

Kaiyi Yuan  
1003084716

**Assignment 2**

ECE1756  
Nov 1<sup>st</sup> 2023

**ECE1756 Assignment 2**

**Image Convolution on FPGAs, CPUs, and GPUs**

**Kaiyi Yuan**  
**1003084716**

**November 2<sup>nd</sup> 2023**

## Part I: FPGA Implementation

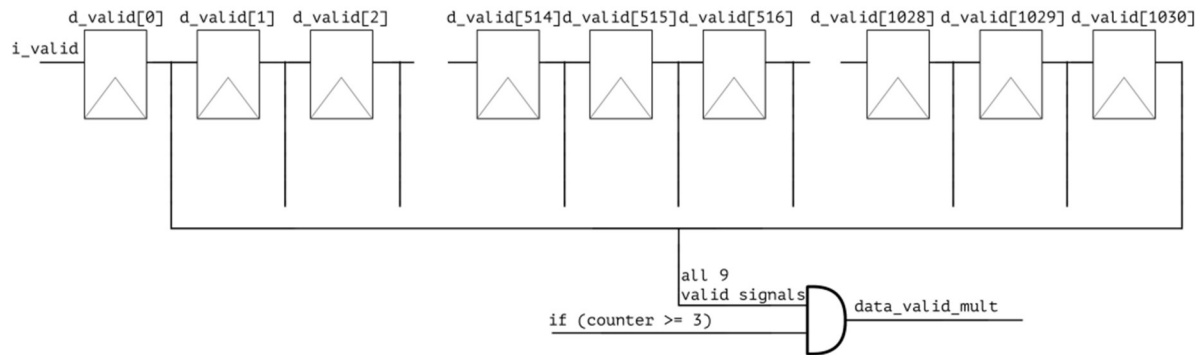


Figure 1: FPGA structure

Due to the nature of the given filter size of 3, I come up with this specific datapath for FPGA application. It has 9 multiplier performing the multiplications in a parallel fashion, and their outputs are summed in adder, data is pipelined to achieve max efficiency between each stage of multiplier and adders.

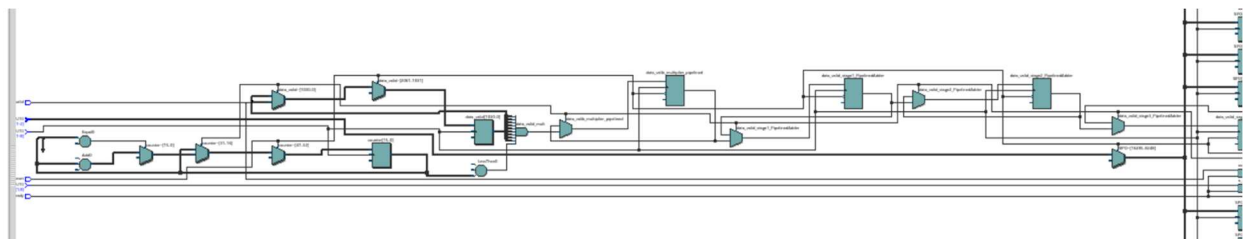


Figure 2: RTL view of the FPGA structure

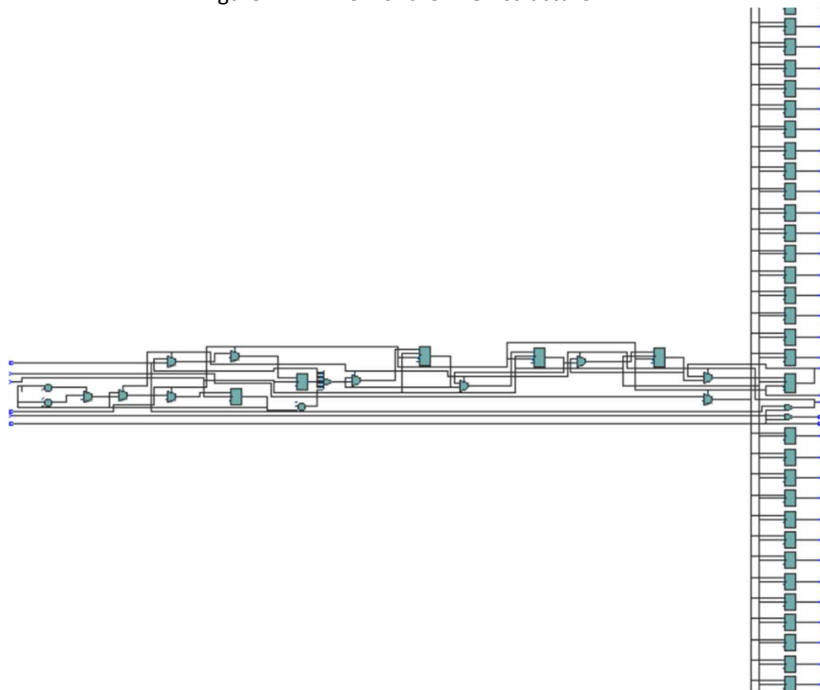


Figure 3: Full RTL view of the FPGA Structure

Above are the RTL view of generated structures, SIPO shift registers are employed in the datapath, size 3 filter corresponding to the 9 pixels involved in the convolution are gathered in an AND gate that keeps track of the input column position with a counter.

