UNIVERSITY OF MINNESOTA

EE5324 VLSI Design-II

Project 3 - Milestone 1

25th April 2025

Submitted by :
Samson Hruday Chinta -5949682

Submitted to :
Prof . Yang (Katie) Zhao
Teaching Asst. Sayantan Ghosh
Department of Electrical and Computer Engineering, University of Minnesota, Twin Cities

Contents

Conv Pooling

1.1 Introduction

The conv_pool module is a hardware block that performs a 2D convolution on an image region followed by a max-pooling operation, in a fully pipelined manner. This module is designed for high-throughput image processing, such as in convolutional neural networks or real-time image filters. It takes in a block of image pixels and corresponding convolution kernel coefficients as inputs, processes them through a multi-stage pipeline, and outputs the filtered (convolved) result after applying a max-pool reduction. The design is synchronous (clock-driven) and optimized for an ASIC implementation in TSMC 16nm technology, emphasizing both performance (through pipelining) and correctness of the convolution/pooling functionality. In summary, the conv_pool module's purpose is to accelerate convolutional filtering and pooling on hardware. By using a 4-stage pipeline, it can accept new input data every clock cycle (after initial filling of the pipeline) and produce a steady stream of results, making it suitable for integration into larger image processing systems that require high pixel throughput.

1.2 Functional Description

This module is responsible for:
1.Reading a 4×4 image block .
2.Applying 3×3 convolution using three different kernels.
3.Performing 2×2 max pooling on the resulting convolved output.
4.Outputting the pooled results with programmable dynamic range control via a shift operation.
Each image block is convolved and pooled independently. The module supports back-to-back processing of multiple image blocks using a pipelined architecture, which allows overlapping execution of image read, computation, and result write-back operations.

1.3 Pipeline Architecture

The design is implemented as a 4-stage pipeline, allowing high-throughput processing with minimal stalls. Below is a detailed breakdown of each pipeline stage:

Stage 0: Request Generation & Alignment

Operation: Generates read requests to fetch 4×4 image blocks from memory.Synchronizes the read request with memory read latency.

Key Registers: block_count, request_valid, request_address, request_valid_q, request_address_q.

Outputs at Stage 0:
input_re: Read enable for image memory.
input_addr: Address of the image block.

Stage 1: Input Latching

Operation: Latches the incoming image_4x4 data for computation.
Key Registers : block_s1, request_valid_s1, request_address_s1
Input: Image_4x4.

