Danmarks
Tekniske
Universitet

DTU

# 02203 Design of Digital Systems
# Assignment 2

Authors

Myrsini Gkolemi - s233091
Christopher Mardones-Andersen - s205119
Michele Bandini - s243121

December 6, 2024

# Contents

# 1 Task 0

## Task Zero: Familiarization with the Memory Interface

In Task Zero, we designed a simple solution for the pixel inversion task to familiarize ourselves with the memory layout. Our approach involved four distinct states: `Idle`, `Read`, `Invert and Write` and `Done` which are described below.

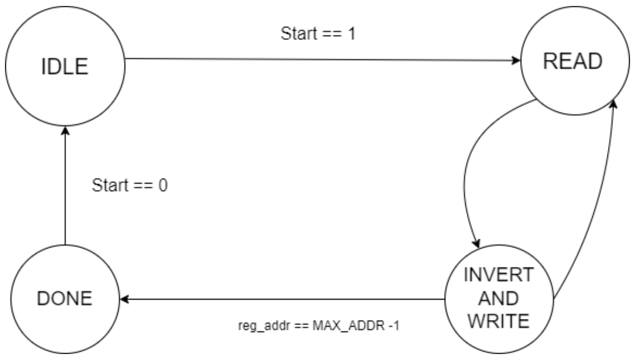


Figure 1: Finite State Machine (FSM) for Task 0

### Idle State

The `Idle` state waits for the user input signal, `start`, to transition into the next state.

### Read State

In the `Read` state, the memory at address `reg_addr` is accessed. Due to the latency of the memory interface, the data cannot be retrieved and processed within the same clock cycle. Instead, the data becomes available in the subsequent clock cycle, ready for use in the next state.

### Invert and Write State

In the `Invert and Write` state, the data fetched in the `Read` state is processed. Each byte of the pixel data is inverted using the following formula:

$$\begin{aligned}
\texttt{Inverted Byte 0} &<= 255 - \texttt{dataR}(7 \texttt{ downto } 0),\\
\texttt{Inverted Byte 1} &<= 255 - \texttt{dataR}(15 \texttt{ downto } 8),\\
\texttt{Inverted Byte 2} &<= 255 - \texttt{dataR}(23 \texttt{ downto } 16),\\
\texttt{Inverted Byte 3} &<= 255 - \texttt{dataR}(31 \texttt{ downto } 24).
\end{aligned}$$

The inverted pixel data is then written back to the memory at the address `reg_addr + MAX_ADDR`. Afterwards, the `reg_addr` is incremented by one, and the system transitions back to the `Read` state. This process repeats until the address reaches `MAX_ADDR - 1`.

## Done State

Once the address `MAX_ADDR - 1` is reached, the system transitions to the `Done` state. In this state, the `finish` flag is raised, signaling that the image processing task has been successfully completed. The complete state flow is illustrated in Figure 1.

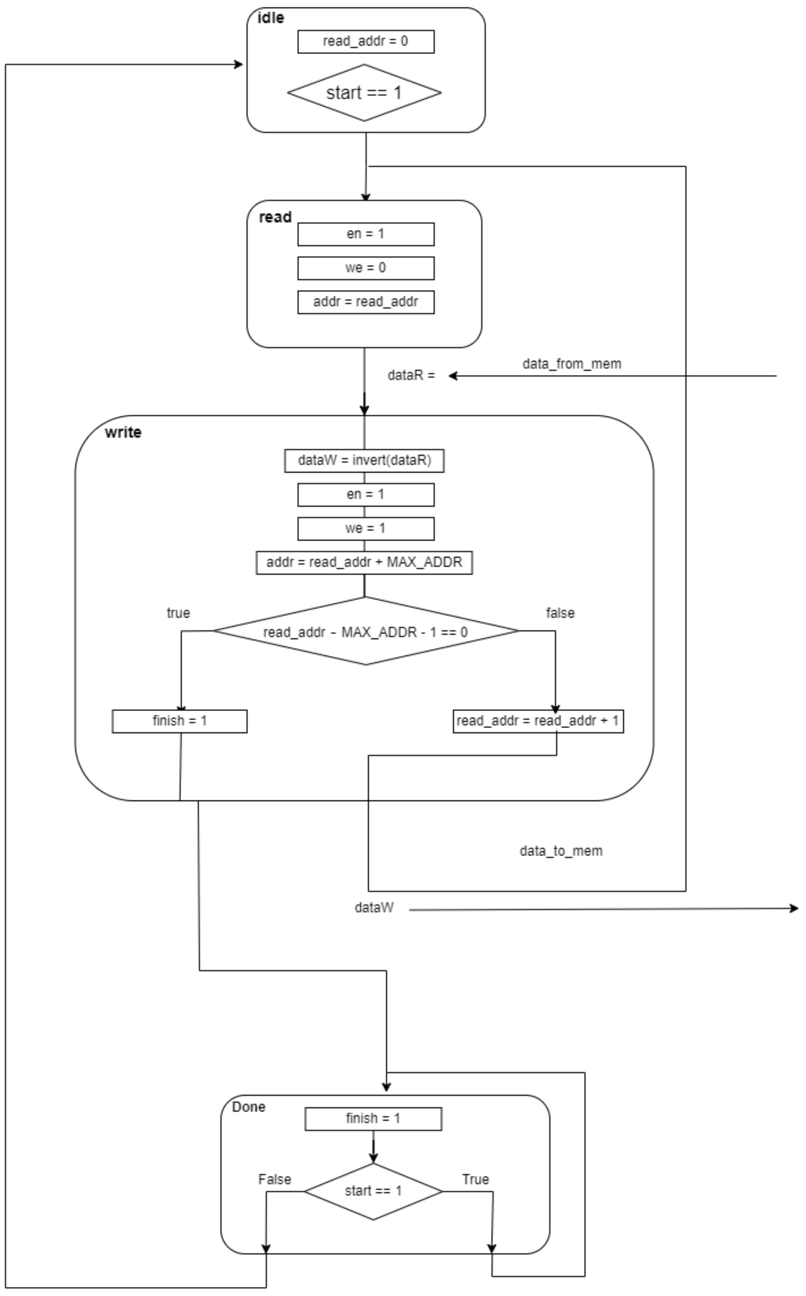


Figure 2: ASM chart of task 0

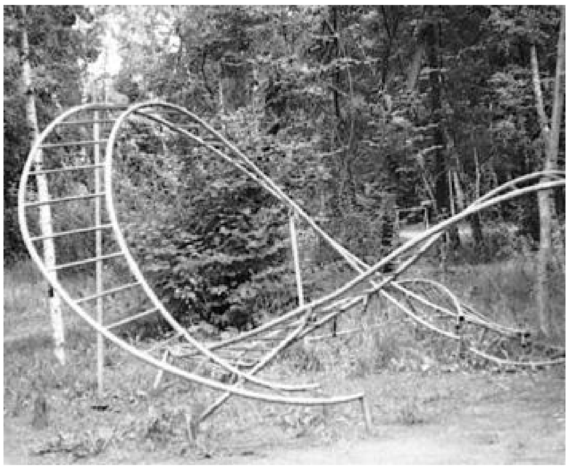


Figure 3: Starting image

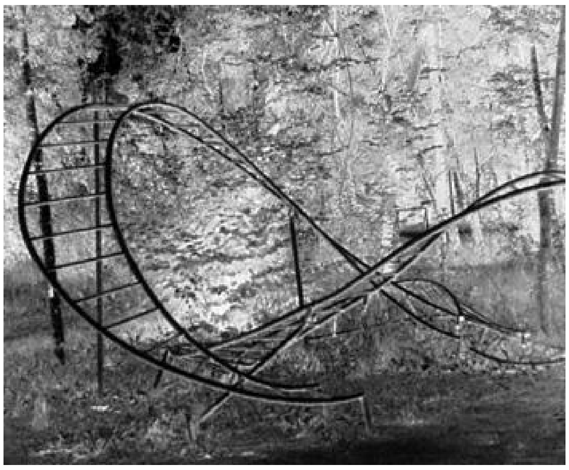


Figure 4: Result image

## 2 Task 1

In Task 1, we focused on designing the edge detector accelerator. The following images illustrate the memory layout, showing how each pixel is stored and the process of fetching them. This visual representation serves as a foundation for understanding the operation of the accelerator. In the sections that follow, we provide a detailed explanation of the accelerator's functionality and the steps involved in edge detection.

