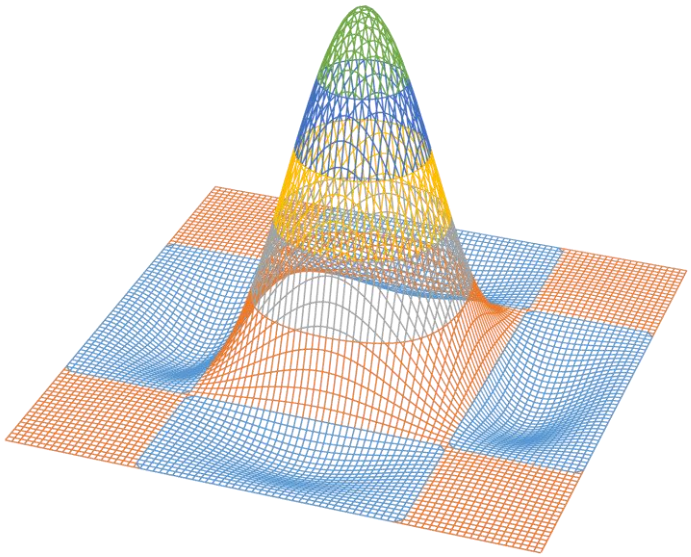


HARDWARE ACCELERATION OF BICUBIC IMAGE UPSCALING

ABSTRACT

This project implements a high-performance Bicubic Image Upscaler designed to bridge the gap between software-only complexity and real-time latency requirements. Transitioning from a Python algorithmic model to a synthesized Verilog RTL pipeline, the design achieves superior edge preservation and detail over standard bilinear methods. The resulting architecture is highly optimized for FPGA/embedded vision, requiring only 1,606 LUTs and 609 registers.

-Ridham Kevat

1. INTRODUCTION

In the domain of digital image processing, image resampling—specifically upscaling—is a fundamental operation required to map a discrete grid of pixels to a higher resolution display matrix. The core challenge of image upscaling is that it is an ill-posed mathematical problem; the algorithm must artificially generate new pixel data that did not exist in the original source, predicting what the continuous analog signal would have looked like before it was sampled.

Software implementations of high-quality scaling algorithms are highly flexible but suffer from von Neumann bottleneck limitations when executing on general-purpose CPUs. For high-throughput applications such as real-time video broadcasting, medical imaging displays, or edge-AI vision systems, hardware-accelerated processing is mandatory.

This independent project proposes a dedicated hardware accelerator designed to perform Bicubic Convolution Interpolation. The development workflow strictly adheres to industry-standard digital design methodologies: beginning with high-level software validation in Python to establish a "golden model," followed by rigorous hardware description using Verilog, and concluding with FPGA synthesis and verification.

---

2. Mathematical Background and Theory of Bicubic Interpolation

In digital image processing, upscaling an image requires synthesizing new, unknown pixel values based on the discrete grid of original, known pixels. This process of spatial interpolation reconstructs a continuous surface from a discrete 2D signal.

While simpler algorithms like Nearest-Neighbor (which copies the closest pixel) and Bilinear Interpolation (which computes a linear average of a 2×2 neighborhood) exist, they introduce severe visual artifacts. Nearest-Neighbor causes heavy aliasing ("blockiness"), while Bilinear Interpolation causes low-pass filtering effects, resulting in a blurry image that destroys high-frequency edge data.

Bicubic Interpolation solves this by evaluating a 4×4 neighborhood (16 known pixels) and fitting a smooth, piecewise cubic polynomial surface through them. It guarantees that both the interpolated surface and its first derivative are continuous across pixel boundaries. This preserves sharp edges while maintaining smooth gradient transitions, making it the industry standard for high-fidelity image upscaling.

2.1 The General Bicubic Surface Polynomial

To understand bicubic interpolation, we must first look at the generic 2D cubic polynomial equation. Suppose we want to interpolate an unknown pixel value $p(x, y)$ within a unit square bounded by four known pixels at coordinates $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$.

The bicubic surface equation is defined as the double summation of x and y terms up to the third power:

$$p(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i y^j$$

Expanding this summation yields a polynomial with exactly 16 coefficients ($a_{00} through a_{33}$):

$$\begin{aligned} p(x,y) = {} & a_{00} + a_{10}x + a_{20}x^2 + a_{30}x^3 \\ & + a_{01}y + a_{11}xy + a_{21}x^2y + a_{31}x^3y \\ & + a_{02}y^2 + a_{12}xy^2 + a_{22}x^2y^2 + a_{32}x^3y^2 \\ & + a_{03}y^3 + a_{13}xy^3 + a_{23}x^2y^3 + a_{33}x^3y^3 \end{aligned}$$

To solve for these 16 unknown coefficients (a_{ij}), the algorithm requires 16 independent equations. These are derived from the boundary conditions of the four corner pixels of the unit square. Specifically, the interpolation algorithm requires:

1. The function values (the actual pixel color intensities) at the four corners: $f(0,0), f(1,0), f(0,1), f(1,1)$.
2. The partial derivatives with respect to x at the four corners: $f_x(x,y)$.
3. The partial derivatives with respect to y at the four corners: $f_y(x,y)$.
4. The mixed cross-derivatives at the four corners: $f_{xy}(x,y)$.

