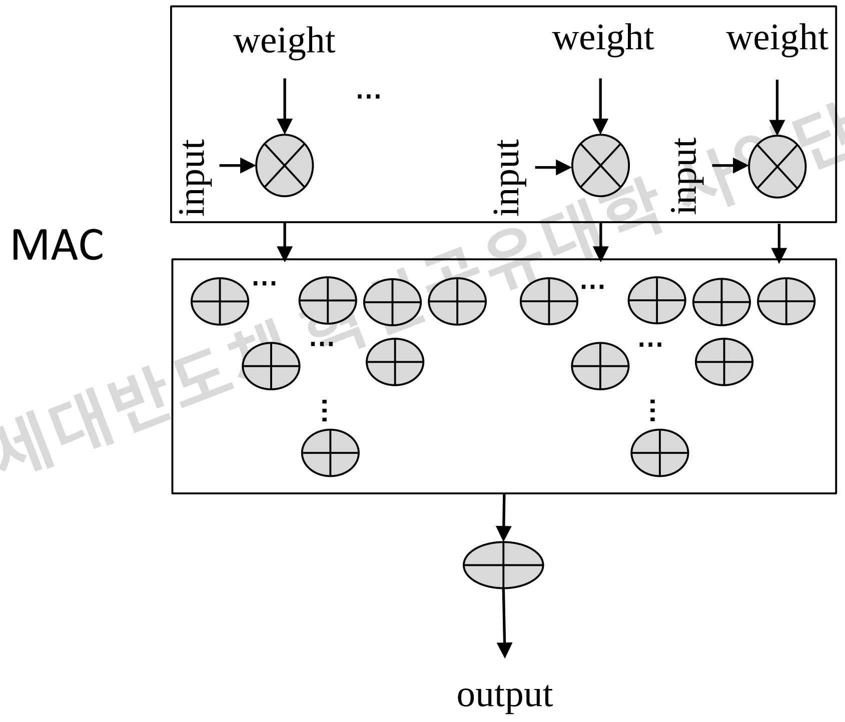
MAC

- 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

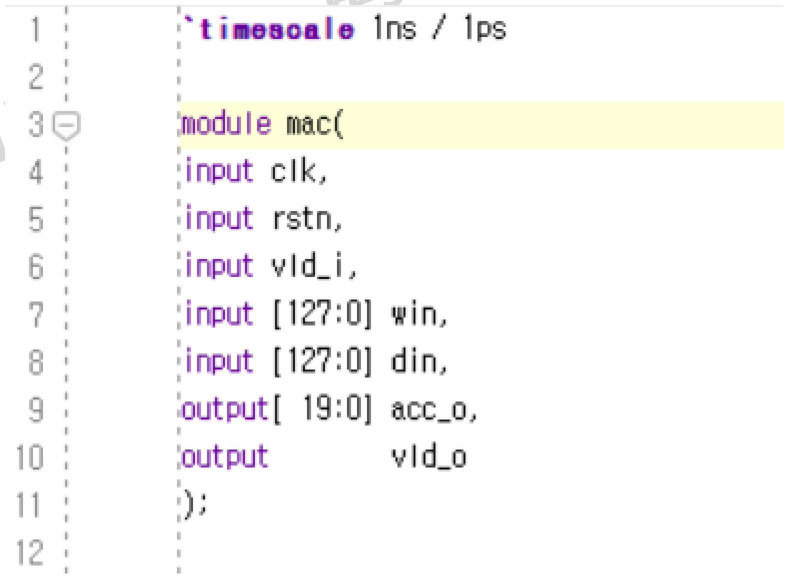
```
$y_0^{(0)} = w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15}$        // N multipliers
$y_0^{(1)} = y_0^{(0)} + y_1^{(0)}, \dots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$        // N/2 adders
$y_0^{(2)} = y_0^{(1)} + y_1^{(1)}, \dots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$        // N/4 adders
$y_0^{(3)} = y_0^{(2)} + y_1^{(2)}, \dots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$        // 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)

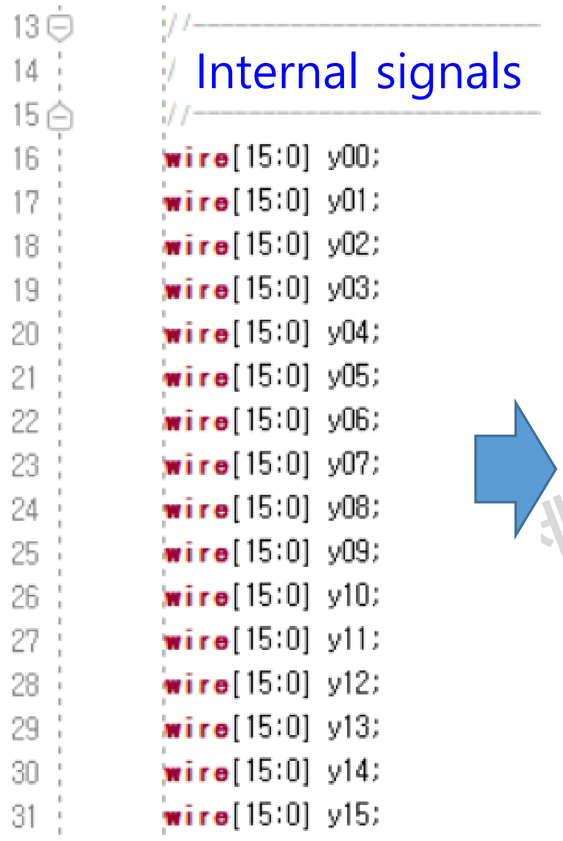