### 2.1 Fetching

Building on the image used in Task 0, we now focus on the top-left corner to examine how the pixels are stored in memory.

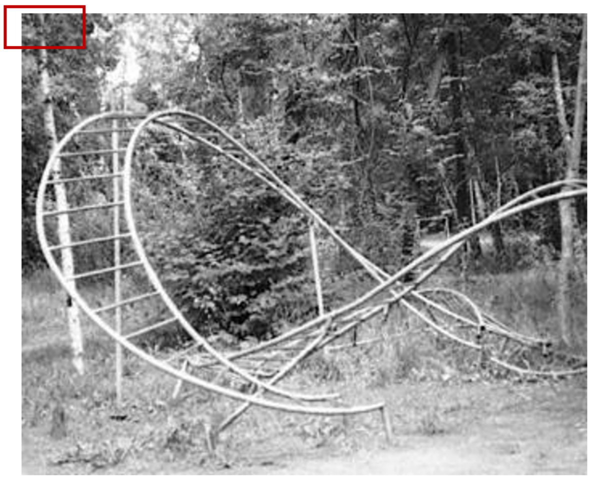
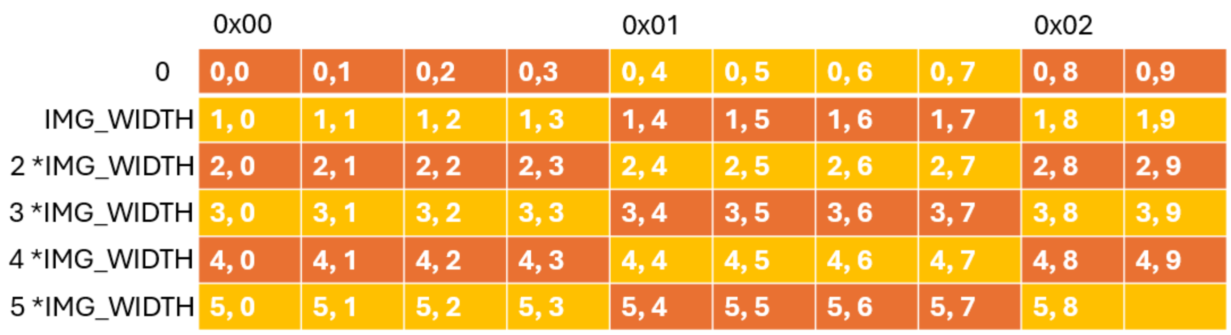


Figure 5: Example image

Pixels are stored in memory row by row, meaning that pixels on the same row are stored consecutively in adjacent memory addresses. Conversely, pixels in the same column have memory addresses offset by `IMG_WIDTH`, which in this case is 352, as the image dimensions are 352x288 pixels. Since the memory word size is 32 bits, each memory address stores four pixels, with each pixel occupying 8 bits (1 byte).



| | 0x00 | | | | 0x01 | | | | 0x02 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0,0 | 0,1 | 0,2 | 0,3 | 0, 4 | 0, 5 | 0, 6 | 0, 7 | 0, 8 | 0,9 |
| IMG_WIDTH | 1, 0 | 1, 1 | 1, 2 | 1, 3 | 1, 4 | 1, 5 | 1, 6 | 1, 7 | 1, 8 | 1,9 |
| 2 *IMG_WIDTH | 2, 0 | 2, 1 | 2, 2 | 2, 3 | 2, 4 | 2, 5 | 2, 6 | 2, 7 | 2, 8 | 2, 9 |
| 3 *IMG_WIDTH | 3, 0 | 3, 1 | 3, 2 | 3, 3 | 3, 4 | 3, 5 | 3, 6 | 3, 7 | 3, 8 | 3, 9 |
| 4 *IMG_WIDTH | 4, 0 | 4, 1 | 4, 2 | 4, 3 | 4, 4 | 4, 5 | 4, 6 | 4, 7 | 4, 8 | 4, 9 |
| 5 *IMG_WIDTH | 5, 0 | 5, 1 | 5, 2 | 5, 3 | 5, 4 | 5, 5 | 5, 6 | 5, 7 | 5, 8 | |

Figure 6: Pixels distribution in the memory

The pixels read from memory are stored in a 3x6 matrix, which buffers 18 pixels, totaling 144 bits. This matrix holds the pixels needed to apply the Sobel operator for multiple pixels.

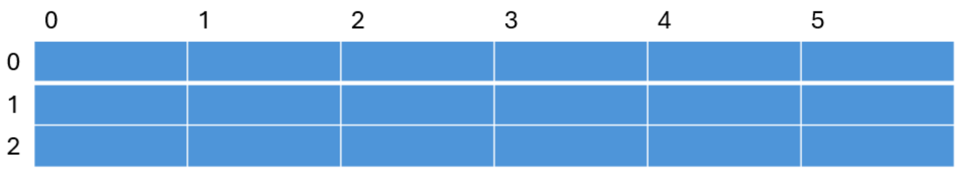
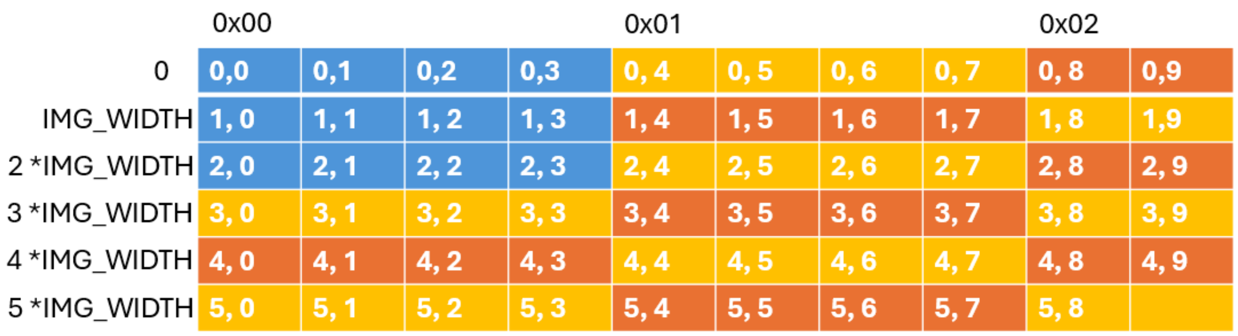


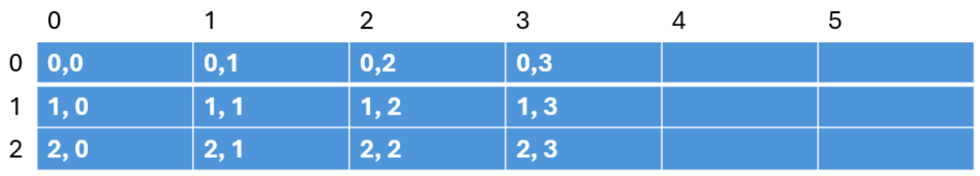
Figure 7: 3x6 matrix

We fetch the first three words from the first column and store them in the first four columns of the matrix. This is done only for the first column. For all the others, the words read from memory are stored in the last four columns of the matrix. The reason for this will be explained in more detail later in the report.



| | 0x00 | | | | 0x01 | | | | 0x02 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0,0 | 0,1 | 0,2 | 0,3 | 0, 4 | 0, 5 | 0, 6 | 0, 7 | 0, 8 | 0,9 |
| IMG_WIDTH | 1, 0 | 1, 1 | 1, 2 | 1, 3 | 1, 4 | 1, 5 | 1, 6 | 1, 7 | 1, 8 | 1,9 |
| 2 *IMG_WIDTH | 2, 0 | 2, 1 | 2, 2 | 2, 3 | 2, 4 | 2, 5 | 2, 6 | 2, 7 | 2, 8 | 2, 9 |
| 3 *IMG_WIDTH | 3, 0 | 3, 1 | 3, 2 | 3, 3 | 3, 4 | 3, 5 | 3, 6 | 3, 7 | 3, 8 | 3, 9 |
| 4 *IMG_WIDTH | 4, 0 | 4, 1 | 4, 2 | 4, 3 | 4, 4 | 4, 5 | 4, 6 | 4, 7 | 4, 8 | 4, 9 |
| 5 *IMG_WIDTH | 5, 0 | 5, 1 | 5, 2 | 5, 3 | 5, 4 | 5, 5 | 5, 6 | 5, 7 | 5, 8 | |

