DSP Manual

With Nexys A7-100T

서울대학교 차세대반도체 혁신공유대학

Xuan-Truong Nguyen (응웬트렁)

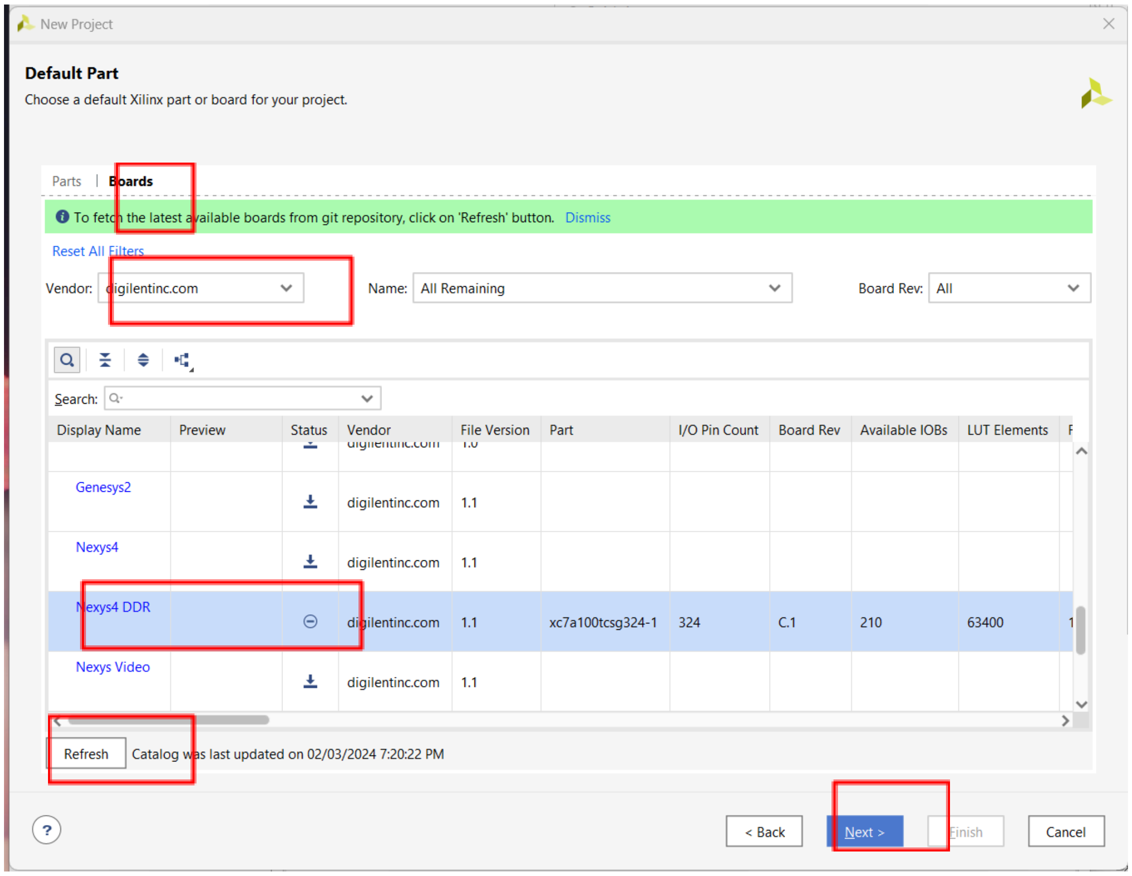
truongnx@snu.ac.kr

1. Nexys A7-100T in Vivado

To examine the board Nexys A7-100T in Vivado, you can do the following steps:

- **Open a new project** with Vivado
- Click **Next** until it opens the window “**Default Part”**, “Choose a default Xilinx part or board for your project”
- Select Tab **“Boards”** → Click “**Refresh**” □ Select **Vendor** “diligentinc.com” → Select “**Nexys4 DDR”**□ Install

⇒ You can examine many parameters of the FPGA chip xc7a100tcsg324-1 like 324 IO pins, 210 IOBs, 63,400 (=15,850×4) LUTs, 126,800 (=15,850×8) FFs, 135 BRAMs (each BRAM = 36kbit), and 240 DSPs.



New Project

Default Part

Choose a default Xilinx part or board for your project.

Parts | **Boards**

To fetch the latest available boards from git repository, click on 'Refresh' button. Dismiss

Reset All Filters

Vendor: digilentinc.com Name: All Remaining Board Rev: All

Search:

Display Name	Preview	Status	Vendor	File Version	Part	I/O Pin Count	Board Rev	Available IOBs	LUT Elements
Genesys2		⤓	digilentinc.com	1.1					
Nexys4		⤓	digilentinc.com	1.1					
Nexys4 DDR		⊖	digilentinc.com	1.1	xc7a100tcsg324-1	324	C.1	210	63400
Nexys Video		⤓	digilentinc.com	1.1					

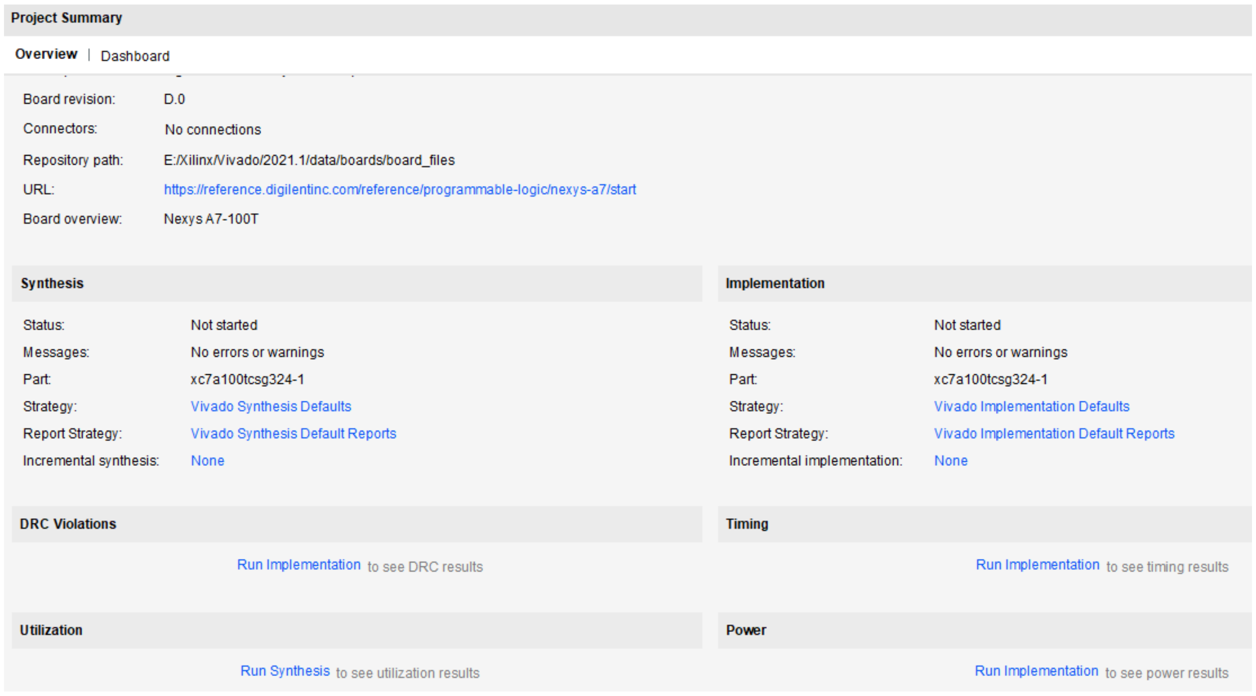
Refresh Catalog was last updated on 02/03/2024 7:20:22 PM

< Back Next > Finish Cancel

- Now, you can select Nexys4 DDR(or Nexys A7-100T in the later version Vivado, since the Nexys 4 DDR has been replaced by the Nexys A7)

  ** When you actually use the board later, you can download&add the external IP and use the same 'Nexys A7-100T' board in the old version Vivado too.*

- Click Next to create a new project.