```
`timescale 1ns / 1ps

module mac(
input clk,
input rstn,
input vld_i,
input [127:0] win,
input [127:0] din,
output[ 19:0] acc_o,
output        vld_o
);

```

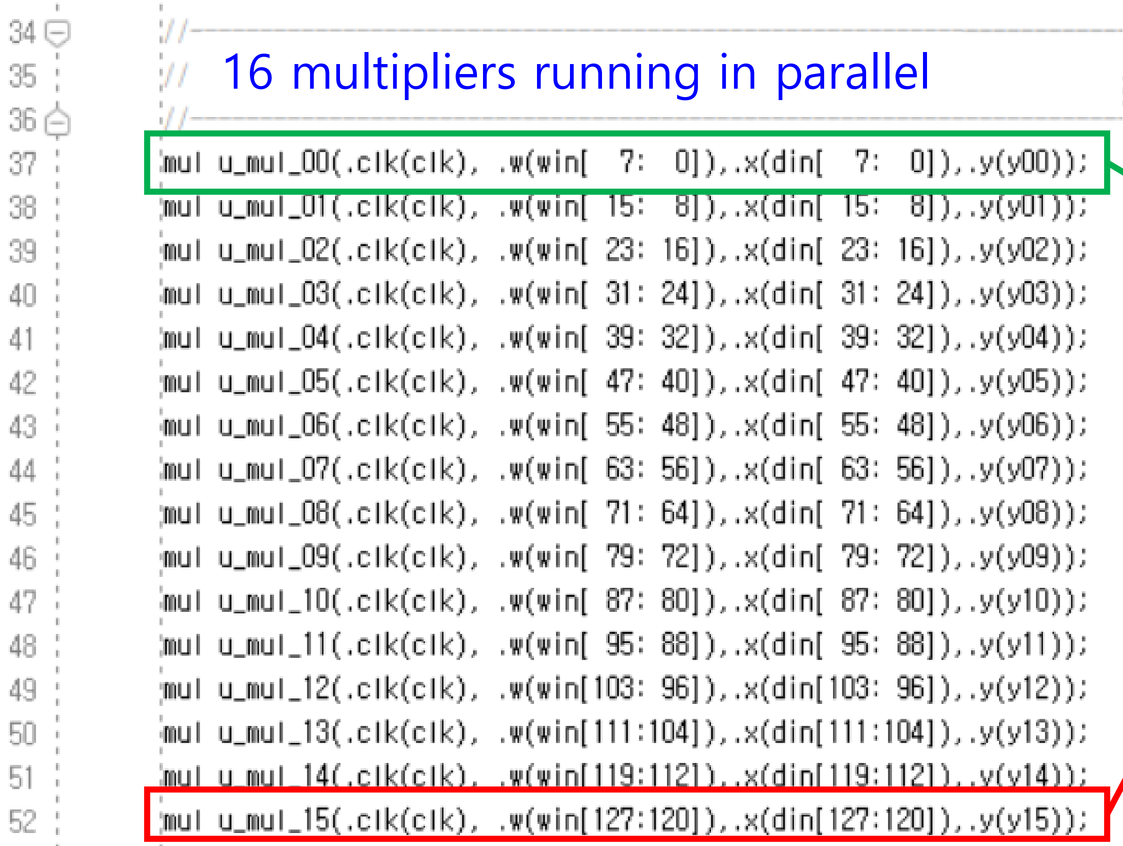
Multiplication (mac.v)

- Compute a sum of 16 products

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



```
//---------------------------
// Internal signals
//---------------------------
wire[15:0] y00;
wire[15:0] y01;
wire[15:0] y02;
wire[15:0] y03;
wire[15:0] y04;
wire[15:0] y05;
wire[15:0] y06;
wire[15:0] y07;
wire[15:0] y08;
wire[15:0] y09;
wire[15:0] y10;
wire[15:0] y11;
wire[15:0] y12;
wire[15:0] y13;
wire[15:0] y14;
wire[15:0] y15;
```



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

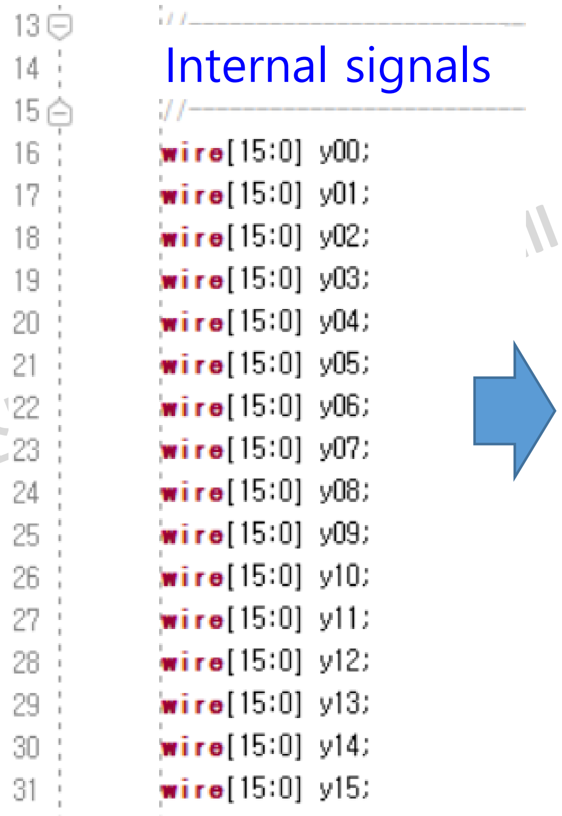
Accumulation (mac.v)

- Compute a sum of 16 products

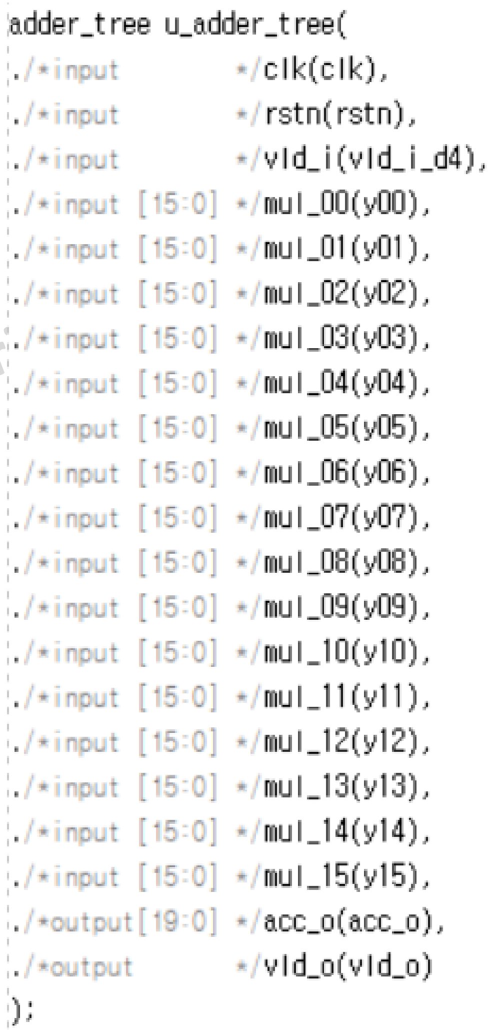
$$Y = \sum_{i=0}^{15} w_i * x_i$$

- Pseudo code

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



```
//-----------------------------------
Internal signals
//-----------------------------------
wire[15:0] y00;
wire[15:0] y01;
wire[15:0] y02;
wire[15:0] y03;
wire[15:0] y04;
wire[15:0] y05;
wire[15:0] y06;
wire[15:0] y07;
wire[15:0] y08;
wire[15:0] y09;
wire[15:0] y10;
wire[15:0] y11;
wire[15:0] y12;
wire[15:0] y13;
wire[15:0] y14;
wire[15:0] y15;
```

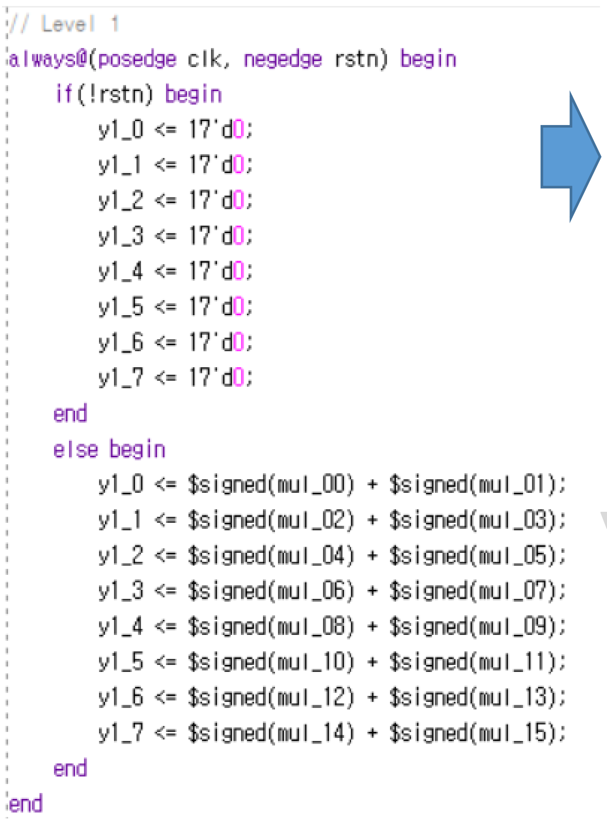