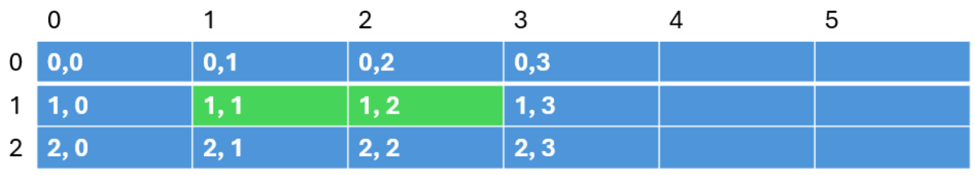
Figure 8: First memory fetch



| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0,0 | 0,1 | 0,2 | 0,3 | | |
| 1 | 1, 0 | 1, 1 | 1, 2 | 1, 3 | | |
| 2 | 2, 0 | 2, 1 | 2, 2 | 2, 3 | | |

Figure 9: First matrix update

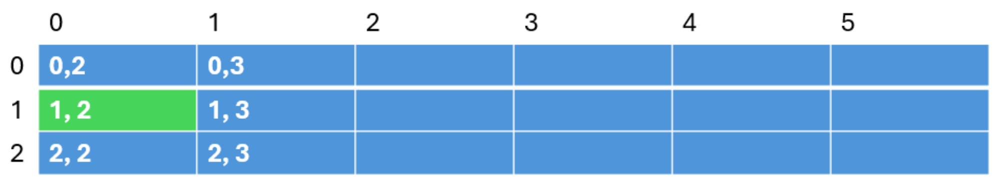
Now we can compute the pixels `1,1` and `1,2`, in green in figure.



| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0,0 | 0,1 | 0,2 | 0,3 | | |
| 1 | 1, 0 | 1, 1 | 1, 2 | 1, 3 | | |
| 2 | 2, 0 | 2, 1 | 2, 2 | 2, 3 | | |

Figure 10: First two pixels computation

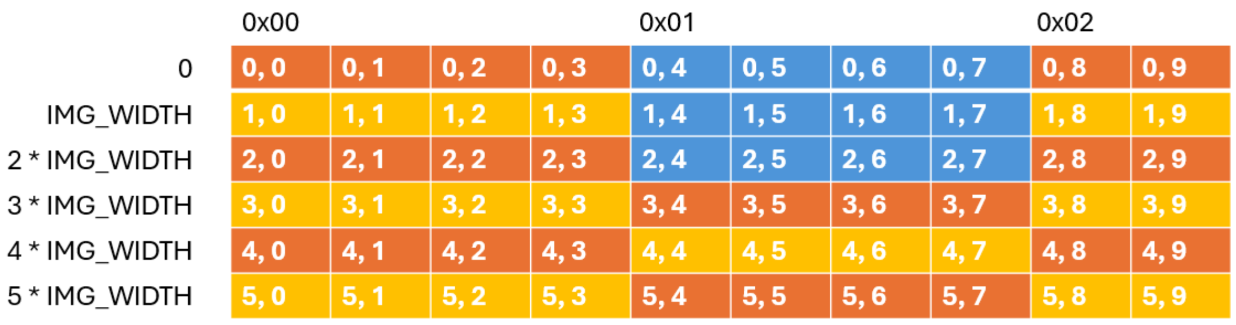
At this stage, we are not ready to write back to memory because we also need to compute the `1,3` pixel. To achieve this, we must fetch three additional words from the next column. Before fetching, we shift the current data in columns 2 and 3 to positions 0 and 1, respectively, replicating the sliding window mechanism. This operation creates space in the matrix for the new three words to be fetched from memory.



| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0,2 | 0,3 | | | | |
| 1 | 1, 2 | 1, 3 | | | | |
| 2 | 2, 2 | 2, 3 | | | | |

Figure 11: First shift

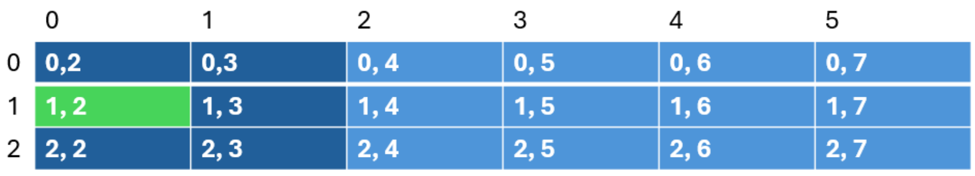
Now we are ready to read:



| | 0x00 | | | | 0x01 | | | | 0x02 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0, 0 | 0, 1 | 0, 2 | 0, 3 | 0, 4 | 0, 5 | 0, 6 | 0, 7 | 0, 8 | 0, 9 |
| IMG_WIDTH | 1, 0 | 1, 1 | 1, 2 | 1, 3 | 1, 4 | 1, 5 | 1, 6 | 1, 7 | 1, 8 | 1, 9 |
| 2 * IMG_WIDTH | 2, 0 | 2, 1 | 2, 2 | 2, 3 | 2, 4 | 2, 5 | 2, 6 | 2, 7 | 2, 8 | 2, 9 |
| 3 * IMG_WIDTH | 3, 0 | 3, 1 | 3, 2 | 3, 3 | 3, 4 | 3, 5 | 3, 6 | 3, 7 | 3, 8 | 3, 9 |
| 4 * IMG_WIDTH | 4, 0 | 4, 1 | 4, 2 | 4, 3 | 4, 4 | 4, 5 | 4, 6 | 4, 7 | 4, 8 | 4, 9 |
| 5 * IMG_WIDTH | 5, 0 | 5, 1 | 5, 2 | 5, 3 | 5, 4 | 5, 5 | 5, 6 | 5, 7 | 5, 8 | 5, 9 |

Figure 12: Second fetch from the memory

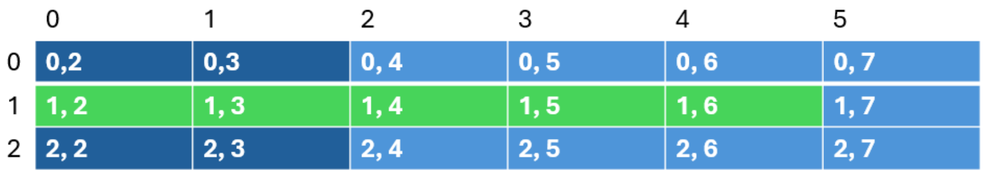
We add the three new words to the matrix. However, since we read from a column other than the first one, we start storing from the column with index 2. This is because the first two columns contain the pixels shifted from the previous memory fetch. In the figure, the computed pixels are marked in green, but the matrix still holds the original values, not the computed ones.



|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0,2 | 0,3 | 0, 4 | 0, 5 | 0, 6 | 0, 7 |
| 1 | 1, 2 | 1, 3 | 1, 4 | 1, 5 | 1, 6 | 1, 7 |
| 2 | 2, 2 | 2, 3 | 2, 4 | 2, 5 | 2, 6 | 2, 7 |

Figure 13: Population of 3x6 matrix

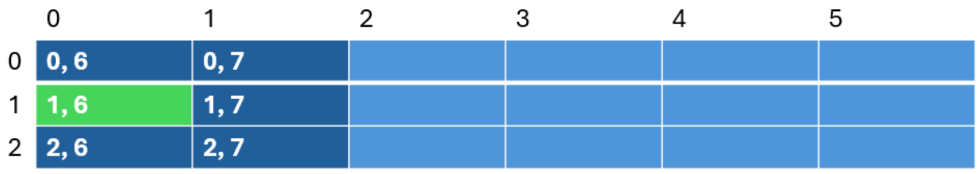
At this point, we can compute the pixels, completing the entire word. Once the computation is finished, the processed word is ready to be written back to memory.



|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0,2 | 0,3 | 0, 4 | 0, 5 | 0, 6 | 0, 7 |
| 1 | 1, 2 | 1, 3 | 1, 4 | 1, 5 | 1, 6 | 1, 7 |
| 2 | 2, 2 | 2, 3 | 2, 4 | 2, 5 | 2, 6 | 2, 7 |