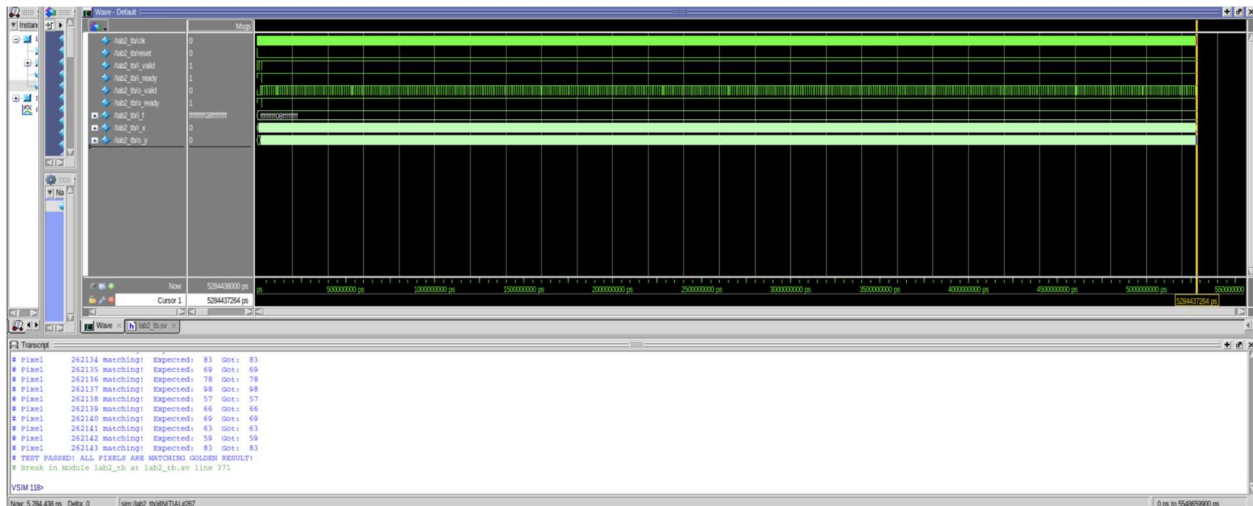


Figure 4: Test 7a waveform result

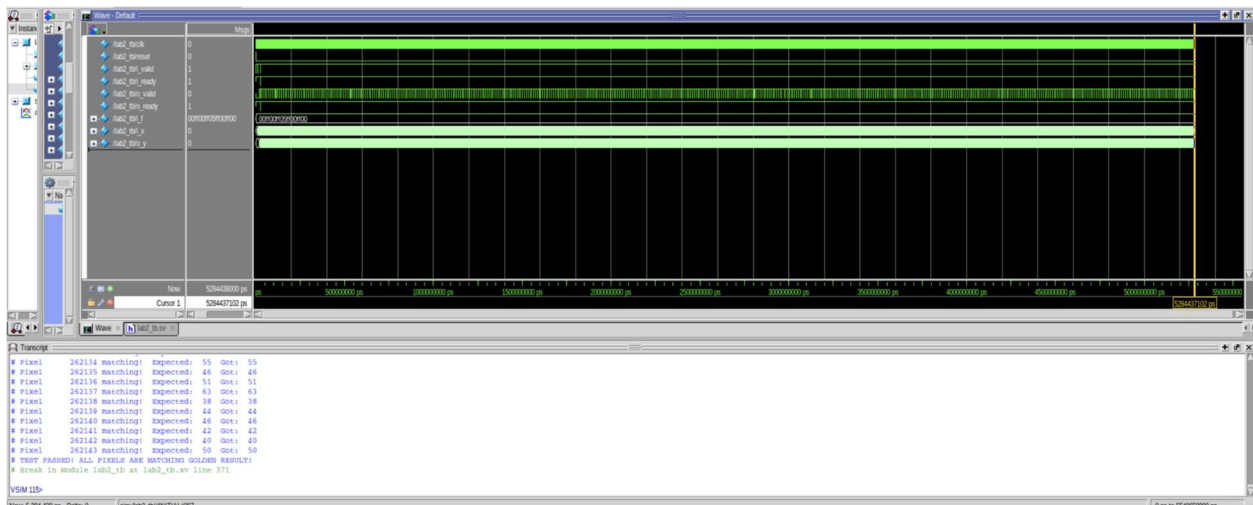


Figure 5: Test 7b waveform result

Total Dynamic Power for one module @ 50 MHz is  $0.36 + 2.21 + 3.84 + 0.13 + 6.12 = 12.66$  mW

So the power @ 428.82 MHz is  $12.66 \times (428.82/50) = 108.577224$  mw

### Throughput of one module (GOPS)

$$(512 * 512) * (9 + 8) = 4456448 \text{ operations}$$

$$(1/428.82 \times 10^6) * 264216 \approx 6.161 \times 10^{-4} \text{ seconds}$$

$$4456448 / 6.161 \times 10^{-4} \approx 7.2333 \times 10^9 = 7.2333 \text{ GOP}$$

### Throughput of full device (GOPS)

ALM count  $427200/2497 \approx 171$

DSP count  $1518/9 \approx 168$

DSP count is less so it's the limit factor.

