Hardware Accelerator for Sobel Edge Detection

Final Report

Submitted by:

Bar Riesel Yanir Vaisman Zanco

Supervised by:

Shahar

Submitted on: 26/03/2025

Contents

List of Figures

List of Tables

1 Introduction

1.1 AMBA APB

The Advanced Microcontroller Bus Architecture (AMBA) Advanced Peripheral Bus (APB) is a low-power, simple protocol designed for low-bandwidth peripherals in System-on-Chip (SoC) designs. It operates in a master-slave model, where the master (typically a CPU) communicates with peripherals (slaves) by reading from or writing to control and status registers.

APB supports a straightforward two-phase transaction:

- Setup Phase: The master sets the address and control signals to prepare for the transfer.
- Access Phase: Data is transferred between the master and slave, completing the transaction.

This simplicity makes APB ideal for configuration and control tasks, such as setting parameters and monitoring status in specialized hardware modules. Its low power consumption and reduced complexity ensure efficient operation in low-bandwidth scenarios.

The APB protocol is a simple and synchronous interface that is not pipelined, and every transfer requires at least two clock cycles to complete. Its design is tailored for accessing the programmable control registers of peripheral devices. APB peripherals are commonly connected to the main system memory via an APB bridge, which acts as an interface to higher-performance buses such as AXI. For instance, a bridge from AXI to APB is used to connect multiple APB peripherals to an AXI memory system. Within this architecture, the APB bridge serves as the *Requester*, while the peripheral devices act as *Completers*, facilitating efficient communication between the two.

The APB protocol supports both write and read transfers, which can occur either with no wait states or with wait states, depending on the peripheral response time. Signals involved in transfers are sampled at the rising edge of the clock ($PCLK$), ensuring synchronization. For a typical write transfer with no wait states, the address ($PADDR$) is stable at the beginning of the transfer cycle, followed by the data ($PWDATA$) being written in subsequent cycles. Control signals such as $PSEL$ (select), $PWRITE$ (write), $PENABLE$ (enable), and $PREADY$ (ready) govern the timing and validity of the transfer. This operation is illustrated in Figure 1.

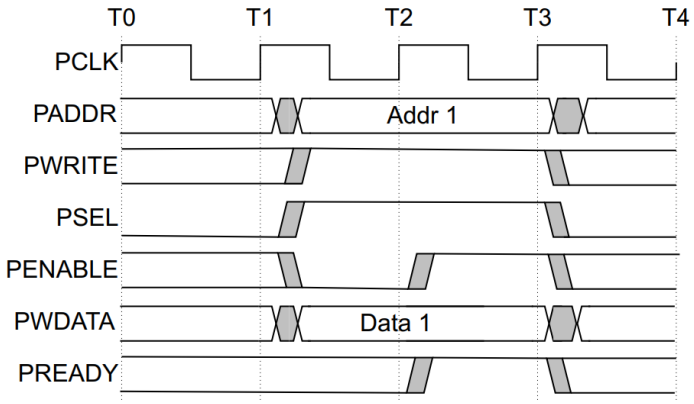


Figure 1: Write transfer with no wait states.

Similarly, during a read transfer, the address is provided first, and the read data ($PRDATA$) is sampled after acknowledgment from the peripheral. The timing for a typical read transfer with no wait states is shown in Figure 2.

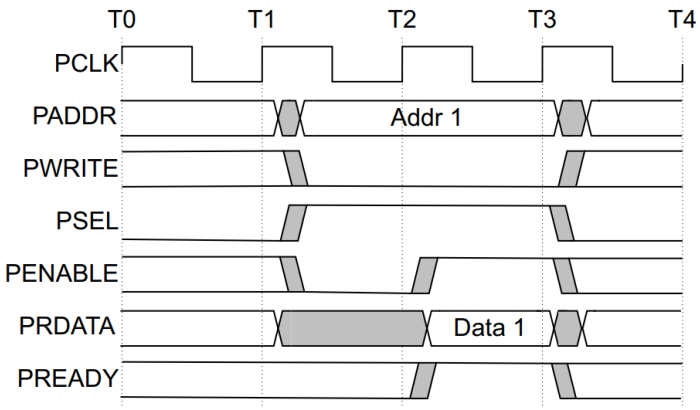


Figure 2: Read transfer with no wait states.

The operation of the APB protocol is managed by a three-state state machine that governs the behavior of the bus interface. The default state, known as *IDLE*, is where no transfer occurs. When a transfer is initiated, the state machine transitions to the *SETUP* state, where the select signal ($PSEL$) is asserted, and the enable signal ($PENABLE$) is deasserted. Following this, the state moves to *ACCESS*, where data transfer takes place, and $PENABLE$ is asserted. The transition back to IDLE or SETUP depends on the $PREADY$ signal, which indicates whether the peripheral is ready to proceed. If $PREADY$ is low, the system remains in the ACCESS state, while a high $PREADY$ signals the completion of the transfer. The state transitions of the APB protocol are depicted in Figure 3, showing the interaction between control signals and bus operation.

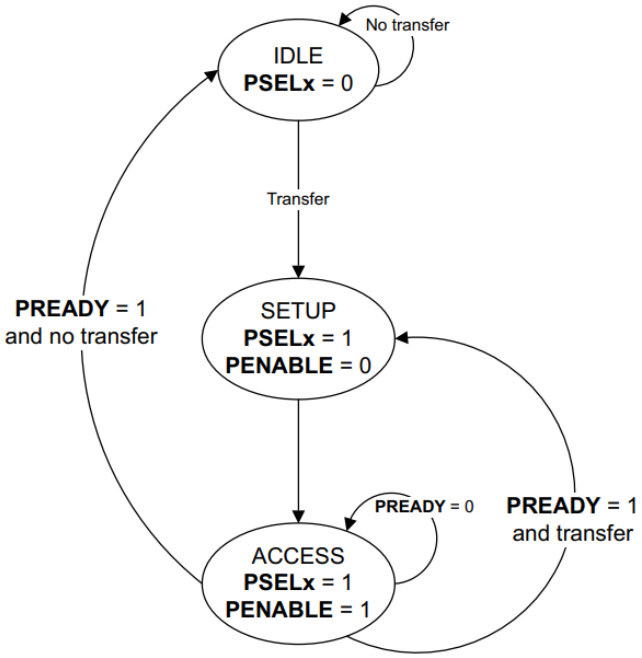


Figure 3: State diagram of the APB protocol

Key signals in the APB protocol include:

- $PADDR$: The address bus that specifies the target peripheral register.
- $PWDATA$: The write data bus used for transferring data to peripherals.
- $PRDATA$: The read data bus used to retrieve data from peripherals.

- $PSEL$: The select signal that activates the desired peripheral.
- $PENABLE$: The enable signal that indicates an active data phase.
- $PWRITE$: A control signal that specifies whether the operation is a write ($PWRITE = 1$) or a read ($PWRITE = 0$).
- $PREADY$: A signal from the peripheral that indicates its readiness for data transfer.

In AMBA, a slave refers to a peripheral device or functional module that responds to requests from the master. These are typically hardware components like timers, UARTs, GPIO controllers, or sensors that require configuration or provide data to the system. The slave operates reactively, waiting for commands from the master and performing specific actions based on the request.

The AMBA slave operates in a master-slave communication model, where the master initiates transactions and the slave processes the corresponding requests. The communication is regulated by control signals that synchronize the flow of data and ensure proper operation within the bus protocol.

Key characteristics of AMBA slaves include:

- **Request-Response Mechanism:** The slave responds to commands initiated by the master, such as read or write operations, driven by signals like:
  - $PSEL$: Selects the slave for communication.
  - $PENABLE$: Enables the active transfer phase.
  - $PWRITE$: Specifies whether the operation is a read or a write.
- **Address Decoding:** The slave decodes the address ($PADDR$) provided by the master to determine if it is the intended recipient.
- **Data Transfer:**
  - For write operations, the master sends data via $PWDATA$, which the slave stores.
  - For read operations, the slave provides data to the master on $PRDATA$.
- **Ready Signal ($PREADY$):** The slave uses $PREADY$ to indicate whether it is prepared for a transaction. A low $PREADY$ introduces wait states.
- **Simplicity:** AMBA slaves are designed for low-power, low-bandwidth applications, without complex pipelining or handshaking mechanisms.

The APB protocol's simplicity and efficiency make it ideal for accessing configuration registers, monitoring hardware modules, and low-power peripherals in modern SoCs [1].

1.2 Sobel Edge Detection

Edges are fundamental features in images, representing regions with significant changes in intensity or color. These boundaries often indicate transitions between objects or between an object and its background, making edge detection a critical task in image processing. By focusing on edges, the amount of data in an image can be significantly reduced while preserving its essential structural information.

Edge detection methods are widely applied in tasks such as object recognition, image segmentation, and feature extraction. However, noise in images can create false edges, making the process more challenging. Edge detection techniques are typically categorized into:

- **Gradient-based methods:** These detect edges by identifying rapid changes in pixel intensity, corresponding to the maxima and minima of the first derivative of the image.
- **Laplacian-based methods:** These identify edges by detecting zero crossings in the second derivative of the image.

The **Sobel filter** is a gradient-based method that efficiently detects edges and their directional properties.

An edge in an image represents a boundary where there is a rapid variation in pixel intensity or color. It marks the transition between different objects or between an object and its background. From a human visual perspective, edges naturally attract attention. Noise in images can cause abrupt changes in pixel values that mimic edges but lack meaningful information. Effective edge detection must overcome these challenges by:

1. Reducing noise in the image while preserving actual edges.
2. Enhancing edge regions while suppressing non-edge areas, often using high-pass filters.
3. Identifying true edge locations by detecting peaks in the filtered output and eliminating false edges caused by noise.

To estimate the intensity gradient at a pixel in an image $f(x, y)$, the following approximations are used for the partial derivatives:

$$\frac{\partial f}{\partial x} \approx f(x+1, y) - f(x-1, y)$$
$$\frac{\partial f}{\partial y} \approx f(x, y-1) - f(x, y+1)$$

To improve accuracy, noise smoothing can be applied using a low-pass filter (e.g., Gaussian) before calculating the gradient. The gradient components g_x and g_y can then be computed using convolution with the appropriate filters h_x and h_y:

$$g_x = h_x * f(x, y)$$
$$g_y = h_y * f(x, y)$$

The Sobel filter is a widely used method for edge detection. It estimates the intensity gradient at each pixel, identifying the steepest changes in intensity and their magnitude. The output provides information on how sharply the intensity changes, indicating the likelihood of an edge, and the probable orientation of the edge.

The Sobel filter uses two 3×3 kernels to approximate the intensity derivatives in horizontal and vertical directions:

$$G_x = A * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = A * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Here, A represents the input image. The G_x kernel detects horizontal edges, while the G_y kernel emphasizes vertical edges.

To compute G_x and G_y, the corresponding kernel is convolved with the input image. For a given pixel, the kernel is placed over the pixel, the convolution is performed, and the result is stored in the output image. The kernel then shifts to the next pixel, and the process repeats for all pixels in the image.

For example, the value of b_{22} in the output image (G_x) is calculated as:

$$b_{22} = -a_{13} + a_{11} + 2a_{23} - 2a_{21} - a_{33} + a_{31}$$

The Sobel kernels produce both positive and negative gradient values. To visualize these gradients:

1. Map zero gradients to a mid-gray tone, where negative gradients appear darker, and positive gradients appear brighter.
2. Use absolute gradient values scaled to a range of 0 to 255, ensuring both strong negative and positive gradients are displayed as bright regions.

The characteristics of an edge can be described by its **magnitude** and **direction**. These properties are computed using G_x and G_y:

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Here, G represents the gradient magnitude, and θ represents the gradient direction. A θ value of 0 indicates a vertical edge that is darker on the left side.

The following example in figure 4 demonstrates the Sobel edge detection process, showing the gradient computation in the x- and y-directions, and their combination to produce the final edge-detected image.

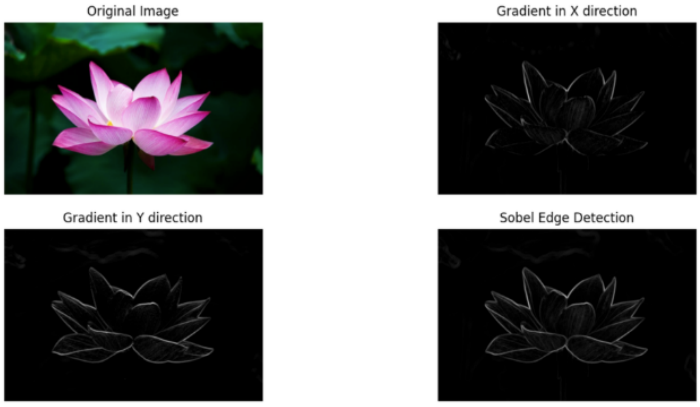


Figure 4: Example of Sobel Edge Detection.

1.3 Paper summary

The paper "A Design of Zynq-based Medical Image Edge Detection Accelerator" [2] explores the acceleration of Sobel edge detection for medical images using the Zynq platform, which integrates ARM processors with FPGA logic. Edge detection is a critical process in medical imaging, highlighting structural details while suppressing less relevant information. The Sobel operator, widely used for edge detection, is enhanced in this design to detect edges in four directions: horizontal, vertical, 45°, and 135°. By computing the maximum absolute value across these directions for each pixel, the system improves edge continuity and accuracy, especially for complex medical images.

The Sobel edge detection module is implemented in Verilog on the programmable logic (PL) of the Zynq platform. The design includes 3x3 convolution kernels for the four edge directions, with the results processed in parallel to maximize efficiency. A line buffer is used to manage pixel data for the sliding convolution window, enabling high-throughput processing. The module computes the gradient magnitude for each pixel and applies a threshold to determine significant edges. The processed edge-detected image is written back to memory and can be visualized on an external display.

Experiments demonstrate that the system achieves significant performance improvements compared to traditional CPU-based implementations, processing high-resolution medical images nearly seven times faster. The Verilog-based design efficiently utilizes the hardware resources of the FPGA while achieving accurate and detailed edge detection, making it a powerful solution for medical imaging applications.