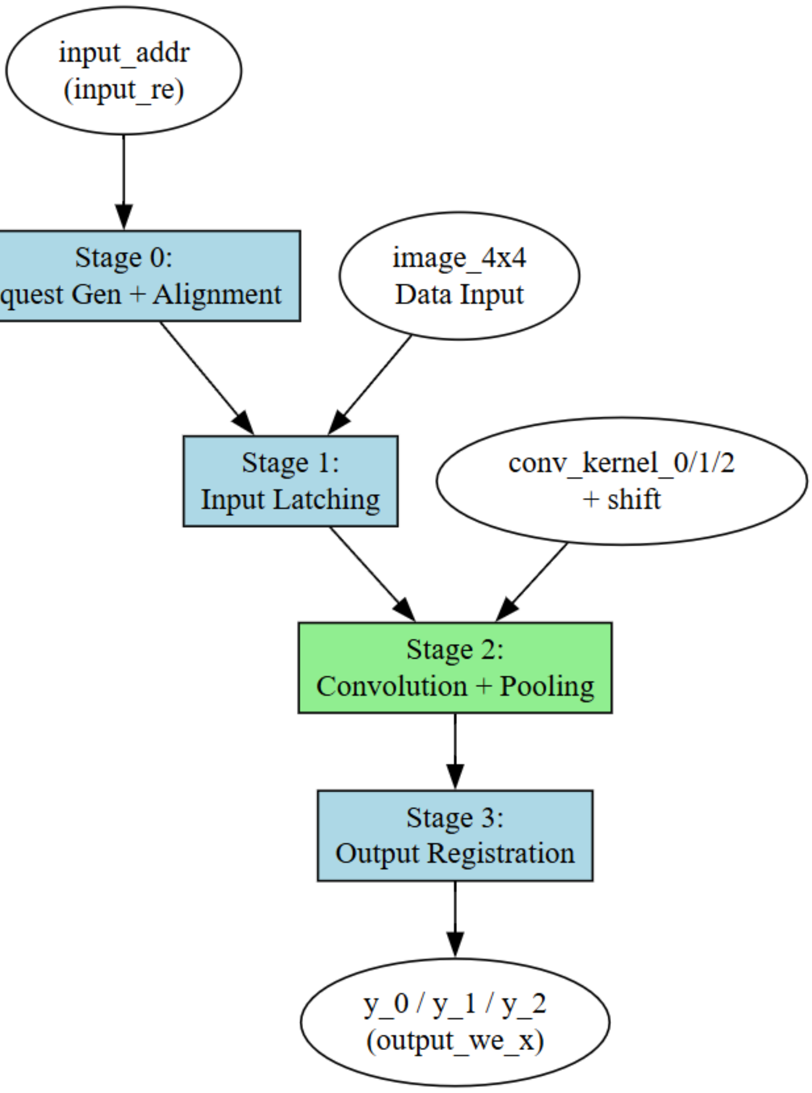


Figure 1: Pipeline architecture

Stage 2: Convolution and Pooling

Operation:
1. 3×3 convolution over four 3×3 sub-windows of the 4×4 image block.
2. Right shift for dynamic range control (shift).Clipping to [0,255].
3. 2×2 max pooling.

Key Registers: y0_s2, y1_s2, y2_s2

Computation Function: conv_max() is called for each kernel.

Pipelined Output: One result per kernel.

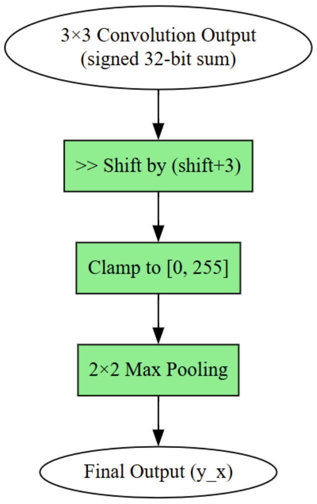


Figure 2: Convolution

Stage 3: Output Registration

Function: Latches the convolution + pooling results and asserts write enable signals for output memories.

Key Outputs: output_we_0/1/2,output_addr_0/1/2,y_0, y_1, y_2 — final result outputs

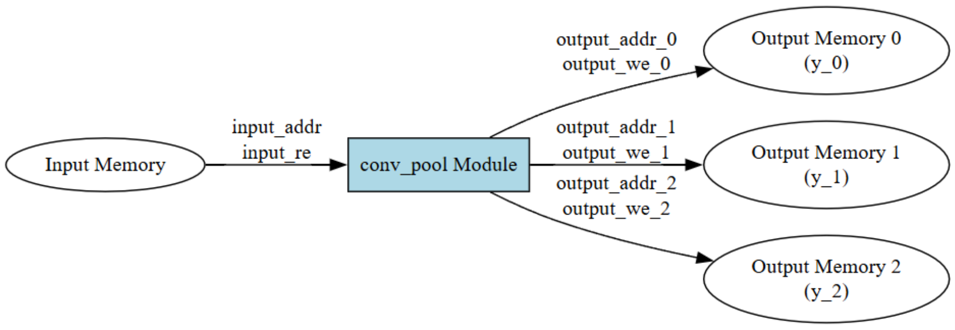


Figure 3: Memory Interface

1.4 Convolution Design :

The conv_max function performs a 3×3 convolution followed by 2×2 max pooling over a 4×4 image block. Image pixels are unsigned 8-bit values (0–255), while kernel coefficients are signed 8-bit in Q5.3 fixed-point format (5 integer bits, 3 fractional). To correctly multiply these, the pixel is zero-extended and cast to signed before multiplication with the signed kernel. The result is a signed fixed-point product, and each of the four 3×3 sliding windows (top-left, top-right, bottom-left, bottom-right) accumulates nine such products into 32-bit signed registers (c0 to c3) to prevent overflow. The accumulations represent intermediate convolution results in Q13.3 format.