Project Summary

Overview | Dashboard

Board revision:	D.0
Connectors:	No connections
Repository path:	E:/Xilinx/Vivado/2021.1/data/boards/board_files
URL:	https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/start
Board overview:	Nexys A7-100T

Synthesis

Status:	Not started
Messages:	No errors or warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	None

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Reports
Incremental implementation:	None

DRC Violations

Run Implementation to see DRC results

Timing

Run Implementation to see timing results

Utilization

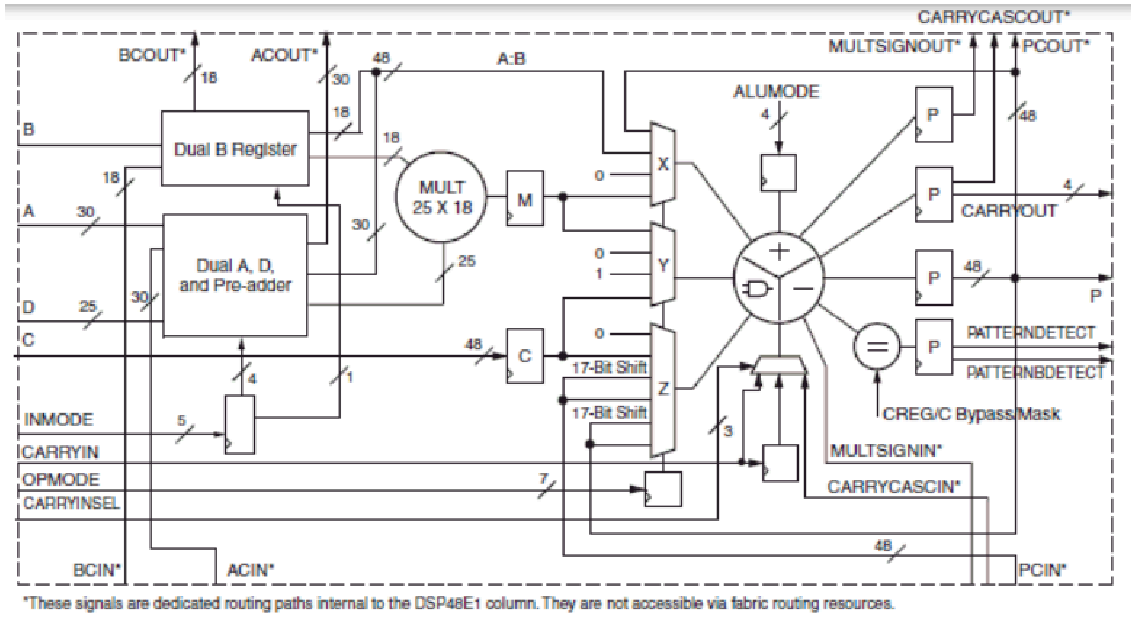
Run Synthesis to see utilization results

Power

Run Implementation to see power results

2. Computing units and DSP

As described in the Tutorial 05, the primary operations in the CNN network are **multiplication and accumulation (MAC)** that can be mapped to the **pre-built DSP on an FPGA board [1].** The DSP block is built on a **25 x 18 bit multiplier** (for details see [1]).

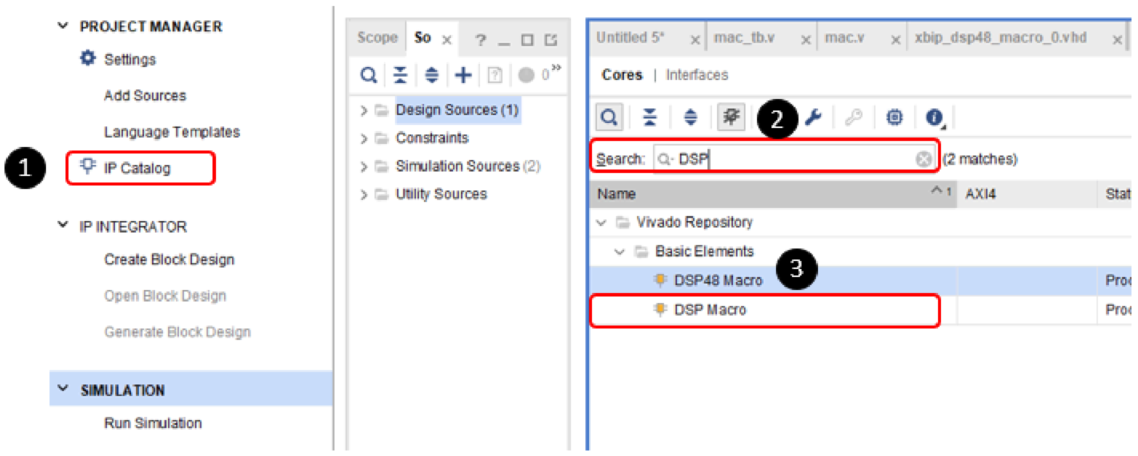


Let’s implement a multiplication based on DSP. To do this, you need to…

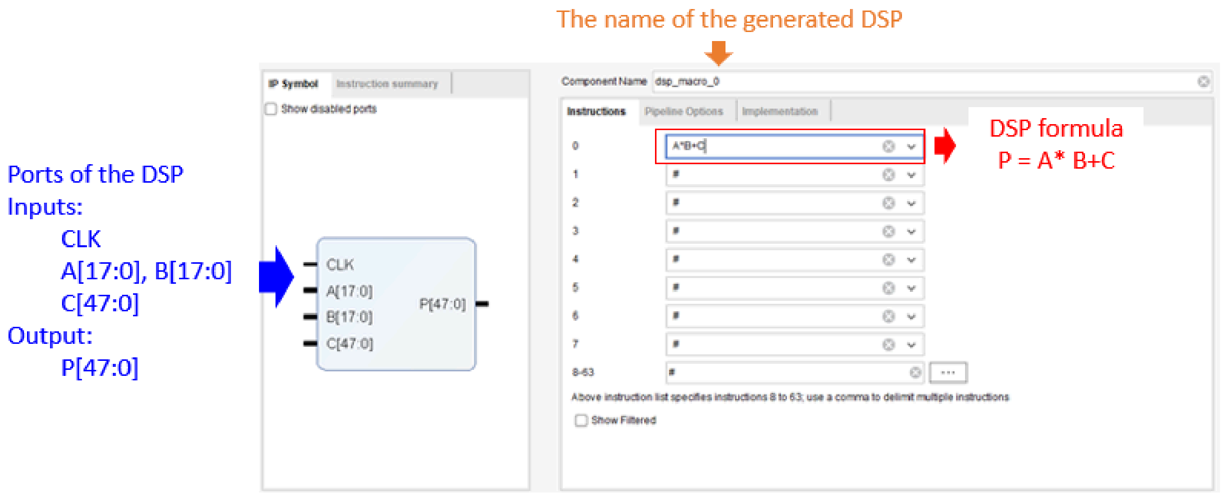
1) **Build a DSP using Xilinx IP generator**
2) **Use(instantiate) the pre-built DSP to implement a multiplier**

1) Build a DSP using Xilinx IP generator

First, you can use “IP Catalog” and find “DSP Macro”.



Select **Instructions**, for example, A * B + C



Component Name: dsp_macro_0

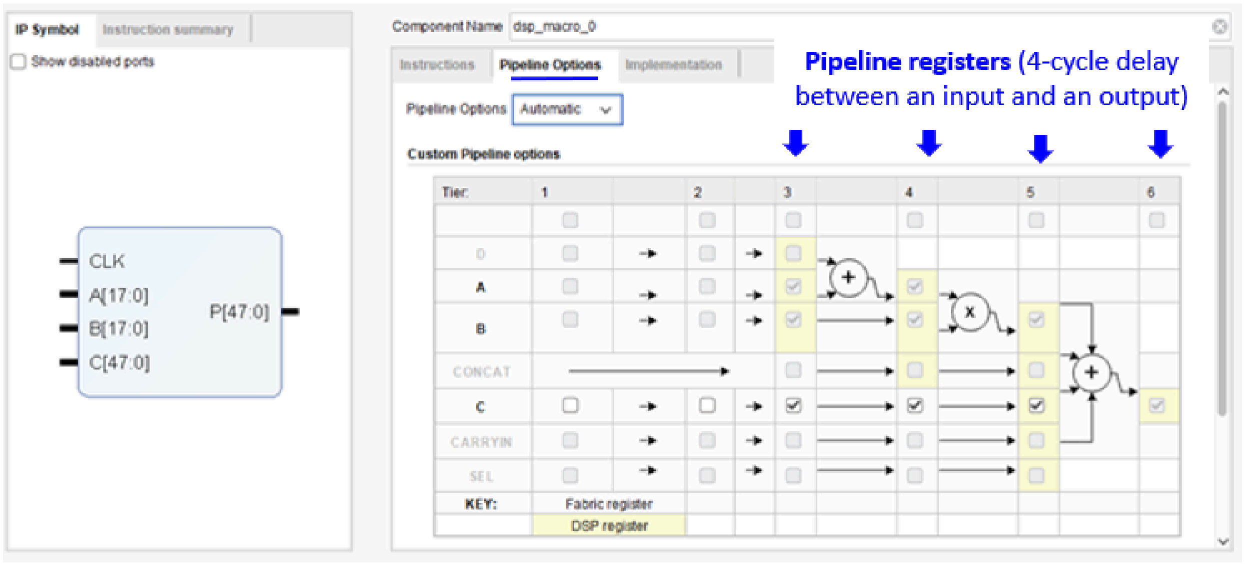
0	A*B+C
1	#
2	#
3	#
4	#
5	#
6	#
7	#
8-63	#

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

Configure **Pipeline Options** for a DSP block.

- Note that you can change **Automatic** into **Expert** to remove some pipeline registers.

- For example, the default DSP takes **four cycles to complete a MAC**. However, if you uncheck all pipeline registers, it takes zero-cycle to execute MAC as a combinational circuit.