$7.2333 * 168 = 1215.144$  GOP

### Total Power for a full device

Total Dynamic Power at 50MHz is  $(0.36 + 2.21 + 3.84) * 168 + 0.13 + 6.12 = 1083.13$  mW

Total Dynamic power at 428.82MHz =  $1083.13 * (428.82/50) = 9289.356$  mW

Total Power at 428.82MHz =  $9289.356 + (0.43 + 2.37 + 11.51) * 168 + 0.14 + 0.05 + 11.21 = 11731.836$  mW  $\approx 11.731$  W

	Result
ALM Utilization	2499
DSP Utilization	9
BRAM (M20K) Utilization	0
Maximum Operating Frequency (MHz)	428.82
Cycles for Test 7a(Hinton)	264216
Dynamic Power for one module @ max frequency	108.577224 mw
Throughput of one module (GOPS)	7.2333
Throughput of a full device (GOPS)	1215.144
Total Power for a full device (W)	11731

Table 1: FPGA implementation results

## Part II: Efficiency Comparison

CPU convolution is straightforward. Both single threaded and multithreaded loop for filter\_id with basic and hand vectorized versions, and it outputs rows and columns respectively; after that, dot product computation commence for 3 by 3 convolution window. Basic implementation travels through all 9 elements (3x3 filter) and accumulates in a single stage of load, dot product and store; then the results of 3 rows are accumulated together. However, multi thread method is performed with OpenMP directives where it applied to the most outside layer of filter\_id dimension.

GPU convolution takes less stages compare with CPU; first each thread loads up a couple pixels from the current tile into shared memory; when all the threads of the current block are done, each thread calculates the dot product for the output pixel all at once with filter\_id assigned to it; for which the filters are being stored in the GPU constant memory.

No. of Filters	Runtime(ms)			
	1	4	16	64
GPU	0.0149	0.0422	0.1325	0.5112
CPU (basic - no opt - 1 thread)	6.0956	24.365	97.335	391.31
CPU (vectorized - no opt - 1 thread)	3.2056	13.626	51.589	211.36
CPU (basic - O2 - 1 thread)	1.2316	4.8963	20.011	81.362
CPU (vectorized - O2 - 1 thread)	0.8554	3.6262	13.125	53.930
CPU (basic - O3 - 1 thread)	0.5326	2.1120	8.3326	35.989
CPU (vectorized - O3 - 1 thread)	0.9001	3.4362	14.011	56.632
CPU (basic - O3 - 4 threads)	0.5669	0.5998	1.2416	4.8623
CPU (vectorized - O3 - 4 threads)	0.9023	1.1221	1.8534	7.6385

Table 2: Runtime of different versions of the CPU and GPU implementations of 2D convolution with different number of filters

From the table above, it's clear that for any given test case, runtime increases as the number of filters increases with a linear relation. For any optimization level, the vectorized method of convolution computation always take less time to complete the task compared with the basic version. Except when the -O3 flag is enabled. As we keep going on with version O2, there's a significant reduce of the gap between the result of vectorized and basic method; however when the -O3 version is implemented, basic performs better than vectorized method; it's interesting to notice that for filter size of 1 there's not difference between 1 thread and 4 thread; but as the filter size increases, 4 threads achieves higher efficiency with lower runtime.

	Throughput (GOPS)	Power (W)	Energy Efficiency (GOPS/W)	Area Efficiency (GOPS/mm <sup>2</sup> )
FPGA (20 nm)	4689.2	10.188	460.27	11.723
CPU (14 nm)	58.771	65	0.904 GOPS/W	0.213
GPU (8 nm)	563.63	220	2.562 GOPS/W	1.434
FPGA (scaled to 8nm)	7502.7	6.875	1091.3 GOPS/W	75.027
CPU (scaled to 8nm)	73.464	56.875	1.292 GOPS/W	0.532

Table 3: Comparison between the 3 compute platforms implementing 2D convolution with 64 filters