```
adder_tree u_adder_tree(
./*input          */clk(clk),
./*input          */rstn(rstn),
./*input          */vld_i(vld_i_d4),
./*input  [15:0]  */mul_00(y00),
./*input  [15:0]  */mul_01(y01),
./*input  [15:0]  */mul_02(y02),
./*input  [15:0]  */mul_03(y03),
./*input  [15:0]  */mul_04(y04),
./*input  [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)
);
```

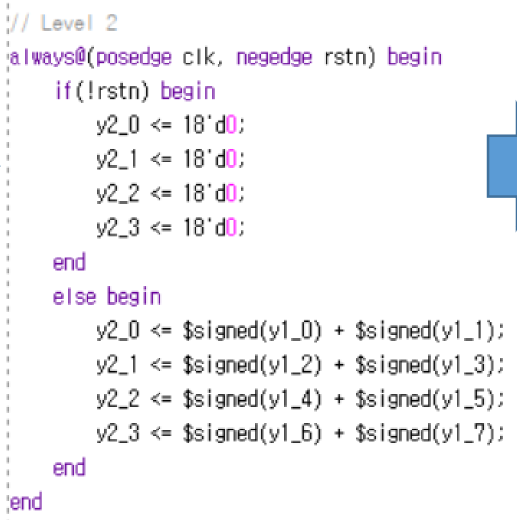
Accumulation (adder_tree.v)

- Compute a sum of 16 products

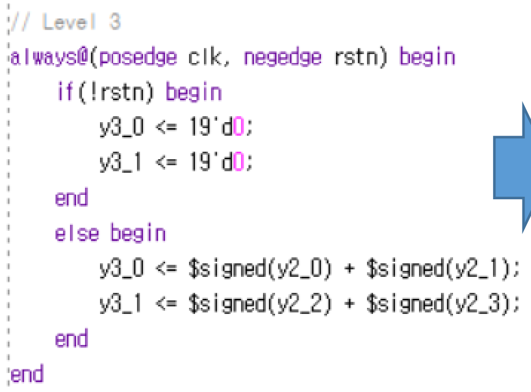
$$Y = \sum_{i=0}^{15} w_i * x_i$$



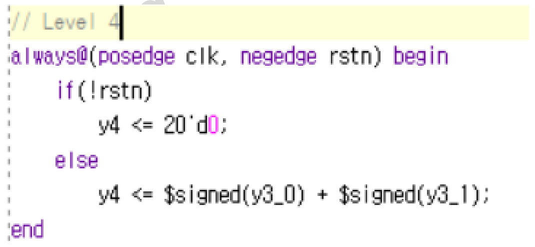
```
// Level 1
always@(posedge clk, negedge rstn) begin
    if(!rstn) begin
        y1_0 <= 17'd0;
        y1_1 <= 17'd0;
        y1_2 <= 17'd0;
        y1_3 <= 17'd0;
        y1_4 <= 17'd0;
        y1_5 <= 17'd0;
        y1_6 <= 17'd0;
        y1_7 <= 17'd0;
    end
    else begin
        y1_0 <= $signed(mul_00) + $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
```



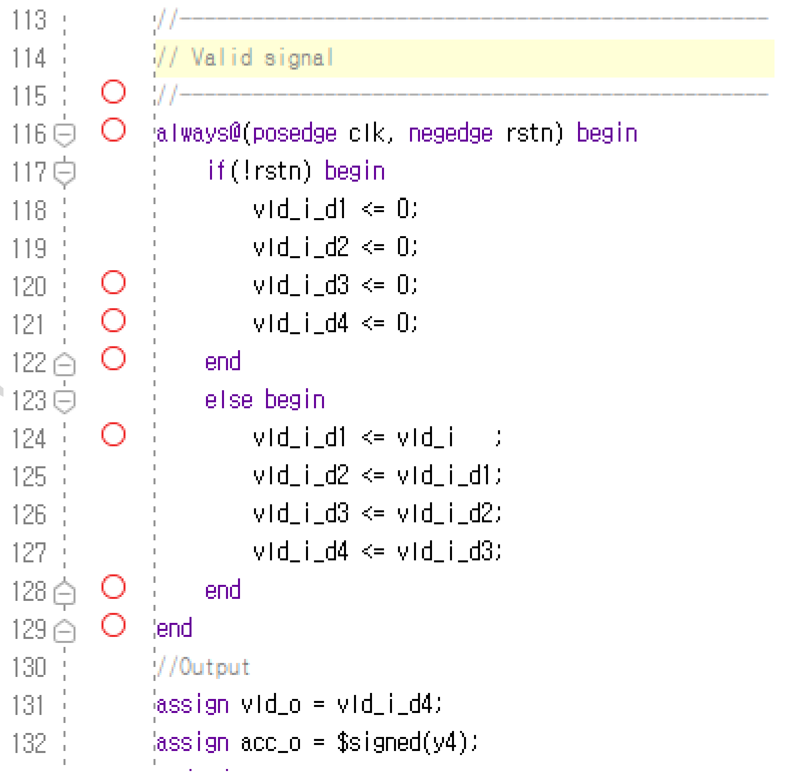
```
// Level 3
always@(posedge clk, negedge rstn) begin
    if(!rstn) begin
        y3_0 <= 19'd0;
        y3_1 <= 19'd0;
    end
    else begin
        y3_0 <= $signed(y2_0) + $signed(y2_1);
        y3_1 <= $signed(y2_2) + $signed(y2_3);
    end
end
```



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

Delays

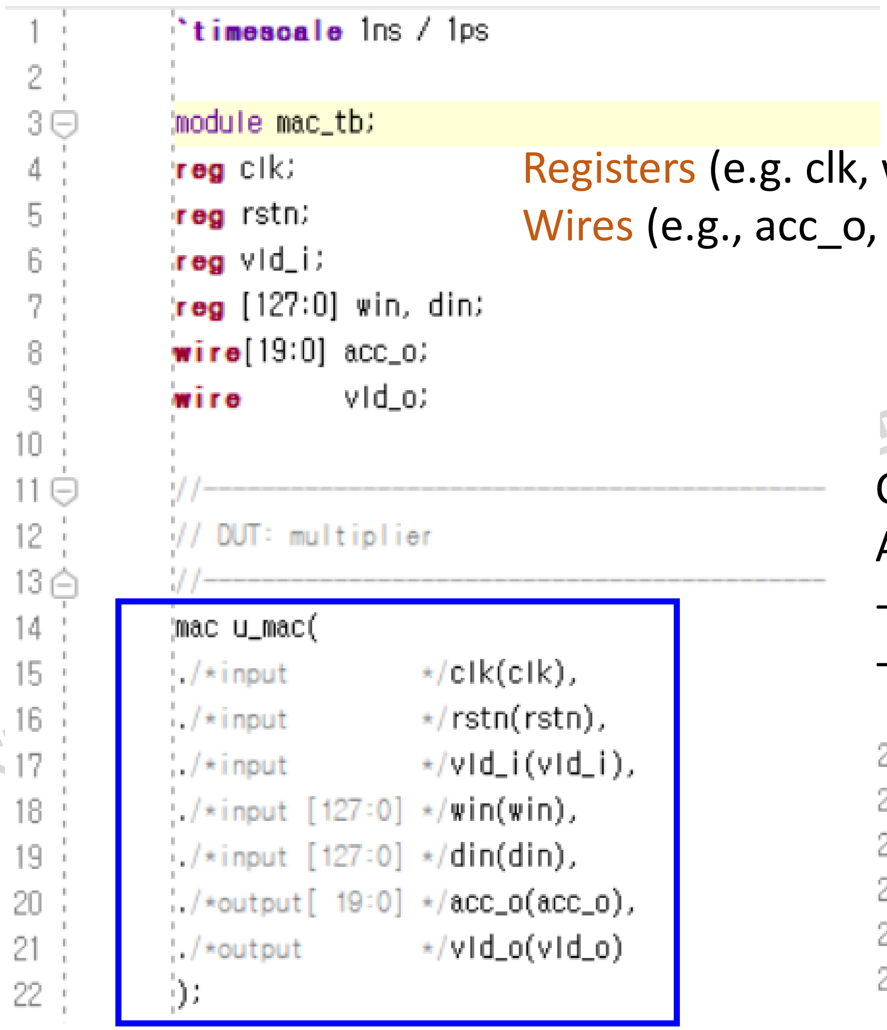
- Calculating delayed valid signals
- vld_i: when high, indicates that win and din are valid inputs, and must be calculated
- Delays in the adder tree (adder_tree.v)
  - vld_d: an internal variable to calculate the timing for vld_o to be high
  - vld_o: when high, indicates that output acc_o is a valid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