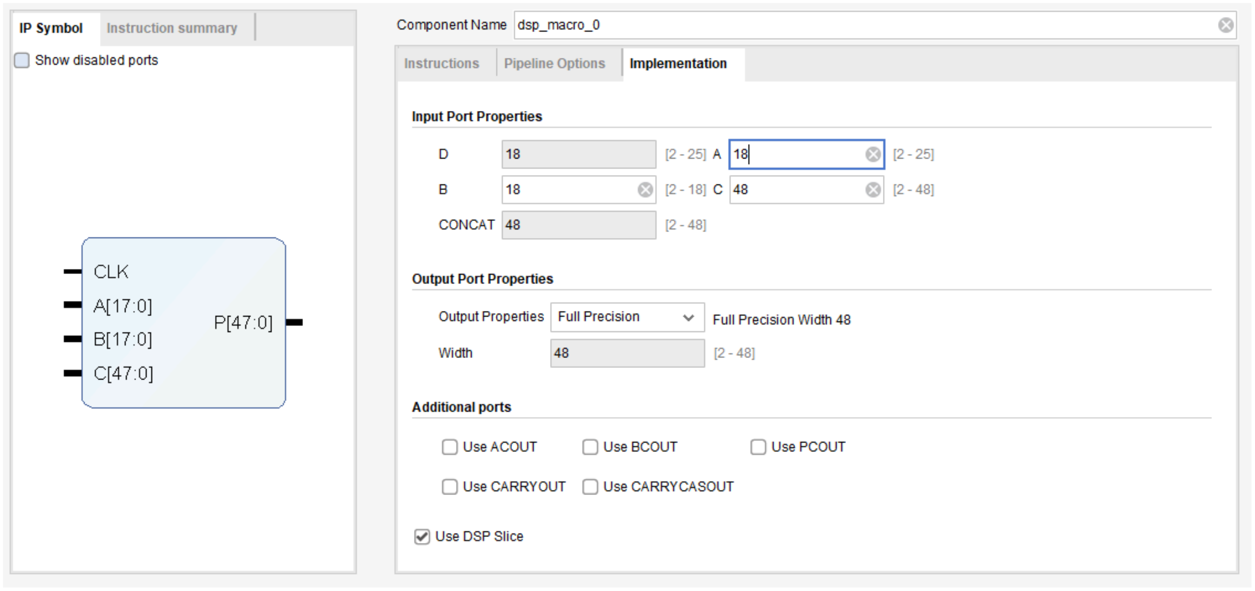
Component Name dsp_macro_0

Pipeline Options: Automatic

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

Step 4) Configure **Implementation** for a DSP block.

You can change the size of inputs and outputs. Although it **does not change the DSP cost,** it saves some resources of the external module that uses a DSP because it works with fewer bits.



Component Name: dsp_macro_0

Input Port Properties

D	18	[2 - 25]	A	18	[2 - 25]
B	18	[2 - 18]	C	48	[2 - 48]
CONCAT	48	[2 - 48]			

Output Port Properties

Output Properties: Full Precision — Full Precision Width 48

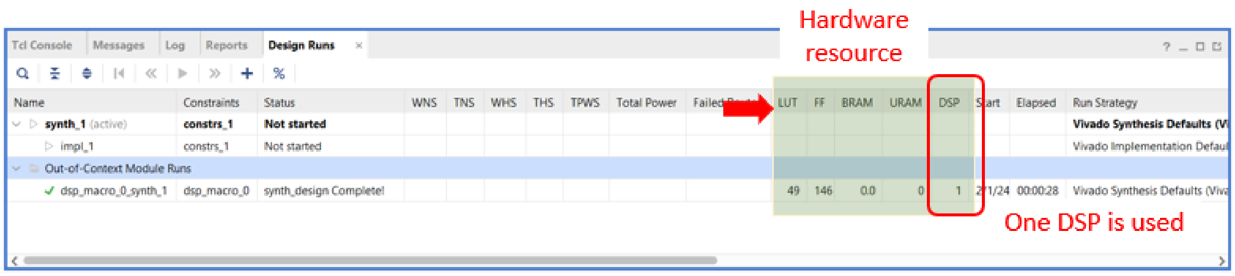
Width: 48 [2 - 48]

Additional ports

- [ ] Use ACOUT
- [ ] Use BCOUT
- [ ] Use PCOUT
- [ ] Use CARRYOUT
- [ ] Use CARRYCASOUT
- [✓] Use DSP Slice

Step 5) Click OK and “Generate” on the popup window

<DSP Generated>

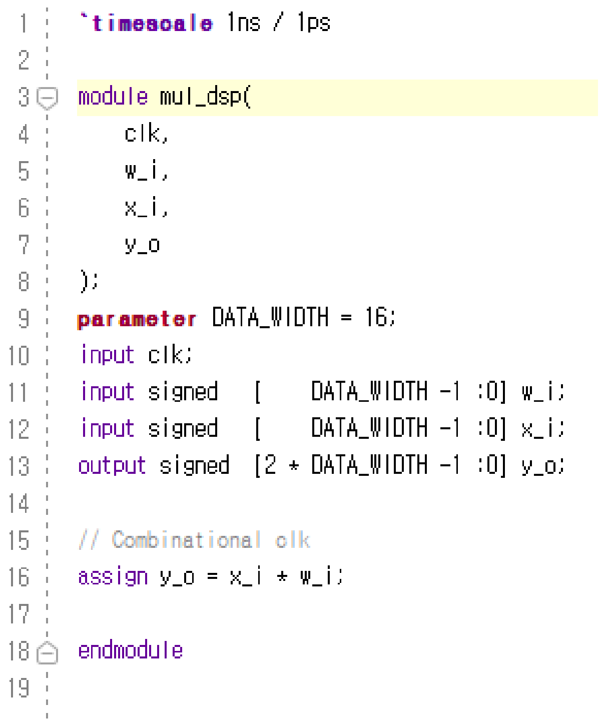


Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1 (active)	constrs_1	Not started															Vivado Synthesis Defaults (Vi
impl_1	constrs_1	Not started															Vivado Implementation Defaul
Out-of-Context Module Runs																	
✓ dsp_macro_0_synth_1	dsp_macro_0	synth_design Complete!								49	146	0.0	0	1	2/1/24	00:00:28	Vivado Synthesis Defaults (Viva

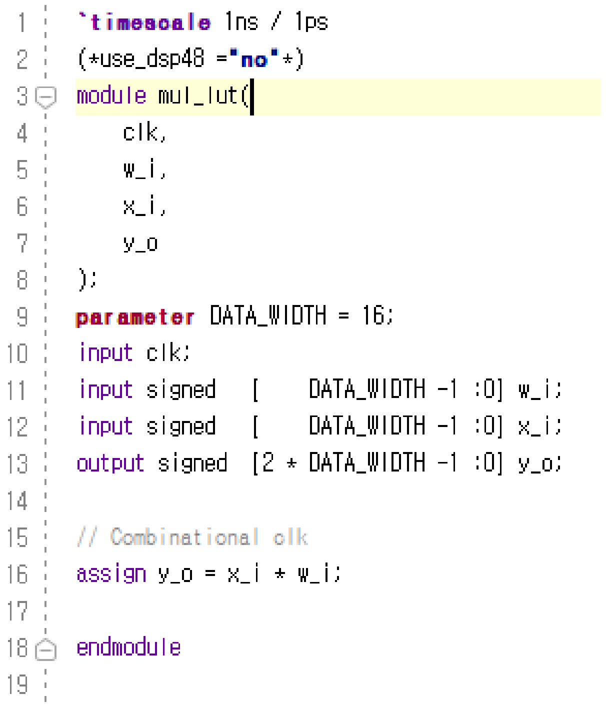
2) Use the pre-built DSP to implement a multiplier

Now that we generated our DSP, we can use it by ‘instantiating’ the module in verilog code.

Before going to the instantiation, note that you can implement a multiplier either (1) with prebuilt DSP or (2) with LUTs.



```
 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  input clk;
11  input signed   [    DATA_WIDTH -1 :0] w_i;
12  input signed   [    DATA_WIDTH -1 :0] x_i;
13  output signed  [2 * DATA_WIDTH -1 :0] y_o;
14
15  // Combinational clk
16  assign y_o = x_i * w_i;
17
18  endmodule
19
```



```
 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  input clk;
11  input signed   [    DATA_WIDTH -1 :0] w_i;
12  input signed   [    DATA_WIDTH -1 :0] x_i;
13  output signed  [2 * DATA_WIDTH -1 :0] y_o;
14
15  // Combinational clk
16  assign y_o = x_i * w_i;
17
18  endmodule
19
```

(1) Multiplier with DSP (2) multiplier with LUT

The following codes are two multiplier designs, one using DSP and another using LUT. To force Vivado to implement a multiplier **without using DSP, you can use the flag (*use_dsp48=”no”*) as shown in Line 2 of Version (b).**

Without using the flag, if DATA_WIDTH is less than or equal to 10, a multiplier is **automatically mapped to LUTs**. Meanwhile, if DATA_WIDTH is greater or equal to 11, **a multiplier is mapped to DSPs and LUTs**.