Above results are tabulated based on the equations below

**Scaled Throughput (GOPS):**  $OriginalThroughput \times ClockSpeedScaling$

**Scaled Power:**  $OriginalPower \times ClockSpeedScaling \times Power@sameclockrateScaling$

**Scaled Area:**  $OriginalArea \times AreaScaling$

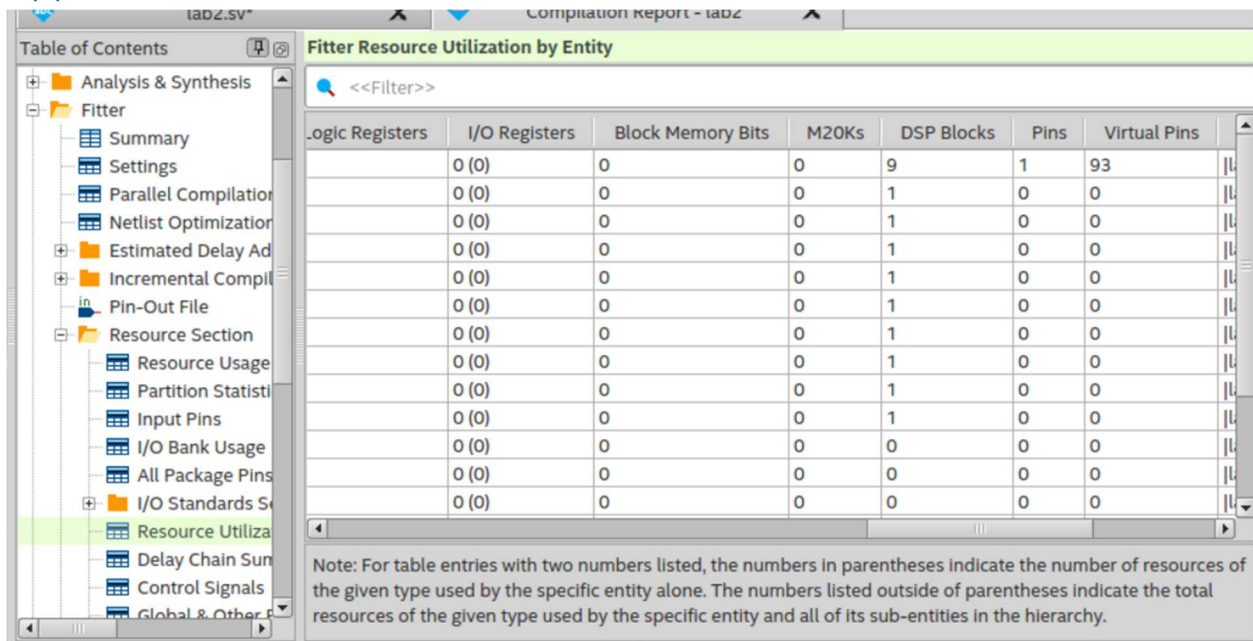
Figure 6: Calculation equation references

FPGA is outstanding compared with CPU and GPU, leading by a significant amount at energy efficiency and area efficiency due to the nature of FPGA for it's dedicated purpose of computing convolutions; FPGA outputs a pixel at each clock cycle while GPU takes several cycle to allow the thread fill up the

entire pixel for the output. GPU is still ahead of CPU in terms of performance due to the massive parallelism.

FPGA is almost a thousand time ahead of CPU and 400 times ahead of GPU when it comes to energy efficiency, because FPGA is more specific at it's given applications, for which enables FPGA to avoid a lot of unnecessary cost associated with generality, and GPU is still better than CPU because the implementation of wider SIMD. The overhead cost reduction is also the reason for outstanding area efficiency of FPGA when compared with GPU and CPU

## Appendix



Logic Registers	I/O Registers	Block Memory Bits	M20Ks	DSP Blocks	Pins	Virtual Pins
0 (0)	0	0	0	9	1	93
0 (0)	0	0	0	1	0	0
0 (0)	0	0	0	1	0	0
0 (0)	0	0	0	1	0	0
0 (0)	0	0	0	1	0	0
0 (0)	0	0	0	1	0	0
0 (0)	0	0	0	1	0	0
0 (0)	0	0	0	1	0	0
0 (0)	0	0	0	0	0	0
0 (0)	0	0	0	0	0	0
0 (0)	0	0	0	0	0	0

Note: For table entries with two numbers listed, the numbers in parentheses indicate the number of resources of the given type used by the specific entity alone. The numbers listed outside of parentheses indicate the total resources of the given type used by the specific entity and all of its sub-entities in the hierarchy.

Figure 7: DSP and M20ks simulation result

Table of Contents		Fitter Summary	
<ul style="list-style-type: none"> <li>Flow Summary</li> <li>Flow Settings</li> <li>Flow Non-Default Glob</li> <li>Flow Elapsed Time</li> <li>Flow OS Summary</li> <li>Flow Log</li> <li>Analysis &amp; Synthesis</li> <li>Fitter <ul style="list-style-type: none"> <li>Summary</li> <li>Settings</li> <li>Parallel Compilation</li> <li>Netlist Optimization</li> </ul> </li> <li>Estimated Delay Ad</li> <li>Incremental Compil</li> </ul>		<<Filter>>	
		Fitter Status	Successful - Wed Nov 1 22:46:28 2023
		Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Standard Edition
		Revision Name	lab2
		Top-level Entity Name	lab2
		Family	Arria 10
		Device	10AX115N1F45I1SG
		Timing Models	Final
		Logic utilization (in ALMs)	2,499 / 427,200 (< 1 %)
		Total registers	9572
		Total pins	1 / 992 (< 1 %)
		Total virtual pins	93
		Total block memory bits	0 / 55,562,240 (0 %)
		Total RAM Blocks	0 / 2,713 (0 %)
		Total DSP Blocks	9 / 1,518 (< 1 %)