The following figure 5 presents an example taken from the paper summary, demonstrating the results of applying the Sobel edge detection algorithm. It showcases the effects of horizontal, vertical, and combined Sobel operators, along with an improved variant for edge enhancement.

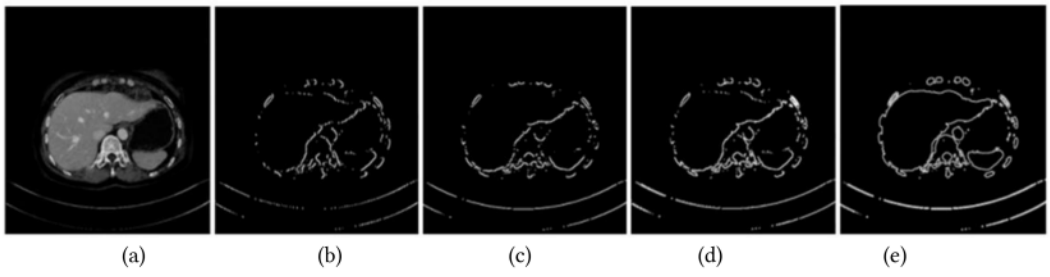


Figure 5: Example from the article: The original image and the processed image :(a) the original image;(b) Image processed by horizontal Sobel operator;(c) images processed by the vertical Sobel operator;(d) Images processed by horizontal and vertical operators;(e) Image processed by the improved Sobel operator.

Figure 6 illustrates the system architecture for real-time Sobel edge detection implemented on the Zynq platform. The workflow begins with the OV5640 camera module, which captures image data. The captured data is converted into AXI4-Stream format using the Video2AXI module. This AXI4-Stream

data is then processed by the Improved Sobel Edge Detection Module, implemented on the FPGA's programmable logic (PL), to compute edges in the image.

Once processed, the edge-detected data is sent to the VDMA (Video Direct Memory Access) module, where it is converted into AXI4 Memory Map format and written to DDR memory. The AXI-SmartConnect module facilitates communication between the VDMA and the Zynq Cortex-A9 processor, allowing the processor to manage and process the data. The processed data is then retrieved from DDR memory, converted back into AXI4-Stream format by the AXI2Video module, and displayed on an HDMI interface.

This architecture demonstrates the Zynq platform's efficient combination of FPGA logic and ARM processing for high-performance, real-time Sobel edge detection in medical imaging applications.

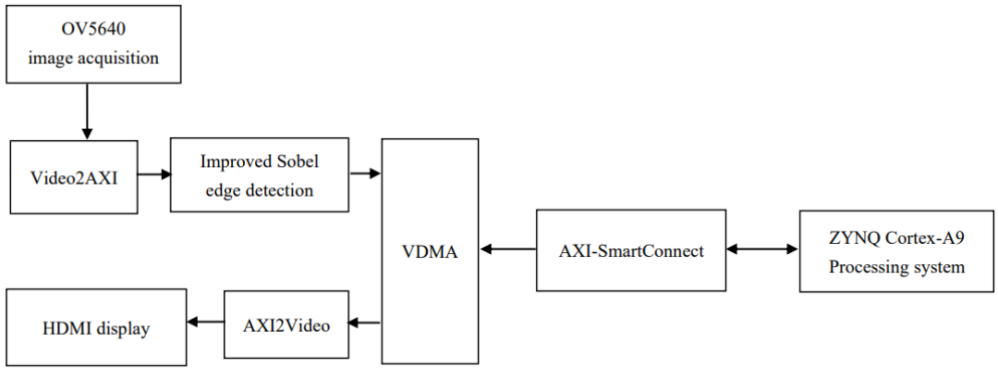


Figure 6: System architecture block diagram.

2 Implementation

2.1 Block diagram

The block diagram in Figure 7 illustrates the architecture of the Sobel Edge Detection Accelerator. Each block performs a specific function within the processing pipeline, ensuring efficient and real-time edge detection.

- **APB Interface:** Acts as the configuration interface, allowing the CPU to write parameters such as image dimensions, threshold values, and the start signal into the Regfile. Additionally, the APB interface allows the CPU to load up to four custom Sobel kernels if it does not wish to use the default Sobel kernel, enabling greater flexibility in edge detection applications.
- **Regfile:** Stores the configuration parameters received from the APB interface and provides them to the controller for synchronization and processing.
- **DMA Interface:** Handles the streaming of 8-bit grayscale pixel data into the system and writes back the processed edge-detected output.
- **Controller:** Manages the state transitions of the accelerator, including Line Buffer loading, processing, and data output. It ensures proper timing and synchronization across all processing stages.
- **Line Buffer:** Temporarily stores incoming pixels from the DMA using three SRAM blocks, maintaining a sliding 3×3 window of pixel data required for convolution.
- **Convolution Block:** Computes the gradient magnitude by applying four Sobel operator kernels to the 3×3 pixel window and summing the absolute values.
- **Threshold Block:** Compares the computed gradient magnitude against the user-defined threshold value and determines whether the pixel should be classified as an edge.
- **Output Interface:** Sends the processed pixel data to the DMA for storage, transmitting either the gradient magnitude for edges or zero for non-edges.

Each block operates in a fully pipelined manner, ensuring that a new pixel is processed every clock cycle, optimizing throughput and efficiency.

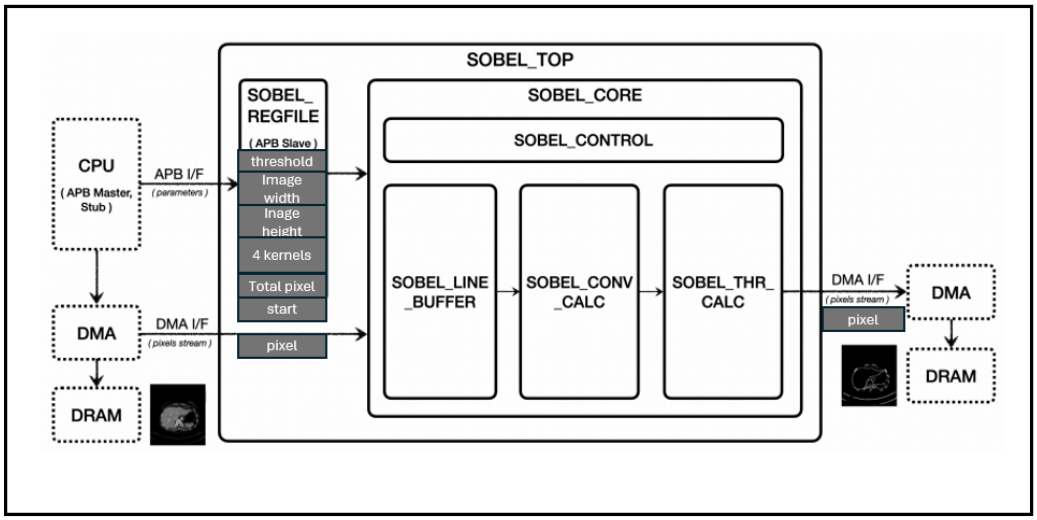


Figure 7: Architecture of the Sobel Edge Detection Accelerator.

2.2 Block description (+flow-chart)

The Sobel Edge Detection Accelerator operates as a fully pipelined system, efficiently processing incoming pixel data and computing edge features in real-time. The system's functionality is structured into distinct operational stages, each controlled by a dedicated controller that transitions between states based on input signals.

The accelerator receives a continuous **stream of 8-bit grayscale pixels** from the **DMA interface**, with **one pixel arriving per clock cycle**. This pixel stream first enters the **Line Buffer Block**, where it is temporarily stored to construct a **3×3 sliding window** necessary for convolution.

In addition to the pixel stream, configuration parameters are retrieved from the **Regfile** via the **APB protocol**. These parameters include:

- **Image Dimensions:** Specifies the width and height of the image.
- **Threshold Value:** Defines the minimum gradient magnitude required to classify a pixel as an edge.
- **Custom Sobel Kernels:** Allows the CPU to define up to four custom 3×3 kernels, which can be used instead of the default Sobel kernel.
- **Start Signal:** Indicates when the processing pipeline should begin.

Once the configuration data is received, the **Start Signal** is forwarded to the controller, which initiates the processing sequence.

Upon receiving the **Start Signal**, the controller transitions to the **Line Buffer Loading Mode**. In this state, the Line Buffer is enabled to start accepting incoming pixel data from the DMA and storing it in three separate **SRAM blocks**, each corresponding to a single image row. The Line Buffer continues to accumulate pixels until it has stored **Image Width + 3** pixels, ensuring that a full 3×3 convolution window can be formed.

Once this condition is met, a signal is raised to indicate readiness, and the controller transitions to the next operational phase.

The controller switches to the **Processing State**, enabling the Convolution Block to begin operations. The convolution computation proceeds as follows:

- The **Convolution Block** retrieves the 3×3 pixel window from the Line Buffer.

- The system applies convolution using **four kernels** to compute gradient values in different orientations.
- The absolute values of the computed gradients are taken.
- The individual gradients are summed to obtain the **final gradient magnitude**.

While in the **Processing State**, the system **continues receiving new pixels from the DMA** at each clock cycle. As each new pixel arrives, it is stored in the Line Buffer, ensuring that the convolution window updates dynamically. This allows the convolution operation to proceed without interruption, maintaining continuous real-time processing.

The computed **gradient magnitude** is then passed to the **Threshold Block**, where it is compared against the **threshold value** received from the Regfile. Based on this comparison:

- If the magnitude exceeds the threshold, the pixel is classified as an **edge** and its gradient magnitude is forwarded.
- If the magnitude is below the threshold, the pixel is set to **zero** (non-edge).

The final processed pixel is then transmitted to the DMA interface, which writes the edge-detected result back to memory, completing the pipeline.

2.3 Pins description

The Sobel Edge Detection Accelerator has a set of input and output pins for configuration, data streaming, and control signals. These pins allow seamless integration into a system for edge detection in grayscale images. The following table provides a detailed description of each pin.

Pin Name	Direction	Width	Description
`clk`	Input	1	Clock signal that drives the accelerator, ensuring synchronous operation.
`reset_n`	Input	1	Active-low reset signal to initialize the module to a known state.
`psel`	Input	1	APB peripheral select signal, used to indicate the accelerator is selected for an APB transaction.
`penable`	Input	1	APB enable signal, used to indicate the data phase of an APB transaction.
`pwrite`	Input	1	APB write signal, indicating whether the transaction is a write (`1`) or read (`0`).
`paddr`	Input	32	APB address bus, specifies which internal register to access during a transaction.
`pwdata`	Input	32	APB write data bus, carries configuration data (e.g., threshold, image dimensions) from the master.
`pixel_in`	Input	8	8-bit input pixel data representing grayscale pixel values from the input image stream.
`valid_in`	Input	1	Indicates that the pixel data on `data_in` is valid and ready for processing by the accelerator.
`pixel_out`	Output	8	8-bit processed pixel data representing the Sobel edge detection result for the current pixel.
`valid_out`	Output	1	Indicates that the data on `data_out` is valid and ready to be consumed by downstream systems.

Table 1: Pins Description for Sobel Edge Detection Accelerator

These pins collectively allow the Sobel Edge Detection Accelerator to receive configuration and data input through the APB and DMA interfaces and output the processed pixel data with proper control

signals. The APB interface ensures the accelerator can be easily configured for different thresholds and image sizes, while the DMA interface provides a streamlined mechanism for pixel data processing.

2.4 Clocks and Resets

The Sobel Edge Detection Accelerator relies on synchronized clock and reset signals to ensure proper operation and coordination among its internal modules. The clock signal (`clk`) is a global input that drives the operation of all internal modules, including the Line Buffer, Convolution Calculator, Threshold Calculator, Regfile, and Controller. It ensures that all modules operate in a synchronized manner, maintaining the pipelined data flow throughout the accelerator. The system is designed to operate at a fixed clock frequency, which is determined based on the performance and throughput requirements of the target application.

The reset signal (`reset_n`) is an active-low asynchronous input used to initialize the entire accelerator to a known state. When the reset signal is asserted, all internal modules are reset to their default states. Specifically, the Line Buffer is cleared, the Convolution Calculator and Threshold Calculator transition to idle states, the registers in the Regfile are either cleared or set to default values, and the Controller moves to its idle state. This initialization ensures that the system starts from a clean and predictable state when powered on or when a reset is triggered by the host processor.

The use of an active-low reset signal (`reset_n`) provides several advantages. Active-low resets are inherently robust against power-on glitches. When the system powers up, the reset signal is naturally held low, ensuring proper initialization of all modules before operation begins. Many standard logic gates and flip-flops are optimized for active-low reset inputs, making their integration straightforward. Additionally, using pull-down resistors ensures that the reset signal remains low during power-on until the clock signal stabilizes, at which point the reset is de-asserted to allow normal operation.

The clock and reset signals are distributed across all sub-modules of the accelerator using a centralized distribution network. Care is taken to ensure glitch-free operation during reset transitions, with proper synchronization mechanisms implemented to avoid metastability issues. Internal logic ensures that no processing begins until the reset signal is de-asserted and the clock signal is stable. This approach guarantees reliable operation and maintains the integrity of the pipelined architecture.

2.5 Memory

The Sobel edge detection accelerator requires an efficient memory architecture to store and process image data in real time. To achieve this, the system utilizes **three separate dual-port SRAM modules**, each responsible for holding a single row of pixel data. This allows for efficient memory access and seamless shifting of pixel values as new data arrives.

2.5.1 Choice of SRAM

The decision to use three independent dual-port SRAM blocks is driven by the following requirements:

- **Efficient Row Storage:** Each SRAM block holds a single row of pixel data, simplifying access patterns and allowing seamless shifting between rows.
- **Parallel Read/Write Access:** Dual-port SRAM allows new pixel data to be written while simultaneously reading.
- **Low Latency:** Ensures that convolution operations can be performed in real-time without delays.
- **Reduced Memory Overhead:** Using separate SRAM blocks for each row eliminates the need for a larger, monolithic memory structure.