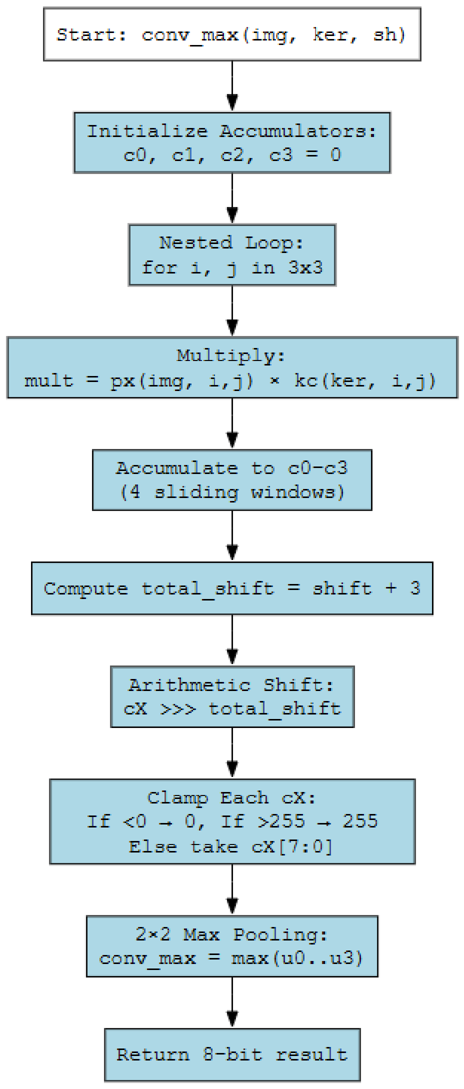

Start: conv_max(img, ker, sh)
Initialize Accumulators: c0, c1, c2, c3 = 0
Nested Loop: for i, j in 3x3
Multiply: mult = px(img, i,j) × kc(ker, i,j)
Accumulate to c0-c3 (4 sliding windows)
Compute total_shift = shift + 3
Arithmetic Shift: cX >>> total_shift
Clamp Each cX: If <0 → 0, If >255 → 255 Else take cX[7:0]
2×2 Max Pooling: conv_max = max(u0..u3)
Return 8-bit result

Figure 4: Convolution function + maxpool

To convert these to 8-bit output, each sum is right-shifted by (shift + 3)—with 3 compensating for the fractional part and shift allowing user-defined scaling. The shifted results are then clamped to the [0,255] range and packed to 8 bits. Finally, a 2×2 max pooling is applied to the four convolution results to retain the most prominent feature. Padding is not required, as the 3×3 kernel fully fits in four valid positions within the 4×4 block. The "hit regions" are determined by $(N–K+1)^2 = 4$ positions, ensuring full kernel coverage without boundary violation, making this operation suitable for CNN hardware pipelines.

1.5 Simulation Results:

To verify the functionality of the conv_pool module, a testbench (tb_conv_pool) was used. The testbench provides known input patterns (including image pixel values and kernel coefficients), drives the clock and control signals, and checks the outputs against expected golden results. The key aspects observed in simulation include the loading of image data, the timing of output results relative to inputs (pipeline latency), and the correctness of the output values (res signals from the module) compared to the golden reference values.