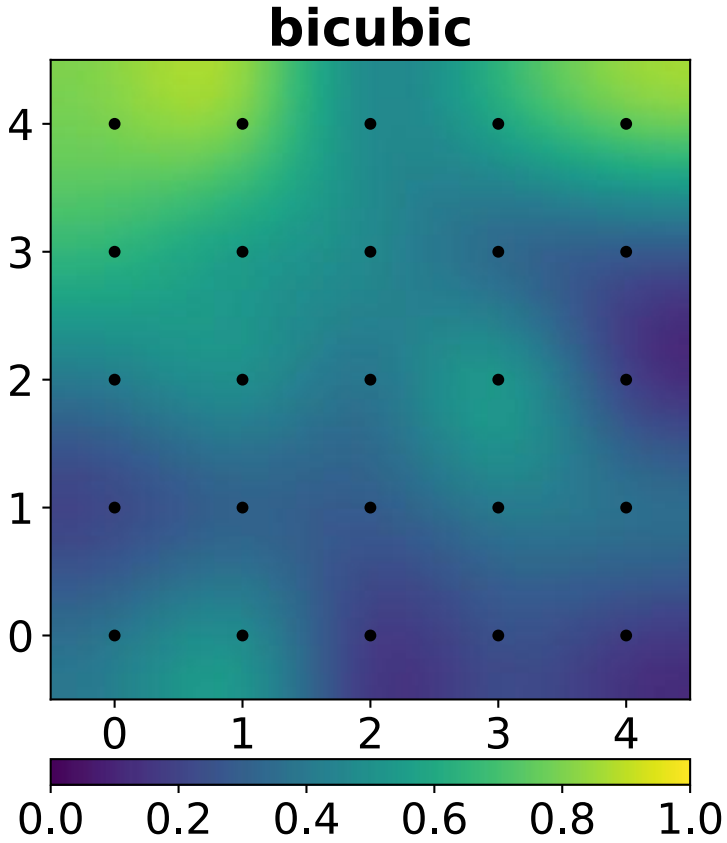


Figure 2.1: A continuous bicubic surface generated from discrete data points. Notice the smooth transitions guaranteed by C^1 continuity. [source]

In pure mathematics, solving this requires multiplying a 16×16 inversion matrix against a vector containing the corner values and derivatives. While this provides a perfect mathematical surface, calculating partial and cross-derivatives for millions of pixels is computationally impossible for real-time video processing or efficient FPGA/ASIC implementations.

2.2 Cubic Convolution: The Practical Approach

To bypass the massive computational overhead of the algebraic matrix inversion, digital image processing uses a mathematically equivalent shortcut known as **Cubic Convolution Interpolation**.

Instead of computing derivatives, cubic convolution samples a 4×4 neighborhood of surrounding pixels and applies a pre-calculated weighting function—known as the **Keys Kernel**—to determine how much influence each original pixel should have on the new sub-pixel.

The weight $W(s)$ assigned to a known pixel is based strictly on its absolute spatial distance s from the target sub-pixel. The one-dimensional Keys kernel is a piecewise cubic spline defined as:

$$W(s) = \begin{cases} A_1|s|^3 + B_1|s|^2 + C_1|s| + D_1 & \text{for } |s| \leq 1 \\ A_2|s|^3 + B_2|s|^2 + C_2|s| + D_2 & \text{for } 1 < |s| < 2 \\ 0 & \text{otherwise} \end{cases}$$