2.5.2 Memory Size and Depth Requirements

Since the Line Buffer stores the first three rows of pixel data for Sobel convolution, each SRAM block must be sized accordingly:

- **Image Width:** 1920 pixels
- **Rows Stored:** 3 rows (each in a separate SRAM block)
- **Pixel Bit Depth:** 8 bits per pixel (assuming grayscale input)

Each individual SRAM block requires:

$$1920 \times 8 = 15360 \text{ bits} = 1.92 \text{ KB}$$

Since there are **three SRAM blocks**, the total memory required is:

$$3 \times 1.92 \text{ KB} = 5.76 \text{ KB}$$

SRAM Configuration:

- **Number of SRAM Blocks:** 3 (each storing a single row)
- **Word Size:** 8 bits (1 byte per pixel)
- **Depth per SRAM:** 1920 words (corresponding to one row)
- **Total SRAM Storage:** 5.76 KB

By employing **three separate SRAM modules**, each dedicated to a single row, the system achieves a highly efficient and structured memory layout. The dual-port feature allows the convolution operation to access multiple rows simultaneously, ensuring smooth and uninterrupted processing of the image.

2.6 Interfaces description

The Sobel Edge Detection Accelerator uses the APB interface for configuration and control. The APB interface allows the host processor, in our case the CPU, to communicate with the accelerator by reading from and writing to internal registers. This interface ensures that the accelerator's parameters, such as the threshold value and image dimensions, can be configured dynamically during operation.

The following registers are accessible through the APB interface:

- **Control Register** Used to start the accelerator.
- **Threshold Register** Configures the threshold value for edge detection.
- **Image Width Register** Specifies the width of the input image in pixels.
- **Image Height Register** Specifies the height of the input image in pixels.

Table 2 summarizes the signals used in the APB interface for the Sobel Edge Detection Accelerator. These signals enable configuration, control, and data transfer between the accelerator and the CPU, which acts as the host processor.

Signal Name	Direction	Width	Description
`psel`	Input	1	Selects the APB slave (Sobel Accelerator) for the current transaction.
`penable`	Input	1	Indicates the data phase of the APB transaction.
`pwrite`	Input	1	Specifies whether the transaction is a write (`1`) or read (`0`).
`paddr`	Input	32	The address of the register being accessed (e.g., Control, Threshold).
`pwdata`	Input	32	Data to be written to the specified register.
`prdata`	Output	32	Data read from the specified register.
`pready`	Output	1	Indicates the slave is ready to complete the transaction.
`pslverr`	Output	1	Indicates an error in the transaction (e.g., invalid address).

Table 2: APB Interface Signals for Sobel Edge Detection Accelerator

Each signal in the APB interface plays a key role in facilitating communication between the host processor and the Sobel Accelerator. The host processor can use these signals to configure the accelerator and monitor its status during operation.

The Sobel Edge Detection Accelerator uses the DMA (Direct Memory Access) interface to handle the streaming of pixel data during processing. The DMA interface allows efficient transfer of data between the accelerator and memory without burdening the processor, enabling real-time edge detection.

The input pixel data is received as a continuous stream through the `data_in` signal, with each pixel represented as an 8-bit grayscale value. The `valid_in` signal ensures that the data on `data_in` is valid and ready for processing. The accelerator processes one pixel at a time in a pipelined manner, enabling efficient handling of the incoming data. The DMA controller streams the input image to the accelerator row by row, allowing the line buffer to generate the 3x3 sliding window required for the Sobel convolution.

Once the processing is complete, the edge-detected pixel data is output as an 8-bit grayscale value through the `data_out` signal. The `valid_out` signal indicates when the data on `data_out` is valid and ready to be written back to memory. The DMA controller writes the processed pixel stream back to memory row by row, ensuring that the output image is stored in the same order as the input, maintaining the structure and alignment of the original image.

The following table 3 summarizes the signals used in the DMA interface for the Sobel Edge Detection Accelerator. These signals facilitate the streaming of input pixel data from memory and the writing of processed edge-detected data back to memory.

Signal Name	**Direction**	**Width**	**Description**
`pixel_in`	Input	8	8-bit input grayscale pixel data received from the memory via the DMA controller.
`valid_in`	Input	1	Indicates that the data on `data_in` is valid and ready for processing.
`pixel_out`	Output	8	8-bit processed pixel data (edge-detected) sent back to memory via the DMA controller.
`valid_out`	Output	1	Indicates that the data on `data_out` is valid and ready to be written back to memory.

Table 3: DMA Interface Signals for Sobel Edge Detection Accelerator

2.7 Sub-units description

The Sobel Edge Detection Accelerator is a hardware system designed for high-speed, real-time edge detection in grayscale images. It is composed of several modular blocks that work together in a pipelined manner to process image data and detect edges using the Sobel operator. Each block in the system is optimized for specific tasks, enabling efficient and synchronized processing. The blocks are designed to operate in a pipelined manner. The Line Buffer continuously streams input data and provides the sliding window to the Convolution Calculator. The Convolution Calculator processes the data in parallel and forwards the gradient magnitudes to the Threshold Calculator. The Threshold Calculator produces the final edge-detected output pixel, which is then streamed back to memory via the DMA interface.

The modular design of the Sobel Edge Detection Accelerator ensures efficient data processing and high performance. Each block plays a critical role in the pipeline and contributes to the overall functionality of the system.

2.7.1 Regfile

The **Regfile** serves as the configuration unit for the accelerator, storing essential parameters that control the processing pipeline. It holds values such as:

- **Threshold Value:** The minimum gradient magnitude required for edge detection.
- **Image Dimensions:** The width and height of the input image.
- **Total Pixels:** Specifies the total number of pixels to process, typically calculated as width $\times$ height.

- **Custom Sobel Kernels:** Optionally allows up to four user-defined 3×3 convolution kernels to be loaded, providing flexibility beyond the default Sobel operator.
- **Start Signal:** A control signal indicating when the Sobel processing should begin.

The Regfile is accessed via the APB interface, allowing the CPU to write configuration values before processing begins. Once the required parameters are set, the Start Signal is issued, triggering the controller to transition from the idle state to the Line Buffer Loading Mode.

This block provides the necessary configuration parameters directly to the core processing blocks, ensuring proper synchronization between the image dimensions, thresholding logic, and convolution operations. By passing the Start Signal, it enables the accelerator to initiate pixel streaming and edge detection in a controlled manner.

Figure 8 illustrates the detailed interface of the Regfile.

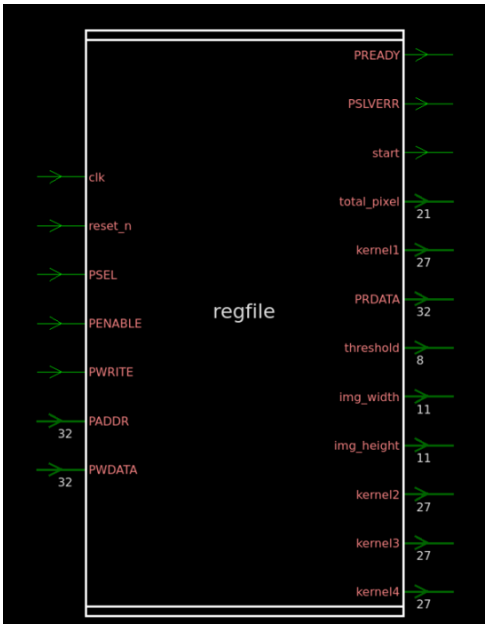


Figure 8: Regfile I/O

The following Table 4 summarizes the input and output signals used in the Regfile:

Signal Name	Direction	Width	Description
clk	Input	1	Clock signal to synchronize register operations.
reset_n	Input	1	Active-low reset signal to initialize the registers to default values.
psel	Input	1	APB peripheral select signal, used to indicate the Regfile is selected for an APB transaction.
penable	Input	1	APB enable signal, used to indicate the data phase of an APB transaction.
pwrite	Input	1	APB write signal, indicating whether the transaction is a write (1) or read (0).
paddr	Input	32	APB address bus, specifies which register is being accessed (e.g., control, threshold, image width).
pwdata	Input	32	APB write data bus, used to write data into the selected register.
prdata	Output	32	APB read data bus, used to read data from the selected register.
pready	Output	1	Indicates the Regfile is ready to complete the transaction.
pslverr	Output	1	Indicates an error in the transaction (e.g., invalid address).
start	Output	1	Signal to indicate that all configuration registers are set and valid data can be processed.
threshold	Output	8	8-bit threshold value for edge detection.
image_width	Output	10	Specifies the width of the input image in pixels.
image_height	Output	10	Specifies the height of the input image in pixels.
total_pixels	Output	20	Specifies the total number of pixels (width × height), used to control processing completion.
sobel_kernel[3:0]	Output	27 × 4	Optional user-defined 3×3 Sobel kernels (signed 3-bit values), allowing up to 4 custom kernels.

Table 4: Regfile Input and Output Signals

2.7.2 Line Buffer

The **Line Buffer**, is responsible for managing incoming pixel data and generating the 3×3 sliding window required for Sobel convolution. It receives a new pixel from the DMA every clock cycle and continuously updates its stored rows to maintain a receptive field that moves across the image.

To ensure low-latency processing while optimizing hardware resource utilization, the Line Buffer is implemented using dual-port SRAM instead of registers. This choice significantly reduces area consumption, as SRAM provides a much more space-efficient storage solution compared to flip-flop-based registers, especially when dealing with large images. The architecture is designed such that:

- **Writes are synchronized** with the system clock, ensuring stable memory updates.
- **Reads are synchronous**, with the system clock, ensuring stable memory updates.
- A **shift-based approach** minimizes memory accesses, ensuring that the correct 3×3 window is available in a single cycle.

The system is designed to process 1920×1080.

Zero Padding Strategy To properly handle edge pixels, the system logically determines when to insert zero values rather than explicitly storing a padded image. This approach optimizes memory usage by only storing valid pixel data while dynamically generating padding as needed.

For a 1920×1080 image, the system stores the 1920×1080 data in SRAM and inserts zero values dynamically when required. This reduces memory usage while still ensuring that convolution can be applied uniformly across the image.

Efficient Row Transition Handling At the end of each row, there is no need to read new data from memory for the output, as zero-padding is applied. However, to ensure that the convolution window remains valid and outputs correctly in a single cycle at the start of the next row, the system implements the following mechanism:

- The next row's pixels are **preloaded in advance** when reaching the end of the current row and are temporarily stored in a `tmp_reg` register.
- This mechanism allows the **output to remain continuous**, avoiding stalls when transitioning between rows.

By leveraging this preloading approach, the system eliminates unnecessary memory fetch delays, maintaining a consistent processing rate across the image.

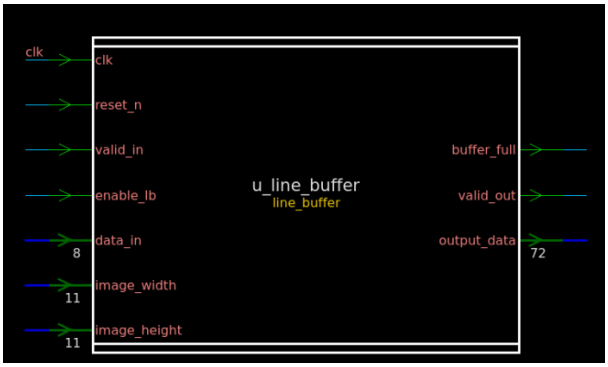


Figure 9: Line Buffer I/O

The following Table 5 summarizes the input and output signals used in the Line Buffer:

Signal Name	Direction	Width	Description
`clk`	Input	1	Clock signal to synchronize line buffer operations.
`reset_n`	Input	1	Active-low reset signal to initialize the buffer.
`enable_line_buffer`	Input	1	Signal from the Regfile, indicating that all configuration registers are set and valid data can be processed.
`image_width`	Input	11	Image width parameter received from the Regfile, specifying the number of pixels in a row of the image.
`pixel_in`	Input	8	8-bit input pixel data received from the DMA interface.
`valid_in`	Input	1	Indicates that the pixel data on `data_in` is valid.
`data_out`	Output	72	3x3 sliding window of pixel data (8 bits per pixel) sent to the Convolution Calculator.
`valid_out`	Output	1	Indicates that the sliding window data on `data_out` is valid and ready for processing.
`line_buffer_full`	Output	1	Asserted when the line buffer has received `image_width` + 3 pixels, indicating that it is fully initialized and the system can transition to the Processing State.

Table 5: Line Buffer Input and Output Signals

In the Sobel Edge Detection Accelerator, boundary pixels require special handling to ensure that convolution operations remain valid at the edges of the image. This is achieved through padding, where

extra pixels (e.g., zeros) are inserted dynamically at the boundaries of the input image. A column counter and row counter track the pixel position, while comparator units (`CMP`) determine whether the current pixel lies at the edges. If the pixel is at the start or end of a row or column, zero-padding is applied by inserting a zero value before the valid pixel data. This ensures that the output remains aligned and convolution operations produce accurate results even at the image boundaries.

Figure 10 illustrates the logic for detecting boundary conditions and applying zero-padding. When a boundary is detected, a multiplexer dynamically inserts a zero value, maintaining the integrity of the output and enabling valid data flow for subsequent processing.

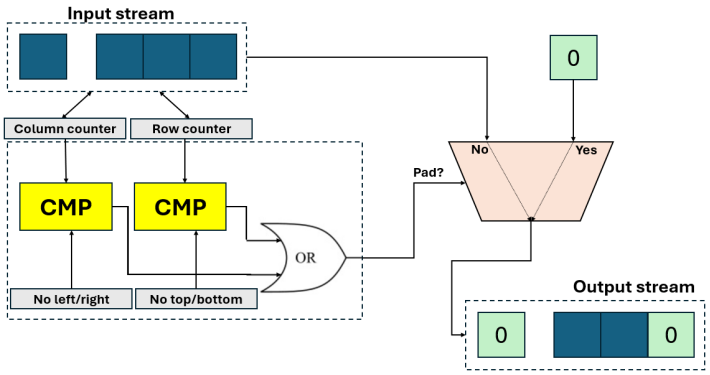


Figure 10: Padding Logic for Line Buffer Operation

Figure 11 provides a visual representation of the Line Buffer's operation. The orange dashed area represents the three rows of data stored in the Line Buffer, while the green area corresponds to the 3x3 sliding window, or receptive field, currently being processed. As the Line Buffer shifts horizontally across the image, new pixel data is streamed in, and the sliding window is updated dynamically. The blue square highlights the central pixel of the sliding window, which corresponds to the output pixel being processed at any given moment.

