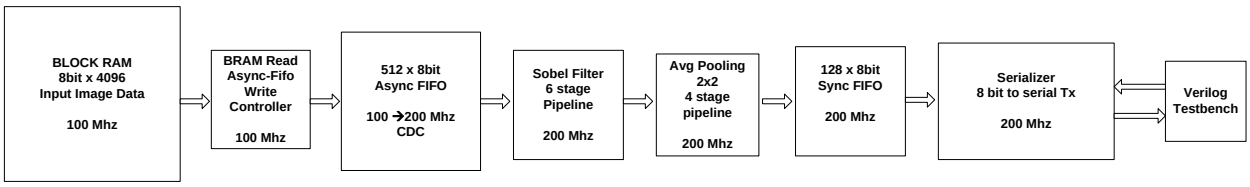
Project Report: Image Processing Pipeline using FPGA

1. Introduction

This project implements a real-time image processing pipeline using an FPGA. The pipeline consists of reading pixel data from Block RAM (BRAM), applying the Sobel edge detection algorithm, performing average pooling down-sampling, and outputting the processed image via a serializer. The design uses multiple clock domains, asynchronous and synchronous FIFOs for data buffering, and a pipelined architecture for efficient processing.

2. System Architecture



The architecture consists of the following main components:

1. **Block RAM (BRAM)**
   - Stores the input image data.
   - Read by a 100 MHz controller.
   - Implemented using Vivado IP catalog's BRAM generator. Single port memory, latency=1clk for read.
   - Configured with total 5000 addressed locations, of which **4096 locations each store 8-bit pixel data.**
   - Pixel data is stored in **input_pixels.coe** file which is stored into BRAM using customize IP
2. **BRAM FIFO Controller**
   - Reads data from BRAM and writes it to the asynchronous FIFO.
   - A simple read and write state machine.
3. **Asynchronous FIFO**
   - Buffers data between 100 MHz BRAM reader and 200 MHz processing unit.
   - Implemented using Vivado FIFO Generator IP.
   - Configured as **8-bit wide, 512 deep**.
   - Block RAM based, Read latency 1 clk, two independent clocks
4. **FIFO Reader**
   - A simple circuit that reads data from the asynchronous FIFO and passes it to the Sobel processing unit.
5. **Sobel Edge Detection Unit**
   - Processes data at 200 MHz.
   - Detects edges using Gx and Gy gradient computations.

- Uses a **3x3 window** of pixel values and only starts computation after loading minimum 2 rows and 2 columns of pixels.
- The input 64x64 pixel image is converted to a 62x62 pixel image by sobel.
- Total Pipeline depth: **6 stages** for optimized performance.
- To simplify gradient computation, register shifting is used instead of multiplication with coefficients.
- The addition path is pipelined to reduce critical path.
- Instead of using square root calculation to find edge weight, the absolute values of Gx and Gy are found and summed to create edge weight.
- The sobel output is not clamped to 8 bits, instead its kept as 12 bits to keep resolution and more data from the edge weight pixel.

6. **Pooling Unit**
   - Take the **12 bit sobel pixel** and converts it to **8 bit output pixel**.
   - Performs **2x2 average pooling with stride 2**.
   - Reduces 62x62 edge-detected image to 31x31 pixels (961 pixels total).
   - Operates at 200 MHz.
   - Total Pipeline depth: **4 stages** for optimized performance.
7. **Synchronous FIFO**
   - Buffers data before serialization.
   - Implemented using Vivado FIFO Generator IP.
   - Configured as **8-bit wide, 128 deep**, Read latency 1 clk
   - Backpressure is applied using the condition *write_data_count > 119*, which signals ***not ready*** directly to the asynchronous FIFO when fewer than 9 locations are empty. These empty locations help to buffer in the valid data from the sobel and pooling pipelines. Theoretically this valid data in pipelines can be around maximum 4-5. But a larger count of 9 is used to prevent any problems such as missing data and ensure smooth operation.
8. **Serializer**
   - Converts parallel 8-bit data into a serial stream.
   - Operates at 200 MHz.
   - Outputs processed image data.
   - Handshakes with the behavioral verilog Testbench with **valid/data** output and **ready** input.
9. **Top module**
   - Instantiates the device under test.
   - Includes all modules above.
10. **Verilog Testbench**
    - Initializes the device under test with an Asynchronous reset.
    - Asynchronous reset is applied **Async** to the dut and is **Sync-Deasserted** within the DUT module.
    - Generates and feeds 100 and 200 Mhz clocks to the dut.
    - After reset process is completed, testbench sends a **Start** signal to the dut to start processing.
      Waits in a process loop to accept serially received 8 bit pixel data with validation signal for each bit.
    - After receiving all 31x31=961 pixels, the testbench stops the run.