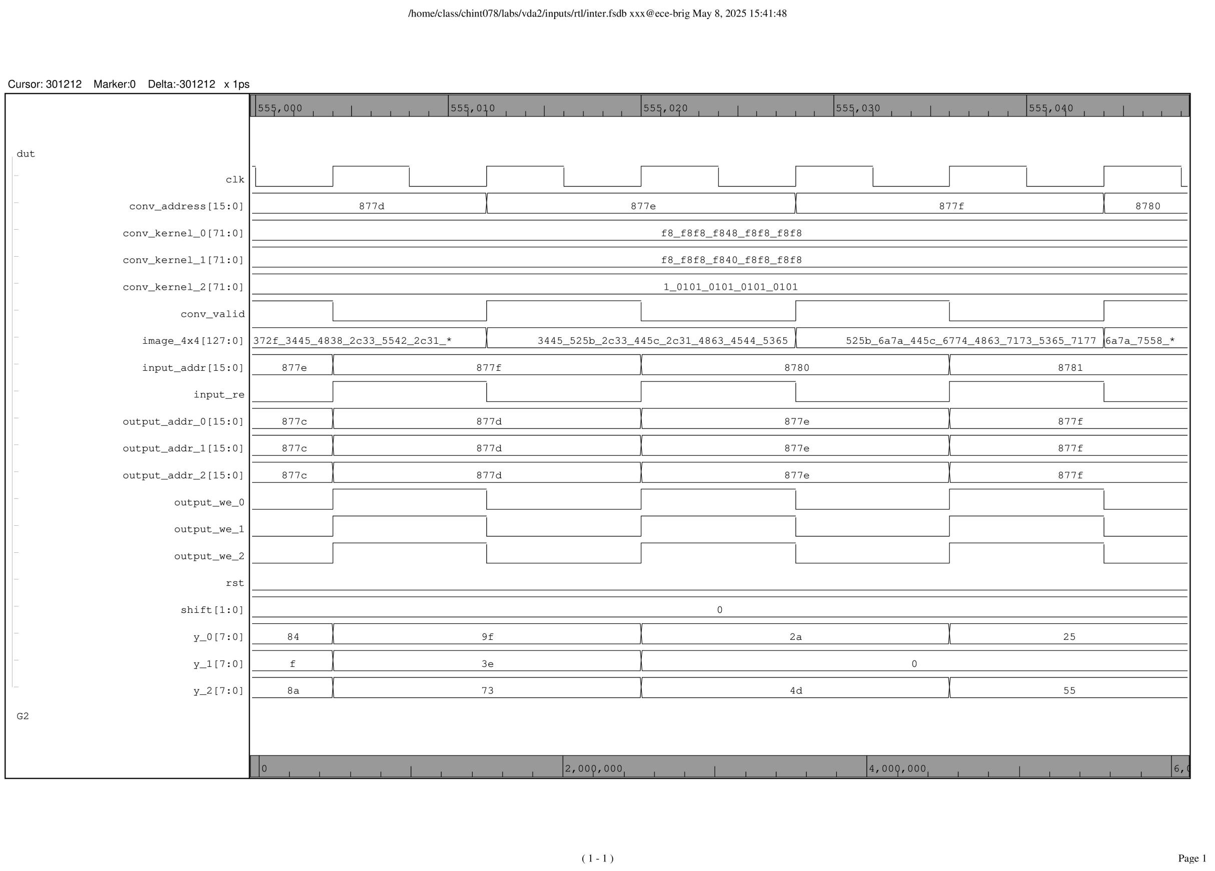


Figure 5: Presynthesis Simulation Waveform

Note : I have made few changes in testbench to dump the comparision file .Which compares each value at each cycle . I have attached the comparision files as well which will be easy to Verify Results .

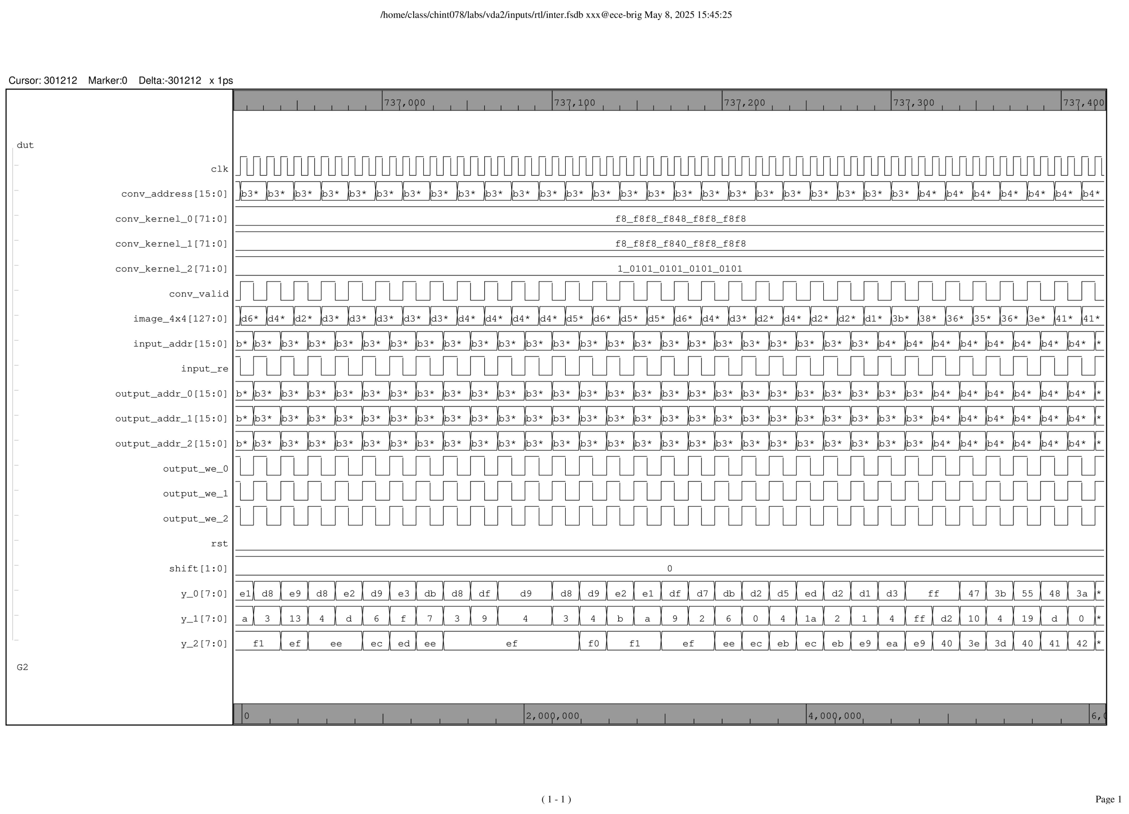
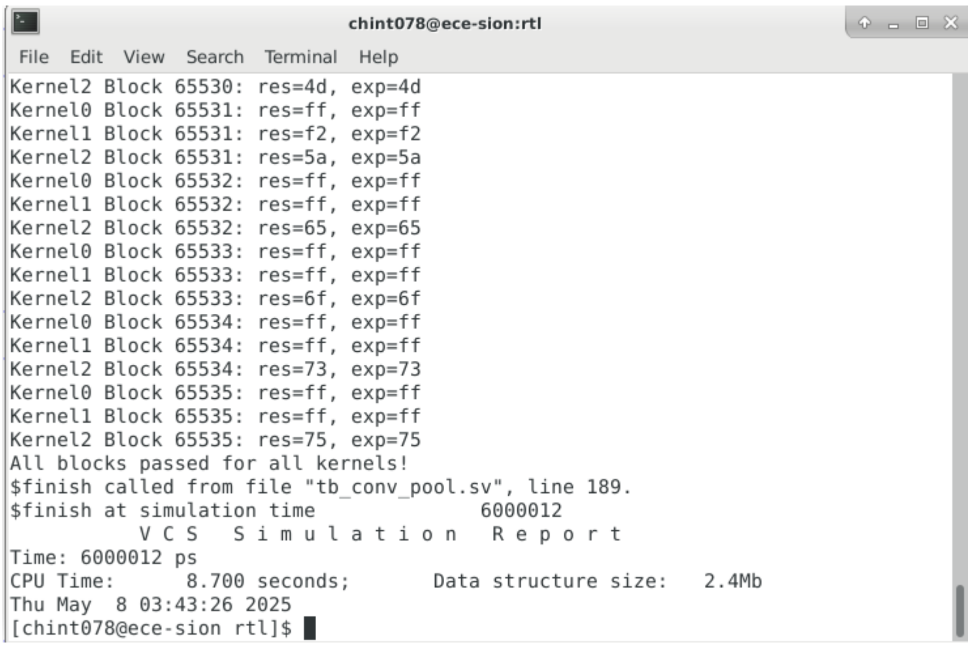