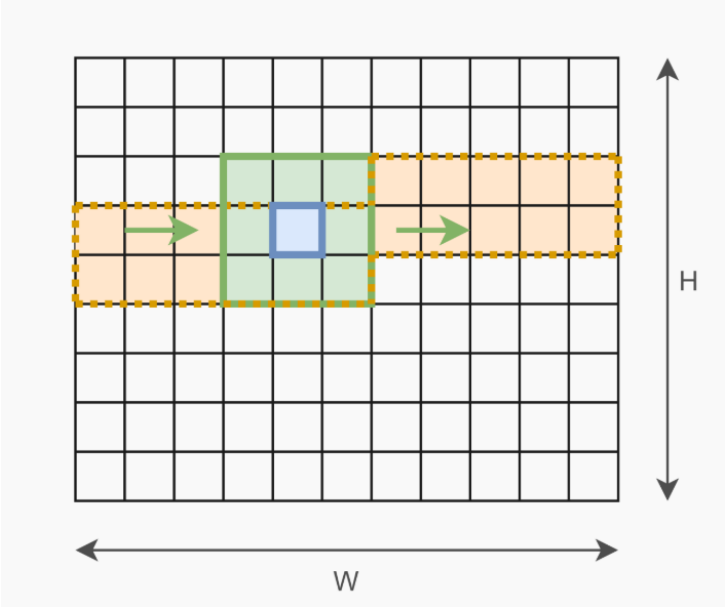


Figure 11: Line Buffer Operation: Generating the 3x3 Sliding Window

2.7.3 Convolution Calculator

The **Convolution Calculator** computes gradient magnitudes in multiple directions using Sobel kernels, including the traditional X (horizontal - 0°) and Y (vertical - 90°) directions, as well as 45° and 135°

directions. It receives the 3×3 sliding window from the Line Buffer and applies the Sobel filter for each of these directions to calculate gradients.

To achieve a more robust edge detection, the accelerator utilizes **four convolution kernels**, which allow for detecting both horizontal, vertical, and diagonal edges with higher accuracy. By incorporating diagonal gradient computations (45 and 135), the system can identify edges that might be missed by the traditional X and Y kernels alone.

Additionally, the Convolution Calculator computes the combined gradient magnitude using the formula:

$$|G| = |G_x| + |G_y| + |G_{45}| + |G_{135}|$$

This provides a more comprehensive representation of edge intensity by aggregating gradient information from multiple orientations.

The resulting gradient magnitudes are then forwarded to the Threshold Calculator for further processing. This multi-directional convolution approach enhances edge detection by capturing diagonal edges more effectively, reducing the likelihood of missing fine structural details in the image.

Optionally, the four default Sobel kernels can be replaced with user-defined 3×3 kernels, loaded via configuration registers through the APB interface. This allows the CPU to tailor the edge detection behavior for specific applications, such as detecting texture boundaries, emphasizing particular angles, or adapting to domain-specific image characteristics.

Figure 12 illustrates the detailed interface of the Conv Calc.

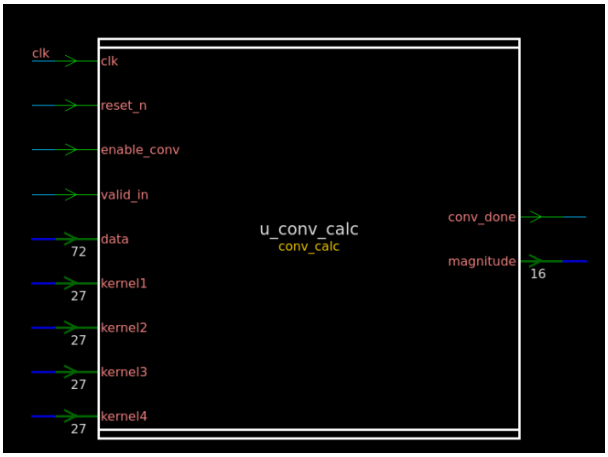


Figure 12: Convolution Calculator I/O.

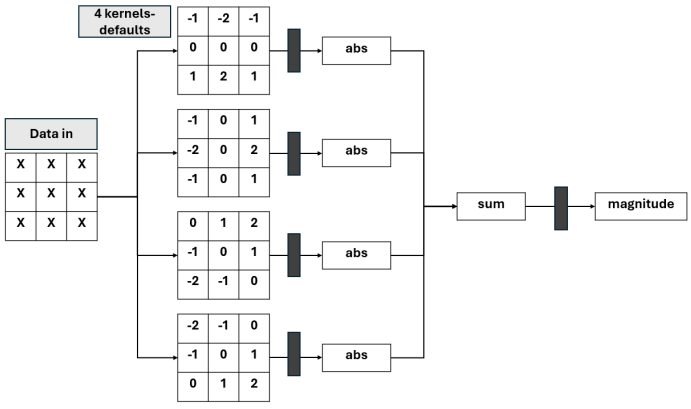


Figure 13: Internal Architecture of the Convolution Calculator, demonstrating parallel processing of Sobel kernels (0°, 90°, 45°, 135°) and gradient magnitude computation using absolute value and addition blocks.

The input and output signals of this block are summarized in Table 6

Signal Name	Direction	Width	Description
clk	Input	1	Clock signal to synchronize convolution operations.
reset_n	Input	1	Active-low reset signal to initialize the block to its default state.
valid_in	Input	1	Indicates that the input pixel data is valid and ready for processing.
data_in	Input	72	3×3 sliding window of pixel data (8 bits per pixel) received from the Line Buffer.
sobel_kernel0	Input	27	First user-defined 3×3 Sobel kernel (signed 3-bit values per cell, packed row-wise).
sobel_kernel1	Input	27	Second user-defined 3×3 Sobel kernel.
sobel_kernel2	Input	27	Third user-defined 3×3 Sobel kernel.
sobel_kernel3	Input	27	Fourth user-defined 3×3 Sobel kernel.
magnitude	Output	16	16-bit computed gradient magnitude combining gradients from the X and Y directions.
valid_out	Output	1	Indicates that the computed gradient magnitude is valid and ready for further processing.

Table 6: Convolution Calculator Input and Output Signals

To achieve real-time performance, the **Convolution Calculator** employs multiple **parallel convolution units**, each dedicated to processing one of the Sobel kernels (0, 90, 45, or 135) simultaneously. This parallel architecture enables the computation of gradient magnitudes in all directions concurrently, ensuring high throughput and minimizing latency. Additionally, the **absolute value** of each convolution result is computed in parallel, avoiding sequential dependencies that could introduce processing delays.

Figure 13 illustrates the internal design of the Convolution Calculator. Each Sobel kernel is implemented as an independent unit, allowing simultaneous gradient computations. The outputs of these kernels are passed through absolute value blocks and summed together using adders to compute the combined gradient magnitude. This design leverages the pipelined nature of the accelerator, ensuring efficient edge detection and real-time processing.

2.7.4 Threshold Calculator

The **Threshold Calculator** processes the gradient magnitudes received from the Convolution Calculator to generate the final edge-detected pixel data. This block serves as the final stage in the edge detection pipeline, ensuring that only significant edges are highlighted in the output.

The key operation within this block involves comparing the computed gradient magnitude of each pixel against a user-defined threshold value, which is configurable through the APB interface. If the magnitude is greater than or equal to the threshold, the pixel is classified as part of an edge and is passed to the output with the following adjustment:

- If the computed magnitude is ≥ 255, the output pixel value is clamped to 255.
- Otherwise, the output is the 8 least significant bits (LSBs) of the magnitude.

If the magnitude is less than the threshold, the pixel is considered a non-edge and is set to zero.

This ensures that only significant edges are preserved while weak gradients are suppressed. The clamping mechanism prevents overflow and ensures the output remains within the valid 8-bit range. By applying thresholding and scaling selectively, this method enhances edge detection accuracy while retaining gradient intensity for stronger edges.

Additionally, the Threshold Calculator maintains an **output pixel counter** that increments for each valid output pixel when valid_out is asserted. Once the number of output pixels reaches **image width**× **image height**, indicating that the entire image has been processed, the block asserts a "threshold done" signal. This signal is sent to the controller, marking the completion of the edge detection process and

allowing the system to transition to the next state. Figure 14 illustrates the detailed interface of the Threshold Calc.

Figure 14: Threshold Calculator I/O

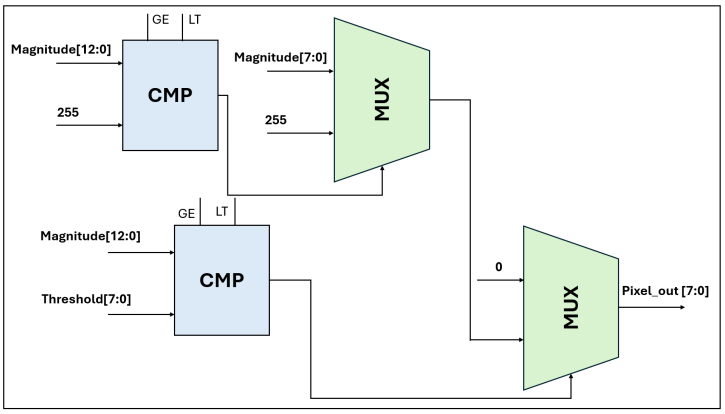


Figure 15: Internal Logic of the Threshold Calculator Block.

The following table (Table 7) summarizes the input and output signals for the Threshold Calculator block, detailing how the block interacts with the rest of the system.

Signal Name	Direction	Width	Description
`clk`	Input	1	Clock signal to synchronize thresholding operations.
`reset_n`	Input	1	Active-low reset signal to initialize the block to its default state.
`valid_in`	Input	1	Indicates that the input gradient magnitude is valid and ready for processing.
`magnitude`	Input	16	16-bit gradient magnitude calculated by the Convolution Calculator.
`threshold`	Input	8	8-bit user-defined threshold value to determine edge detection.
`data_out`	Output	8	8-bit scaled pixel data, clamped to 255 if necessary.
`valid_out`	Output	1	Indicates that the data on `data_out` is valid and ready for storage.
`thr_done`	Output	1	Asserted when thresholding is completed for all required pixels, signaling readiness to transition to the next state.

Table 7: Threshold Calculator Signal Descriptions

Figure 15 illustrates the internal logic of the Threshold Calculator. The block primarily consists of

two **Comparator Blocks** and two **Multiplexers (MUXes)**, ensuring that the gradient magnitude is both thresholded and limited to an 8-bit output.

- **First Comparator (Threshold Comparison):** This comparator determines whether the pixel should be classified as an edge by comparing the **13-bit gradient magnitude** to the **8-bit threshold value** received from the Regfile.
  - If the magnitude is **greater than or equal to the threshold** (`GE`), the pixel is considered an edge candidate.
  - If the magnitude is **less than the threshold** (`LT`), the pixel is suppressed to zero.
- **Second Comparator (Clamping to 8-bit Range):** Since the gradient magnitude is 13 bits but the output must be 8 bits, this comparator ensures that the output value does not exceed 255.
  - If the magnitude is **greater than 255** (`GE`), the value is clamped to 255.
  - Otherwise, the **8 least significant bits (LSBs)** of the magnitude are used.
- **First Multiplexer (Saturation Control):** This MUX ensures that the output remains within the valid 8-bit range by selecting between:
  - The 8 least significant bits (LSBs) of the magnitude when it is 255.
  - The fixed value 255 when the magnitude exceeds 255.
- **Second Multiplexer (Final Thresholding Decision):** This MUX determines the final pixel output:
  - If the pixel was classified as an edge (magnitude $\geq$ threshold), it forwards the saturated magnitude value.
  - If the magnitude was less than the threshold, the output is set to zero.

This two-stage comparison and selection process ensures that only significant edge pixels are retained, while weaker gradients are suppressed. Additionally, the saturation mechanism prevents overflow by ensuring that no pixel value exceeds 255, preserving a valid 8-bit output format.

This design ensures that the thresholding operation is performed efficiently and aligns with the pipelined architecture of the Sobel Edge Detection Accelerator.

2.8 Controller

The controller functions as a centralized Finite State Machine (FSM), orchestrating the operation of all units within the chip. It generates enable and control signals for the sub-units, ensuring seamless coordination and communication between them. By managing the data flow across the entire system, the controller ensures synchronized operation, efficient processing, and smooth transitions between different stages of the pipeline.

Signal Name	Direction	Width	Description
`clk`	Input	1	Clock signal to synchronize the controller's operation.
`reset_n`	Input	1	Active-low reset signal to initialize the controller to its default state.
`start`	Input	1	Signal to start the processing of the pipeline.
`enable_line_buffer`	Output	1	Enables the Line Buffer block for loading pixel data.
`buffer_full`	Input	1	Indicates that the Line Buffer is full and ready for processing.
`enable_conv`	Output	1	Enables the Convolution Calculator block for processing.
`threshold_done`	Input	1	Indicates that the Threshold Calculator has completed its processing.
`done`	Output	1	Signals that the entire processing pipeline is complete.

Table 8: Controller Input and Output Signals

2.8.1 Controller as a FSM

The **Controller's Finite State Machine (FSM)**, is a finite state machine implemented as a moore machine.