Once you choose to use multiplier based on DSP, you can manually **instantiate** the DSP modules to use. When you want to utilize DSP for your eight-bit multiplier, you can **call a DSP instance** as follows:

```
dsp_macro_0 u_dsp(
.CLK(clk),
.A(dsp_A),
.B(dsp_B),
.C(48'b0),
.P(dsp_P)
);
```

The following table shows the resources for different DATA_WIDTHs. If you implement 240 multipliers without using DSPs, it will cost 14,640 LUTs accounting for 23.09% of the total LUTs in a Nexys A7-100T board. **In other words, if we utilize the existing DSPs for 240 multipliers, we will save 23.09% of the LUTs for other modules.**

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

Let’s make **a multiplier.**

Step 1: Generate a DSP from IP catalog

Step 2: Make “mul.v”

- Inputs: clk, w[7:0], x[7:0]
- Output: y[15:0]

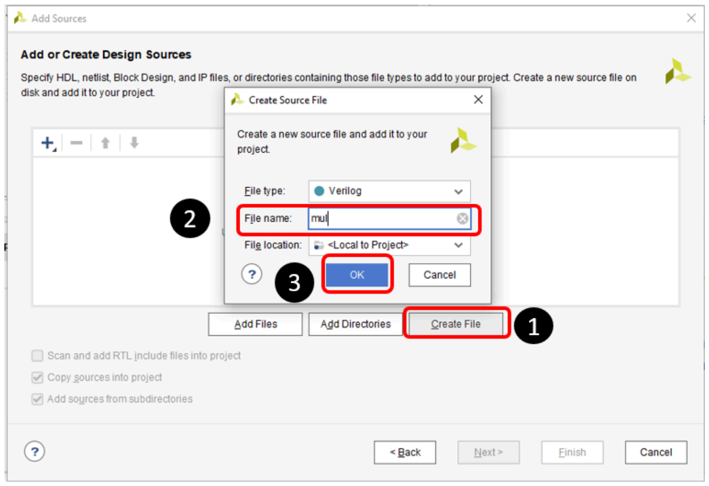
Step 3: Make inputs and outputs for the DSP

Step 4: Call a DSP instance

(This time, you don’t have to use DSP generator because you can just use the one you made before. So, all you need to do here is to create a new code to instantiate the DSPs.)

To start with, “Add source”>> Click “Create File” >> Make a file name “mul” in “Verilog” >> click “OK”

- Note: You can just download add the source code(mul.v)


Add Sources

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 Source File

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

File type: Verilog

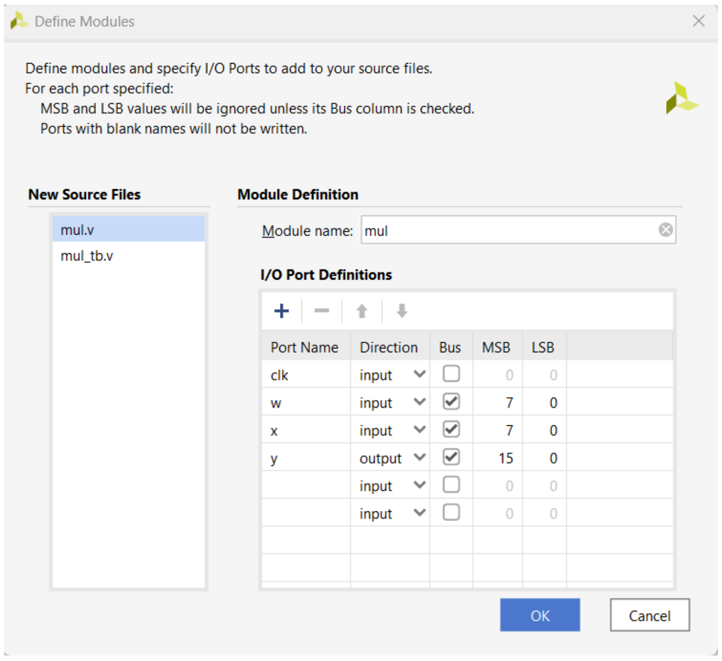
File name: mul

File location: <Local to Project>

Scan and add RTL include files into project

Copy sources into project

Add sources from subdirectories


Define Modules

Define modules and specify I/O Ports to add to your source files.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

New Source Files

mul.v
mul_tb.v

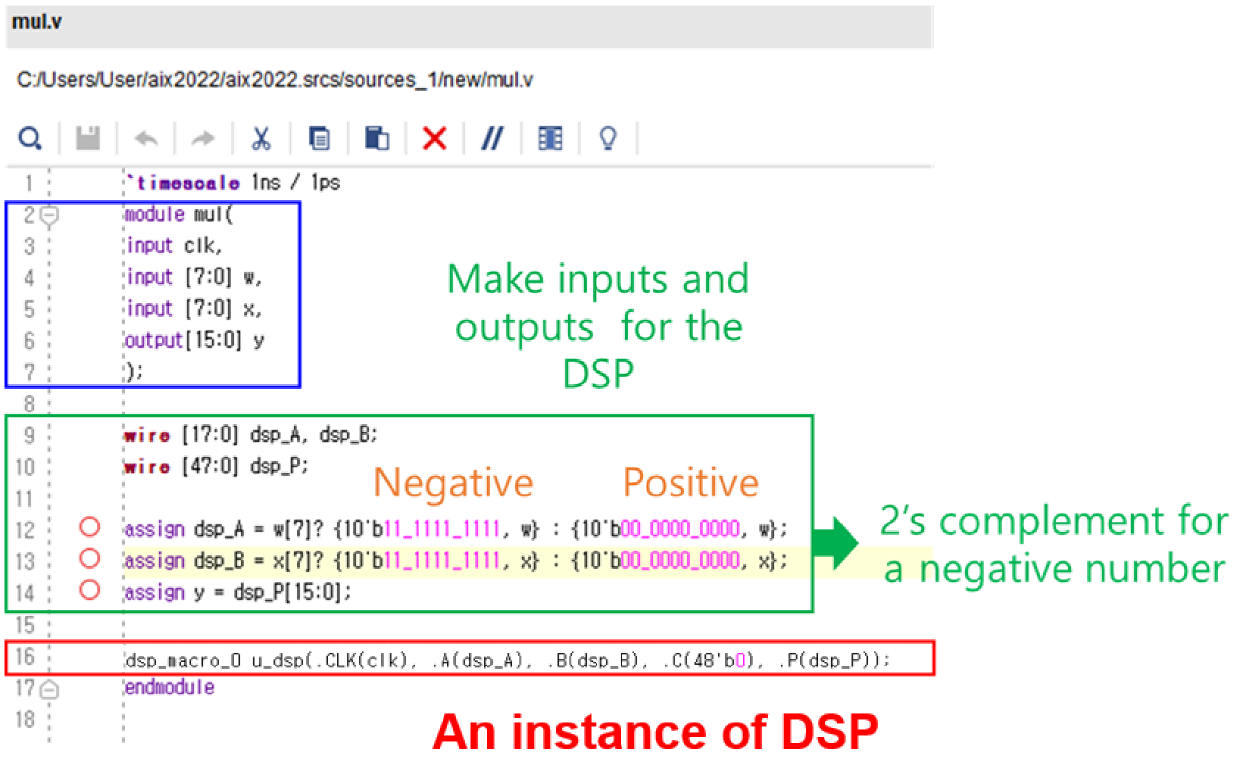
Module Definition

Module name: mul

I/O Port Definitions

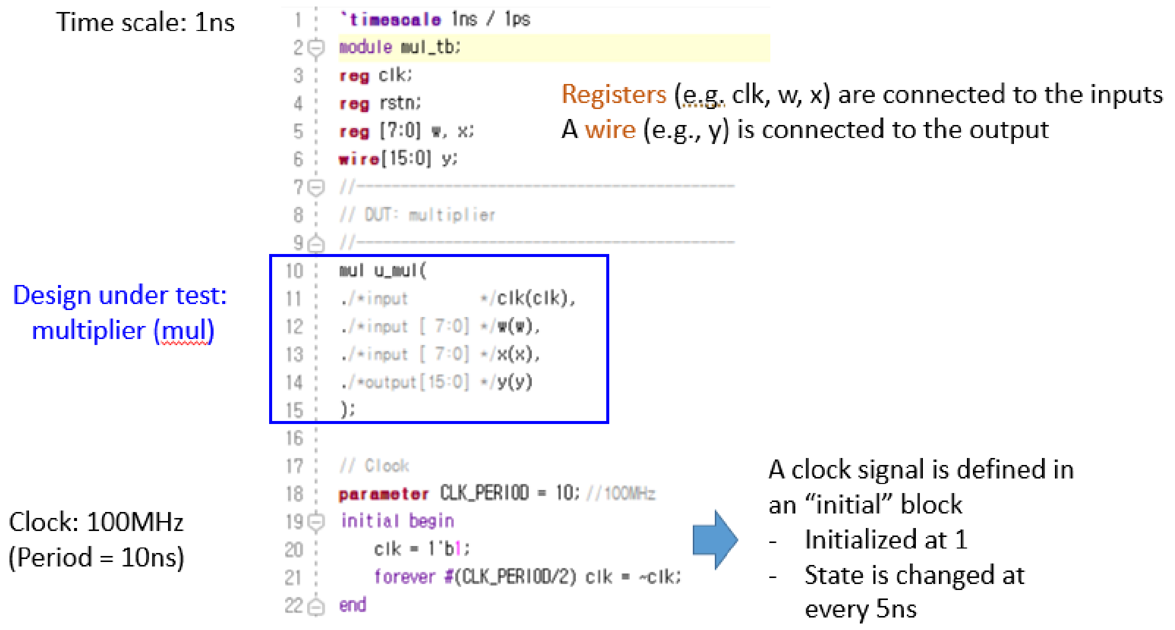
Port Name	Direction	Bus	MSB	LSB
clk	input	☐	0	0
w	input	✓	7	0
x	input	✓	7	0
y	output	✓	15	0
	input	☐	0	0
	input	☐	0	0

You can make an **8-bit-by-8-bit multiplier** as follows.