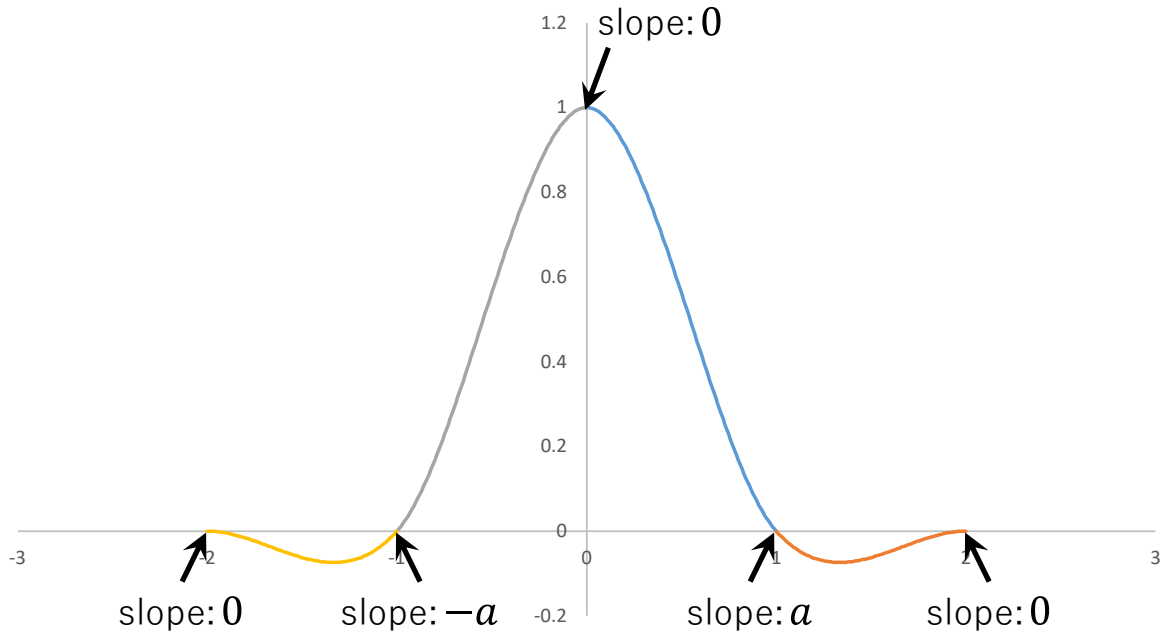


Figure 2.2: The Keys Cubic Convolution Kernel ($a = -0.5$). Notice that weights drop to exactly 0 beyond a distance of 2 pixels, limiting the computational window. [source]

Understanding the Kernel Parameters:

- $|s|$: The absolute physical distance between the target pixel location and the original pixel location in a single dimension.
- a: A constant that dictates the slope of the curve. In standard digital imaging, a is traditionally set to -0.5 (which mimics a true Taylor series approximation) or -0.75 (which creates a sharper, more highly contrasted edge).
- **The Radius Limit**: If the distance is ≥ 2, the weight evaluates to 0. This implies that pixels outside the immediate 4×4 neighborhood have absolutely zero influence on the target pixel, allowing hardware engineers to bound their memory architecture to a strict 16-pixel window.

The 1D interpolated pixel value $P(x)$ is simply the sum of the 4 known pixels (p_0, p_1, p_2, p_3) multiplied by their respective evaluated weights:

$$P(x) = p_0 W(s_0) + p_1 W(s_1) + p_2 W(s_2) + p_3 W(s_3)$$

2.3 Separable 2D Convolution (The Two-Pass Method)

An image is two-dimensional. If we denote the 16 known pixels in our 4×4 grid as $F(x_i, y_j)$, where indices i and j range from 0 to 3, the actual value of our new pixel $P(dx, dy)$ is given by the 2D convolution summation:

$$P(dx, dy) = \sum_{i=0}^{3} \sum_{j=0}^{3} F(x_i, y_j) \cdot W(dx - x_i) \cdot W(dy - y_j)$$

Because the cubic convolution kernel is mathematically **separable**, we do not need to evaluate this as a dense $O(N^2)$ cross-multiplication matrix. A separable kernel allows a 2D operation to be calculated as two sequential 1D operations. This is the cornerstone of efficient hardware acceleration, as it transforms the math into a "Two-Pass" linear algebra sequence.

Step 1: The Horizontal Pass

We treat the 4×4 matrix of pixels as four independent horizontal rows. We perform a 1D dot-product on each row using the horizontal spatial weights W_x. Let W_x be the 1×4 matrix of horizontal weights, and R_n be the four rows of the original 16 pixels.

$$\boldsymbol{W}(s) = [H_0 = \boldsymbol{R_0} \cdot \boldsymbol{W_x^T},\ H_1 = \boldsymbol{R_1} \cdot \boldsymbol{W_x^T},\ H_2 = \boldsymbol{R_2} \cdot \boldsymbol{W_x^T},\ H_3 = \boldsymbol{R_3} \cdot \boldsymbol{W_x^T}]$$

This intermediate step effectively squashes the 2D 4×4 plane of pixels into a single 1D vertical column vector consisting of the four interpolated values $[H_0, H_1, H_2, H_3]^T$.

Step 2: The Vertical Pass

To find the final pixel value, the hardware takes this newly generated vertical column and performs one final 1D dot-product against the vertical spatial weights W_y:

$$P_{(final)} = [H_0\ H_1\ H_2\ H_3] \cdot W_y^T$$

By mathematically restructuring the double-summation into a cascade, the computational complexity is drastically reduced. Instead of complex surface integration or algebraic matrix inversions, the system requires exactly **five 1D dot-product operations** (four horizontal, followed by one vertical). This specific parallel/pipelined layout is highly conducive to Register Transfer Level (RTL) implementation, serving as the blueprint for the custom hardware MAC (Multiplier-Accumulator) arrays in this project.