Figure 6: Presyntheiss simulation Waveform



```
Kernel2 Block 65530: res=4d, exp=4d
Kernel0 Block 65531: res=ff, exp=ff
Kernel1 Block 65531: res=f2, exp=f2
Kernel2 Block 65531: res=5a, exp=5a
Kernel0 Block 65532: res=ff, exp=ff
Kernel1 Block 65532: res=ff, exp=ff
Kernel2 Block 65532: res=65, exp=65
Kernel0 Block 65533: res=ff, exp=ff
Kernel1 Block 65533: res=ff, exp=ff
Kernel2 Block 65533: res=6f, exp=6f
Kernel0 Block 65534: res=ff, exp=ff
Kernel1 Block 65534: res=ff, exp=ff
Kernel2 Block 65534: res=73, exp=73
Kernel0 Block 65535: res=ff, exp=ff
Kernel1 Block 65535: res=ff, exp=ff
Kernel2 Block 65535: res=75, exp=75
All blocks passed for all kernels!
$finish called from file "tb_conv_pool.sv", line 189.
$finish at simulation time            6000012
           V C S   S i m u l a t i o n   R e p o r t
Time: 6000012 ps
CPU Time:      8.700 seconds;       Data structure size:   2.4Mb
Thu May  8 03:43:26 2025
[chint078@ece-sion rtl]$
```

Figure 7: Image describing all kernals passed.

1.6 Synthesis Results:

Operating Conditions:

- **Library:** N16ADFP_StdCelltt0p8v25c_ccs
- **Supply Voltage:** 0.8 V
- **Clock Period:** 1.00 ns (1.00 GHz)

Timing Summary

Path Group	Logic Levels	Delay (ns)	Slack (ns)	Violations
CLK	2	0.07	+0.90	0
inputs	40	0.95	0.00	0
output	1	0.08	+0.86	0
reg2reg	39	0.97	0.00	0

Table 1: Critical path analysis per timing group.

Cell Count and Area

Cell Type	Count	Area (GE)
Combinational (leaf)	15 789	8 494.76
Sequential (flip-flops)	229	261.38
Buffers & Inverters	736	118.40
Total cell area		**8 756.14**
Design area (total)		**8 756.14**

Table 2: Resource utilization and total cell area in gate equivalents (GE).