    - Outputs processed image data into an output_pixels.txt file.
11. **Python2 scripts**
    - Used to create both input data file from a test image and output image file from output pixel data.
    - Input file is loaded into BRAM.
    - Output file is processed to a 31x31 grayscale image.
    - Input test image and output image are visually compared.
    - Was in the process of creating scripts and verilog testbench for pixel wise comparison of expected
    - (Old PC, so Python3 was not installed.)

3. Implementation Details

- The **Asynchronous FIFO** ensures smooth data transfer between the 100 MHz and 200 MHz clock domains, removing any serious issues **with CDC clock domain crossing**.
- The **Synchronous FIFO** prevents data loss before serialization.
- The **BRAM** stores the input image in **row-major order**.
- **Sobel** and **Pooling unit** are pipelined to minimize critical path delay and for efficient processing high data throughput.
- **Python2 scripts** are used to generate the input image data and reconstruct the output image from the serialized data.
- **Verilog testbench** verifies the design by capturing the output pixels and writing them to a text file.

4. Vivado Synthesis and Implementation Flow

Used **Vivado 2019 HLS** Webpack free edition

- The FPGA software was used to create the design.
- Target device was **Atrix 7 xc7a200tsbv484-2L**
- Design was implemented in Verilog RTL.
- BRAM was loaded with an input_pixels.coe pixel data file via customize IP window.
- The simulation was first verified via behavioral simulation.
- Design constraints file was created to constrain clocks and inputs and assign pins to be used on FPGA.
- RTL was Analysed and Elaborated
- Design was synthesized and Implemented
- Finally Post Synthesis and Post Implementation Behavioral Simulations were run. The reception of 961 pixels was verified in Testbench and the image output data was converted to a grayscale 31x31 pixel image to be visually checked.

5. Verification Methodology

1. **Input Image Generation**

    - A **Python2 script** generates an image in `.coe` format for BRAM initialization.
    - Pixels are stored in **row-major order**, with one pixel per line.
2. **FPGA Processing**
    - The FPGA processes the image, applying **Sobel filtering and pooling**.
    - The output is serialized and captured in a **Verilog testbench**.
3. **Output Image Reconstruction**
    - The serialized output is stored in a text file.
    - A **Python2 script** reads the file and reconstructs the output image for visual comparison.

Input 64x64 image

Output of sobel filter below 62x62 (obtained by bypassing pooling layer while simulating)

Output of pooling layer below 31x31 pixel after simulation

The image maybe shifted to the right by about 2 columns. This is because the sobel filter verilog only starts processing after column and row count are both >2, to form the sampling 3x3 window with valid data. Similarly pooling layer only starts processing after column and row are both >1.