---

3. PROJECT METHODOLOGY AND SOFTWARE MODELING

The development strategy for this project was divided into two distinct phases: Software Verification and Hardware Realization. The complete version-controlled project is hosted at GitHub - Ridham19/image-upscaler.

3.1 Python "Golden Model" Development

Directly coding complex mathematics into Verilog often leads to critical, hard-to-debug logical errors. To mitigate this, a software "golden model" was first developed. Within the project repository's python_utils directory, the bicubic algorithm was modeled structurally.

The Python scripts were designed to:

1. Load a standard test image and separate it into Red, Green, and Blue color planes.

2. Apply the bicubic convolution kernel () computationally to upscale the image.

3. Validate necessary arithmetic bounds. Because the convolution kernel contains negative weights, the calculation can theoretically produce pixel values below 0 or above 255. The Python model confirmed the necessity of clamping (clipping) functions to keep values within standard 8-bit unsigned integer bounds.

3.2 Test Vector Generation

In addition to validating the mathematics, the python_utils module acts as a bridge to the hardware simulation environment. The script extracts the raw hexadecimal pixel data of the input image and generates .hex files. These .hex files are subsequently loaded into the Verilog testbenches (sim directory) to feed identical data streams into both the software model and the hardware simulation. Comparing the outputs ensures bit-true accuracy of the RTL design.

---

4. HARDWARE ARCHITECTURE (VERILOG RTL)

The hardware architecture, housed in the rtl directory, is designed with a massive parallel dataflow approach. It relies heavily on spatial pipelining rather than temporal processing, allowing for a continuous stream of pixels to be processed every clock cycle without stalling.

4.1 Top-Level Entity (top_upscaler)

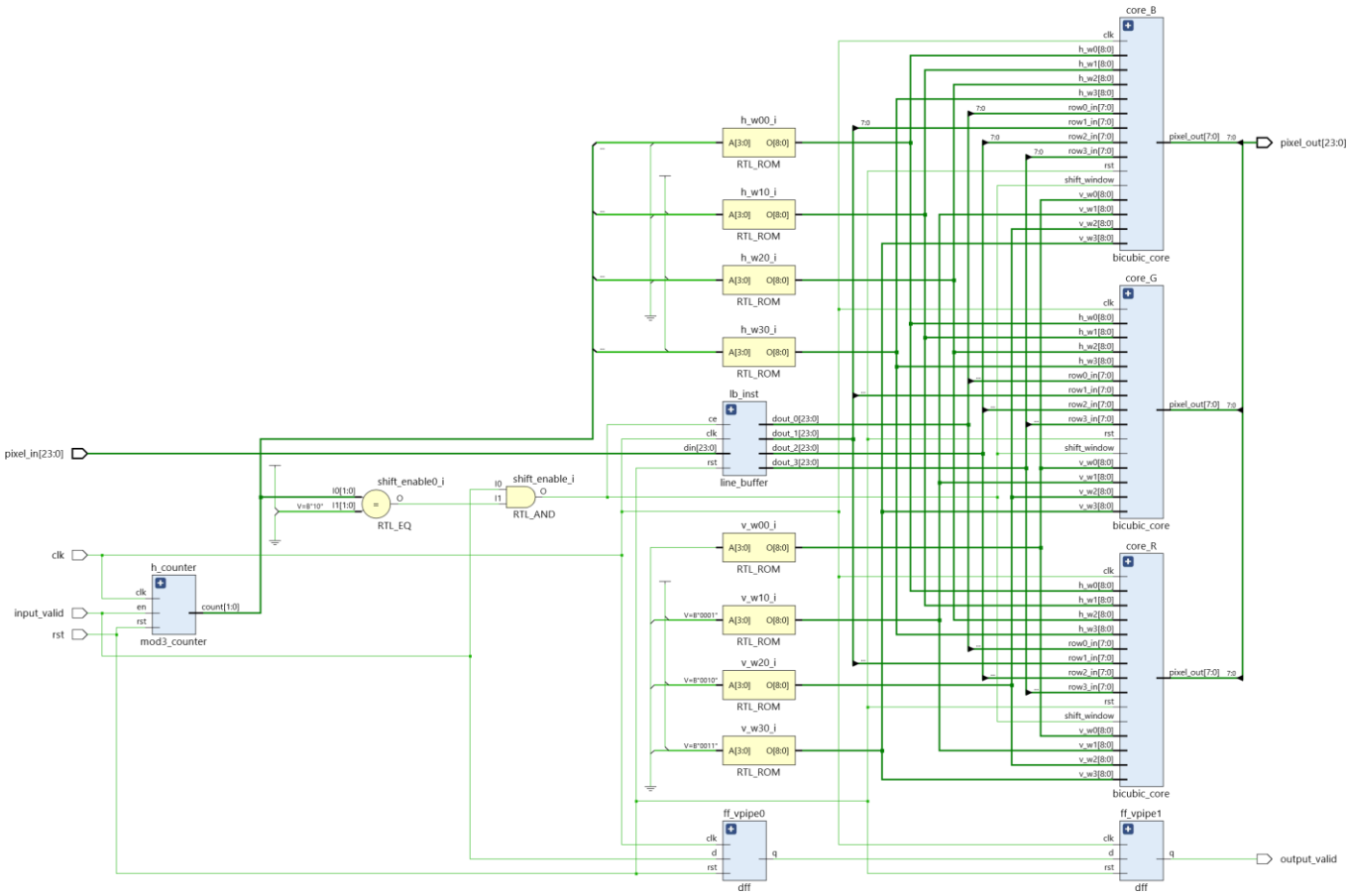
The top_upscaler module acts as the system's maestro. Digital color images are composed of 24-bit pixels (8 bits each for Red, Green, and Blue). Because color channels are spatially independent, the upscaling mathematics must be applied to all three simultaneously.