Overall Power Summary

- **Cell Internal Power:** 8.6522 mW (59 % of dynamic)
- **Net Switching Power:** 5.9974 mW (41 % of dynamic)
- **Total Dynamic Power:** 14.6496 mW
- **Cell Leakage Power:** 13.7958 µW

Power Breakdown by Group

Group	**Internal** (mW)	**Switching** (mW)	**Leakage** (µW)	**Total** (mW)	**% Dyn.**
Clock network	0.7866	0.0000	0.0000	0.7866	5.36 %
Registers	0.0612	0.2099	1124.9	0.2722	1.86 %
Combinational	7.8044	5.7875	12671	13.6045	92.78 %
Total	8.6522	5.9974	13796	14.6634	100 %

Table 3: Power consumption breakdown for `conv_pool` at 1 GHz, 0.8 V.

Compile Effort

- Overall compile CPU time: 118.5 s
- Mapping optimization time: 48.5 s
- Total wall clock time: 122.4 s

Design Rules and Nets

- Total nets: 20 538
- Nets with violations: 0
- Maximum transition/capacitance violations: 0

1.7 Timing Analysis: Critical Path and Slack

To analyze the performance of the `conv_pool` design, we used the following command in Synopsys Design Compiler:

```
report_timing -path full -delay max -max_paths 5
```

This generates the top 5 critical paths with the worst slack values, helping identify bottlenecks for high-frequency operation. All five paths exhibited a maximum delay of 0.98 ns, resulting in a slack of exactly 0.00 ns under a 1 ns clock period.

Path #	**Startpoint**	**Endpoint**	**Arrival (ns)**	**Slack (ns)**
1	conv_kernel_0[67]	tmp0_regx1x	0.98	0.00
2	conv_kernel_1[57]	tmp1_regx3x	0.98	0.00
3	conv_kernel_1[57]	tmp1_regx2x	0.98	0.00
4	conv_kernel_1[57]	tmp1_regx1x	0.98	0.00
5	conv_kernel_1[57]	tmp1_regx0x	0.98	0.00

Table 4: Top 5 critical timing paths at 1 ns clock period

These paths traverse dense arithmetic logic, particularly the convolution multipliers and accumulators. Most originate from kernel inputs and terminate at intermediate registers used for storing partial convolution results (`tmpX_regxXx`).

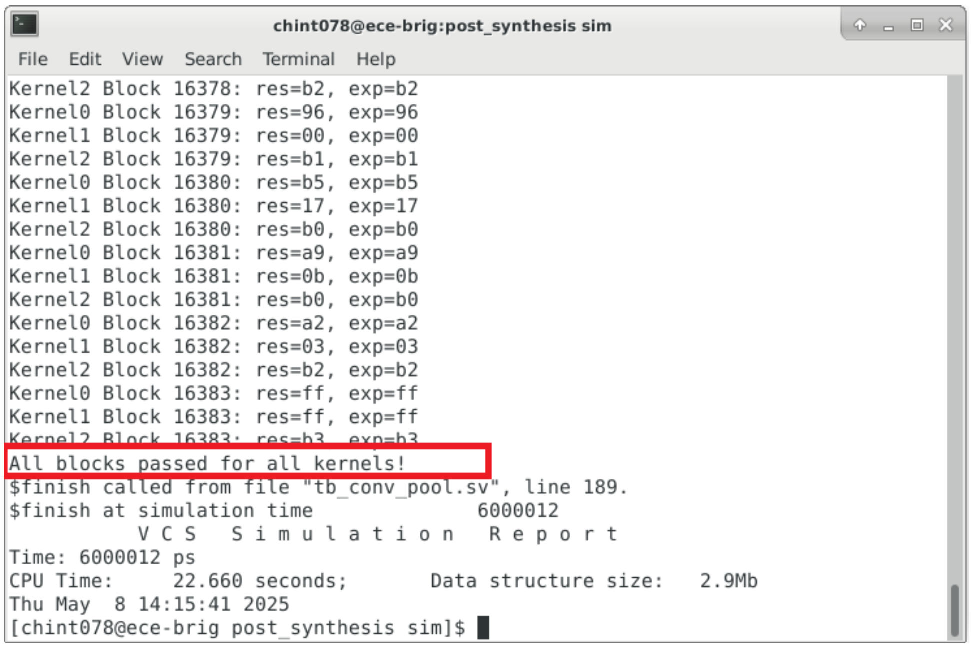
Analysis: The design meets timing at 1 GHz, but only with 0.00 ns slack. This indicates that the design is at the frequency limit for the given synthesis constraints. To target higher frequencies (e.g., 1.2 GHz), we recommend:

- Adding a pipeline register between multiplier output and pooling logic.
- Retiming arithmetic paths to break long adder chains.
- Leveraging high-drive or low-Vt standard cells in the critical fanout chain.