```
//----------------------------------------------------------------
// Valid signal
//----------------------------------------------------------------
always@(posedge clk, negedge rstn) begin
    if(!rstn) begin
        vld_i_d1 <= 0;
        vld_i_d2 <= 0;
        vld_i_d3 <= 0;
        vld_i_d4 <= 0;
    end
    else begin
        vld_i_d1 <= vld_i    ;
        vld_i_d2 <= vld_i_d1;
        vld_i_d3 <= vld_i_d2;
        vld_i_d4 <= vld_i_d3;
    end
end
//Output
assign vld_o = vld_i_d4;
assign acc_o = $signed(y4);
```

Test bench (mac_tb.v)

Time scale: 1ns

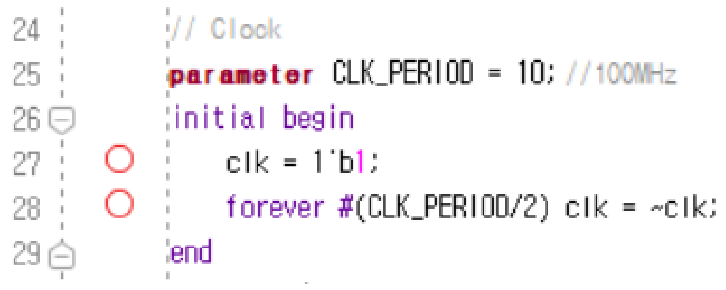


```
`timescale 1ns / 1ps

module mac_tb;
reg clk;
reg rstn;
reg vld_i;
reg [127:0] win, din;
wire[19:0] acc_o;
wire       vld_o;

//-----------------------------------------------------
// DUT: multiplier
//-----------------------------------------------------
mac u_mac(
./*input              */clk(clk),
./*input              */rstn(rstn),
./*input              */vld_i(vld_i),
./*input  [127:0]     */win(win),
./*input  [127:0]     */din(din),
./*output[ 19:0]      */acc_o(acc_o),
./*output             */vld_o(vld_o)
);
```

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

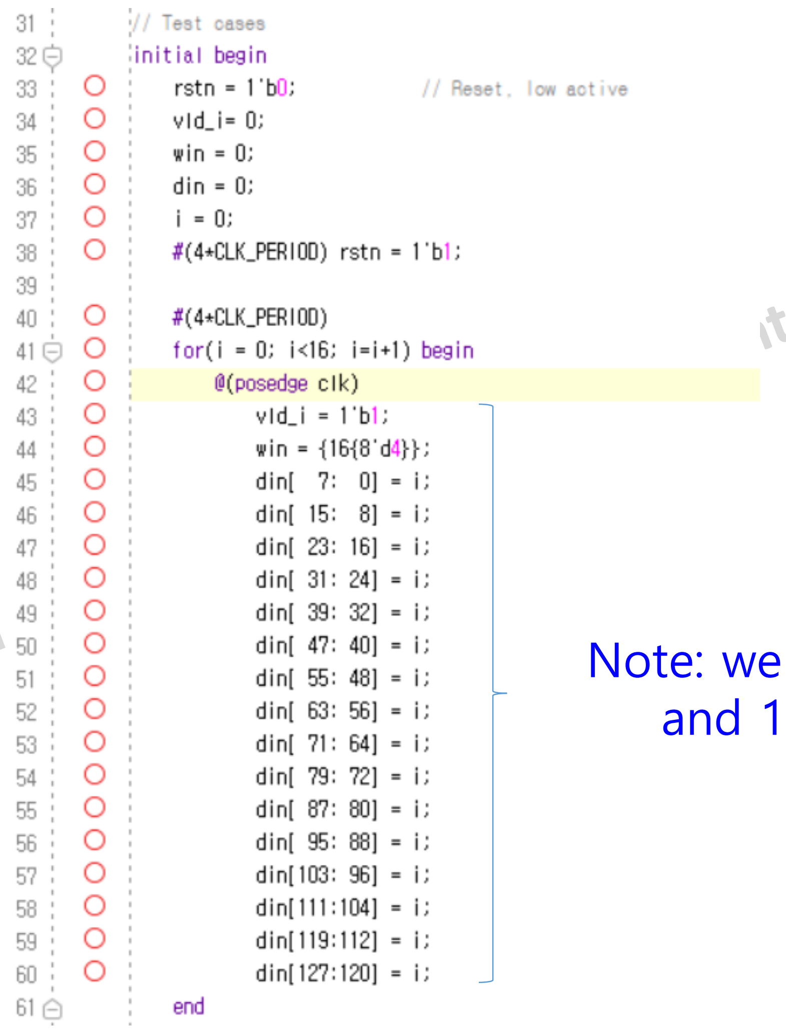
Clock: 100MHz
A clock signal is defined in an "initial" block
- Initialized at 1
- State is changed at every 5ns