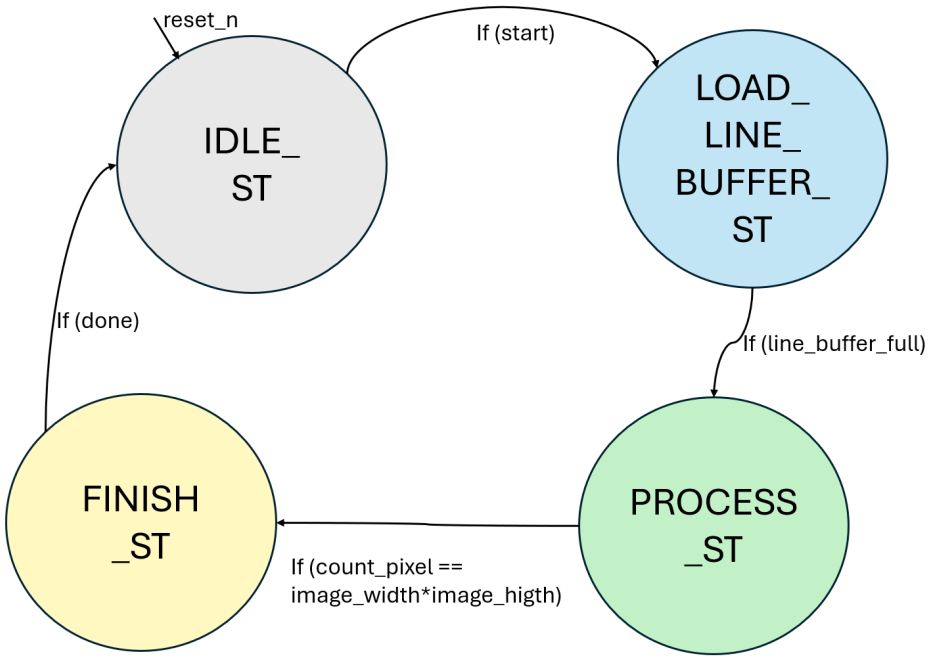


Figure 16: Controller's Finite State Machine.

- **IDLE_ST:** This is the default state of the FSM. The controller enters this state upon reset (`reset_n`) or when the processing of the image is complete. The FSM remains in the IDLE state until the `start` signal is asserted, indicating that the system is ready to begin operation.
- **LOAD_LINE_BUFFER_ST:** In this state, the controller enables the Line Buffer to start loading pixel data into its memory. The FSM remains in this state until the Line Buffer signals that it is full (`line_buffer_full`). This ensures that sufficient pixel data is available to generate the required 3x3 sliding window for the Convolution Calculator.

- **PROCESS_ST:** Once the Line Buffer is full, the FSM transitions to the PROCESS state. In this state, the controller enables both the Convolution Calculator and the Threshold Calculator to begin pixel processing. **This initiates the pipelined stage of the accelerator, where a new output pixel becomes available every clock cycle.**
- **FINISH_ST:** After all pixels have been processed, the FSM transitions to the FINISH state. Here, the controller performs any finalization tasks, such as signaling the completion of processing (`done`). Once all tasks are complete, the FSM transitions back to the IDLE state.

2.8.2 Controller's Interface with Subunits

The Controller plays a crucial role in orchestrating the operation of all subunits within the system, including the **Regfile**, **Line Buffer (LB)**, **Convolution Calculator (Conv)**, and **Threshold Calculator (Thr)**. It ensures proper coordination and synchronization by managing the flow of data and control signals.

Regfile The **Regfile** holds the configuration parameters, such as image dimensions (`image_width`, `image_height`), and the threshold value for edge detection. The Regfile passes the `start` signal to the Controller to initiate the program. Once this signal is received, the Controller manages the pipeline execution independently.

Line Buffer (LB) The **Line Buffer** temporarily stores rows of pixel data and generates the 3x3 sliding window required by the Convolution Calculator. The Controller manages the loading and readiness of the Line Buffer through dedicated control signals.

- **Input Signals to the Controller:**
  - `line_buffer_full`: Indicates that the Line Buffer has enough data to generate a 3x3 sliding window.
- **Output Signals from the Controller:**
  - `enable_line_buffer`: Enables the Line Buffer to load pixel data.

Convolution Calculator (Conv) The **Convolution Calculator** processes the 3x3 window of pixel data to compute gradient magnitudes using Sobel kernels. The Controller coordinates its operation with the Line Buffer and the Threshold Calculator.

- **Input Signals to the Controller:**
  - `conv_ready`: Indicates that the Convolution Calculator has completed processing the current 3x3 window.
- **Output Signals from the Controller:**
  - `enable_conv`: Enables the Convolution Calculator to process pixel data from the Line Buffer.

Threshold Calculator The **Threshold Calculator** applies the user-defined threshold to the gradient magnitudes generated by the Convolution Calculator, producing the final edge-detected output.

- **Input Signals to the Controller:**
  - `threshold_done`: Indicates that the Threshold Calculator has finished processing the all image.

Summary of Signal Flow Table 8 summarizes the key signals between the Controller and its subunits.

Subunit	Signal Name	Direction	Description
Regfile	`start`	Input	Indicates that the system is ready to start processing.
Line Buffer	`line_buffer_full`	Input	Indicates that the Line Buffer is ready to output a 3x3 sliding window.
Line Buffer	`enable_line_buffer`	Output	Enables the Line Buffer to load pixel data.
Convolution Calculator	`conv_ready`	Input	Indicates that the Convolution Calculator has completed processing.
Convolution Calculator	`enable_conv`	Output	Enables the Convolution Calculator to process data.
Threshold Calculator	`threshold_done`	Input	Indicates that the Threshold Calculator has completed processing.

Table 9: Controller Interface with Subunits

3 Performance

The performance of the Sobel Edge Detection Accelerator is evaluated based on its latency and throughput, which determine the efficiency of edge detection in real-time applications. By analyzing the pipeline execution, latency, and throughput, we assess the accelerator's ability to process high-resolution images with minimal delay while maintaining maximum pixel processing efficiency.

3.1 Pipeline Diagram

The pipeline diagram illustrates the sequential flow of data through the Sobel Edge Detection Accelerator (figure 17). Each stage in the pipeline is designed to operate concurrently, ensuring that a new pixel is processed every clock cycle. The architecture follows a structured data path, where pixel data flows through the **DMA Interface**, is temporarily stored in the **Line Buffer**, and is then processed by the **Convolution Block**. The **Threshold Block** determines edge classification before the final output is written back via the **Output Interface**.

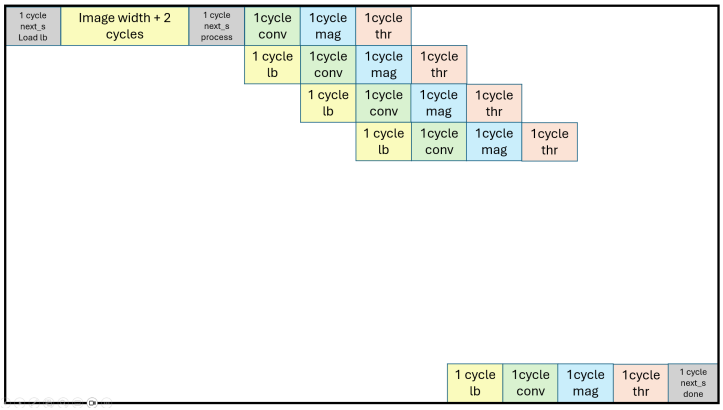


Figure 17: Pipeline execution timing diagram of the Sobel Edge Detection Accelerator, illustrating the sequential processing stages, including line buffering, convolution, gradient magnitude computation, and thresholding.

The pipeline design optimizes both latency and throughput by enabling continuous data processing. The diagram provides a visual representation of these stages, highlighting dependencies and concurrent execution.

3.2 Latency and Throughput

3.2.1 Latency Calculation

The latency of the Sobel Edge Detection Accelerator is defined as the total number of clock cycles required to process the entire image, divided by the number of pixels:

$$\text{Latency} = \frac{\text{Execution Cycles}}{\text{Number of Pixels}} = \frac{(1 + (\text{image width} + 2) + 1 + 1) + (\text{image width} \times \text{image height}) + 2 + 1 + 1}{\text{image width} \times \text{image height}}$$

Where:

- $1 + (\text{image width} + 2) + 1 + 1$: Line Buffer loading phase
- $(\text{image width} \times \text{image height})$: Pixel stream processing
- $+2$: Convolution calculation latency
- $+1$: Thresholding latency
- $+1$: Finalization overhead

Latency for 1920×1080 image For a Full HD image of size 1920×1080, the latency in clock cycles is calculated as:

$$\text{Latency} = \frac{(1 + (1920 + 2) + 1 + 1) + (1920 \times 1080) + 2 + 1 + 1}{1920 \times 1080}$$

$$\text{Latency} = \frac{(1 + 1922 + 1 + 1) + 2,073,600 + 2 + 1 + 1}{2,073,600} = \frac{2,075,529}{2,073,600} \approx 1.00093 \text{ cycles/pixel}$$

Assuming a clock period of 10 ns, the latency in time per pixel is:

$$\text{Latency}_{\text{time}} \approx 1.00093 \times 10 \text{ ns} \approx 10.0093 \text{ ns/pixel}$$

This demonstrates that the pipeline becomes highly efficient with large images, processing nearly **one pixel per cycle** on average.

3.2.2 Throughput Calculation

Throughput is defined as the number of instructions (pixels) executed divided by the total execution time:

$$\text{Throughput} = \frac{\text{Number of Pixels}}{\text{Execution Cycles} \times T_{\text{clk}}} = \frac{W \times H}{((W \times H) + W + 9) \times T_{\text{clk}}}$$

Where:

- $W \times H$ is the total number of pixels in the input image
- $(W \times H) + W + 9$ is the total number of execution cycles, derived from:
  - $W + 5$: Line buffer loading and start-up overhead
  - $W \times H$: Pixel processing
  - $+4$: Final convolution, thresholding, and done signaling cycles
- T_{clk} is the clock period (e.g., 5 ns)

Throughput for 1920×1080 image For a Full HD image of size 1920×1080, the throughput is:

$$\text{Throughput} = \frac{1920 \times 1080}{(1920 \times 1080 + 1920 + 9) \times 10 \text{ ns}} = \frac{2,073,600}{(2,073,600 + 1920 + 9) \times 10 \text{ ns}}$$

$$\text{Throughput} = \frac{2,073,600}{2,075,529 \times 10 \text{ ns}} \approx \frac{2,073,600}{20,755,290 \text{ ns}} \approx 99.93 \text{ MegaPixels/sec}$$

As the image size increases, the overhead becomes negligible, and the throughput approaches:

$$\text{Throughput} \approx \frac{1}{T_{\text{clk}}} = \frac{1}{10 \text{ ns}} = 100 \text{ MegaPixels/sec}$$

This indicates that the accelerator can sustain near-maximum throughput, processing almost one pixel per clock cycle.

3.3 Synthesis

The synthesis process was carried out using **Synopsys Design Vision**, targeting the **Tower TSL018** technology library. The design was synthesized with a **100 MHz clock constraint**, corresponding to a clock period of 10 ns. These conditions were used for evaluating area, timing, and power consumption under realistic operational scenarios.

Figure 18 illustrates the synthesized RTL schematic of the `sobel_top` module. It shows the hierarchical structure of the design, including key submodules such as the Regfile, Line Buffer, Controller, Convolution Calculator, and Threshold Calculator. The figure also highlights the interconnections between blocks, verifying correct integration and signal flow as intended in the top-level architecture.

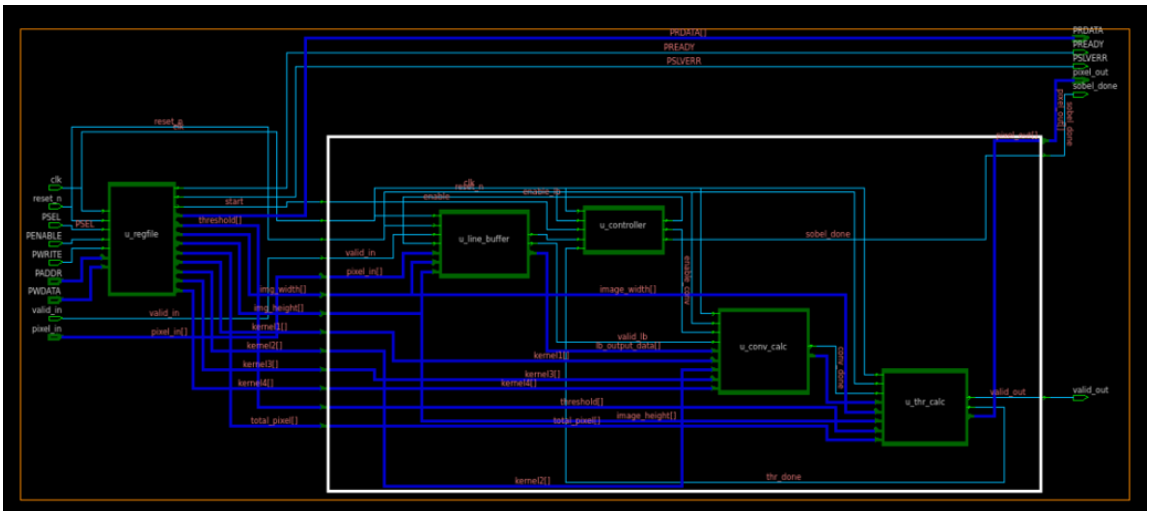


Figure 18: Gate-level schematic of the synthesized `sobel_top` module.

3.3.1 FloorPlan

The synthesized floorplan of the Sobel Edge Detection Accelerator illustrates the placement and routing of key functional blocks. The design is optimized to ensure minimal routing congestion and efficient area utilization. As shown in Figure 19, the layout includes the main computational logic along with dedicated SRAM blocks for efficient storage and processing.

A notable feature of this floorplan is the presence of three SRAM blocks, labeled as `sram_line0`, `sram_line1`, and `sram_line2`. These SRAM blocks are used to store intermediate pixel data for the Line Buffer, ensuring that a 3×3 pixel window is continuously available for convolution operations. The placement of these SRAM blocks is strategically optimized to minimize routing congestion and ensure efficient data access.