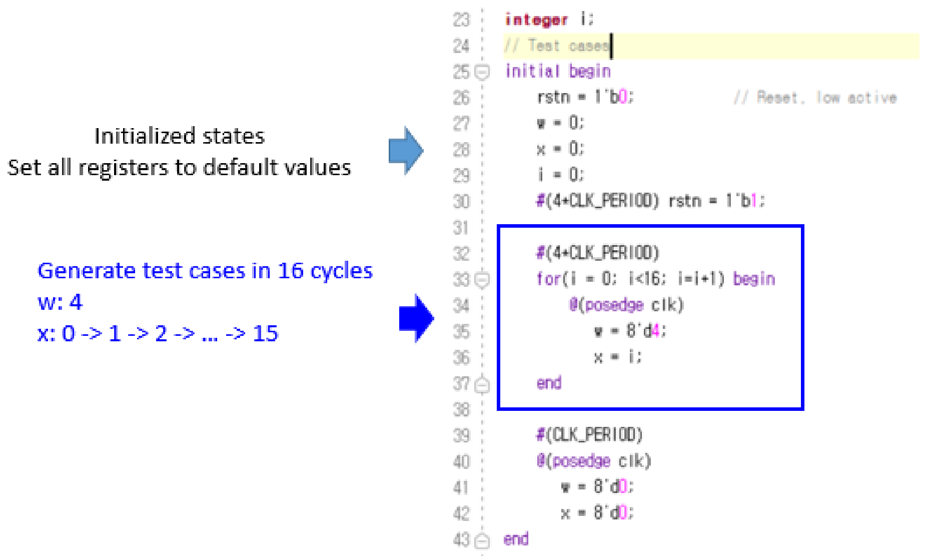

mul.v
C:/Users/User/aix2022/aix2022.srcs/sources_1/new/mul.v
Make inputs and outputs for the DSP
Negative
Positive
2's complement for a negative number
An instance of DSP

```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];

dsp_macro_0 u_dsp(.CLK(clk), .A(dsp_A), .B(dsp_B), .C(48'b0), .P(dsp_P));
endmodule
```

Once you've created the design(mul.v), you need to make corresponding testbench(mul_tb.v) to simulate it. In general, clock generator, DUT(Design Under Test), and input stimulus generator is included in testbench.


Time scale: 1ns
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

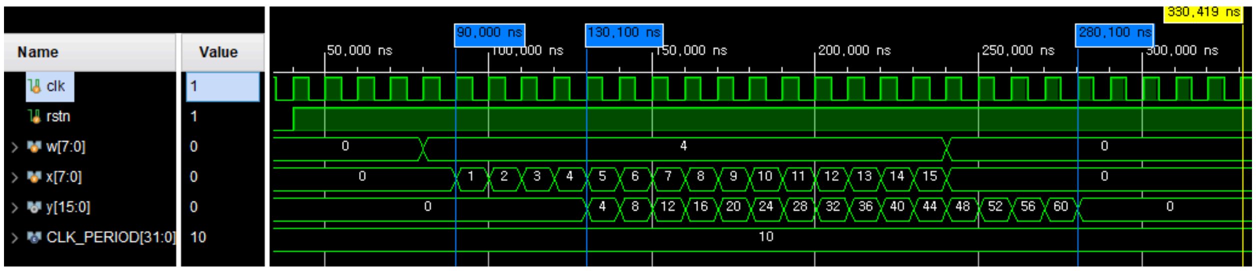
```
`timescale 1ns / 1ps
module mul_tb;
reg clk;
reg rstn;
reg [7:0] w, x;
wire[15:0] y;
//-----------------------------------------------
// DUT: multiplier
//-----------------------------------------------
mul u_mul(
./*input         */clk(clk),
./*input  [ 7:0] */w(w),
./*input  [ 7:0] */x(x),
./*output[15:0]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

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


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
```

Now, you can run the simulation.

In the Tab "Simulation", click on "run simulation" >> "Run behavioral simulation".



Next, Let's make **an array of multiplier with MAC**(Multiplication and ACcumulation). You can download the source codes. They are for example MAC module and its testbench that calculates **(1)16 multiplications(mac.v)** and **(2)accumulations(adder_tree.v) every cycle**.

Assume that we compute the inner product of two vectors *W* and *x,* each with 16 elements:

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

To do so, we'll instantiate sixteen multipliers that can execute sixteen multiplication, with an adder tree to add all the products. Inputs are 8 bit 2's complement numbers and the output is a 20 bit 2's complement number.

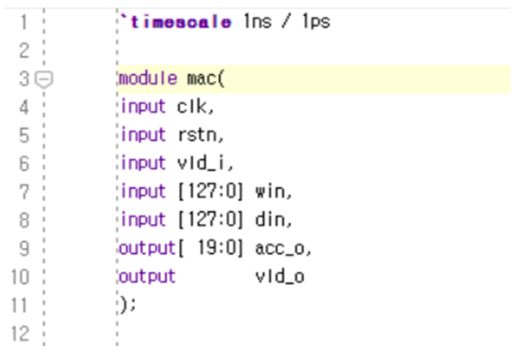
[mac.v]

- Input

clk, rstn	Clock and reset. Active low
vld_i	input valid signal
win	128 bits for weights → 16 eight-bit weights
din	128 bits for input pixels → 16 eight-bit pixels

- Output

vld_o	Output valid signal
acc_o	Accumulated output that has 20 bits.



```
`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
);
```

- 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
```

[adder tree.v]

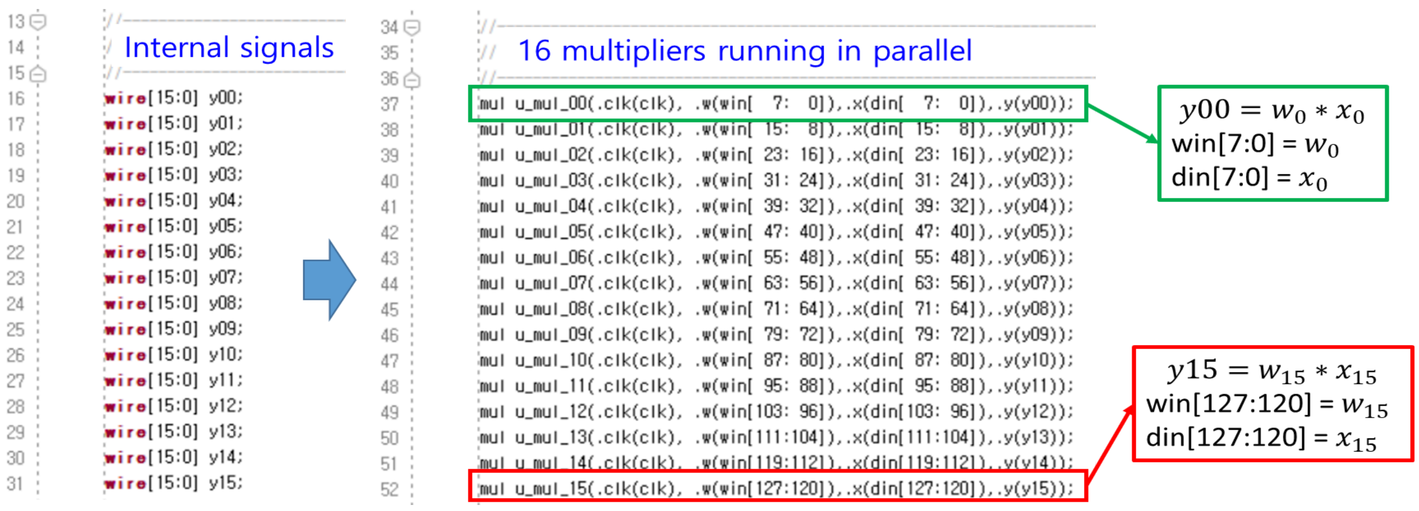
The outputs from the sixteen multipliers above are forwarded to an adder tree for accumulation. Use ‘adder_tree.v’ to compute a sum of 16 products.

To obtain an output, the adder tree uses **fifteen adders to perform hierarchical addition.**

At the first level, sixteen inputs are paired and then forwarded to **eight** adders that generate eight intermediate results. At the second level, the eight results are paired and moved to **four** adders that produce four results. Next, the four outputs are paired and inserted into **two** adders that output two results.

Finally, the results are forwarded to the **last** adder to generate the accumulated output.

 8+4+2+1=15