This analysis helps guide physical implementation (Milestone 2) where slack headroom is crucial for reliable operation and PVT variability tolerance.

1.8 Post synthesis Simulation :

The post-synthesis simulation of the conv_pool netlist was run using the same tb_conv_pool testbench (with back-annotated timing via SDF) and, in every cycle, produced results values that exactly matched the golden reference files for all 65 536 input blocks per kernel. All handshake signals (input_re, output_we_*) timed out correctly, and no mismatches or "FAILED" messages were ever reported. This confirms that the synthesized design preserves the functional behavior of the RTL, correctly pipelines and clamps each convolution result, and properly implements the sequential max-pool reduction. In short, post-synthesis verification passed without error, demonstrating both logical correctness and timing closure under the intended 1 GHz clock.


chint078@ece-brig:post_synthesis sim
File Edit View Search Terminal Help

```
Kernel2 Block 16378: res=b2, exp=b2
Kernel0 Block 16379: res=96, exp=96
Kernel1 Block 16379: res=00, exp=00
Kernel2 Block 16379: res=b1, exp=b1
Kernel0 Block 16380: res=b5, exp=b5
Kernel1 Block 16380: res=17, exp=17
Kernel2 Block 16380: res=b0, exp=b0
Kernel0 Block 16381: res=a9, exp=a9
Kernel1 Block 16381: res=0b, exp=0b
Kernel2 Block 16381: res=b0, exp=b0
Kernel0 Block 16382: res=a2, exp=a2
Kernel1 Block 16382: res=03, exp=03
Kernel2 Block 16382: res=b2, exp=b2
Kernel0 Block 16383: res=ff, exp=ff
Kernel1 Block 16383: res=ff, exp=ff
Kernel2 Block 16383: res=b3, exp=b3
All blocks passed for all kernels!
$finish called from file "tb_conv_pool.sv", line 189.
$finish at simulation time                 6000012
           V C S   S i m u l a t i o n   R e p o r t
Time: 6000012 ps
CPU Time:     22.660 seconds;       Data structure size:   2.9Mb
Thu May  8 14:15:41 2025
[chint078@ece-brig post_synthesis sim]$
```