```
// Clock
parameter CLK_PERIOD = 10; //100MHz
initial begin
    clk = 1'b1;
    forever #(CLK_PERIOD/2) clk = ~clk;
end
```

Test bench (mac_tb.v)

Initialized states
Set all registers to default values

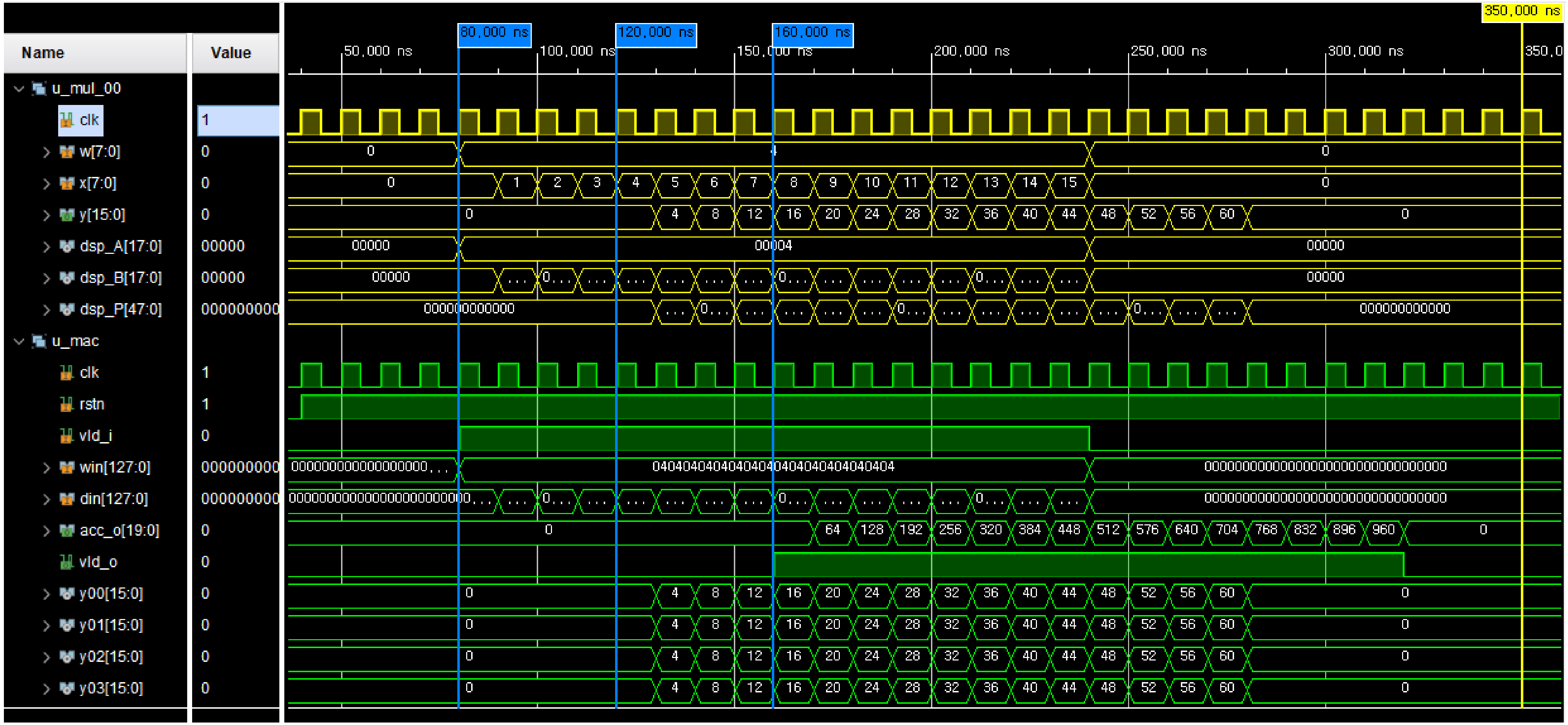


```
// Test cases
initial begin
    rstn = 1'b0;            // Reset, low active
    vld_i= 0;
    win = 0;
    din = 0;
    i = 0;
    #(4*CLK_PERIOD) rstn = 1'b1;