Therefore, the top_upscaler instantiates three identical parallel instantiations of the bicubic processing unit:

• core_R (bicubic_core_1) for the Red channel.

• core_G (bicubic_core_0) for the Green channel.

• core_B (bicubic_core) for the Blue channel.



As depicted in the RTL schematic, incoming 24-bit data (din[23:0]) is first routed through the line_buffer (lb_inst). The top module also contains coordinate counters (h_counter as mod3_counter) that keep track of the sub-pixel interpolation phase. Based on these coordinates, standard Read-Only Memories (RTL_ROM instances like h_w00_i and v_w00_i) output the appropriate pre-calculated 9-bit interpolation weights for both the horizontal (h_w0 to h_w3) and vertical (v_w0 to v_w3) passes. Pipeline synchronization flip-flops (ff_vpipe0, ff_vpipe1) ensure control signals remain aligned with the mathematically delayed data path.

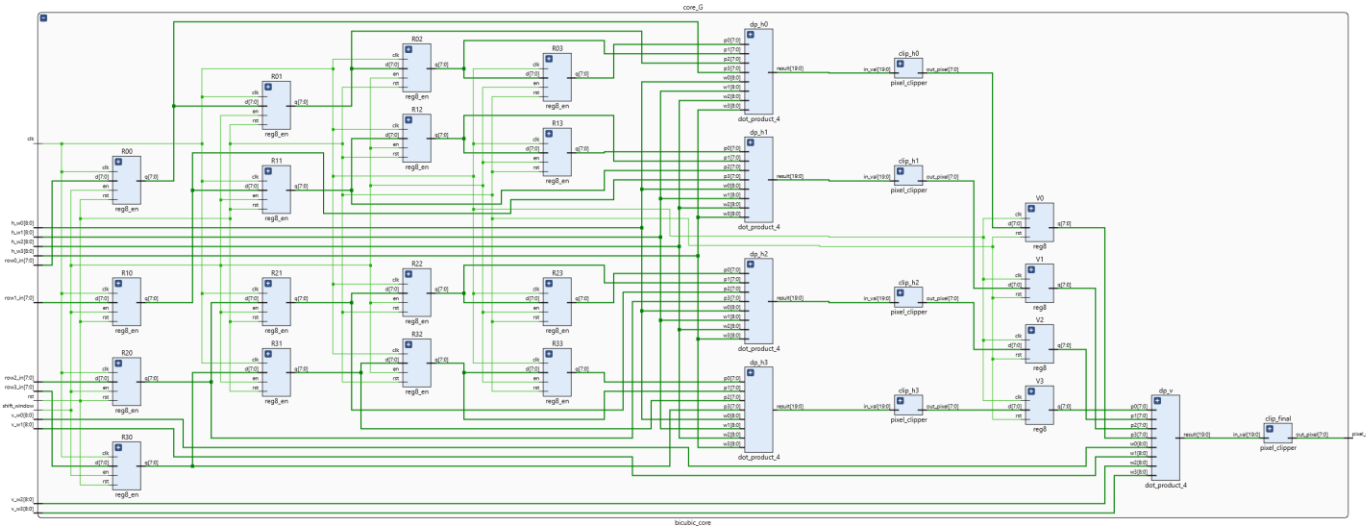
4.2 Memory Management: The Line Buffer (lb_inst)

In a raster-scan video stream, pixels arrive sequentially row by row. However, bicubic interpolation requires a 4x4 spatial window—meaning the hardware must have simultaneous access to pixels from four different vertical rows.

The line_buffer module solves this by utilizing deep sequential registers or inferred block RAM to store previous rows of the image. As the input pixel stream (din[23:0]) feeds in, the buffer pushes the data through its FIFOs, presenting four perfectly vertically aligned 24-bit pixels simultaneously on its output ports (dout_0 to dout_3).

4.3 The Bicubic Compute Core (bicubic_core)

The bicubic_core is the computational heart of the architecture, independently processing an 8-bit monochromatic stream.