Figure 8: Simulation summary window

Type	ID	Message
⚠	332148	Timing requirements not met
	11105	For recommendations on closing timing, run Report Timing Closure Recommendations in the Timing Analyzer.
ⓘ	332146	Worst-case setup slack is -1.230
	332119	Slack End Point TNS Clock
	332119	=====
	332119	-1.230 -174.735 clk
ⓘ	332146	Worst-case hold slack is 0.045
	332119	Slack End Point TNS Clock
	332119	=====
	332119	0.045 0.000 clk
ⓘ	332140	No Recovery paths to report
ⓘ	332140	No Removal paths to report
ⓘ	332146	Worst-case minimum pulse width slack is -1.150
	332119	Slack End Point TNS Clock

Figure 9: Simulation time result

Type	ID	Message
ⓘ	332146	Worst-case minimum pulse width slack is -1.150
	332119	Slack End Point TNS Clock
	332119	=====
	332119	-1.150 -31.600 clk
ⓘ		Analyzing Slow 900mV 0C Model
ⓘ	332146	Worst-case setup slack is -1.332
	332119	Slack End Point TNS Clock
	332119	=====
	332119	-1.332 -172.148 clk
ⓘ	332146	Worst-case hold slack is 0.045
	332119	Slack End Point TNS Clock
	332119	=====
	332119	0.045 0.000 clk
ⓘ	332140	No Recovery paths to report