    #(4*CLK_PERIOD)
    for(i = 0; i<16; i=i+1) begin
        @(posedge clk)
            vld_i = 1'b1;
            win = {16{8'd4}};
            din[  7:  0] = i;
            din[ 15:  8] = i;
            din[ 23: 16] = i;
            din[ 31: 24] = i;
            din[ 39: 32] = i;
            din[ 47: 40] = i;
            din[ 55: 48] = i;
            din[ 63: 56] = i;
            din[ 71: 64] = i;
            din[ 79: 72] = i;
            din[ 87: 80] = i;
            din[ 95: 88] = i;
            din[103: 96] = i;
            din[111:104] = i;
            din[119:112] = i;
            din[127:120] = i;
    end
```

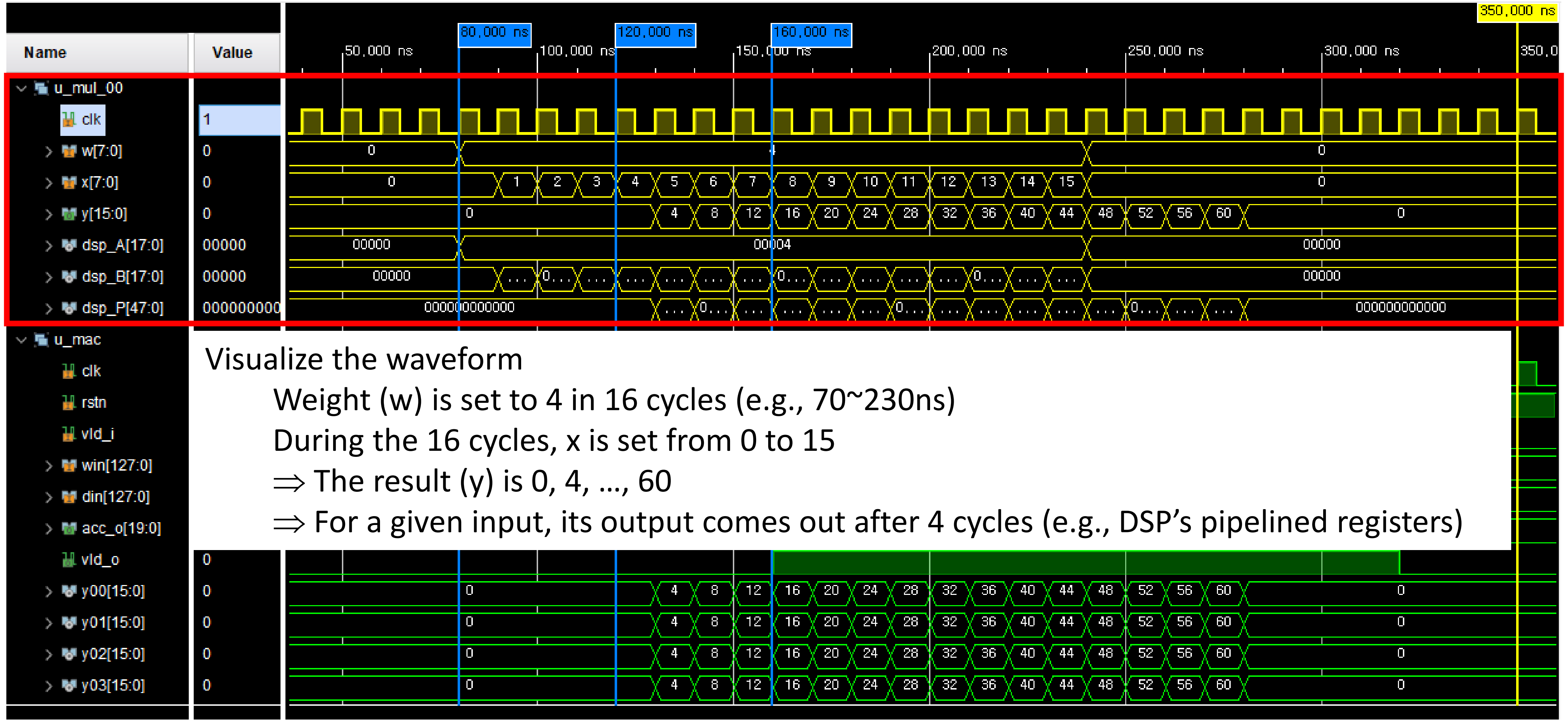
Generate test cases in 16 cycles
w: 4
x: 0 -> 1 -> 2 -> … -> 15

Note: we have 16 weights
and 16 inputs now.

Waveform


80,000 ns
120,000 ns
160,000 ns
350,000 ns
50,000 ns
100,000 ns
150,000 ns
200,000 ns
250,000 ns
300,000 ns
Name
Value
u_mul_00
clk 1
w[7:0] 0
x[7:0] 0
y[15:0] 0
dsp_A[17:0] 00000
dsp_B[17:0] 00000
dsp_P[47:0] 000000000
u_mac
clk 1
rstn 1
vld_i 0
win[127:0] 000000000
din[127:0] 000000000
acc_o[19:0] 0
vld_o 0
y00[15:0] 0
y01[15:0] 0
y02[15:0] 0
y03[15:0] 0

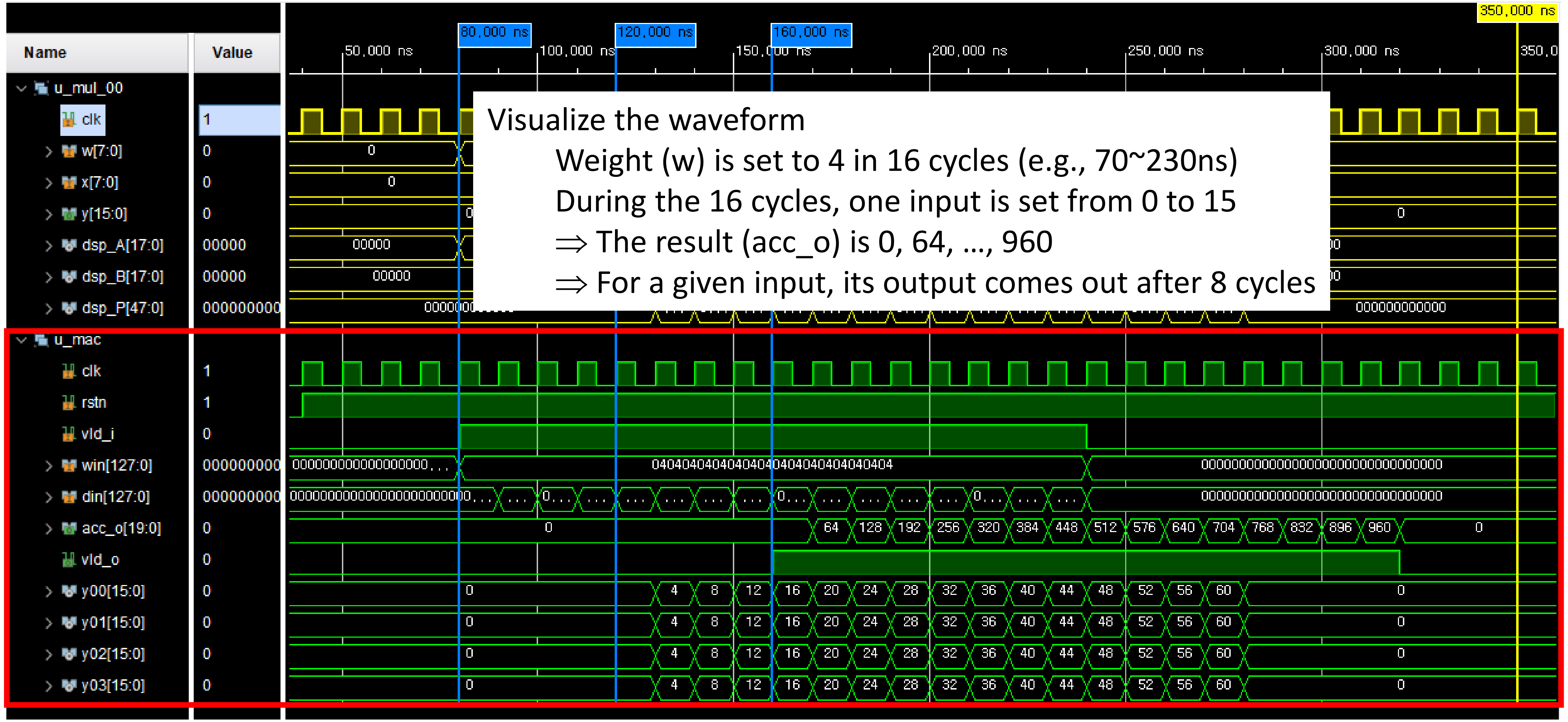
Waveform



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



Visualize the waveform

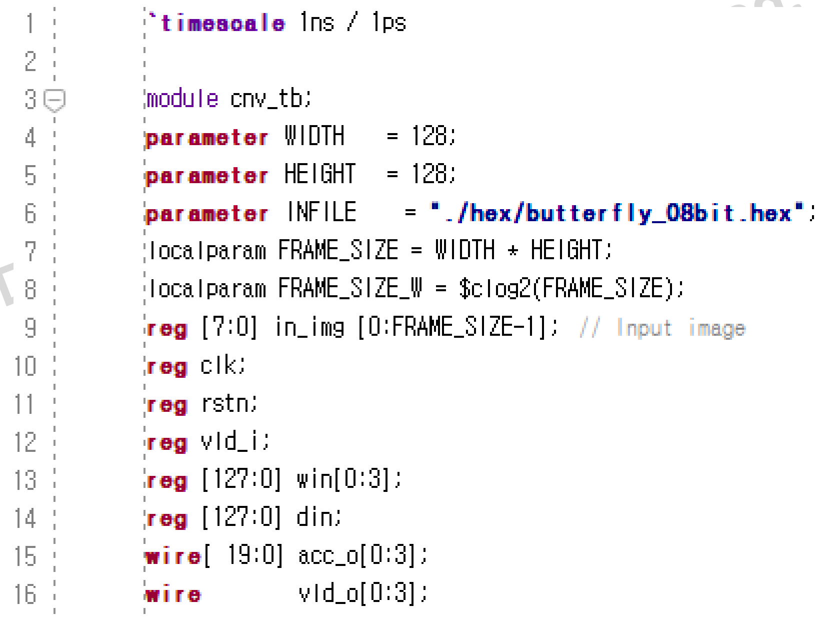
- Weight (w) is set to 4 in 16 cycles (e.g., 70~230ns)
- During the 16 cycles, one input is set from 0 to 15
- ⇒ The result (acc_o) is 0, 64, …, 960
- ⇒ For a given input, its output comes out after 8 cycles

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)

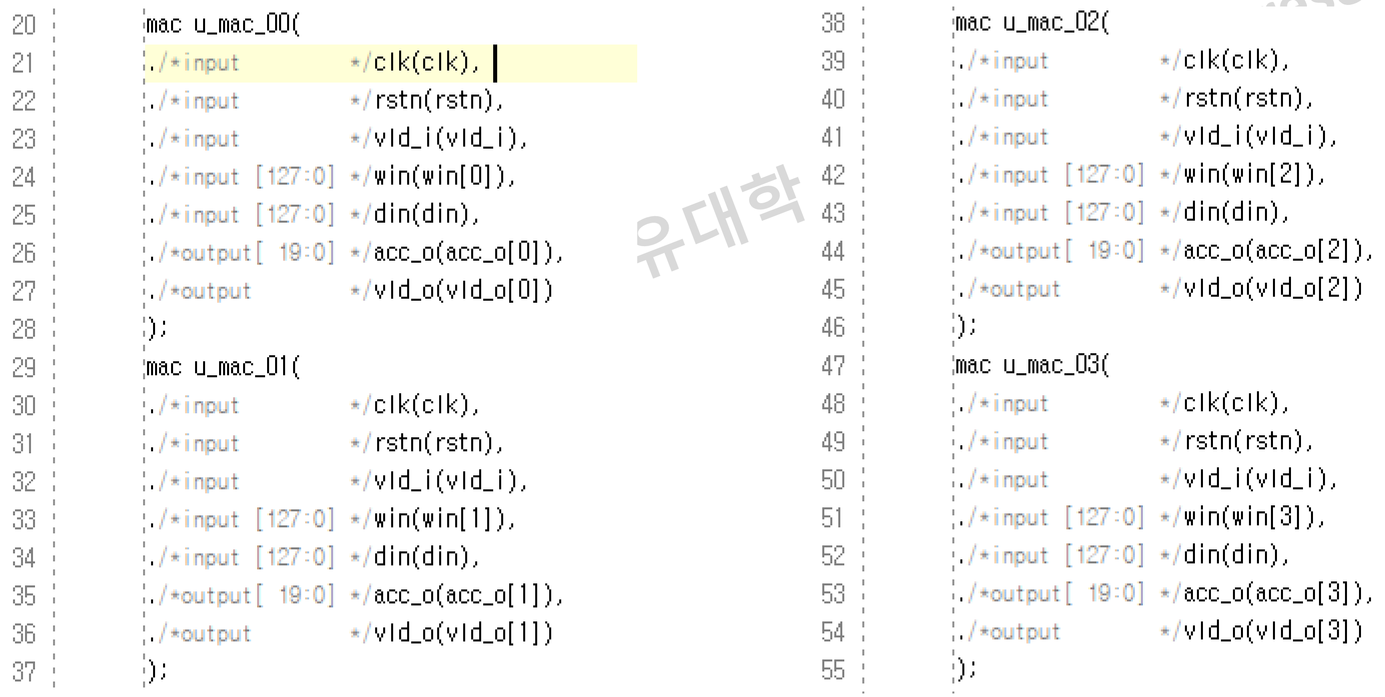
- Parameters for an image file, e.g., WIDTH, HEIGHT, file location (INFILE), frame/image size
- A buffer to store an image (in_img)
- Internal signals
    - clk, rstn,
    - vld_i, din[127:0]
    - Four sets of filters win[0:3][127:0]
    - acc_o[0:3][19:0]
    - vld_o[0:3]