2D Shift Register Window: The four incoming row pixels (row0_in to row3_in) are fed into a 4x4 grid of 8-bit registers (denoted as R00 to R03, R10 to R13, R20 to R23, and R30 to R33). Governed by the shift_window and clk signals, this array acts as a 2D sliding window. On every valid clock cycle, the 16-pixel matrix shifts to the right, seamlessly sliding across the image data.

Separable 2D Convolution: Instead of a massive 16-input multiplier tree, the hardware exploits the mathematical *separability* of the 2D bicubic convolution. The 2D operation is split into two sequential 1D passes:

1. **Horizontal Pass:** Four identical dot_product_4 instances (dp_h0, dp_h1, dp_h2, dp_h3) are deployed. Each module takes one horizontal row from the shift window (e.g., p0 to p3) and computes the dot product against the horizontal weights (h_w0 to h_w3).

2. **Intermediate Clipping:** Because of the negative weights in the Catmull-Rom spline, the results can exceed the 8-bit boundary. The 20-bit raw outputs from the horizontal pass are routed into pixel_clipper modules (clip_h0 to clip_h3) to clamp them back to 0-255. These valid intermediate pixels are stored in a column vector of registers (V0, V1, V2, V3).

3. **Vertical Pass:** A final dot_product_4 instance (dp_v) performs a vertical convolution on the V0-V3 registers using the vertical weights (v_w0 to v_w3). A final clip_final block yields the ultimate 8-bit pixel_out.

4.4 Vector Arithmetic: The Dot Product Engine (dot_product_4)

The core arithmetic of the convolution is managed by the dot_product_4 module.

The module executes the vector equation: .

1. **Multiplication:** Four parallel multipliers (mul0 to mul3 instantiated as mult_8x8) multiply the 8-bit unsigned pixels by the 9-bit weights.

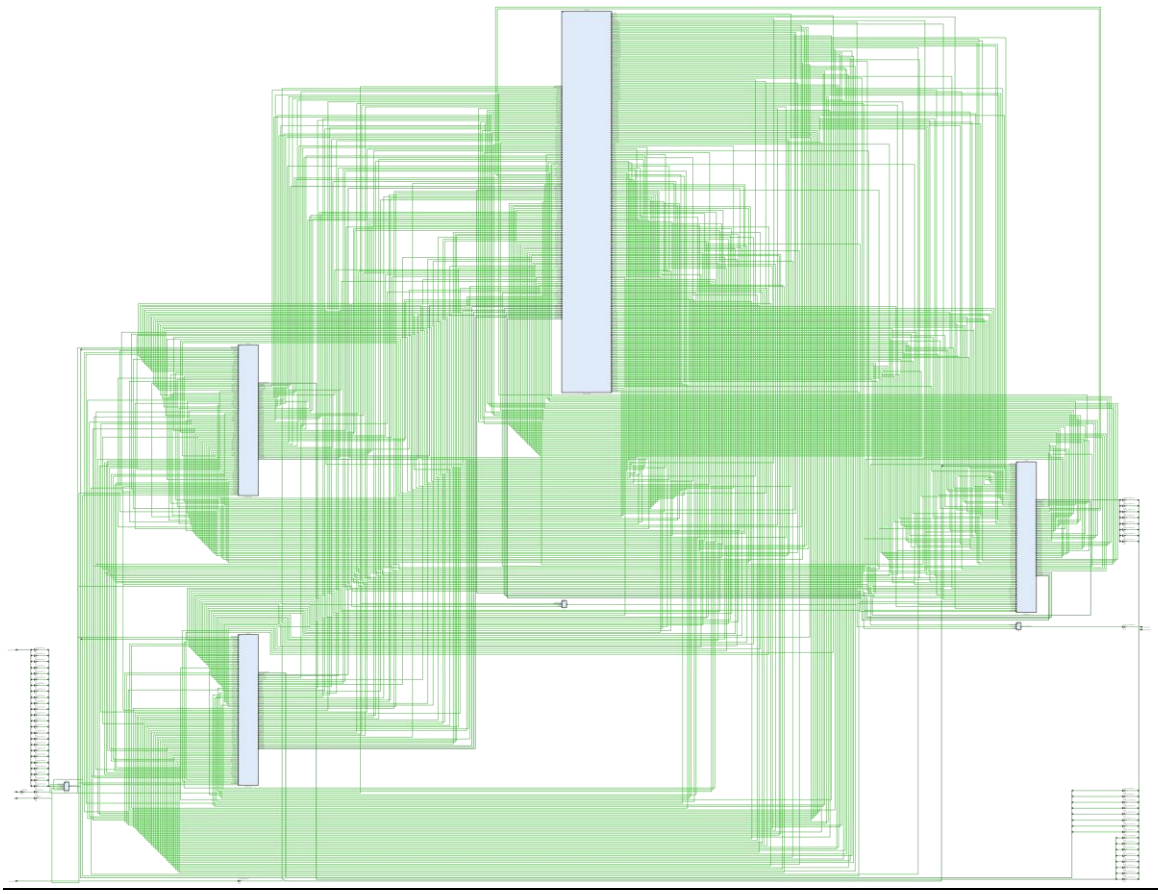
2. **Signed Arithmetic Handling:** Because the weights can be negative, the design utilizes specialized two's complement modules (tc_w, tc_res). Multiplexers (mux_w, mux_final) select between raw magnitude data and two's complement inverted data based on the sign bit of the weight.