```
//--------------------------------
// 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));
```

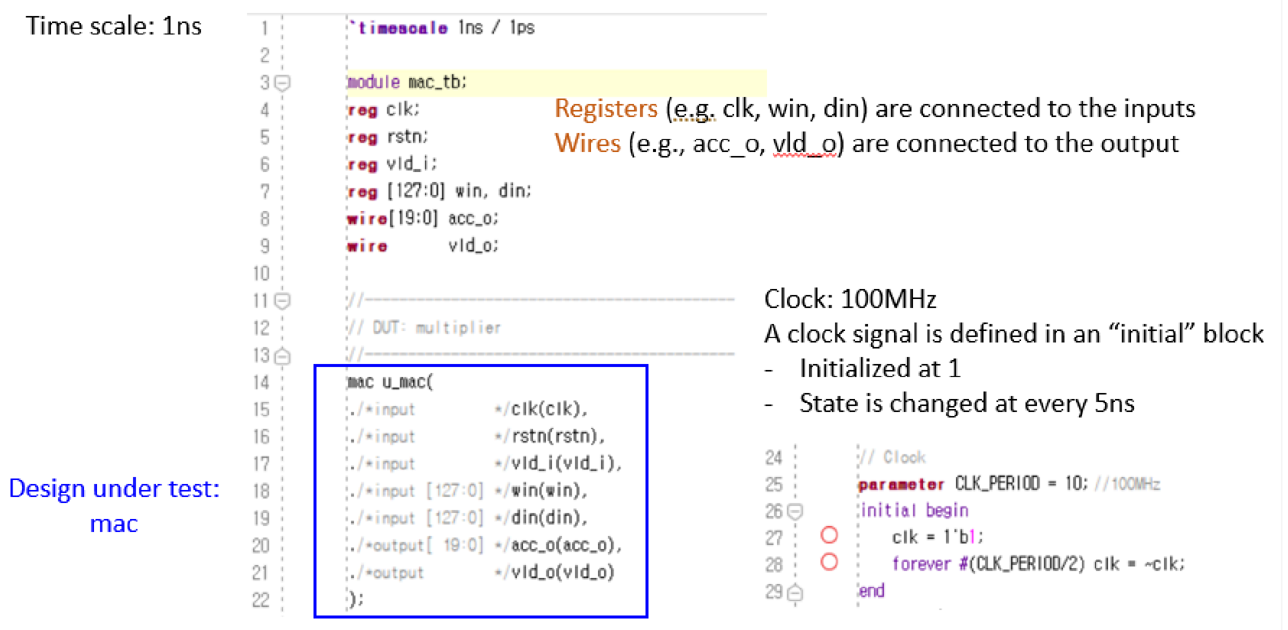
To generate the output valid signal, we use a **delay line**. For example, it takes **four cycles to compute a multiplication** using the default DSP, while it takes **four cycles (=clog2(16)) to accumulate** the sixteen multiplication results to generate an output.

Therefore**,** it takes **eight cycles of latency** to calculate $y = \sum_{i=0}^{15} w_i * x_i$.

- vld_i: when high, this indicates that '**win' and 'din' are valid inputs**, and must be calculated
- vld_d(delay): **an internal variable** to count the delay for vld_o to be high
- vld_o(delay): when high, indicates that **output acc_o is a valid** calculation result

- **Note that the eight cycles are the latency between input and output. If we insert <u>multiple inputs consecutively, it will generate the consecutive outputs too, accordingly.</u>**

Now that you created the module design, you can simulate the module by creating testbench code(mac_tb.v). You can download and add it to your project.



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

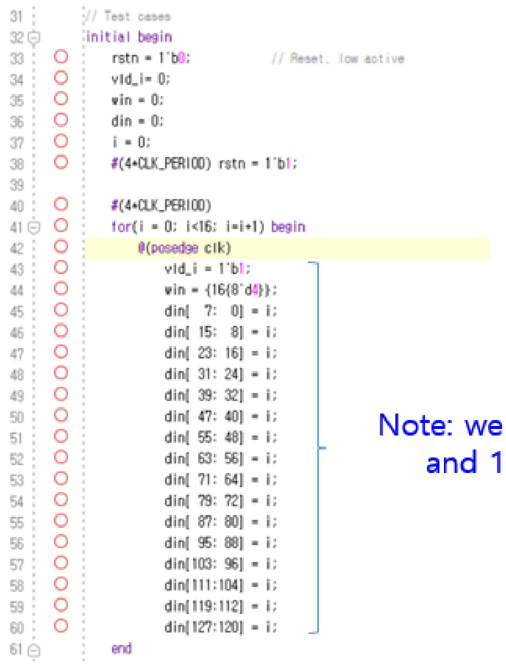
Design under test: mac

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
```

Initialized states
Set all registers to default values

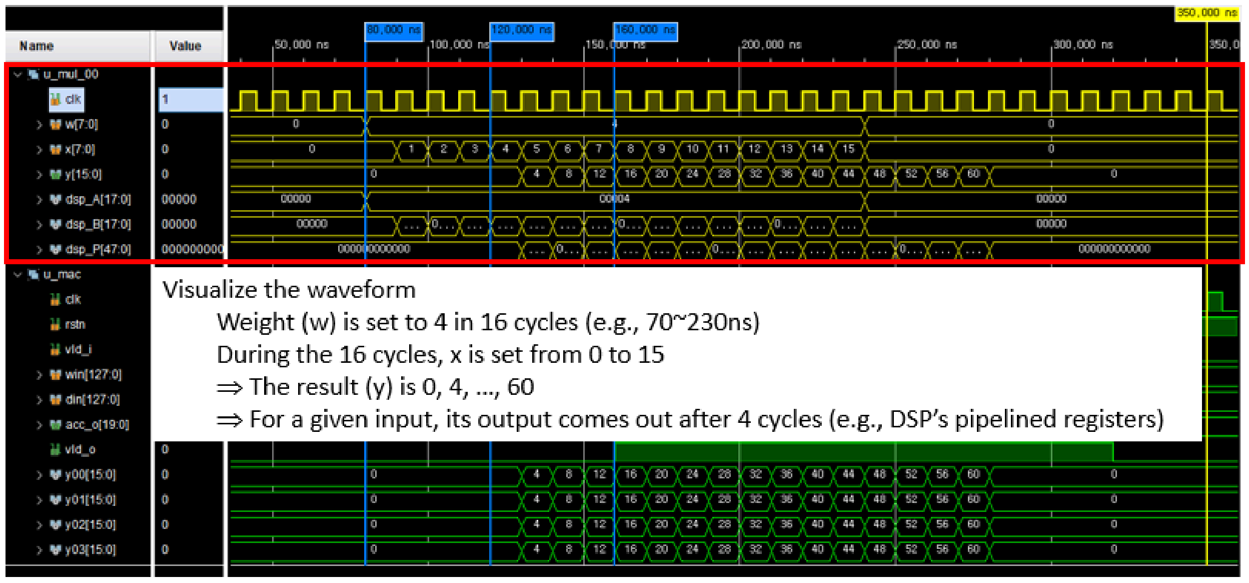
Generate test cases in 16 cycles
w: 4
x: 0 -> 1 -> 2 -> ... -> 15



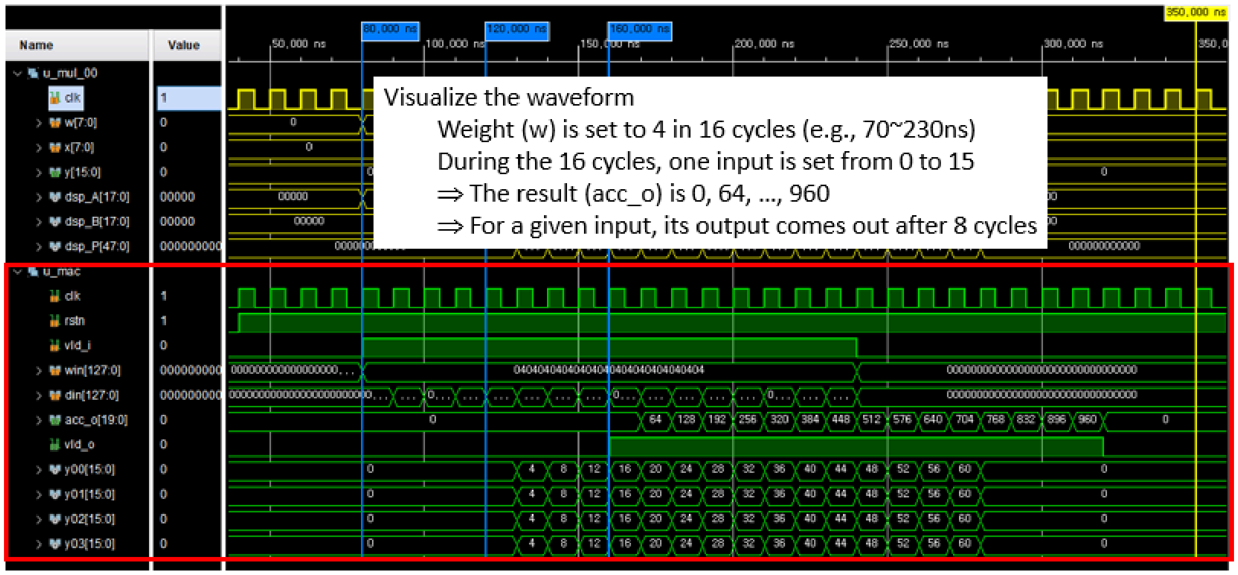
```
// Test cases
initial begin
    rstn = 1'b0;            // Reset, low active
    vld_i= 0;
    win = 0;
    din = 0;
    i = 0;
    #(4*CLK_PERIOD) rstn = 1'b1;

    #(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
```

Note: we have 16 weights
and 16 inputs now.