Figure 14: Pixels computation

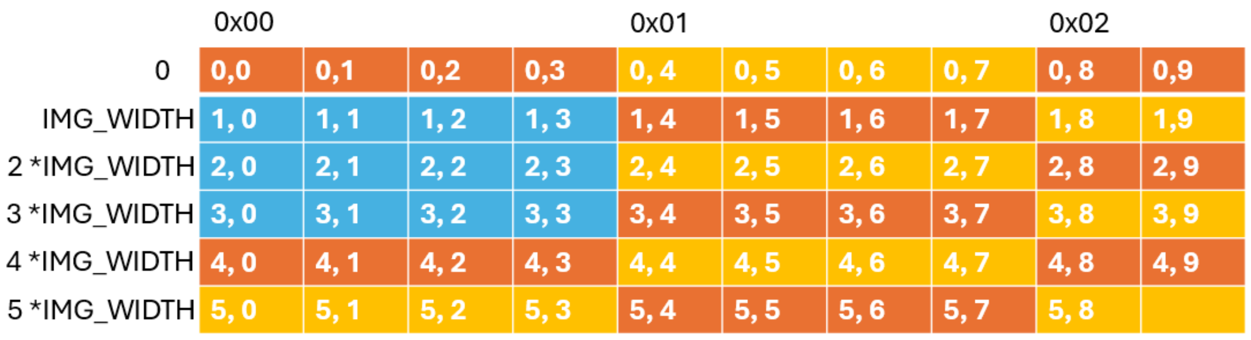
Now we need to perform the shift again. However, since this is no longer the first column, we now shift the data in columns 4 and 5 to positions 0 and 1, respectively.



|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0, 6 | 0, 7 |  |  |  |  |
| 1 | 1, 6 | 1, 7 |  |  |  |  |
| 2 | 2, 6 | 2, 7 |  |  |  |  |

Figure 15: shift operation on 3x6 matrix

We restart the entire process by reading all the columns. Once we reach the edge of the image and compute the final word, we move to the first column of the next row. With this approach, each pixel is read three times from the memory.



| | 0x00 | | | | 0x01 | | | | 0x02 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0,0 | 0,1 | 0,2 | 0,3 | 0, 4 | 0, 5 | 0, 6 | 0, 7 | 0, 8 | 0,9 |
| IMG_WIDTH | 1, 0 | 1, 1 | 1, 2 | 1, 3 | 1, 4 | 1, 5 | 1, 6 | 1, 7 | 1, 8 | 1,9 |
| 2 *IMG_WIDTH | 2, 0 | 2, 1 | 2, 2 | 2, 3 | 2, 4 | 2, 5 | 2, 6 | 2, 7 | 2, 8 | 2, 9 |
| 3 *IMG_WIDTH | 3, 0 | 3, 1 | 3, 2 | 3, 3 | 3, 4 | 3, 5 | 3, 6 | 3, 7 | 3, 8 | 3, 9 |
| 4 *IMG_WIDTH | 4, 0 | 4, 1 | 4, 2 | 4, 3 | 4, 4 | 4, 5 | 4, 6 | 4, 7 | 4, 8 | 4, 9 |
| 5 *IMG_WIDTH | 5, 0 | 5, 1 | 5, 2 | 5, 3 | 5, 4 | 5, 5 | 5, 6 | 5, 7 | 5, 8 | |

Figure 16: New row fetch

## 2.2 Implementation

Now that we understand how the memory is fetched, we can focus on how the edge detection is computed. The core of the process lies in the Sobel Compute Unit, which applies the Sobel operator to the pixel data from the matrix allowing the system to detect edges where there are significant changes in intensity. Once the computations are complete, the results are written back to memory for further use or display.

In our implementation we ignore the first and last column and the first and last row.

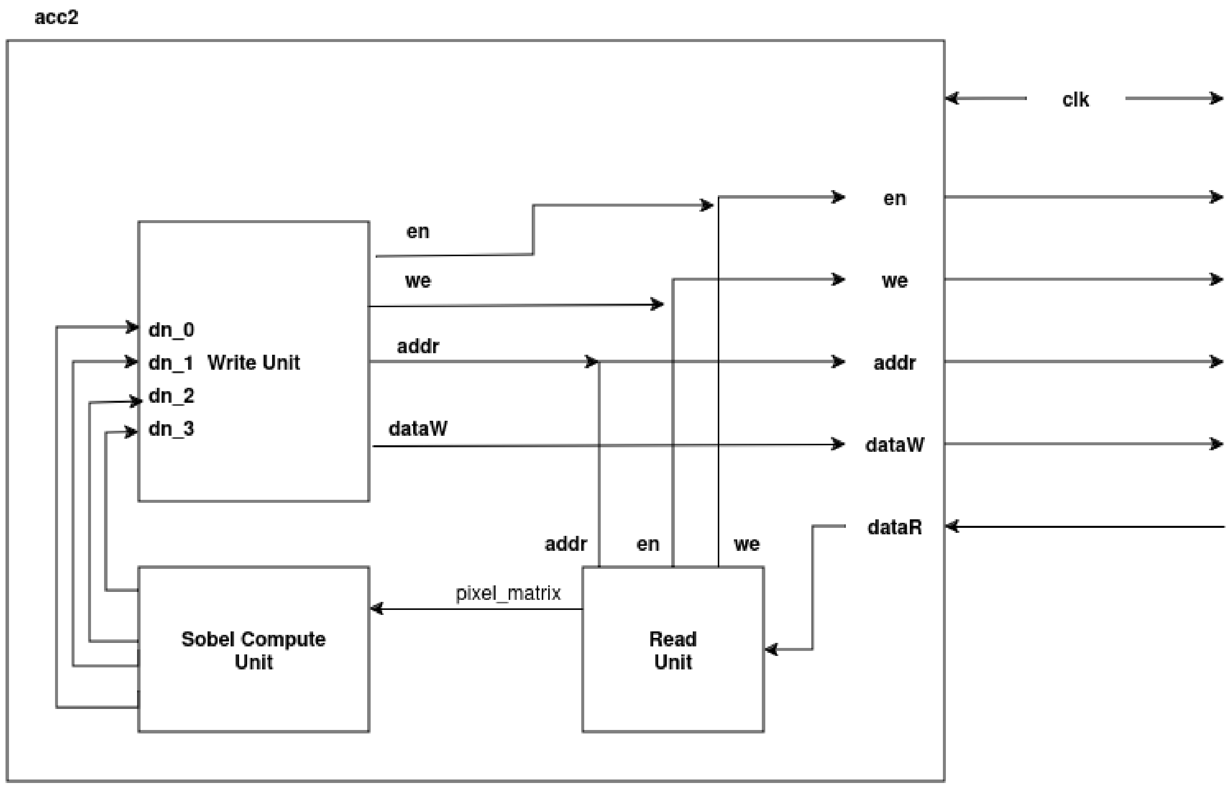


Figure 17: Block Diagram for Data Path

## Block Diagram Description

The block diagram is a simple representation of the datapath for the acc2 edge-detection hardware accelerator.

### Input/Output Signals

- **clk:** The clock signal that drives the entire design.
- **en, we, addr, dataW, dataR:** These signals interface with external memory:
  - **en:** Enables memory transactions.
  - **we:** Indicates whether the operation is a read (‘0‘) or write (‘1‘).
  - **addr:** Specifies the memory address for reading or writing.
  - **dataW:** The data that is written to memory.
  - **dataR:** The data that is read from memory.

### Read Unit

The `Read Unit` is responsible for handling memory read operations. It fetches pixel data from the external memory and loads it into the accelerator. This unit provides the `pixel_matrix` to the `Sobel Compute Unit` for further processing.

### Sobel Compute Unit

The `Sobel Compute Unit` is the core computational block of the accelerator. It performs Sobel edge detection on the input pixel matrix, calculating the edge gradients for the pixels. The outputs of this unit are the computed edge values: `dn_0`, `dn_1`, `dn_2`, and `dn_3`.

### Write Unit

The `Write Unit` formats the computed edge values (`dn_0` to `dn_3`) into the `dataW` signal and manages the write operations. It sends processed results back to memory using the `we`, `addr`, and `dataW` signals.