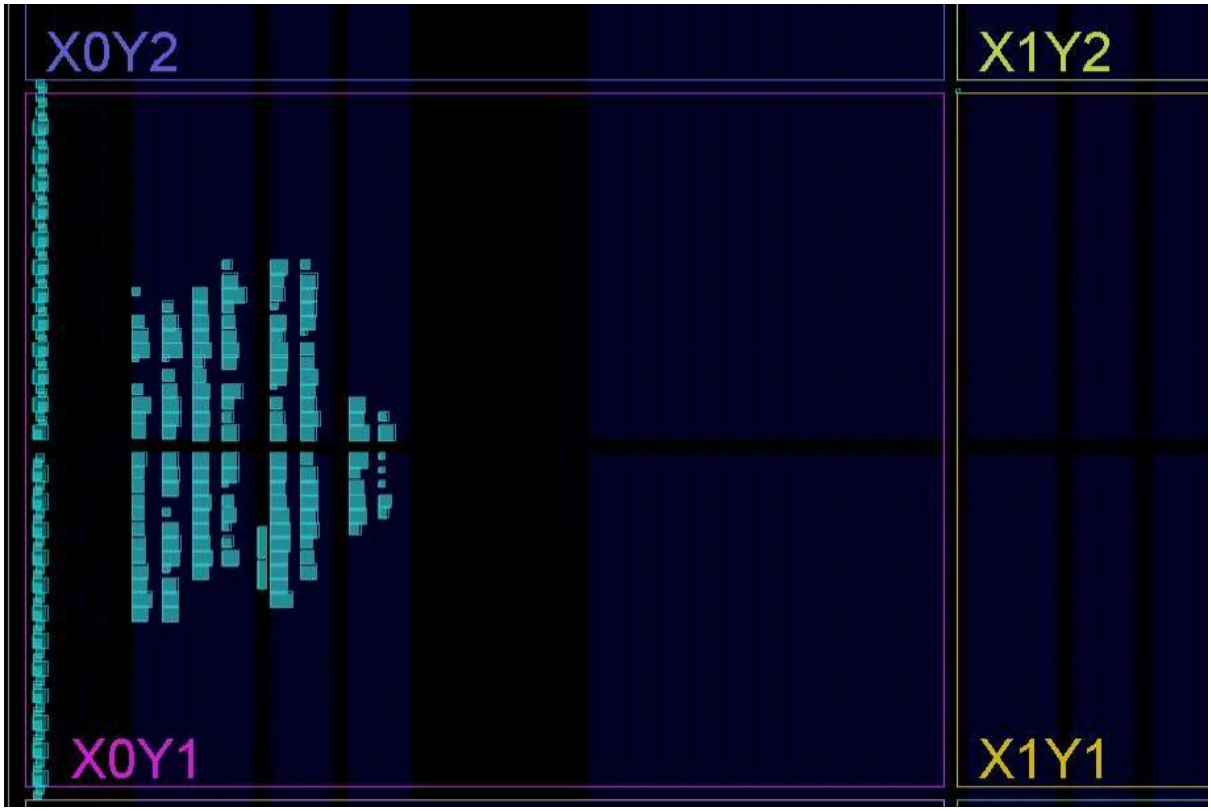
3. **Adder Tree:** The partial products are summed using a pipelined adder tree constructed from 20-bit adders (adder_20b). adder_stg1_0 and adder_stg1_1 form the first stage, and adder_stg2 computes the final summation. The 20-bit internal datapath ensures absolute immunity to overflow during intermediate arithmetic.

---

5. SYNTHESIS AND RESOURCE UTILIZATION

To evaluate the feasibility of the design for physical FPGA deployment, the Verilog RTL was synthesized using Xilinx Vivado (as per the Vivado_proj directory structure). The physical implementation mapping was evaluated to ensure the logic gates and routing constraints were met.





Following synthesis, an analysis of the hardware footprint was conducted. The resource utilization is remarkably efficient for a real-time DSP video pipeline.