When you run the simulation, you can see the waveform like this. You can compare multiplier and mac unit as follows:



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)



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

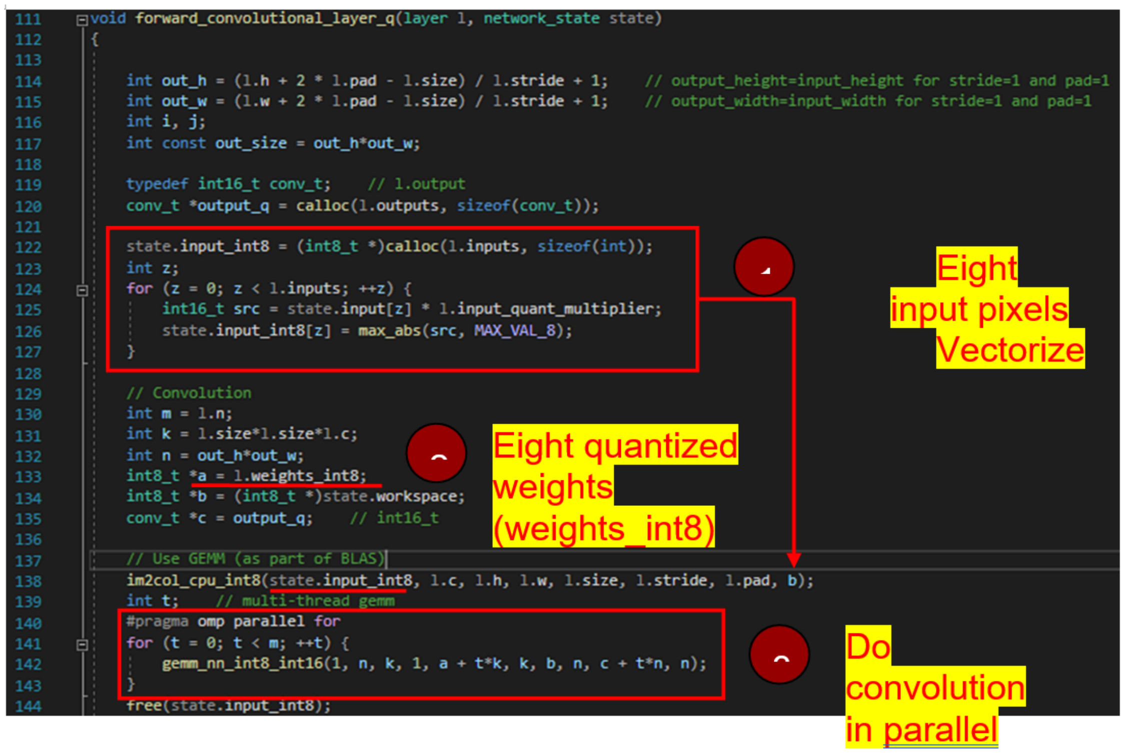
3. Convolutional Kernel(Parallel MAC modules)

We observed that 16 multipliers and 15 adders could be used to calculate an inner product between two vectors with sixteen eight-bit elements. Recall that the AIX2024 SDK does an inference with a quantized model as follows:

- Do activation quantization and vectorize eight-bit input pixels (im2col_cpu_int8) and save them into ***b***.
- Retrieve the quantized int8 weights(weights_int8) and save them into ***a***.
- Do convolution, for example, inner product between ***a(weight)*** and ***b(input activation)***, **in parallel**.

  ** Note that the loop "for (t =0; t <m; ++t)" **iterates among m sets of the filters**. In other words, **each iteration produces one accumulated map.***

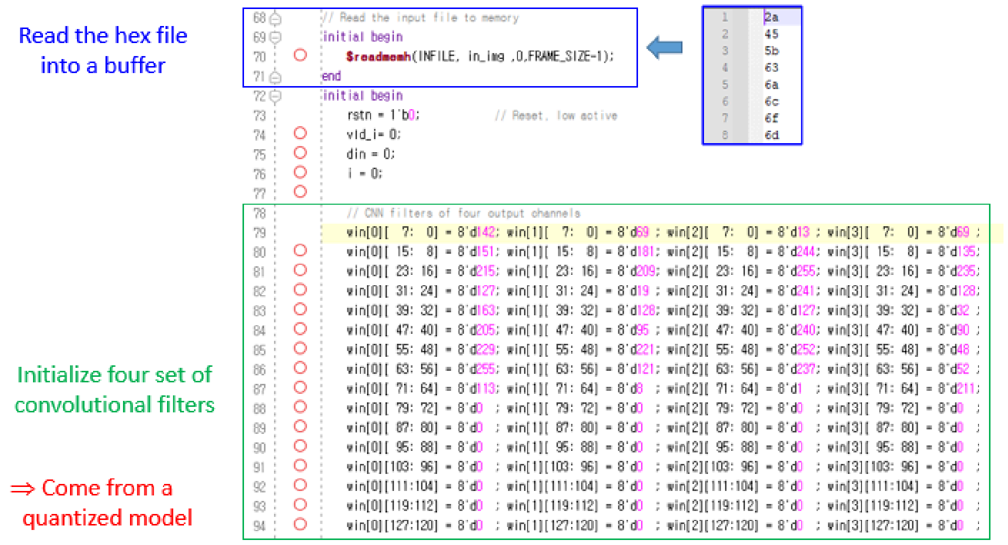
- Meanwhile, 'gemm_nn_int8_int16' takes two vectors from ***a*** and ***b***, and **calculates inner products as [mac.v] and the computed results are stored in *c*.**



```c
void forward_convolutional_layer_q(layer l, network_state state)
{

    int out_h = (l.h + 2 * l.pad - l.size) / l.stride + 1;    // output_height=input_height for stride=1 and pad=1
    int out_w = (l.w + 2 * l.pad - l.size) / l.stride + 1;    // output_width=input_width for stride=1 and pad=1
    int i, j;
    int const out_size = out_h*out_w;

    typedef int16_t conv_t;    // l.output
    conv_t *output_q = calloc(l.outputs, sizeof(conv_t));

    state.input_int8 = (int8_t *)calloc(l.inputs, sizeof(int));
    int z;
    for (z = 0; z < l.inputs; ++z) {
        int16_t src = state.input[z] * l.input_quant_multiplier;
        state.input_int8[z] = max_abs(src, MAX_VAL_8);
    }

    // Convolution
    int m = l.n;
    int k = l.size*l.size*l.c;
    int n = out_h*out_w;
    int8_t *a = l.weights_int8;
    int8_t *b = (int8_t *)state.workspace;
    conv_t *c = output_q;    // int16_t

    // Use GEMM (as part of BLAS)
    im2col_cpu_int8(state.input_int8, l.c, l.h, l.w, l.size, l.stride, l.pad, b);
    int t;    // multi-thread gemm
    #pragma omp parallel for
    for (t = 0; t < m; ++t) {
        gemm_nn_int8_int16(1, n, k, 1, a + t*k, k, b, n, c + t*n, n);
    }
    free(state.input_int8);
```

While **mac.v** does 16 multipliers in parallel, we can **stack multiple MAC modules** to replicate the loop “for (t =0; t <m; ++t)” in the AIX2024 SDK.

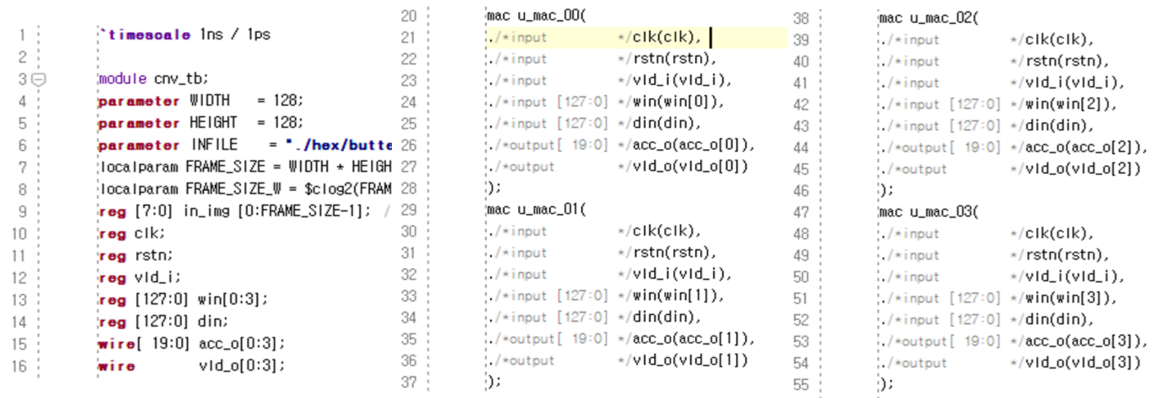
For example, we **stack four MAC kernels (cnv_tb.v),** with each module being responsible for one set of convolutional filters. In particular, there are four streams of weights ***win[0:3]*** that **apply for the same input pixel vector *din*** to produce four different accumulated outputs.