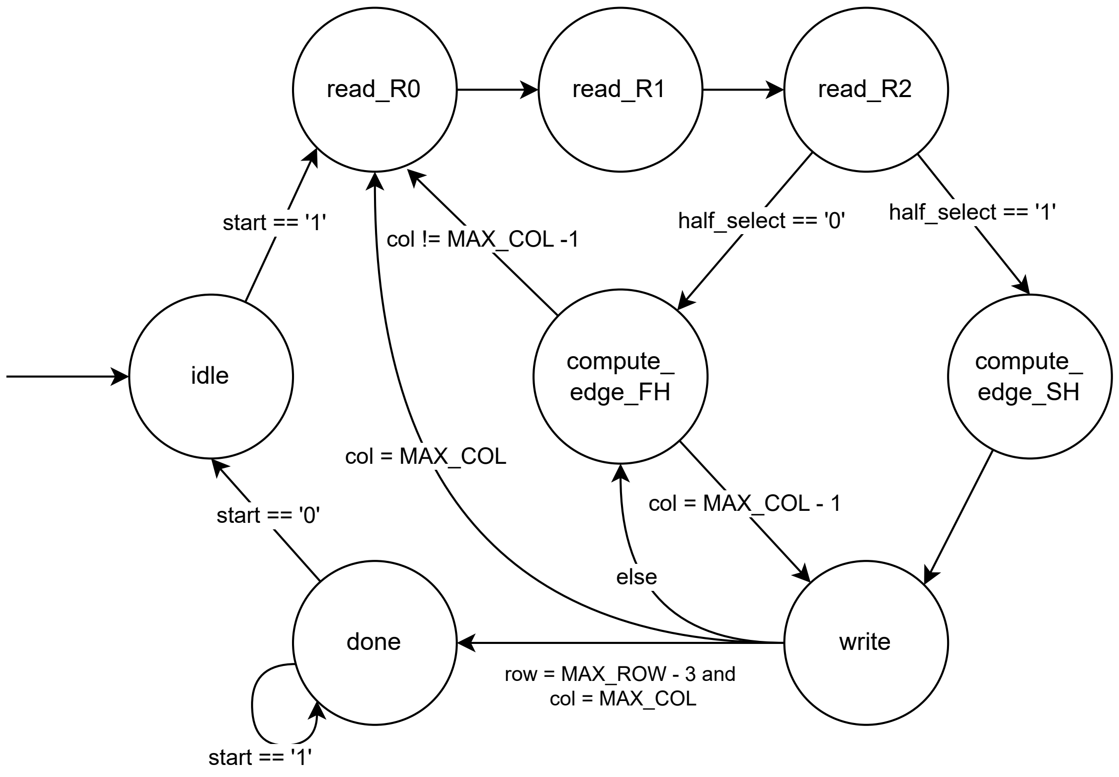


Figure 18: Accelerator FSM

## FSM State Descriptions

The FSM of the accelerator consists of 8 distinct states, each responsible for a specific step in the image processing task. Below is a detailed explanation of each state.

### Idle State

The FSM starts in the `idle` state, waiting for the `start` signal. When detected, it transitions to the `read_R0` state.

### Read_R0 State

In `read_R0`, memory is enabled, and the address is calculated based on the current row and column. After setting the address, the FSM transitions to `read_R1`. The shift operation is done in this state to save a clock cycle. The shift operation is performed only if the signal `half_select` is high. The `half_select` is low only when we are computing the first column of pixels.

### Read_R1 State

The FSM fetches data from memory and stores it in the appropriate positions of the first row of the computation matrix. If it is the first column we store the word in the computation matrix starting from column 0 (see Figure 8), conversely if it is not the word is stored is stored starting from column with index 2 in the computation matrix (see Figure 13). After this, the FSM transitions to `read_R2`.

### Read_R2 State

This state reads the second row of pixels into the computation matrix. What we said for the state `read_R1` applies also for this state.

### Compute_Edge_FH State

In the `compute_edge_FH` state, firstly if the last state was `Read_R2`, the state reads the third row of pixels into the computation matrix. Then the FSM computes the Sobel gradients ($dx$ and $dy$) and gradient magnitudes ($dn$) for the first half of the row. This involves:

- For the first column, gradients are calculated for pixels at (1,1) and (1,2). Pixels outside the matrix boundary (left edge) are ignored.
- For other columns, gradients are calculated for pixels at (1,2), (1,3), and (1,4).

After the computation, the FSM increments the column index and raises the `half_select` signal to indicate the second half of the row needs to be computed.

If the current column is the last column (`col=MAX_COL-1`), it skips the second half and transitions to the `write` state.

### Compute_Edge_SH State

The `compute_edge_SH` state is responsible for calculating the second half of the pixels for the current sliding window and preparing for the next memory write. Specifically, this state:

- Updates the pixel matrix by populating the last row of the computation matrix (row 2) with the most recent pixel data (`dataR`).
- Computes the gradient ($dx$ and $dy$) and gradient magnitude ($dn$) for one additional pixel in the sliding window (pixel at position (1,1)).
- Ensures that the `half_select` signal is reset to 0, indicating that the next computation will start with the first half of the row.
- Transitions to the `write` state to store the computed gradient magnitudes into memory.

For the first column:

- Gradients are calculated for the rightmost edge pixels of the first block.
- The computation incorporates the row's previous gradient data to ensure consistency in edge detection.

For columns beyond the first:

- Gradients and magnitudes are computed for the last available pixel positions in the current half (e.g., (1,4) and (1,5)).
- Ensures that pixel edge data is correctly aligned for subsequent rows.

**Write State**

In the `write` state, the FSM writes the four computed gradient magnitudes (`dn_0` to `dn_3`) to memory in a single operation. After calculating the writing address it resumes with the following logic:

- If the current column is the last column (`col=MAX_COL`), the FSM resets the column index to 0, increments the row index, and transitions to the `read_R0` state for the next row.
- If the current row and column are at their maximum values (`row=MAX_ROW-3` and `col=MAX_COL`), the FSM transitions to the `done` state. It is MAX_ROW-3 as we do not calculate the gradients for the last row.
- Otherwise, it transitions to the `compute_edge_FH` state to continue processing the next portion of the image.

**Done State**

The `done` state signals the completion of processing. The FSM remains in this state until reset. It is responsible for enabling signal `finish` to mark the process done.

For a more detailed overview of the implementation, including the specific state transitions and control logic, please refer to the ASMD chart provided in the appendix 26. The chart offers a comprehensive representation of the design's behavior and execution flow.

## 2.3 Gradient Computation Functions

Listing 1: Gradient Computation

```vhdl
function compute_grad(pixel_matrix : in pixel_matrix_type; row : integer; col
    : integer; op : std_logic) return signed is
    variable gradient : signed(23 downto 0); -- Gradient result
    variable s11, s12, s13, s21, s22, s23, s31, s32, s33 : signed(11 downto 0)
        ; -- 12-bit signed values
begin
    -- Convert 8-bit pixel_matrix values to signed 12-bit values
    s11 := signed("0000" & pixel_matrix(row - 1, col - 1));
    s12 := signed("0000" & pixel_matrix(row - 1, col));
    s13 := signed("0000" & pixel_matrix(row - 1, col + 1));
    s21 := signed("0000" & pixel_matrix(row, col - 1));
    s22 := signed("0000" & pixel_matrix(row, col));
    s23 := signed("0000" & pixel_matrix(row, col + 1));
    s31 := signed("0000" & pixel_matrix(row + 1, col - 1));
    s32 := signed("0000" & pixel_matrix(row + 1, col));
    s33 := signed("0000" & pixel_matrix(row + 1, col + 1));

    if op = '0' then
        gradient := (s13 - s11) + 2 * (s23 - s21) + (s33 - s31);
    else
        gradient := (s11 - s31) + 2 * (s12 - s32) + (s13 - s33);
    end if;
    return gradient(15 downto 0);
end function;

-- Function to compute dn (magnitude of gradient)
function compute_dn(dx : in signed; dy : in signed) return std_logic_vector is
    variable dn : unsigned(15 downto 0);
begin
    -- Normalize by a factor of 4
    dn := unsigned((abs(dx) + abs(dy)) srl 2);
    if dn > 255 then
        dn := THRESHOLD; -- Clamp to threshold
    end if;
    return std_logic_vector(dn(7 downto 0)); -- Return 8-bit result
end function;
```

In this section, we will describe the computation details we needed to figure out to achieve our results (see Listing 1).