Figure 10: Simulation time result for slow 900mV



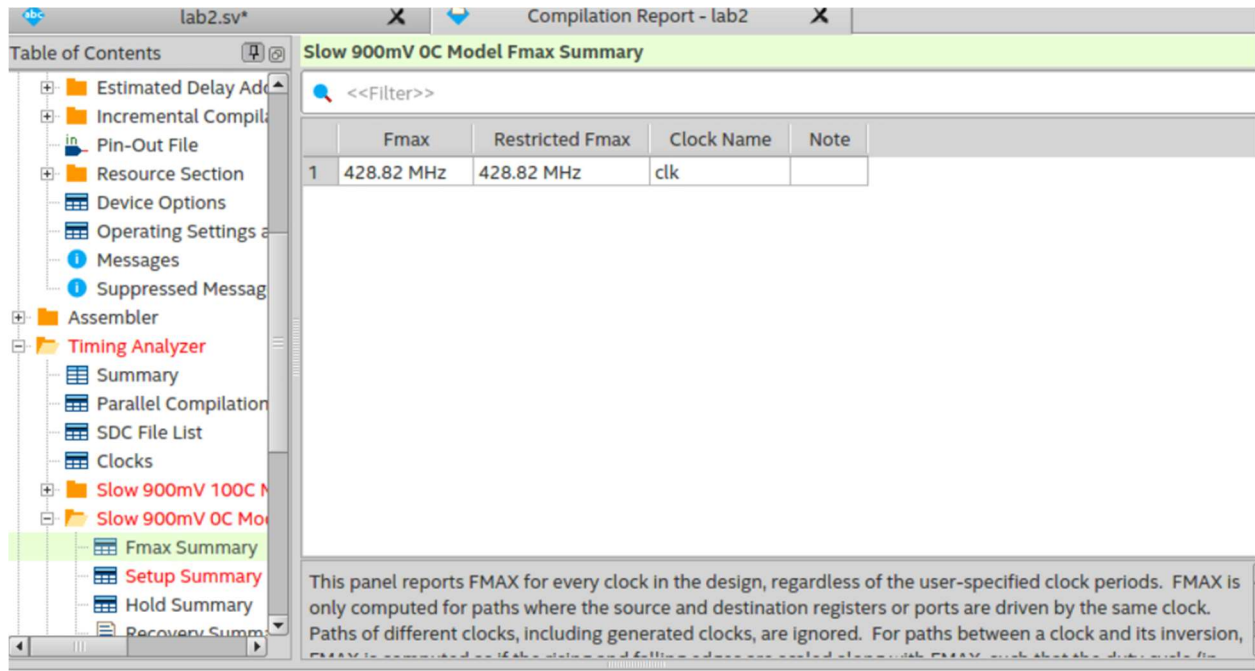


Figure 11: Simulation Frequency result

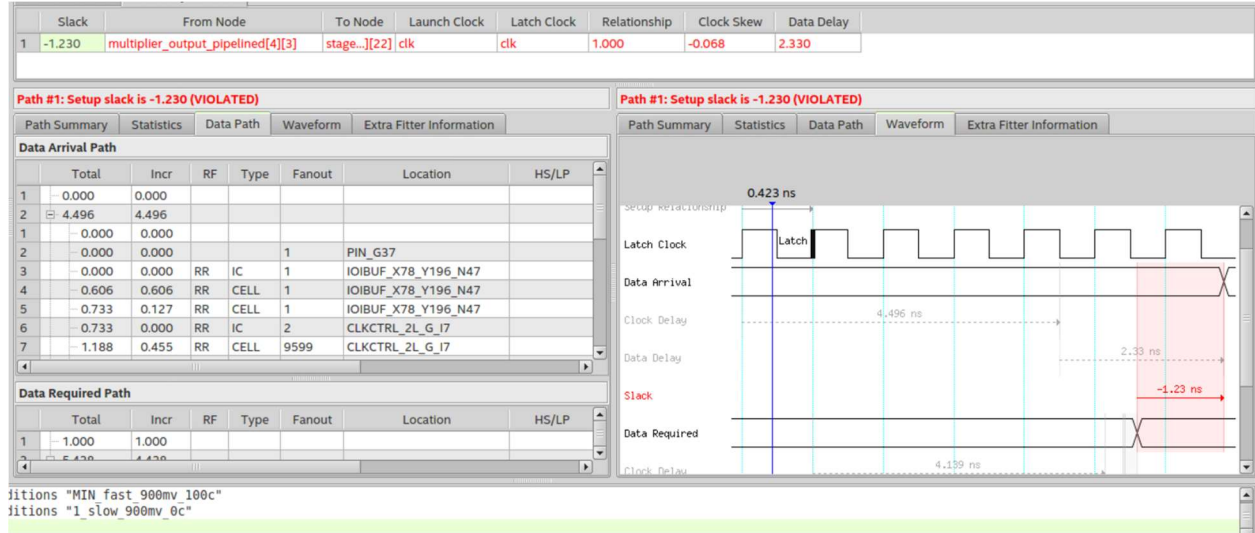


Figure 12: Testbench result



	Block Type	Total Thermal Power by Block Type	Block Thermal Dynamic Power	
1	DSP	0.43 mW	0.36 mW	0.0
2	Combinational cell	2.37 mW	2.21 mW	0.0
3	Register cell	11.51 mW	3.84 mW	0.0
4	IO Analog	0.14 mW	0.13 mW	0.0
5	IO Digital	0.05 mW	0.00 mW	0.0
6	Clock Network	11.21 mW	6.12 mW	0.0

Figure 13: Power dissipation simulation result

## FGPA HDL Code

// This module implements 2D covolution between a 3x3 filter and a 512-pixel-wide image of any height.

// It is assumed that the input image is padded with zeros such that the input and output images have

// the same size. The filter coefficients are symmetric in the x-direction (i.e.  $f[0][0] = f[0][2]$ ,

//  $f[1][0] = f[1][2]$ ,  $f[2][0] = f[2][2]$  for any filter  $f$ ) and their values are limited to integers

// (but can still be positive or negative). The input image is grayscale with 8-bit pixel values ranging

// from 0 (black) to 255 (white).

module lab2 (

input clk, // Operating clock

input reset, // Active-high reset signal (reset when set to 1)

input [71:0] i\_f, // Nine 8-bit signed convolution filter coefficients in row-major format (i.e.  $i\_f[7:0]$  is  $f[0][0]$ ,  $i\_f[15:8]$  is  $f[0][1]$ , etc.)

input i\_valid, // Set to 1 if input pixel is valid

input i\_ready, // Set to 1 if consumer block is ready to receive a new pixel

## Assignment 2

```
input [7:0] i_x,          // Input pixel value (8-bit unsigned value between 0 and 255)
output o_valid,           // Set to 1 if output pixel is valid
output o_ready,           // Set to 1 if this block is ready to receive a new pixel
output [7:0] o_y          // Output pixel value (8-bit unsigned value between 0
and 255)

);

localparam FILTER_SIZE = 3;    // Convolution filter dimension (i.e. 3x3)
localparam PIXEL_DATAW = 8;    // Bit width of image pixels and filter coefficients (i.e. 8 bits)

// The following code is intended to show you an example of how to use parameters and
// for loops in SystemVerilog. It also arranges the input filter coefficients for you
// into a nicely-arranged and easy-to-use 2D array of registers. However, you can ignore
// this code and not use it if you wish to.

logic signed [PIXEL_DATAW-1:0] r_f [FILTER_SIZE-1:0][FILTER_SIZE-1:0]; // 2D array of registers for
filter coefficients

integer col, row; // variables to use in the for loop

always_ff @ (posedge clk) begin
    // If reset signal is high, set all the filter coefficient registers to zeros
    // We're using a synchronous reset, which is recommended style for recent FPGA
architectures

    if(reset)begin
        for(row = 0; row < FILTER_SIZE; row = row + 1) begin
            for(col = 0; col < FILTER_SIZE; col = col + 1) begin
                r_f[row][col] <= 0;
            end
        end

        // Otherwise, register the input filter coefficients into the 2D array signal
    end else begin
```