```
`timescale 1ns / 1ps

module cnv_tb;
parameter WIDTH   = 128;
parameter HEIGHT  = 128;
parameter INFILE   = "./hex/butterfly_08bit.hex";
localparam FRAME_SIZE = WIDTH * HEIGHT;
localparam FRAME_SIZE_W = $clog2(FRAME_SIZE);
reg [7:0] in_img [0:FRAME_SIZE-1]; // Input image
reg clk;
reg rstn;
reg vld_i;
reg [127:0] win[0:3];
reg [127:0] din;
wire[ 19:0] acc_o[0:3];
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

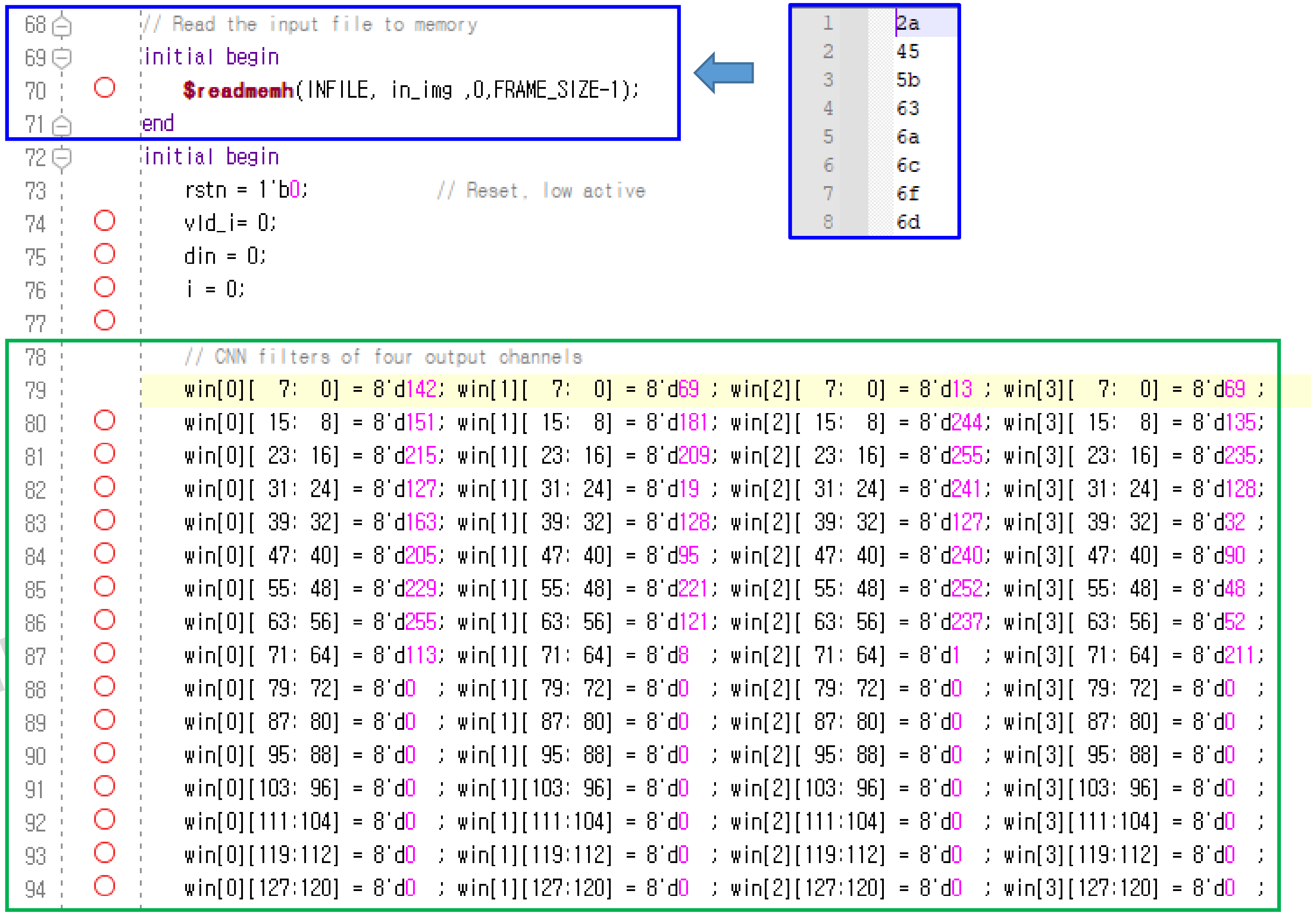


```
mac u_mac_00(
./*input          */clk(clk),
./*input          */rstn(rstn),
./*input          */vld_i(vld_i),
./*input [127:0]  */win(win[0]),
./*input [127:0]  */din(din),
./*output[ 19:0]  */acc_o(acc_o[0]),
./*output         */vld_o(vld_o[0])
);
mac u_mac_01(
./*input          */clk(clk),
./*input          */rstn(rstn),
./*input          */vld_i(vld_i),
./*input [127:0]  */win(win[1]),
./*input [127:0]  */din(din),
./*output[ 19:0]  */acc_o(acc_o[1]),
./*output         */vld_o(vld_o[1])
);
```