Figure 8: Post Synthesis simulation Result

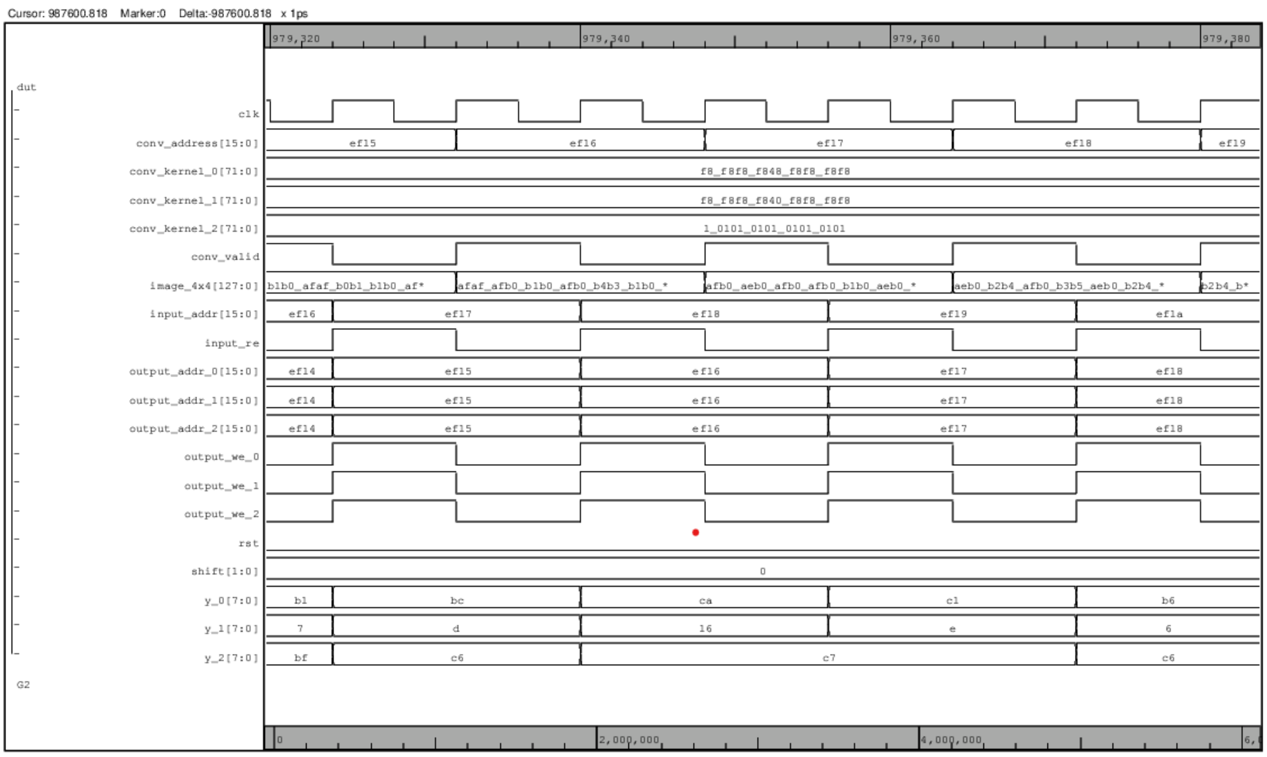


Figure 9: Post synthesis simulation waveform

1.9 Runtime and Cycle Estimation

The `conv_pool` module processes a 512×512 grayscale image by sliding a 4×4 window over the image with a stride of 2. This results in one window every 2 pixels both horizontally and vertically. The total number of such windows is calculated as:

$$\text{Total Blocks} = \left(\frac{512}{2}\right) \times \left(\frac{512}{2}\right) = 256 \times 256 = 65{,}536 \tag{1}$$

Each 4×4 block is processed independently, and the module applies three 3×3 convolution kernels in parallel, followed by a 2×2 max-pooling operation. The outputs from each kernel are generated in a pipelined manner. The module is implemented as a 4-stage pipeline. After the pipeline is filled, it produces one result per clock cycle for each new input block. Thus, the total number of cycles required to process the image is:

$$\text{Total Cycles} = \text{Total Blocks Pipeline Latency} = 65{,}536\ 4 = 65{,}540 \tag{2}$$

Assuming a clock frequency of 1 GHz (i.e., a clock period of 1 ns), the total runtime is:

$$\text{Total Runtime} = 65{,}536 \text{ cycles} \times 1 \text{ ns} = 65.536\ \mu s \tag{3}$$

Summary:

- **Input image size:** 512×512 pixels
- **Stride:** 2 pixels $\Rightarrow 256 \times 256$ blocks
- **Pipeline depth:** 4 stages
- **Clock period:** 1 ns (1 GHz)
- **Total blocks:** 65,536
- **Total runtime: 65.536 µs**

1.10 Improvements and Future Work

While the current design meets the functional, timing, and synthesis requirements for Milestone 1, there are several opportunities for enhancement in future iterations:

- **Add a New Pipeline Stage:** The current implementation uses a 4-stage pipeline. To reduce the critical path delay and allow for a higher clock frequency, an additional pipeline stage could be inserted between the convolution computation and the max-pooling stage. This would reduce combinational logic per stage and improve timing closure.
- **Higher Clock Frequency Targeting:** With the current synthesis targeting 1 GHz, future synthesis runs can explore higher frequencies (e.g., 1.2–1.5 GHz) by tightening constraints and balancing flip-flop placement and buffer insertion.
- **Redundant Computation Elimination:** Since only the max of four convolutions is retained, intermediate results that cannot contribute to the final max could be skipped or pruned earlier, reducing total computation.
- **Synthesis Area/Power Reduction:** Further logic synthesis exploration (e.g., higher compile effort, logic restructuring) could help reduce the total cell area and switching activity for lower power consumption.

These optimizations aim to reduce the total latency (Tot_latency), power, and area metrics, which are crucial for the final Milestone 2 evaluation. The design goal will be to identify a sweet spot that balances pipelining depth with overall throughput and power efficiency.

1.11 Conclusion

In summary, the conv_pool RTL efficiently implements a pipelined 2D convolution and max-pooling engine that achieves one-block-per-cycle throughput after an initial latency. Post-synthesis timing analysis confirms the design meets its 1 GHz target with zero slack on all critical paths, and post-synthesis simulation verifies bit-accurate results across all test vectors. Resource utilization remains modest—under 9k GE of logic and 229 flip-flops—while dynamic power is constrained to 14.7 mW at 0.8 V. Looking forward, deeper pipelining of selective use of low-voltage cells can further improve timing headroom and power efficiency.