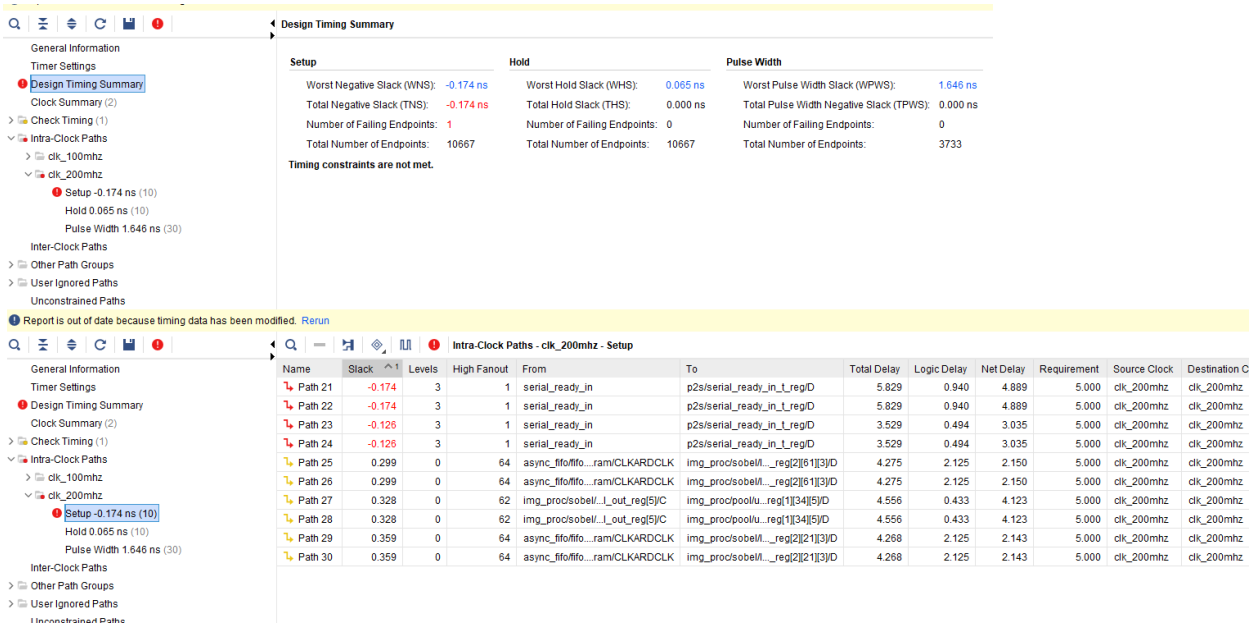
6. Challenges and optimizations

- **Hold and Setup violations**
    - During synthesis and Implementation there were some **time violations**.
    - Implementation also includes placed net delays while synthesis does not take many net delays into account.
    - **So i**nstead of synthesis, Implementation success was targeted, even if synthesis fails.
    - To prevent the software optimizing too much the path delays, and to preserve comb delays to prevent hold time issues, FSM encoding was kept as is by **turning off Auto encode option** in synthesis settings.

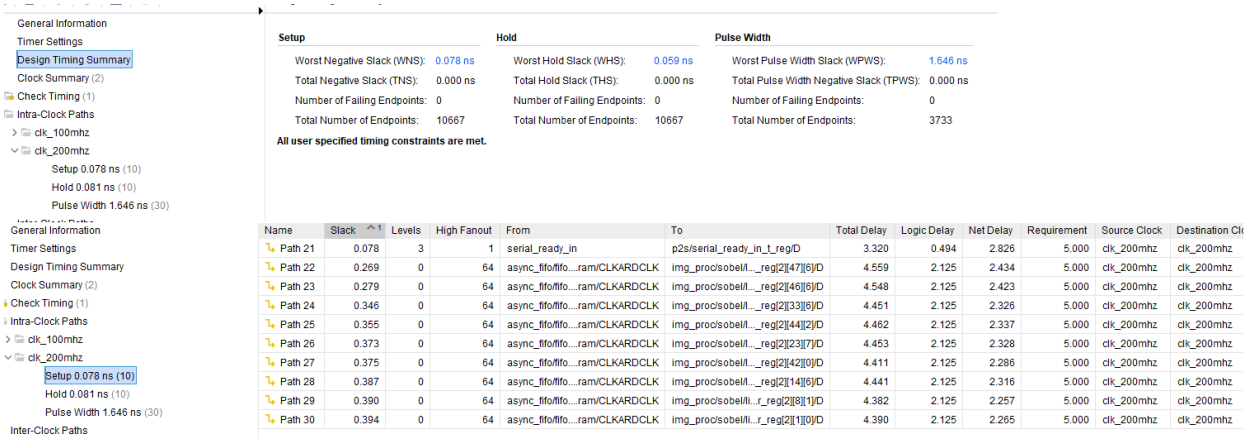
- Some RTL code was modified **with (* dont_touch = "yes" *)** directive and **redundant comb delay** was added wherever combinational delay was desired.
- Some state encoding was **deliberately made to be larger** than needed to add comb delay.
- This was needed in parallel to serial module. There was a lot of trouble to find timing closure for both set up and hold for serial_ready input to parallel to serial module. This module was being run at 200Mhz, so timing closure for hold timing was not being met while synthesis was done.