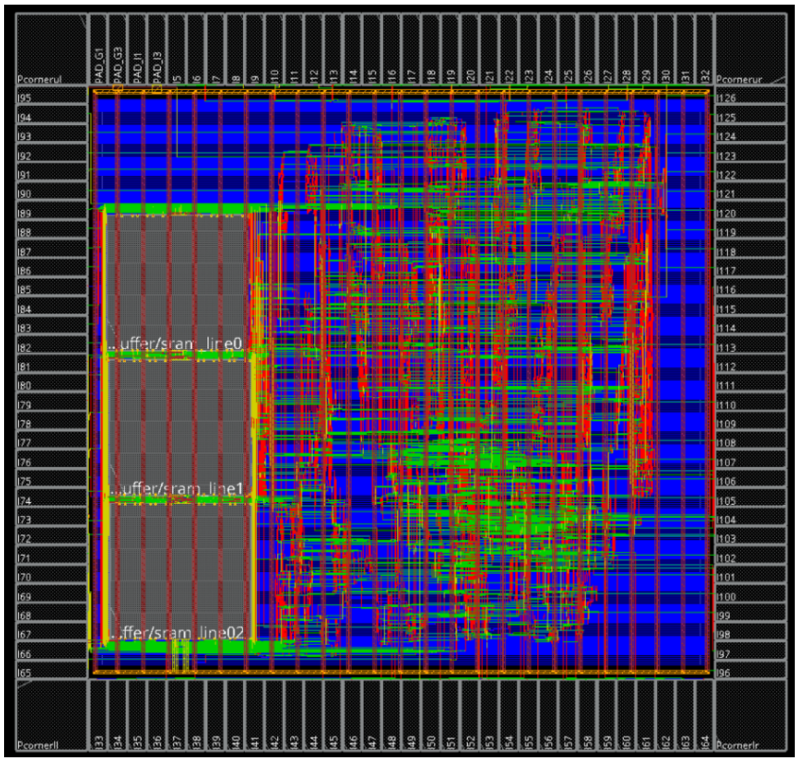


Figure 19: Floorplan of the Sobel Edge Detection Accelerator, highlighting the placement of key processing units. The three SRAM blocks (`sram_line0`, `sram_line1`, and `sram_line2`) are visible in the layout, supporting the Line Buffer for convolution processing.

3.3.2 Clock Tree Synthesis (CTS)

Clock Tree Synthesis (CTS) is a critical step in the physical design process to ensure proper clock distribution across the design. The goal of CTS is to minimize clock skew and ensure balanced clock propagation across all sequential elements. Figure 20 illustrates the synthesized clock tree structure for the Sobel Edge Detection Accelerator.

The clock tree is constructed using buffering and clock gating techniques to optimize power consumption and timing. The hierarchical structure ensures that all registers receive the clock signal with minimal skew, improving synchronization and timing closure. The yellow nodes represent clock buffers, while the red nodes indicate clock sinks (flip-flops or registers receiving the clock).

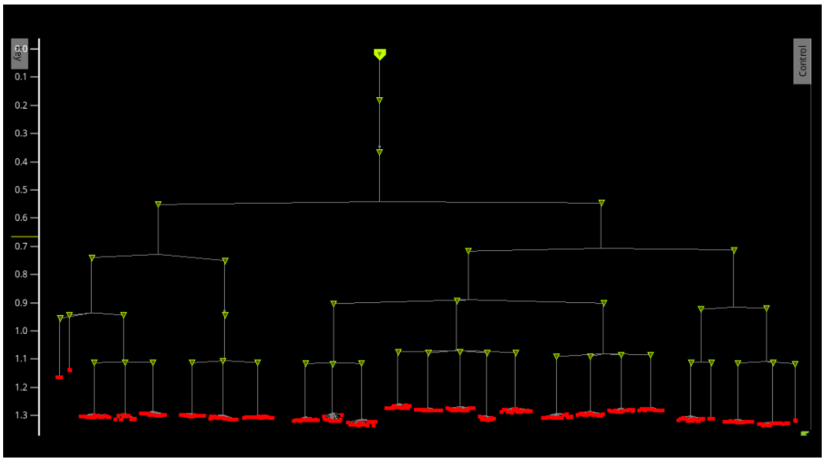


Figure 20: Clock Tree structure of the Sobel Edge Detection Accelerator, showing the distribution of the clock network to minimize skew and optimize power.

3.3.3 Area, Static Power, and Timing Analysis

The area, power, and timing analysis of the Sobel Edge Detection Accelerator were obtained from the synthesis reports. These metrics are crucial for evaluating the design's efficiency in terms of hardware resource utilization, power consumption, and timing performance.

Area Analysis The synthesized design consists of 7,643 total cells, including 6,719 combinational cells and 583 sequential cells. The floorplan contains three memory macros (SRAMs) for buffering image data, contributing to a significant portion of the total area.

The detailed area breakdown is as follows:

- **Macro/Black Box Area:** 57,513.30 μm^2 (dominated by SRAM macros)
- **Net Interconnect Area:** 11,125.87 μm^2
- **Total Cell Area:** 57,513.30 μm^2
- **Total Design Area:** 68,769.17 μm^2

The area utilization is primarily influenced by the memory components, which are necessary for efficient data storage and access in the Line Buffer. The interconnect area accounts for routing overhead, ensuring proper communication between computational units.

```
****************************************
Report : area
Design : sobel_top
Version: U-2022.12
Date   : Mon Mar 17 12:35:07 2025
****************************************

Library(s) Used:

    gtech (File: /eda/synopsys/2022-23/RHELx86/SYN_2022.12/libraries/syn/gtech.db)
    dpram2048x8_CB_typ (File: /users/iit/tower_memories/dpram2048x8_CB/dpram2048x8_CB/dpram2048x8_CB_typ.db)

Number of ports:                         7517
Number of nets:                         16379
Number of cells:                         7643
Number of combinational cells:           6719
Number of sequential cells:               583
Number of macros/black boxes:               3
Number of buf/inv:                        856
Number of references:                       2

Combinational area:                0.000000
Buf/Inv area:                      0.000000
Noncombinational area:             0.000000
Macro/Black Box area:          57513.298828
Net Interconnect area:         11255.868658

Total cell area:               57513.298828
Total area:                    68769.167486
```

Figure 21: Area report generated post-synthesis, showing the distribution of macro area, interconnect, and total cell usage.

Static Power Analysis The power report 22 provides an estimate of the dynamic and leakage power consumption. The power consumption is analyzed at a global operating voltage of 1.8V, with the following key observations:

- **Total Dynamic Power:** 45.25 mW
- **Cell Internal Power:** 45.03 mW (99.53% of total dynamic power)
- **Net Switching Power:** 0.22 mW (0.47% of total dynamic power)
- **Cell Leakage Power:** 11.05 µW

From the breakdown, it is evident that memory cells contribute the most to power consumption, particularly in internal power dissipation. The net switching power is relatively low, indicating that

the design has efficient clock gating and optimized switching activity. The cell leakage power remains minimal at 11.05 µW, ensuring low static power dissipation.

```
Global Operating Voltage = 1.8
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


Attributes
----------
i - Including register clock pin internal power


  Cell Internal Power  =  45.0285 mW  (100%)
  Net Switching Power  = 226.1996 uW    (0%)
                         ---------
Total Dynamic Power    =  45.2547 mW  (100%)

Cell Leakage Power     =  11.0511 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory            45.0285        6.8970e-03          1.1051e+07          45.0464  (  99.53%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)  i
register           0.0000        1.1763e-03             0.0000          1.1763e-03  (   0.00%)
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)
combinational      0.0000           0.2112              0.0000             0.2112  (   0.47%)
--------------------------------------------------------------------------------------------------
Total             45.0285 mW        0.2193 mW      1.1051e+07 pW         45.2588 mW
design_vision>
```

Figure 22: Post-synthesis power report showing dynamic and leakage power contributions, with memory accounting for most of the internal power.

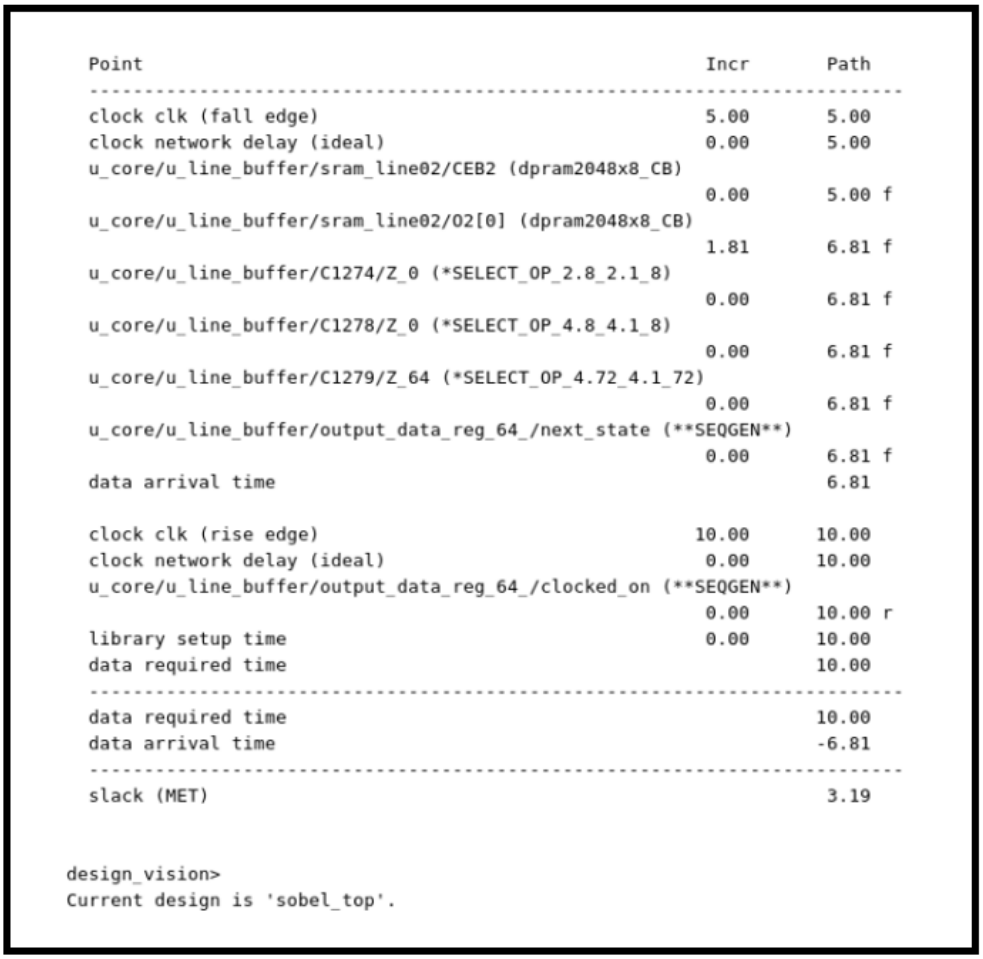
Timing Analysis The timing report 23 provides insights into the longest critical path in the design, which determines the maximum clock frequency the system can operate at.

From the timing report:

- **Startpoint:** `u_core/u_line_buffer/sram_line02` (falling edge-triggered flip-flop clocked by clk).
- **Endpoint:** `u_core/u_line_buffer/output_data_reg_64` (rising edge-triggered flip-flop clocked by clk).
- **Clock Delay:** 5.00 ns (fall edge) and 10.00 ns (rise edge).
- **Data Arrival Time:** 6.81 ns.
- **Data Required Time:** 10.00 ns.
- **Slack (MET):** 3.19 ns.

The longest critical path originates from the SRAM block (`sram_line02`) and ends at the output register (`output_data_reg_64`). This path involves reading data from memory, performing computations, and storing the results in the output register.

Since the slack is 3.19 ns, the design meets the timing constraints, meaning the critical path delay does not exceed the required clock cycle time. This ensures that the design operates reliably at the intended clock frequency.



```
Point                                                    Incr       Path
--------------------------------------------------------------------------
clock clk (fall edge)                                    5.00       5.00
clock network delay (ideal)                              0.00       5.00
u_core/u_line_buffer/sram_line02/CEB2 (dpram2048x8_CB)
                                                         0.00       5.00 f
u_core/u_line_buffer/sram_line02/O2[0] (dpram2048x8_CB)
                                                         1.81       6.81 f
u_core/u_line_buffer/C1274/Z_0 (*SELECT_OP_2.8_2.1_8)
                                                         0.00       6.81 f
u_core/u_line_buffer/C1278/Z_0 (*SELECT_OP_4.8_4.1_8)
                                                         0.00       6.81 f
u_core/u_line_buffer/C1279/Z_64 (*SELECT_OP_4.72_4.1_72)
                                                         0.00       6.81 f
u_core/u_line_buffer/output_data_reg_64_/next_state (**SEQGEN**)
                                                         0.00       6.81 f
data arrival time                                                   6.81

clock clk (rise edge)                                   10.00      10.00
clock network delay (ideal)                              0.00      10.00
u_core/u_line_buffer/output_data_reg_64_/clocked_on (**SEQGEN**)
                                                         0.00      10.00 r
library setup time                                       0.00      10.00
data required time                                                 10.00
--------------------------------------------------------------------------
data required time                                                 10.00
data arrival time                                                  -6.81
--------------------------------------------------------------------------
slack (MET)                                                         3.19


design_vision>
Current design is 'sobel_top'.
```

Figure 23: Timing report of the longest path between SRAM and output register, showing a positive slack of 3.19 ns.

Conclusion The area, power, and timing reports indicate that the Sobel Edge Detection Accelerator is well-optimized for hardware implementation. The SRAM macros contribute significantly to both area and power, but they are necessary for handling high-throughput image processing. The design maintains low leakage power, meets the timing constraints with a positive slack of 3.19 ns, and ensures stable operation within the defined clock cycle.

4 Zero-order ("aliveness") Verification

This section focuses on verifying that the Sobel Edge Detection Accelerator is functionally alive and responsive to basic stimuli. The goal is to confirm that the design correctly receives configuration commands, processes input pixels, and produces valid output data under controlled test conditions.