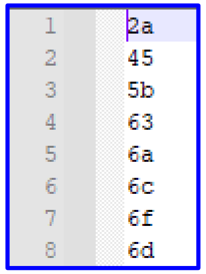
```
mac u_mac_02(
./*input          */clk(clk),
./*input          */rstn(rstn),
./*input          */vld_i(vld_i),
./*input [127:0]  */win(win[2]),
./*input [127:0]  */din(din),
./*output[ 19:0]  */acc_o(acc_o[2]),
./*output         */vld_o(vld_o[2])
);
mac u_mac_03(
./*input          */clk(clk),
./*input          */rstn(rstn),
./*input          */vld_i(vld_i),
./*input [127:0]  */win(win[3]),
./*input [127:0]  */din(din),
./*output[ 19:0]  */acc_o(acc_o[3]),
./*output         */vld_o(vld_o[3])
);
```

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;

    // 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  ;
```

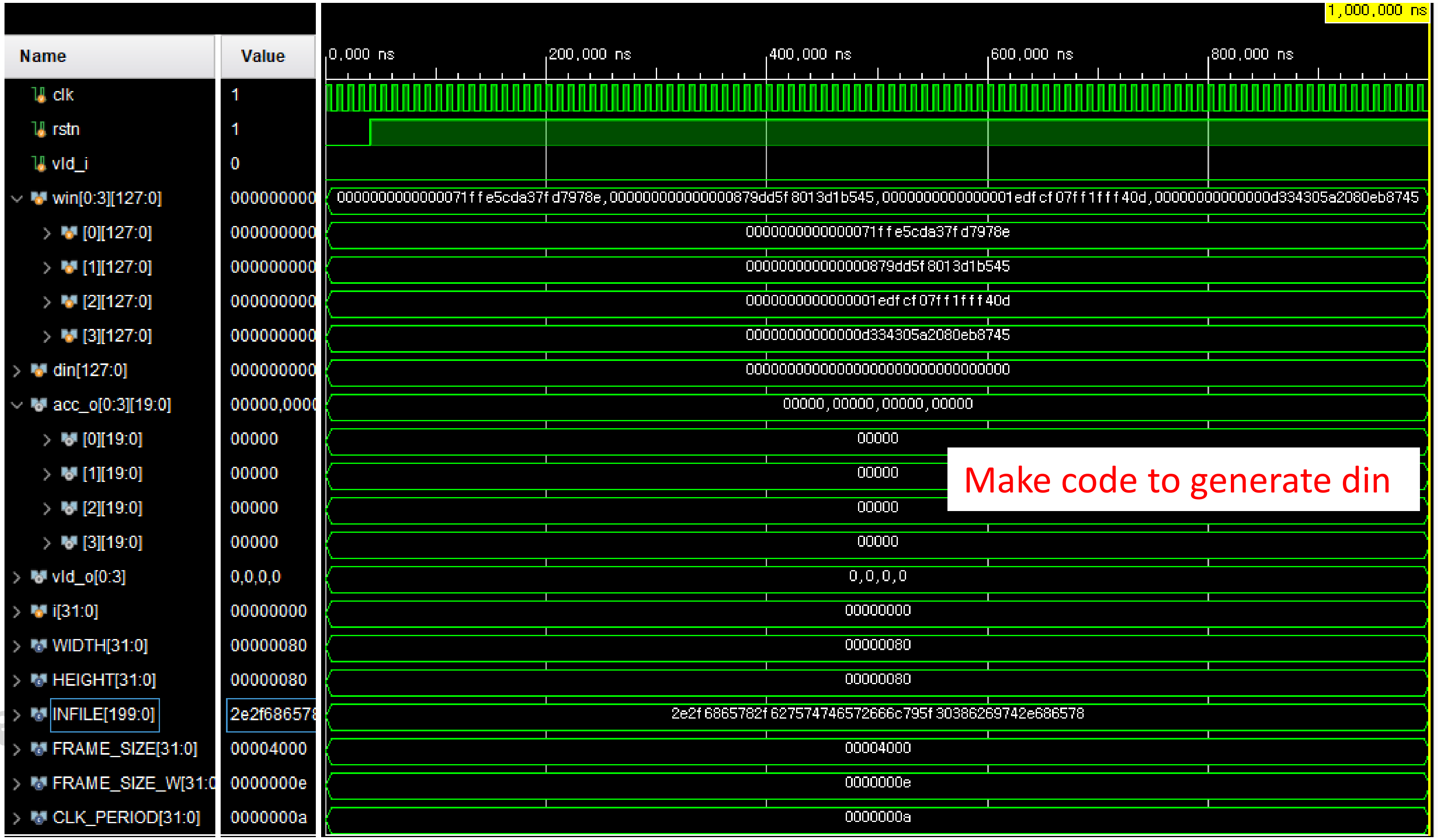


```
2a
45
5b
63
6a
6c
6f
6d
```

Initialize four set of convolutional filters

⇒ Come from a quantized model

Waveform



```
1,000,000 ns
0,000 ns   200,000 ns   400,000 ns   600,000 ns   800,000 ns

Name                Value
clk                 1
rstn                1
vld_i               0
win[0:3][127:0]     000000000     0000000000000071ffe5cda37fd7978e,00000000000000879dd5f8013d1b545,0000000000000001edfcf07ff1fff40d,0000000000000d334305a2080eb8745
  [0][127:0]        000000000     0000000000000071ffe5cda37fd7978e
  [1][127:0]        000000000     00000000000000879dd5f8013d1b545
  [2][127:0]        000000000     0000000000000001edfcf07ff1fff40d
  [3][127:0]        000000000     0000000000000d334305a2080eb8745
din[127:0]          000000000     00000000000000000000000000000000
acc_o[0:3][19:0]    00000,0000    00000,00000,00000,00000
  [0][19:0]         00000         00000
  [1][19:0]         00000         00000
  [2][19:0]         00000         00000
  [3][19:0]         00000         00000
vld_o[0:3]          0,0,0,0       0,0,0,0
i[31:0]             00000000      00000000
WIDTH[31:0]         00000080      00000080
HEIGHT[31:0]        00000080      00000080
INFILE[199:0]       2e2f686578    2e2f6865782f6275747465726666c795f30386269742e686578
FRAME_SIZE[31:0]    00004000      00004000
FRAME_SIZE_W[31:0   0000000e      0000000e
CLK_PERIOD[31:0]    0000000a      0000000a
```

Make code to generate din