- **Not fixed Hold or Setup violation concerning 1 path**
  - The path below doesn't have any combinational delay It's a primary input coming in through pin serial_ready_in to a register that latches it.
  - **-from [get_ports serial_ready_in] -to [get_pins p2s/serial_ready_in_t_reg/D]**
  - This path exists in the serial to parallel module and runs at 200mhz. This path started with a hold violation. So to try to fix it, redundant comb delay was added which was set don't touch for synthesis tools to Not optimize away.
  - This resulted in the path failing either setup or hold even if miniscule delays were added like just adding nets, adding 2 back to back inverting, adding Anding logic with one input tied to 1 etc.
  - One or two times, the Vivado tool actually gave no errors with either setup or hold. In these attempts, there were zero failing end points. But re attempting the synthesis and implementation resulted in the hold or setup fail happening again with the path.
  - **Finally it was decided to ignore the path and simulate.**
  - **The post synthesis and post implementation simulation both yielded correct waveform and ouput pixel data results**

The failing end point in pic



The same end point passed and the design was implemented without errors in another run



- **FIFO Depth**
  - Initially async and sync fifo were both depth 64. Minimum use of logic gates was the objective to reduce resource consumption. But this fifo setup let to errors such as missing data when the design was simulated after doing synthesis and implementation, where gate delays and net delays can cause issues.
  - So the FIFO depth was revised to the current setup to have smoother processing of data stream. See below

Deciding FIFO Depths Based on Data Rate

	Clock	Throughput (Pixels/sec)	Data Rate (MB/s)
BRAM → Async FIFO	100 MHz	100M pixels/sec	100 MB/s
Async FIFO → Sobel	200 MHz	200M pixels/sec	200 MB/s
Sobel → Pooling	200 MHz	200M pixels/sec	200 MB/s
Pooling → Sync FIFO	200 MHz	50M pixels/sec	50 MB/s
Sync FIFO → Serializer	200 MHz	25M pixels/sec	25 MB/s

Choosing 512-depth Async FIFO and 128-depth Sync FIFO

- 4 BRAM pixels = 1 Serializer pixel
- 4 Sobel pixels = 1 Pooling output
- Async FIFO fills 4× faster than Sync FIFO empties
- Sync FIFO fills 2× faster than the Serializer can send
- A 512-depth Async FIFO provides sufficient buffering to handle data flow variations without causing stalls and missing data.
- A 128-depth Sync FIFO ensures a steady stream of pixels to the serializer while preventing frequent backpressure events.

7. Deliverables

The files for the following are attached in the zip file

- **Verilog files** for all the modules and testbench
- **Constraint .xdc file**
- **input_pixels.coe** file for input image pixel data
- input test image **test7.png**
- 31x31 output pixel data after behavioral/post syn/post implementation as **output_pixels.txt**
- Reconstructed **output images** after sobel and after pooling operation
- **Python2 script**s for creating input pixel data and reconstructing output image from pixel data.
- .
- .
- Reports .txt files for **Power, Power Opt and Timing summary, Timing paths and CDC**.
- Waveform picture from post implementation run
- **Waveform .wdb files** which can be loaded via *Flow pull down menu* and *Open static simulation*. **These .wdb files are attached for behavioral, post synthesis functional and post implementation** functional runs

7. Conclusion

This FPGA-based image processing pipeline demonstrates efficient real-time edge detection and downscaling. The implementation utilizes optimized hardware blocks, clock domain crossing FIFOs, and a pipelined approach for high-performance processing. The correctness of the design is verified by comparing input and output images.