The maximum value for $d_x$ and $d_y$ is $1020 = (255-0) + 2 \cdot (255-0) + (255-0)$, which to be represented in binary needs 10 bits, and 1 bit for the sign, which totals to 11 bits. Yet, due to intermediate operations, we extended the values to 12 bits, so that we can prevent possible overflows. This extension ensures that no intermediate result during addition or subtraction exceeds the bit width, avoiding incorrect wrapping of values due to limited representation. The 16-bit gradient returned ensures enough precision for subsequent operations, such as $d_n$ computation, without the risk of truncation errors in scenarios involving high gradients. For $d_n$, we clamp to 255 if the value exceeds the bounds after normalization. This ensures that only extreme gradients are clamped to 255, while maintaining the ability to represent smaller gradients accurately within the 8-bit range. We choose a factor of 4 to the normalization which is explained further in the upcoming sections.

## 2.4 Clock analysis

**Assumptions:**

- The image dimensions are $352 \times 288$
- For every set of 4 pixels processed, we assume an **average of 5 clock cycles** are required.
- The clock frequency is 10 MHz during simulation

### Total Pixel Computation

The total number of pixels in the image is:

$$\text{Total pixels (image)} = 352 \times 288 = 101,376 \text{ pixels.}$$

### Clock Cycles per Image

Given that 4 pixels are computed in 5 clock cycles:

$$\frac{101,376\,\text{pixels}}{4\,\text{pixels/computation}} = 25,344\,\text{computations.}$$

Thus, the total clock cycles required:

$$\text{Total clock cycles} = 25,344\,\text{computations} \times 5\,\text{clock cycles} = 126,720\,\text{clock cycles.}$$

### Time per Image

The clock period at 50 MHz is:

$$\text{Clock period} = \frac{1}{50 \times 10^6} = 20\,\text{ns.}$$

Hence, the time to process one image is:

$$\text{Time per image} = 126,720\,\text{clock cycles} \times 20\,\text{ns} = 2,534,400\,\text{ns} = 2.53\,\text{ms/image.}$$

### Images per Second

The number of images processed per second reuslts in:

$$\text{Images per second} = \frac{1}{2.53 \times 10^{-3}\,\text{s}} \approx 395\,\text{images/second.}$$

**Result:** Approximately 395 images per second are processed under the given assumptions.

### Clock Frequency Choice

From the synthesis results, we identified the critical path of our accelerator, shown in Figure 27 in the appendix. The critical path, located in the compute_edge_FH state, has a total delay of 13.6 ns. To keep the design simple and reliable, we use a 50 MHz clock from the clock divider. This clock provides a safe margin over the delay. The table below shows a slack time of 6.2 seconds.

| Name | Slack | Total Delay | Logic Delay | Src Clk | Dst Clk | Clock Uncertainty |
|---|---|---|---|---|---|---|
| Path 1 | 6.2024 | 13.6079 | 7.3650 | clk | clk | 0.0887 |
| Path 2 | 6.2024 | 13.6079 | 7.3650 | clk | clk | 0.0887 |
| Path 3 | 6.2024 | 13.6079 | 7.3650 | clk | clk | 0.0887 |
| Path 4 | 6.2024 | 13.6079 | 7.3650 | clk | clk | 0.0887 |
| Path 5 | 6.2104 | 13.5999 | 7.3570 | clk | clk | 0.0887 |

## 2.5 Considerations and Optimizations

### Normalization and Clamping

When calculating the Sobel Operator, values may get out of bounds and surpass the 0 to 255 range. Clamping `dn` at 255 ensures that the output remains within the valid range for image pixel intensities.

When we were building our solution, we observed, that the image had very high exposure (a lot of white pixels). To tackle that, we decided to divide by a factor of 4. A smaller divisor like 4 ensures that lower gradient values, corresponding to subtle edges, are not reduced too much. This maintains the visibility of finer details in the image. It is also common to divide by 8, but in our use case 4 seemed to visually resemble more the existing implementation of the edge filter (e.g. Python OpenCV implementation).

To justify the choice of a normalization factor of 4, we consider the scaling required to ensure that $d_n$ remains bounded by 255 after normalization. The normalization process divides $d_x$ and $d_y$ by a factor and clamps the resulting magnitude $d_n$ within the range $[0, 255]$. The maximum possible value for the gradient magnitude $d_n$ is given by $d_n = \sqrt{|d_x| + |d_y|}$. The theoretical maximum possible value for $d_x$ and $d_y$ is 1020, so the maximum magnitude is $d_n^{\max} = |1020| + |1020| = 2040$. To ensure $d_n^{\max} \leq 255$, we apply a normalization factor $k$, such that $k \leq \frac{255}{2040} = \frac{1}{8}$. A normalization factor $k = \frac{1}{8}$ would make sense here as it ensures $d_n$ is within bounds. Yet, $k = \frac{1}{4}$ was chosen to preserve finer details and maintain better visibility of subtle edges in the image. Given the relation we just found, there will be values that exceed the threshold, but those in turn will get clamped, ensuring that the final result remains within the valid intensity range.

### Operator Sharing

To make use of the resources in an optimal manner, we decided to use operator sharing in the computation of the gradient for `dx` and `dy`. We define `op` which takes values 0 and 1 to interchange between calculation for `dx` and `dy` in the same function. By doing that, we aim to use the same hardware resources (adders, subtractors etc.) for both computations, but at different times.

To validate this approach, we synthesized our design for both implementations. Due to default optimizations enabled by Synthesis, the resources used ended up being the same in the Synthesis and Utilization Report. So, we decided to tweak a bit the arguments used in Synthesis to have a less biased comparison. First we disabled `incremental_mode` by setting it to off to disable all optimizations. This approach also lead to identical results. Then we moved on and enabled `keep_equivalent_registers` which prevents registers sourced by the same logic by being merged. As none of the approaches proved fruitful, we finally disabled `resource_sharing` (for sharing arithmetic operators), and it indeed changed the quantity of resources but, in the end, there was no difference in the two implementations we developed. To conclude, using operator sharing by keeping a shared function did not really make a difference in our design in terms of resource number. This could be due to further internal optimizations, or maybe overlapping resource demands.

In Tab. 1 we can see the resources with the default optimizations enabled for both approaches.

| **Resource Type** | **Details** | **Quantity** |
|---|---|---|
| Adders | 2 Input, 16 Bit Adders | 4 |
| | 3 Input, 16 Bit Adders | 4 |
| | 4 Input, 16 Bit Adders | 4 |
| | 3 Input, 12 Bit Adders | 14 |
| Muxes | 8 Input, 32 Bit Muxes | 1 |
| | 8 Input, 16 Bit Muxes | 27 |
| | 2 Input, 16 Bit Muxes | 26 |
| | 8 Input, 15 Bit Muxes | 1 |
| | 8 Input, 8 Bit Muxes | 51 |
| | 2 Input, 8 Bit Muxes | 95 |
| | 8 Input, 7 Bit Muxes | 1 |
| | 8 Input, 1 Bit Muxes | 10 |
| | 2 Input, 1 Bit Muxes | 2 |

Table 1: RTL Component Statistics with Default Optimizations are identical for both approaches (operator sharing present or absent)

# 3 Task 2 & 3

## 3.1 Results

The provided results, displayed below, were calculated using simulation in Vivado. These outputs showcase the transformation achieved through the implementation of the Sobel operator, with the edge-detection process verified through accurate simulation.