Read the hex file into a buffer
Initialize four set of convolutional filters
⇒ Come from a quantized model

```
// 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
```

This time, use the testbench “**conv_tb**” to see how the convolution is parallelly processed with multiple macs and adder trees that we created before.

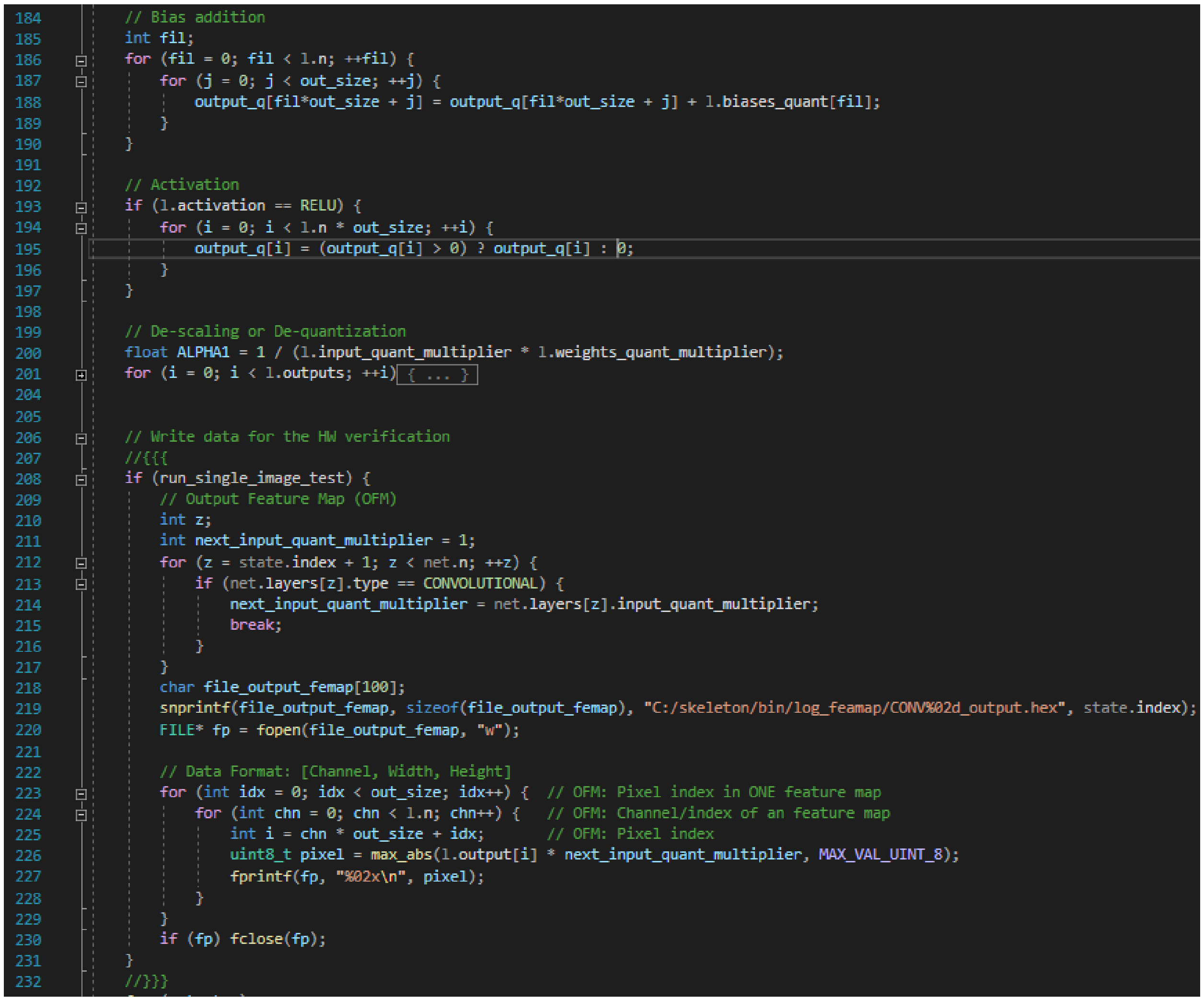


```
`timescale 1ns / 1ps

module cnv_tb;
parameter WIDTH    = 128;
parameter HEIGHT   = 128;
parameter INFILE    = "./hex/butte
localparam FRAME_SIZE = WIDTH * HEIGH
localparam FRAME_SIZE_W = $clog2(FRAM
reg [7:0] in_img [0:FRAME_SIZE-1]; /
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];
```

```
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])
);
```

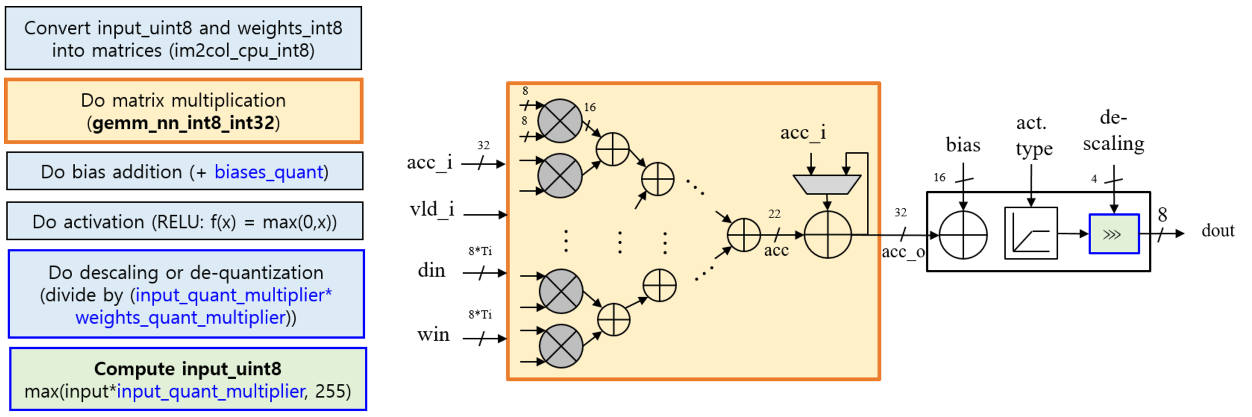
```
mac u_mac_02(
./*input          */clk(clk),
./*input          */rstn(rstn),
./*input          */vld_i(vld_i),
./*input [127:0] */win(win[2]),
./*input [127:0] */din(din),
./*output[ 19:0] */acc_o(acc_o[2]),
./*output         */vld_o(vld_o[2])
);
mac u_mac_03(
./*input          */clk(clk),
./*input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

- In the AIX2023 SDK, after convolution, it adds a bias, performs activation, and then descaling. The hardware module must replicate similar operations.



```c
// Bias addition
int fil;
for (fil = 0; fil < l.n; ++fil) {
    for (j = 0; j < out_size; ++j) {
        output_q[fil*out_size + j] = output_q[fil*out_size + j] + l.biases_quant[fil];
    }
}

// Activation
if (l.activation == RELU) {
    for (i = 0; i < l.n * out_size; ++i) {
        output_q[i] = (output_q[i] > 0) ? output_q[i] : 0;
    }
}

// De-scaling or De-quantization
float ALPHA1 = 1 / (l.input_quant_multiplier * l.weights_quant_multiplier);
for (i = 0; i < l.outputs; ++i) { ... }


// Write data for the HW verification
//{{{
if (run_single_image_test) {
    // Output Feature Map (OFM)
    int z;
    int next_input_quant_multiplier = 1;
    for (z = state.index + 1; z < net.n; ++z) {
        if (net.layers[z].type == CONVOLUTIONAL) {
            next_input_quant_multiplier = net.layers[z].input_quant_multiplier;
            break;
        }
    }
    char file_output_femap[100];
    snprintf(file_output_femap, sizeof(file_output_femap), "C:/skeleton/bin/log_feamap/CONV%02d_output.hex", state.index);
    FILE* fp = fopen(file_output_femap, "w");

    // Data Format: [Channel, Width, Height]
    for (int idx = 0; idx < out_size; idx++) {  // OFM: Pixel index in ONE feature map
        for (int chn = 0; chn < l.n; chn++) {   // OFM: Channel/index of an feature map
            int i = chn * out_size + idx;       // OFM: Pixel index
            uint8_t pixel = max_abs(l.output[i] * next_input_quant_multiplier, MAX_VAL_UINT_8);
            fprintf(fp, "%02x\n", pixel);
        }
    }
    if (fp) fclose(fp);
}
//}}}
```

Here is one example of the final design.


Convert input_uint8 and weights_int8 into matrices (im2col_cpu_int8)
Do matrix multiplication (gemm_nn_int8_int32)
Do bias addition (+ biases_quant)
Do activation (RELU: f(x) = max(0,x))
Do descaling or de-quantization (divide by (input_quant_multiplier* weights_quant_multiplier))
Compute input_uint8 max(input*input_quant_multiplier, 255)
acc_i
32
vld_i
din
8*Ti
win
8*Ti
8
8
16
acc_i
22
acc
32
acc_o
bias
16
act. type
de-scaling
4
>>>
8
dout

References

[1]. DSP48E1

https://docs.xilinx.com/r/2021.1-English/ug1483-model-composer-sys-gen-user-guide/DSP48E1

[2]. Deep Learning with INT8 Optimization on Xilinx Devices

https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8