4.1 Block Diagram

Figure 24 shows the high-level block diagram of the testbench environment used for zero-order (aliveness) verification of the Sobel Edge Detection Accelerator. The goal of this verification stage is to confirm that the design is correctly instantiated, responds to basic stimuli, and is functionally "alive" before proceeding to deeper functional testing.

The testbench includes a **Clock/Reset Generator**, which drives the synchronous components of the system, including the DUT, CPU, and DMA Model. The **CPU** block emulates a host processor and communicates with the DUT via the APB protocol, writing configuration values such as image dimensions, threshold level, and kernel weights.

A **DMA Model** simulates the pixel data stream, feeding grayscale image pixels into the DUT. The **DUT** processes the incoming data and generates the edge-detected output. A **Monitor** captures the DUT's output to verify that it reacts as expected, ensuring that the data path is active and responsive.

This setup confirms proper connectivity between the blocks, clock/reset propagation, and responsiveness of control and data paths.

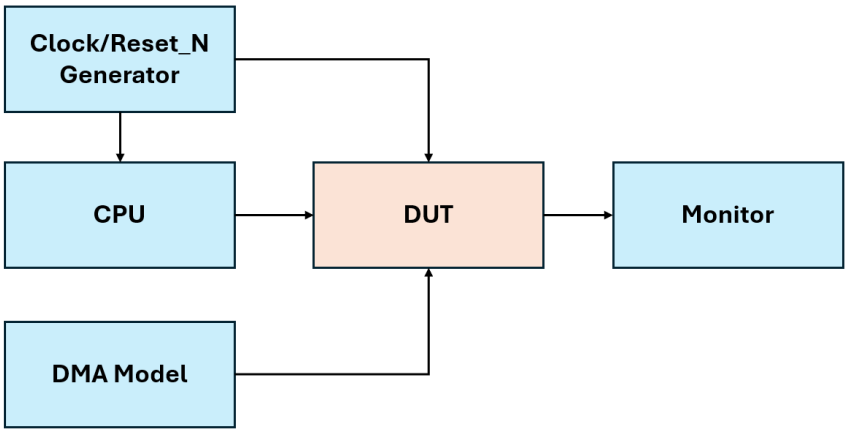


Figure 24: Block diagram of the testbench setup for zero-order (aliveness) verification of the Sobel Edge Detection Accelerator.

4.2 Tests Description

The zero-order verification tests aim to validate the basic functionality of the Sobel Edge Detection Accelerator by ensuring that it correctly processes image data and produces the expected edge-detected output.

A Python script is used to convert a grayscale test image into a stream of pixel intensity values. These pixel values are written to a file named `pixel.txt`, which serves as the input to the testbench. In addition to the pixel stream, the Python script also provides essential configuration parameters including: Image width, Image height, Total number of pixels, Threshold value.

The testbench reads the `pixel.txt` file and applies the pixel stream to the DUT along with the configuration values via the APB interface. The DUT performs Sobel edge detection and generates the corresponding output stream, which is captured and written to `edge.txt`.

Finally, the `edge.txt` file is processed by another Python script to reconstruct the output image, allowing visual comparison with the expected edge-detected result. This setup verifies the correct behavior of the accelerator by validating the end-to-end data path and control configuration.

4.3 Tests Result

To validate the functionality of the Sobel Edge Detection Accelerator, a zero-order verification test was performed using a known grayscale image as input. The DUT was configured using parameters written via the APB interface, including image width, height, threshold, and kernel coefficients. The pixel data was streamed in using the DMA model, and the resulting edge-detected image was saved to an output file for comparison.

Visual Comparison Figure 25 shows the visual output of the hardware-based Sobel accelerator alongside the software-based Python implementation. The image on the left is the original grayscale input, while the center and right images represent the hardware and Python edge detection outputs, respectively. The hardware implementation closely matches the Python-generated reference, demonstrating functional correctness.

Figure 25: Visual comparison between original image, hardware output, and software (Python) Sobel output.

Waveform Analysis The waveform results provide additional evidence of correct operation. Figure 26 shows the initialization of control registers using the APB interface, where threshold, image dimensions, total pixels, and kernel values are sequentially written. The 'start' signal is asserted to begin processing, and pixel values are streamed in through the `pixel_in` and `valid_in` signals.

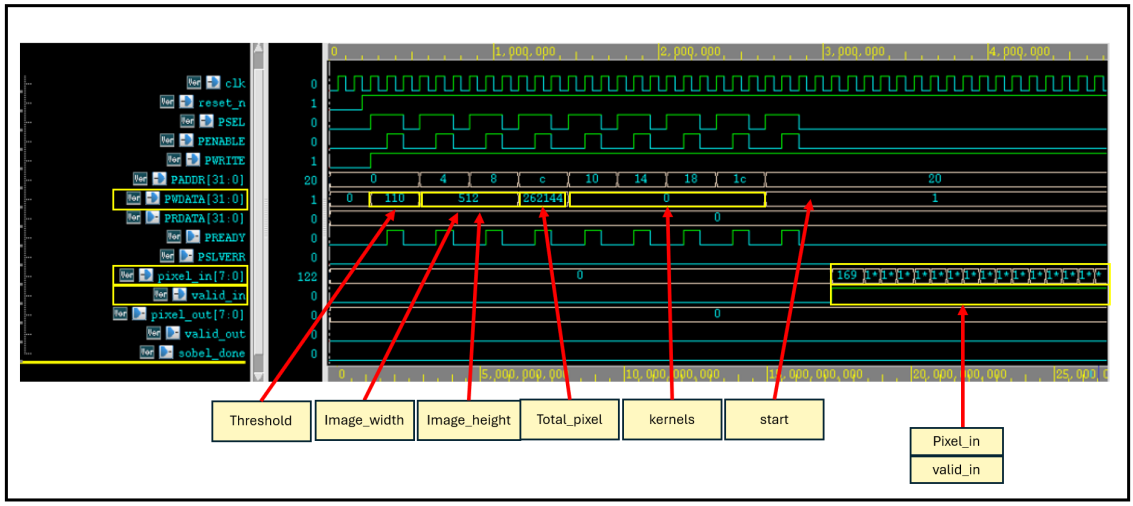


Figure 26: Register initialization and pixel input sequence observed through APB interface and DMA model.

Figure 27 confirms correct output behavior. The DUT begins outputting processed pixels on `pixel_out` along with the `valid_out` signal once processing begins. The data is streamed out continuously until completion.

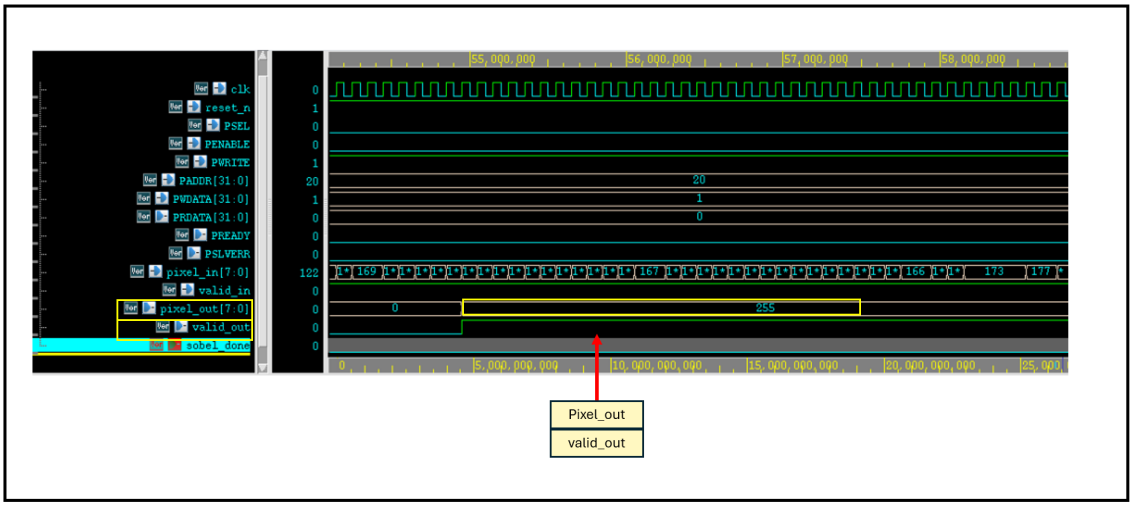


Figure 27: Output pixel stream and validation signal during convolution and threshold stages.

Finally, Figure 28 shows the assertion of the `sobel_done` signal, indicating the end of processing once all pixels have been streamed through and processed by the pipeline.

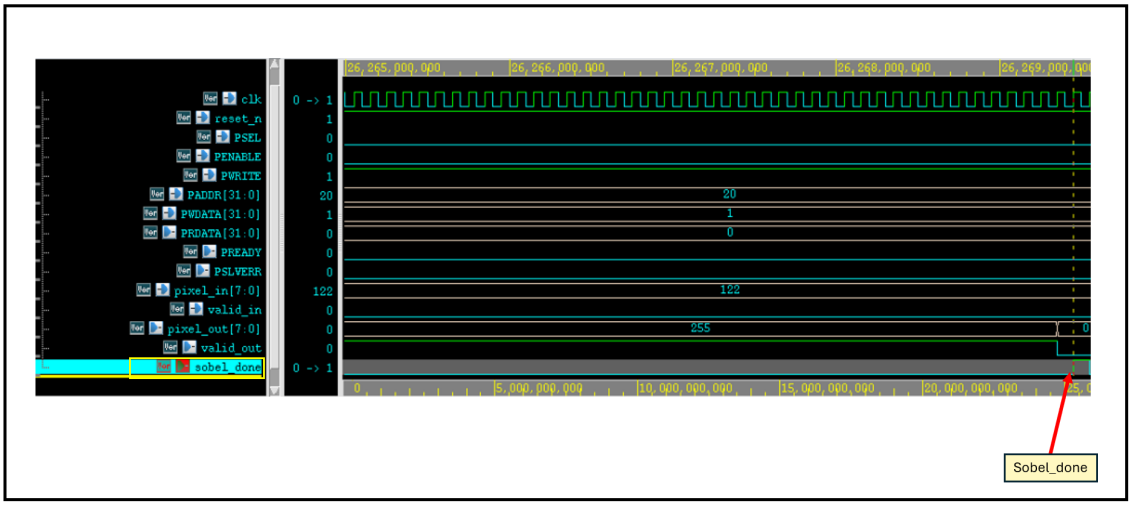


Figure 28: End-of-processing signal (`sobel_done`) indicating successful completion of the image.

Additional Result and Waveforms An additional test was performed to validate the system with a different input image and configuration. The following waveform figures show the register setup, pixel streaming, and output activity. The corresponding image result is also shown for visual confirmation.

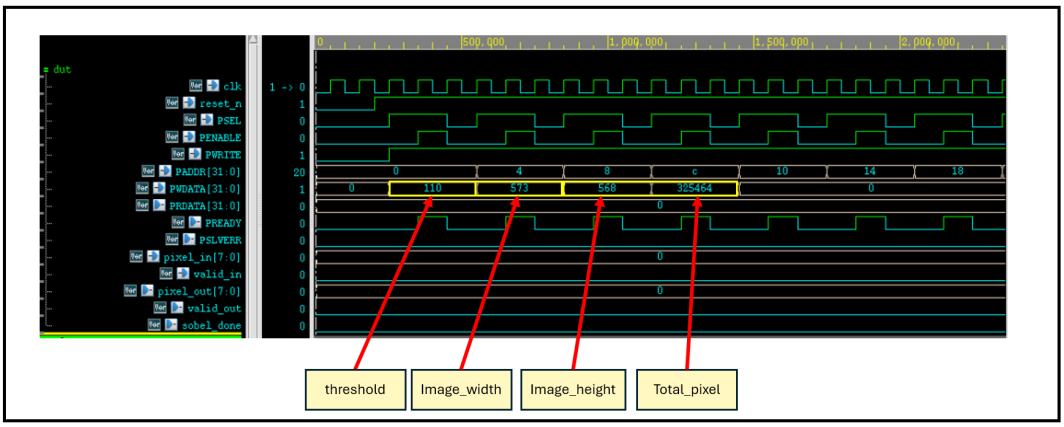


Figure 29: APB configuration waveforms for second test scenario.

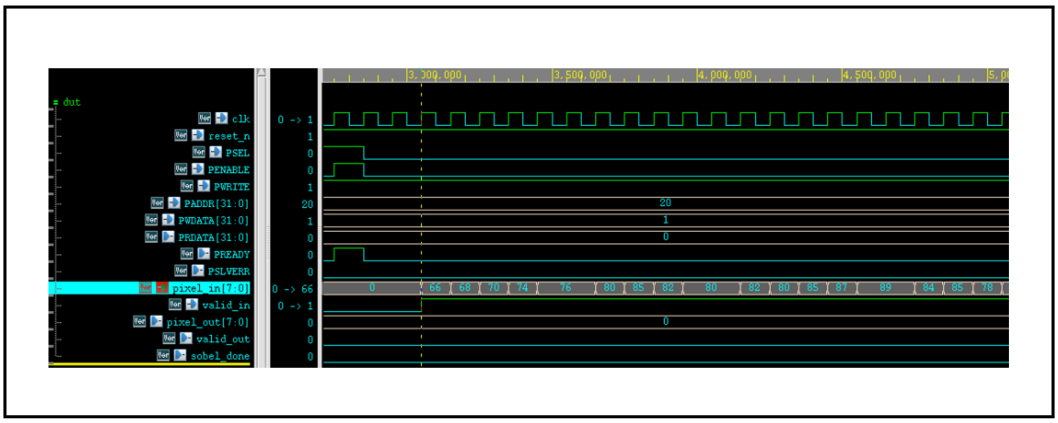


Figure 30: Pixel input waveforms during processing.