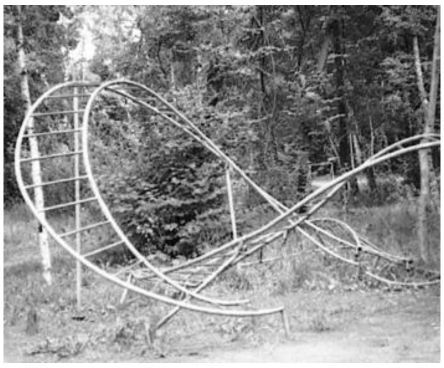


Figure 19: Original pic1

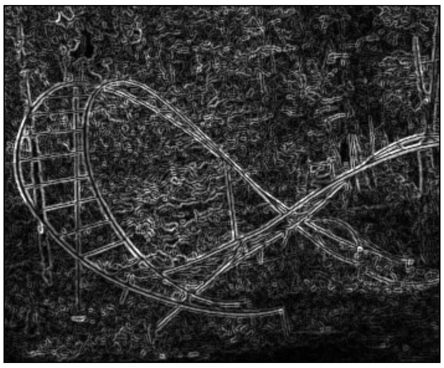


Figure 20: Result pic1

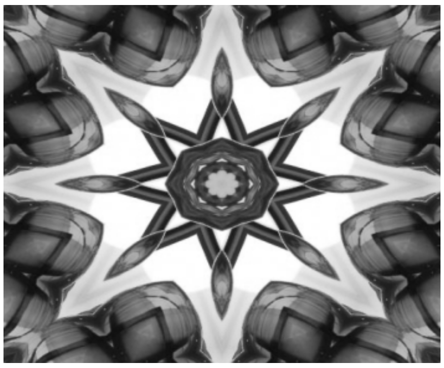


Figure 21: Original kaleidoscope

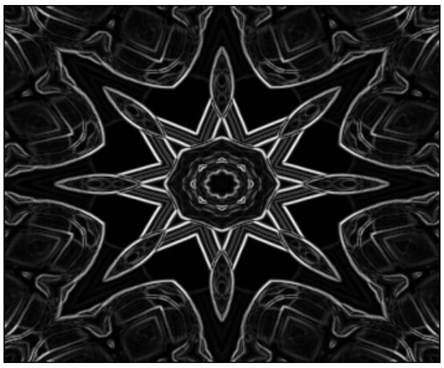


Figure 22: Result kaleidoscope

As previously mentioned, we can achieve results similar to the one shown above in approximately 3 ms. The image below illustrates this performance. In the calculations presented in the Clock Analysis subsection, we estimated 2.53 ms for computing an image. This slight difference arises because we used an average value of 5 clock cycles per pixel, which holds true for most scenarios but not for edge cases, such as processing the first column.

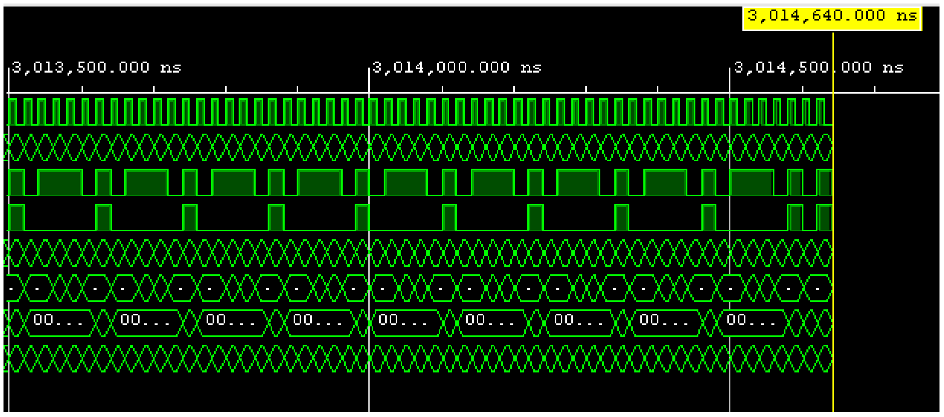


Figure 23: Edge-detection simulation

### Running the implementation on the FPGA

When running the implementation on the FPGA a small change had to be made to the done state. The change ensures that start is 0 before going into the idle state. This is necessary because otherwise the edge-detection will run continuously until the button is released, which results in strange behavior, as the values for reading and writing to memory are not reset.

Listing 2: Running on FPGA

```
when done =>
    if start = '1' then
        finish <= '1';
        next_state <= done;
    else
        next_state <= idle;
    end if;
```

## 3.2 Optional Tasks

### Buffering

Due to time constraints, our buffering solution is not fully functional, but we have included it in a separate folder. Instead of using a block of RAM, we implemented two circular buffers. These custom circular buffers act as FIFO queues to store words fetched from memory. For the first row of image pixels, the process remains unchangedâwe fetch three new rows, requiring three clock cycles for reading. After processing, we push the second row into the first queue and the third row into the second queue. This continues until we reach the last column and return to the image's left edge. At this point, both queues are full, each storing `IMG_WIDTH` pixels, or $352 \times 2 = 704$ pixels in total. Since each pixel is 1 byte, the total buffered size is 704 bytes.

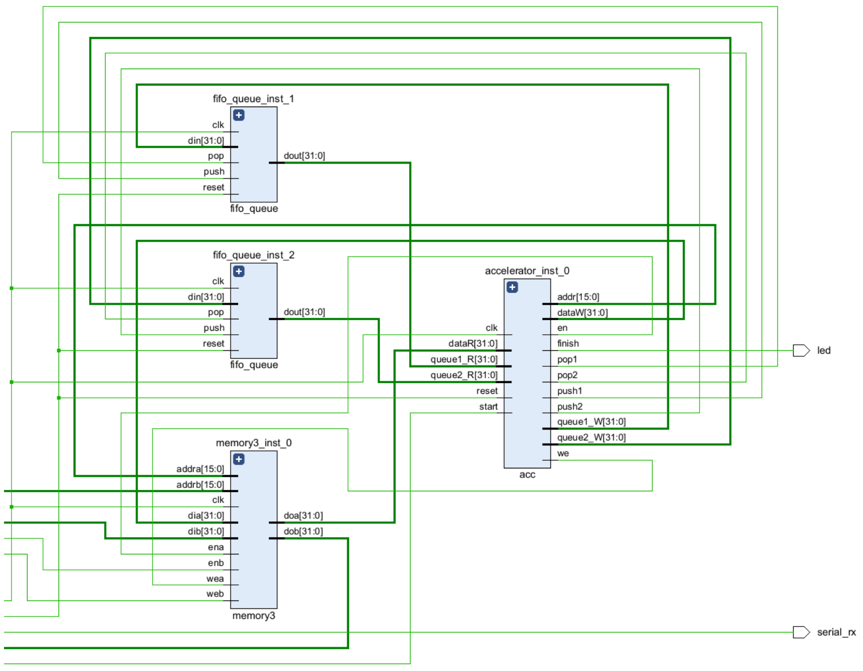


Figure 24: RTL schematic with the two queues

Next, we increment the row by one. Instead of fetching all rows from memory, we retrieve the first row from the first queue and the second row from the second queue. The third row is fetched from memory as usual. This reduces the read time to one clock cycle instead of three. Meanwhile, the second and third rows of the computation matrix are pushed into the first and second queues, respectively.

This approach significantly reduces the time the accelerator needs to process a single image. Simulations show that with buffering, computation time decreases from 3 ms to 2 ms, resulting in a 33.3% reduction in processing time, which corresponds to a 33.3% increase in images per second.

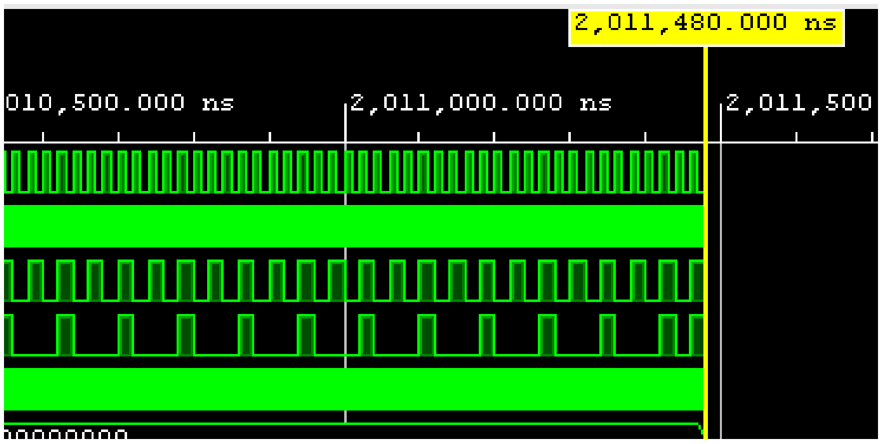


Figure 25: Simulation Window Timing

**Pipelining**

While buffering was intended to reduce memory access times by storing intermediate rows locally, pipelining would focus on overlapping the stages of memory access and computation. In a pipelined implementation, the accelerator would perform computations on buffered data simultaneously with fetching new rows from memory. By decoupling these states, the design ensures that the computation state is never idle, as it can continuously process data available in the buffer while the memory interface prepares the next set of data. This parallelization would have minimized latency and optimized the utilization of the clock.