## Assignment 2

```
for(row = 0; row < FILTER_SIZE; row = row + 1) begin
    for(col = 0; col < FILTER_SIZE; col = col + 1) begin
        // Rearrange the 72-bit input into a 3x3 array of 8-bit filter
        coefficients.

        // signal[a +: b] is equivalent to signal[a+b-1 : a]. You can try to
        plug in

        // values for col and row from 0 to 2, to understand how it
        operates.

        // For example at row=0 and col=0: r_f[0][0] = i_f[0+:8] = i_f[7:0]
        //          at row=0 and col=1: r_f[0][1] = i_f[8+:8] = i_f[15:8]
        r_f[row][col] <= i_f[(row * FILTER_SIZE * PIXEL_DATAW)+(col *
PIXEL_DATAW) +: PIXEL_DATAW];
    end
end
end
end

// Start of your code
localparam PictureWidth = 512+2; // width of our picture
logic [PIXEL_DATAW-1:0] SIPO
[PictureWidth*2+FILTER_SIZE]; // shift register for input data
logic data_valid
[PictureWidth*2+FILTER_SIZE]; // valid data to propagate
// Intermediate outputs
logic signed [31:0] mult_out [9]; // Output for multiplier
logic
data_valid_mult;
logic signed [31:0] stage1_adder_output [4]; // stage 1 adder output
logic signed [31:0] stage2_adder_output [2]; // stage 2 adder output
logic signed [31:0] stage3_adder_output; //stage 3 adder output
```

## Assignment 2

```
logic signed [31:0] stage4_adder_output; // stage 4 adder output

// Pipeline registers

logic signed [31:0] multiplier_output_pipelined [9];

logic data_valib_multiplier_pipelined;

logic signed [31:0] stage1_adder_PipelinedOutput [5];

logic data_valid_stage1_PipelinedAdder;

logic signed [31:0] stage2_adder_PipelinedOutput [3];

logic data_valid_stage2_PipelinedAdder;

logic signed [31:0] stage3_adder_PipelinedOutput [2];

logic data_valid_stage3_PipelinedAdder;

logic unsigned [PIXEL_DATAW-1:0] stage4_adder_PipelinedOutput;

logic data_valid_stage4_PipelinedAdder; // same level as o_valid

// Counter to ensure correctness

logic [15:0] counter;

// Instantiating multipliers. All products are computed in one within the same clock cycle

Multiplier8x8 convo_mult_0
(.i_DataA(SIPO[2+PictureWidth*2]),.i_DataB(r_f[0][0]), .o_Res(mult_out[0])); // Row 0, Column 0

Multiplier8x8 convo_mult_1
(.i_DataA(SIPO[1+PictureWidth*2]),.i_DataB(r_f[0][1]), .o_Res(mult_out[1])); // Row 0, Column 1

Multiplier8x8 convo_mult_2
(.i_DataA(SIPO[0+PictureWidth*2]),.i_DataB(r_f[0][2]), .o_Res(mult_out[2])); // Row 0, Column 2

Multiplier8x8 convo_mult_3
(.i_DataA(SIPO[2+PictureWidth*1]),.i_DataB(r_f[1][0]), .o_Res(mult_out[3])); // Row 1, Column 3

Multiplier8x8 convo_mult_4
(.i_DataA(SIPO[1+PictureWidth*1]),.i_DataB(r_f[1][1]), .o_Res(mult_out[4])); // Row 1, Column 1

Multiplier8x8 convo_mult_5
(.i_DataA(SIPO[0+PictureWidth*1]),.i_DataB(r_f[1][2]), .o_Res(mult_out[5])); // Row 1, Column 2

Multiplier8x8 convo_mult_6 (.i_DataA(SIPO[2]),.i_DataB(r_f[2][0]),.o_Res(mult_out[6])); // Row
2, Column 0

Multiplier8x8 convo_mult_7 (.i_DataA(SIPO[1]),.i_DataB(r_f[2][1]),.o_Res(mult_out[7])); // Row
2, Column 1
```

```
Multiplier8x8 convo_mult_8 (.i_DataA(SIPO[0]),.i_DataB(r_f[2][2]),.o_Res(mult_out[8])); // Row
2, Column 2

// Instantiating adders. each group is in one per cycle

addr32p32 convo_addr_stage_1_0
(.i_DataA(multiplier_output_pipelined[0]),.i_DataB(multiplier_output_pipelined[1]), .o_Res(stage1_adder_output[0]));

addr32p32 convo_addr_stage_1_1
(.i_DataA(multiplier_output_pipelined[2]),.i_DataB(multiplier_output_pipelined[3]), .o_Res(stage1_adder_output[1]));

addr32p32 convo_addr_stage_1_2
(.i_DataA(multiplier_output_pipelined[4]),.i_DataB(multiplier_output_pipelined[5]), .o_Res(stage1_adder_output[2]));

addr32p32 convo_addr_stage_1_3
(.i_DataA(multiplier_output_pipelined[6]),.i_DataB(multiplier_output_pipelined[7]), .o_Res(stage1_adder_output[3]));

addr32p32 convo_addr_stage_2_0
(.i_DataA(stage1_adder_PipelinedOutput[0]),.i_DataB(stage1_adder_PipelinedOutput[1]), .o_Res(stage2_adder_output[0]));

addr32p32 convo_addr_stage_2_1
(.i_DataA(stage1_adder_PipelinedOutput[2]),.i_DataB(stage1_adder_PipelinedOutput[3]), .o_Res(stage2_adder_output[1]));

addr32p32 convo_addr_stage_3_0
(.i_DataA(stage2_adder_PipelinedOutput[0]),.i_DataB(stage2_adder_PipelinedOutput[1]), .o_Res(stage3_adder_output));

addr32p32 convo_addr_stage_4_0
(.i_DataA(stage3_adder_PipelinedOutput[0]),.i_DataB(stage3_adder_PipelinedOutput[1]), .o_Res(stage4_adder_output));

// true if all input values are valid

// else propagate along the computed values

always_comb begin

// Multiplier output only valid if all 9 data bytes are valid

data_valid_mult = data_valid[2+PictureWidth*2] & data_valid[1+PictureWidth*2] &
data_valid[0+PictureWidth*2] & data_valid[2+PictureWidth*1] & data_valid[1+PictureWidth*1] &
data_valid[0+PictureWidth*1] & data_valid[2] & data_valid[1] & data_valid[0] & (counter >=
FILTER_SIZE); //

end
```

```
logic enable;

always_comb begin

enable = i_ready & i_valid;

end

int i;

always_ff @ (posedge clk) begin

if(reset)begin

// Reset counter

counter <= 16'b0;

// Reset SIPO and valid propagation

for(i = 0; i < $size(SIPO); i = i + 1) begin

SIPO[i] <= 8'b0;

data_valid[i] <= 1'b0;

end

end else begin

if(enable)begin

// Counter

//514, current row completed, reset to 1

if (counter == PictureWidth) begin

counter <= 16'b1;

end else begin

counter <= counter + 1'b1;

end

SIPO[0] <= i_x;

// If input is valid, load input into SIPO[0]

data_valid[0] <= i_valid;

for(i = 1; i < $size(SIPO); i = i + 1) begin

SIPO[i] <=

SIPO[i-1]; // Propagate signal is transfered in shift register
```