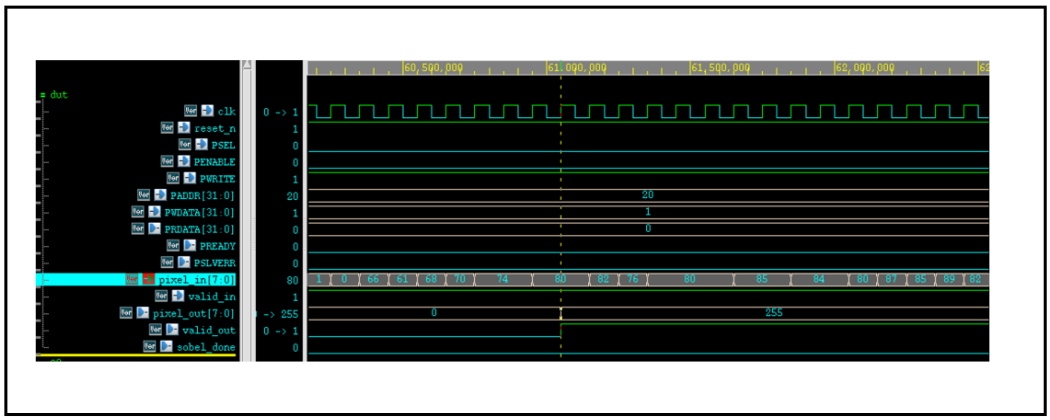


Figure 31: Pixel input and output waveforms during processing.

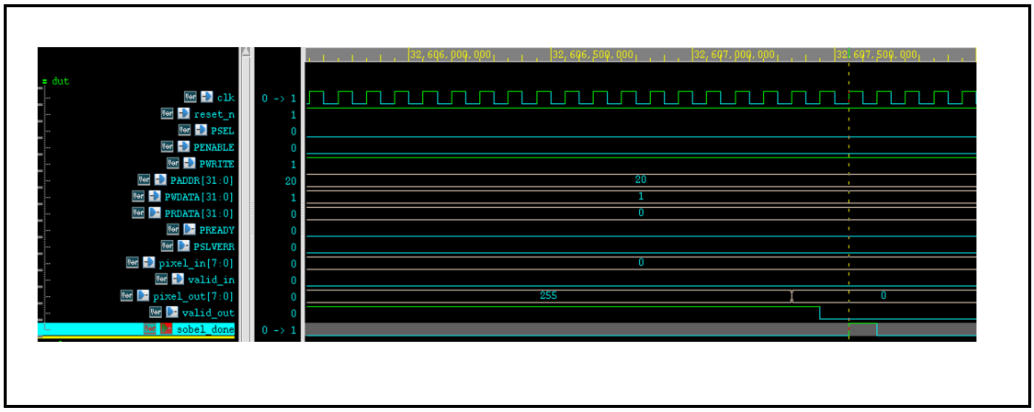


Figure 32: Completion signal (`sobel_done`) asserting after all pixels are processed.

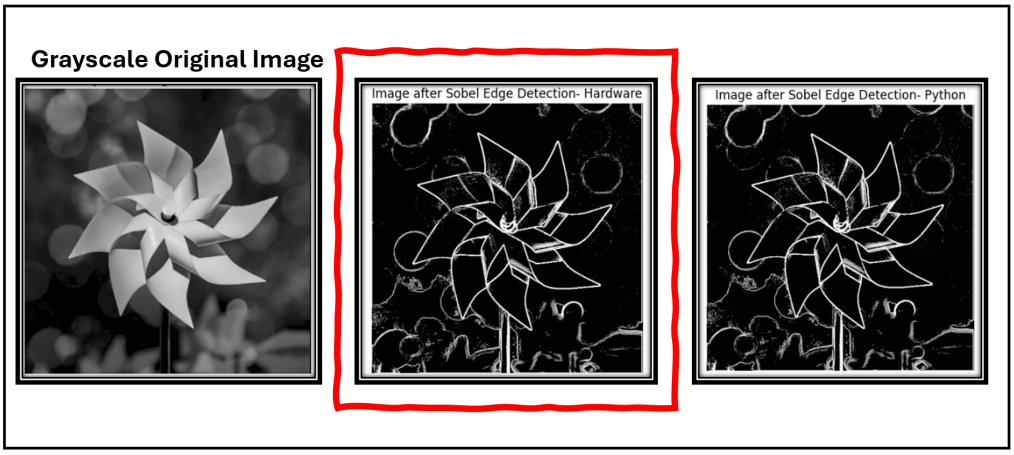


Figure 33: Edge-detected output for second test case.

In addition to the visual and waveform results, we also validated the execution timing by comparing the theoretical cycle count with the actual hardware behavior, as shown below:

Cycle Count Validation The waveform in Figure 34 confirms that the number of cycles measured in hardware matches the expected theoretical calculation. A 4×4 image was used as input, and the cycles were counted from the moment the `start` signal was asserted until the `sobel_done` signal was set high.

The number of cycles is given by the formula:

$$\text{Cycles} = \underbrace{(1 + (\text{image width} + 2) + 1 + 1)}_{\text{Load Line Buffer}} + \underbrace{(\text{image width} \times \text{image height})}_{\text{Streaming Pixels}} + \underbrace{2}_{\text{Conv}} + \underbrace{1}_{\text{Threshold}} + \underbrace{1}_{\text{Finish}}$$

For a 4×4 image:

$$\text{Cycles} = 1 + (4 + 2) + 1 + 1 + (4 \times 4) + 2 + 1 + 1 = \mathbf{29\ cycles}$$

This confirms that the pipelined architecture operates correctly, achieving the expected number of clock cycles under test conditions.

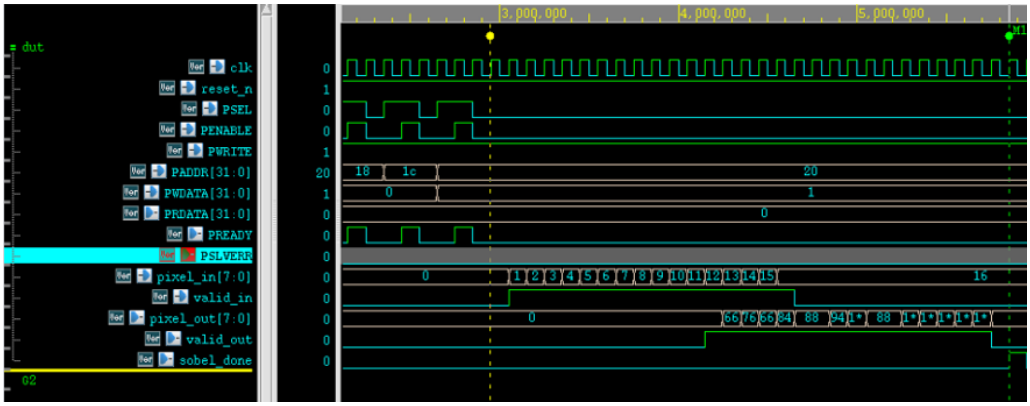


Figure 34: Waveform showing cycle count from `start` to `sobel_done`, confirming theoretical latency of 29 cycles for a 4×4 image.

Conclusion The matching output images and consistent waveform activity confirm that the accelerator is operational and performs Sobel edge detection correctly. The design successfully transitions through configuration, data loading, processing, and result output, as verified both visually and through simulation waveforms.

5 Programmer's Guide

5.1 Registers Table

Table 10 lists all the memory-mapped registers used to configure and control the Sobel Edge Detection Accelerator through the APB interface. Each register is accessed using the standard APB protocol and allows the user to set image parameters, provide custom Sobel kernels, and initiate processing.

Offset	Name	Width	Access	Description
0x00	THRESHOLD	8	R/W	Threshold value used to classify whether a pixel is part of an edge (range: 0–255).
0x04	IMG_WIDTH	16	R/W	Width of the input image in pixels.
0x08	IMG_HEIGHT	16	R/W	Height of the input image in pixels.
0x0C	TOTAL_PIXELS	32	R/W	Total number of pixels in the image, usually IMG_WIDTH × IMG_HEIGHT. Used to determine when to stop processing.
0x10	SOBEL_KERNEL0	27	R/W	First user-defined 3×3 Sobel kernel, packed row-wise (signed 3-bit values).
0x14	SOBEL_KERNEL1	27	R/W	Second user-defined 3×3 Sobel kernel.
0x18	SOBEL_KERNEL2	27	R/W	Third user-defined 3×3 Sobel kernel.
0x1C	SOBEL_KERNEL3	27	R/W	Fourth user-defined 3×3 Sobel kernel.
0x20	START	1	WO	Write '1' to trigger the start of image processing. Must be asserted after all configuration is complete.
0x24	DONE	1	RO	Indicates the end of processing. Becomes '1' when the entire image has been processed.

Table 10: Memory-mapped registers used to configure and control the Sobel Edge Detection Accelerator.

Figure 35 illustrates the memory-mapped register layout of the Sobel Edge Detection Accelerator. This diagram provides a visual summary of the register offsets and their function within the APB configuration space.

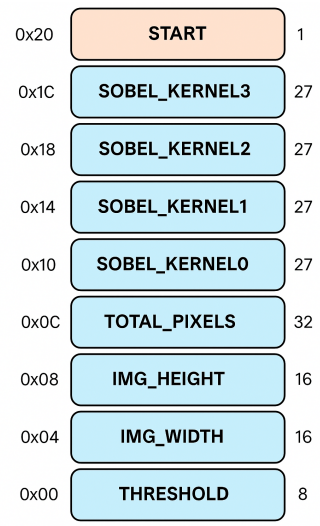


Figure 35: Memory-mapped register layout of the Sobel Edge Detection Accelerator.

5.2 Usage Guidelines

To use the Sobel Edge Detection Accelerator effectively, the user must follow a specific sequence of configuration and data streaming steps via the APB and DMA interfaces. These steps ensure proper initialization, processing, and retrieval of the edge-detected image.

1. **Reset the design.** Assert the system reset (reset_n = 0) for at least one clock cycle, then deassert it (reset_n = 1) to initialize all internal logic.
2. **Configure accelerator registers via the APB interface.** The host processor (CPU) must write to the following memory-mapped registers using the APB protocol:
   - `THRESHOLD (0x00)`: Set the pixel gradient threshold value (0–255).
   - `IMG_WIDTH (0x04)`: Specify the image width in pixels.
   - `IMG_HEIGHT (0x08)`: Specify the image height in pixels.
   - `TOTAL_PIXELS (0x0C)`: Set to IMG_WIDTH × IMG_HEIGHT.
   - `SOBEL_KERNEL`: 3 (0x10 – 0x1C): (Optional) Load custom 3×3 Sobel kernels.
3. **Start processing.** After all configuration values are written, initiate image processing by writing '1' to the `START` register (address '0x20').
4. **Stream pixel data.** Use the DMA or pixel streaming interface to send grayscale image pixels sequentially through the pixel_in and valid_in signals. Pixels should be sent in raster-scan order (row by row). The DUT accepts one pixel per clock cycle when valid_in is high.
5. **Monitor the output.** Once the pipeline starts producing edge results, the processed pixels are available on pixel_out, and the corresponding valid_out signal indicates when the data is valid. One pixel is produced per clock cycle after the initial latency.
6. **Wait for completion.** The accelerator asserts the `DONE` register (readable at 0x24) or the output signal sobel_done once all pixels have been processed.

It is important to follow this configuration and streaming order to ensure correct operation. Writing to configuration registers while processing is active is not recommended. All APB writes must follow the standard APB handshake protocol (PSEL, PENABLE, PWRITE) with proper clock synchronization.

6 Summary

6.1 Project's Summary

This project presented the design, implementation, and verification of a hardware accelerator for Sobel edge detection using SystemVerilog. The accelerator was built with a modular, pipelined architecture capable of processing one pixel per clock cycle after initialization, enabling high-throughput real-time image processing.

The system was configured via an APB interface, allowing the host CPU to control parameters such as image dimensions, threshold values, and custom Sobel kernels. A DMA interface was used to stream image data into the accelerator and retrieve the edge-detected output. Verification was carried out using a custom testbench, waveform analysis, and output comparison against a reference Python model.

In addition to RTL design and functional verification, the project included RTL synthesis, floorplanning, and generation of the initial layout using a standard-cell library. The layout demonstrates successful placement and routing of the design, including three SRAM blocks.

Synthesis and timing reports confirmed that the design meets timing requirements and operates efficiently in terms of area and power. The overall architecture supports flexibility, configurability, and integration into larger SoC systems.

6.2 Take Message Home

Edge detection is a critical step in many real-time vision systems, and hardware acceleration can significantly improve performance and energy efficiency. This project demonstrates how a custom, pipelined Sobel operator can be effectively implemented in hardware using SystemVerilog and controlled through a simple APB interface.

The main takeaway is that a well-designed hardware accelerator can achieve high throughput with low latency and minimal resource overhead, while still offering configurability and compatibility with standard bus protocols.

6.3 Next Steps

While the current implementation successfully demonstrates the design, synthesis, and verification of a Sobel Edge Detection Accelerator, several extensions and improvements can be pursued in future work to enhance functionality, performance, and integration.

- **Complete layout closure:** Finalize the physical design by addressing metal routing constraints, design rule checking (DRC), and layout-versus-schematic (LVS) verification. This step is essential for creating a manufacturable, signoff-ready GDSII layout.
- **Additional filter modes:** Add the ability to toggle between Sobel, Prewitt, and Scharr operators, or support programmable 5×5 kernels for more advanced edge detection.
- **Support for color images:** Extend the architecture to handle RGB image inputs by processing each color channel independently or using a luminance-based approach (e.g., converting RGB to grayscale internally). This would make the accelerator compatible with a wider range of vision applications.

These next steps will help transition the design from a verified RTL prototype to a fully deployable, production-grade accelerator suitable for integration in real-time embedded imaging systems.

References

[1] ARM Ltd., "Amba apb protocol specification," https://developer.arm.com/documentation/ihi0024/latest/, 2023, accessed: Mar. 27, 2025.

[2] B. Li, J. Chen, X. Zhang, X. Xu, Y. Wei, and D. Kong, "A design of zynq-based medical image edge detection accelerator," in *Proceedings of the 5th International Conference on Biological Information Processing and Biomedical Engineering (ICBIP '21)*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3484424.3484434