# 4 Appendix

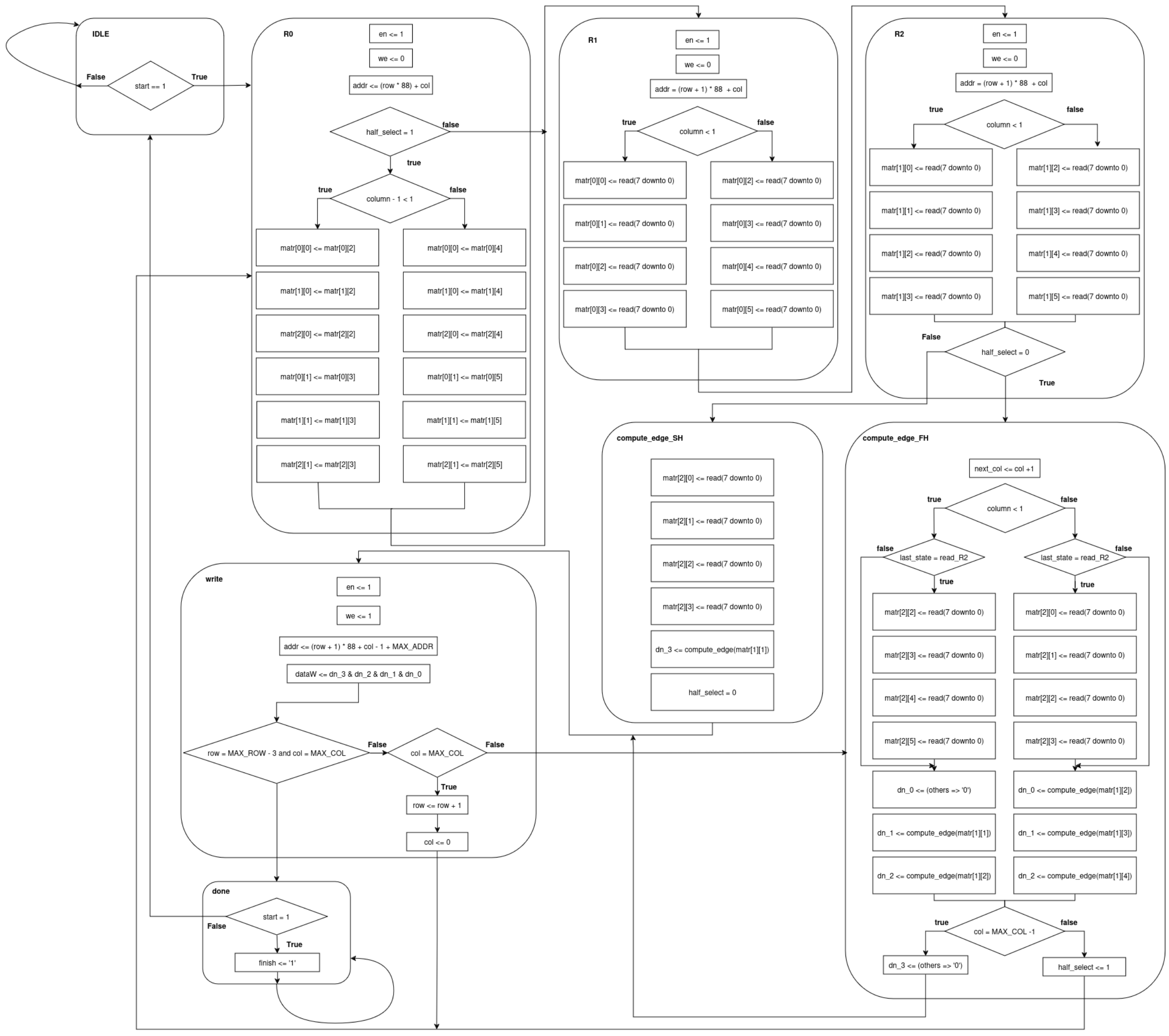

IDLE
start == 1
False
True
R0
en <= 1
we <= 0
addr <= (row * 88) + col
half_select = 1
false
true
column - 1 < 1
true
false
matr[0][0] <= matr[0][2]
matr[1][0] <= matr[1][2]
matr[2][0] <= matr[2][2]
matr[0][1] <= matr[0][3]
matr[1][1] <= matr[1][3]
matr[2][1] <= matr[2][3]
matr[0][0] <= matr[0][4]
matr[1][0] <= matr[1][4]
matr[2][0] <= matr[2][4]
matr[0][1] <= matr[0][5]
matr[1][1] <= matr[1][5]
matr[2][1] <= matr[2][5]
R1
en <= 1
we <= 0
addr = (row + 1) * 88 + col
column < 1
true
false
matr[0][0] <= read(7 downto 0)
matr[0][1] <= read(7 downto 0)
matr[0][2] <= read(7 downto 0)
matr[0][3] <= read(7 downto 0)
matr[0][2] <= read(7 downto 0)
matr[0][3] <= read(7 downto 0)
matr[0][4] <= read(7 downto 0)
matr[0][5] <= read(7 downto 0)
R2
en <= 1
we <= 0
addr = (row + 1) * 88 + col
column < 1
true
false
matr[1][0] <= read(7 downto 0)
matr[1][1] <= read(7 downto 0)
matr[1][2] <= read(7 downto 0)
matr[1][3] <= read(7 downto 0)
matr[1][2] <= read(7 downto 0)
matr[1][3] <= read(7 downto 0)
matr[1][4] <= read(7 downto 0)
matr[1][5] <= read(7 downto 0)
half_select = 0
False
True
compute_edge_SH
matr[2][0] <= read(7 downto 0)
matr[2][1] <= read(7 downto 0)
matr[2][2] <= read(7 downto 0)
matr[2][3] <= read(7 downto 0)
dn_3 <= compute_edge(matr[1][1])
half_select = 0
compute_edge_FH
next_col <= col +1
column < 1
true
false
last_state = read_R2
false
true
last_state = read_R2
false
true
matr[2][2] <= read(7 downto 0)
matr[2][3] <= read(7 downto 0)
matr[2][4] <= read(7 downto 0)
matr[2][5] <= read(7 downto 0)
matr[2][0] <= read(7 downto 0)
matr[2][1] <= read(7 downto 0)
matr[2][2] <= read(7 downto 0)
matr[2][3] <= read(7 downto 0)
dn_0 <= (others => '0')
dn_1 <= compute_edge(matr[1][1])
dn_2 <= compute_edge(matr[1][2])
dn_0 <= compute_edge(matr[1][2])
dn_1 <= compute_edge(matr[1][3])
dn_2 <= compute_edge(matr[1][4])
col = MAX_COL -1
true
false
dn_3 <= (others => '0')
half_select <= 1
write
en <= 1
we <= 1
addr <= (row + 1) * 88 + col - 1 + MAX_ADDR
dataW <= dn_3 & dn_2 & dn_1 & dn_0
row = MAX_ROW - 3 and col = MAX_COL
False
col = MAX_COL
False
True
row <= row + 1
col <= 0
done
start = 1
False
True
finish <= '1'

Figure 26: ASMD Chart Task 1

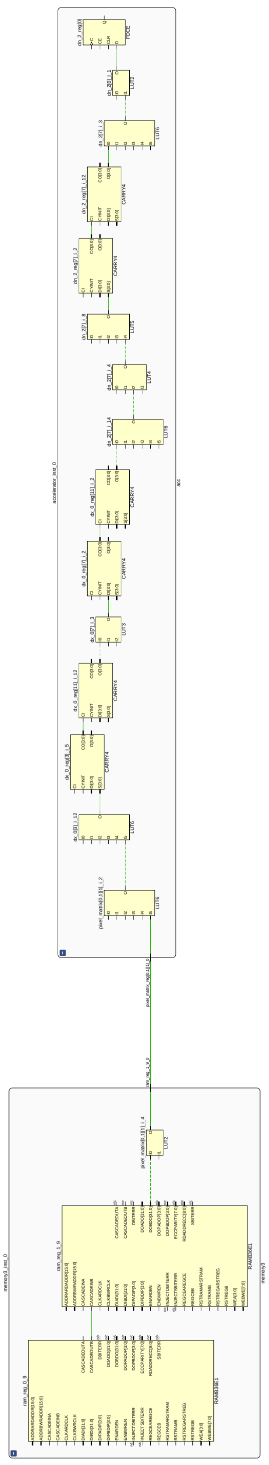


Figure 27: Critical Path

# List of Figures

# List of Tables