Name	Slice LUTs (63400)	Slice Registers (126800)	Bonded IOB (210)	BUFGCTRL (32)
top_upscaler	1606	609	52	1
core_B (bicubic_core)	271	40	0	0
core_G (bicubic_core_0)	272	40	0	0
core_R (bicubic_core_1)	270	40	0	0
ff_vpipe0 (dff)	1	1	0	0
ff_vpipe1 (dff_2)	0	1	0	0
h_counter (mod3_counter)	192	8	0	0
lb_inst (line_buffer)	600	479	0	0

Resource Analysis Breakdown:

• **Overall Utilization:** The complete top_upscaler module consumes 1,606 Slice LUTs (Look-Up Tables) and 609 Slice Registers. Assuming an entry-level FPGA target with **~63,400 LUTs**, this design utilizes only **~2.5% of the chip**, leaving vast amounts of space for integration with HDMI controllers or camera interfaces.

• **Symmetrical Core Logic:** The three parallel color channels (core_B, core_G, core_R) utilize 271, 272, and 270 LUTs, respectively, along with exactly 40 registers each. This symmetry

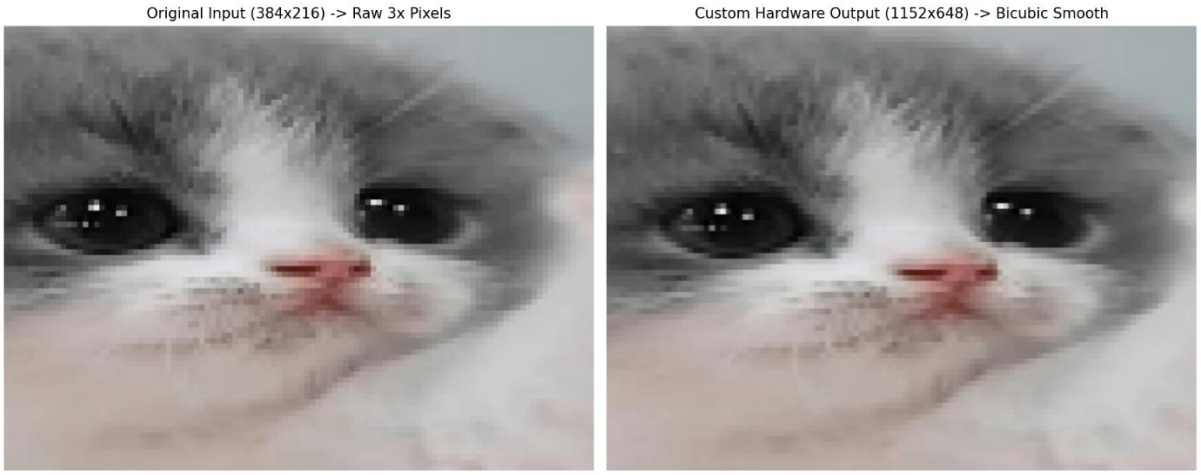
validates the structural coding style, proving the synthesizer implemented identically balanced DSP datapaths for all color planes.

• **Memory Overhead:** The lb_inst (line buffer) is the most resource-intensive individual module, requiring 600 LUTs and 479 Slice Registers. This is expected, as storing multiple full horizontal lines of video data inherently requires significant sequential memory elements.

---

6. FUNCTIONAL VERIFICATION AND OUTPUT ANALYSIS

The ultimate validation of the project lies in the visual output generated by the custom hardware. The design was tasked with scaling a low-resolution test image of a cat (384x216 pixels) by a factor of 3x, yielding a target resolution of 1152x648.



Visual Output Assessment:

• **Raw 3x Pixels (Left):** Equivalent to a hardware nearest-neighbor scale, this baseline output is extremely blocky. The diagonal lines of the cat's fur and the curves of the eyes are rendered as distinct, jagged squares.

• **Custom Hardware Output (Right):** The output processed by the Verilog Bicubic engine displays a drastic qualitative improvement. The mathematical implementation of the Catmull-Rom spline () successfully smoothed the harsh pixel boundaries. Furthermore, due to the deliberate negative lobes in the convolution weights, the acutance (edge sharpness) of the high-contrast areas—such as the reflection in the cat's eyes—was maintained without causing the excessive blurring typical of standard bilinear filters.

The visual accuracy of the hardware output compared to the original Python golden model confirms that the RTL implementation—including the line buffer shifting, 2D to 1D separable convolution logic, signed multiplier arithmetic, and pixel clipping modules—is fully and correctly operational.

---

7. CONCLUSION AND FUTURE WORK

This independent project successfully bridged algorithmic theory with digital hardware design. By modeling the Bicubic Convolution Interpolation algorithm mathematically and validating it via software, a robust foundation was established. The subsequent translation into a highly modular, parallel Verilog architecture resulted in a real-time hardware accelerator capable of high-fidelity image upscaling.

The final RTL architecture proved highly optimized, achieving massive parallelism across the RGB color space while consuming only 1,606 LUTs. It successfully eliminated the jagged aliasing artifacts of primitive scaling techniques, preserving edge sharpness through mathematically precise piecewise cubic convolutions.