## Assignment 2

```
data_valid[i] <= data_valid[i-1]; // transfer valid signal
end

//pipeline register of adder stage 1 input and
multiplier_output_pipelined <= mult_out;
data_valib_multiplier_pipelined <= data_valid_mult;
data_valid_stage1_PipelinedAdder <= data_valib_multiplier_pipelined;
data_valid_stage2_PipelinedAdder <= data_valid_stage1_PipelinedAdder;
data_valid_stage3_PipelinedAdder <= data_valid_stage2_PipelinedAdder;
data_valid_stage4_PipelinedAdder <= data_valid_stage3_PipelinedAdder;

// pipeline register of adder stage 1 output and stage 2 input
stage1_adder_PipelinedOutput[0] <= stage1_adder_output[0];
stage1_adder_PipelinedOutput[1] <= stage1_adder_output[1];
stage1_adder_PipelinedOutput[2] <= stage1_adder_output[2];
stage1_adder_PipelinedOutput[3] <= stage1_adder_output[3];
stage1_adder_PipelinedOutput[4] <= multiplier_output_pipelined[8];

// Pipeline reg for adder stage 2 output, and adder stage 3 input
stage2_adder_PipelinedOutput[0] <= stage2_adder_output[0];
stage2_adder_PipelinedOutput[1] <= stage2_adder_output[1];
stage2_adder_PipelinedOutput[2] <= stage1_adder_PipelinedOutput[4];

// Pipeline reg for adder stage 3 output, adder stage4 input
stage3_adder_PipelinedOutput[0] <= stage3_adder_output;
stage3_adder_PipelinedOutput[1] <= stage2_adder_PipelinedOutput[2];

// Pipeline reg for adder stage 4 output
if (stage4_adder_output > 32'sd255) begin
stage4_adder_PipelinedOutput <= 8'd255;
end else if (stage4_adder_output < 32'sd0) begin
stage4_adder_PipelinedOutput <= 8'd0;
end else begin
stage4_adder_PipelinedOutput <= stage4_adder_output[7:0];
```



## Assignment 2

```
end  
  
end  
  
end  
  
end  
  
assign o_y = stage4_adder_PipelinedOutput;  
assign o_ready = i_ready;  
assign o_valid = data_valid_stage4_PipelinedAdder & i_ready;  
  
// End of your code  
  
endmodule  
  
module Multiplier8x8 ( /* synthesis multstyle = "dsp" */  
    input unsigned [7:0] i_DataA,  
    input signed [7:0] i_DataB,  
    output signed [31:0] o_Res  
);  
  
    logic signed [31:0] result;  
  
    always_comb begin  
  
        result = i_DataB * $signed({1'b0,i_DataA});  
  
    end  
  
    assign o_Res = result;  
  
endmodule  
  
module addr32p32 (  
    input signed [31:0] i_DataA,  
    input signed [31:0] i_DataB,  
    output signed [31:0] o_Res  
);  
  
    assign o_Res = i_DataA + i_DataB;  
  
endmodule
```