Final Report: Depths of SADness

Veloria Pannell
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, US
velrox77@mit.edu

Remi Kuba
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, US
rkuba@mit.edu

***Abstract*—Many 3D rendering and mapping tools utilize depth maps to capture the general topology of objects in images. In software, creating a depth map from two 640 x 360 images takes 3-4 seconds to produce. Our project sought to implement stereo vision and the sum of absolute differences (SAD) algorithm technique to render live depth calculations for incoming video. The two-camera system required two FPGA boards, with one of the boards streaming its frame data to the other. The main board interacted with memory to store and access the raw video frames, which were then used to calculate the optimal offset for each pixel using the SAD algorithm. This optimal offset was used to calculate a relative depth map that was fed into the video pipeline and HDMI output.**

***Index Terms*—sum of absolute differences, stereo vision, relative depth**

I. Introduction

Stereo vision is a process that can capture 3-dimensional information from digital images, calculating distance by comparing the output images of two cameras a set distance from each other. While this can be done using software, an FPGA board specifically programmed to handle the data and calculations in a streamlined manner can accomplish the task faster.

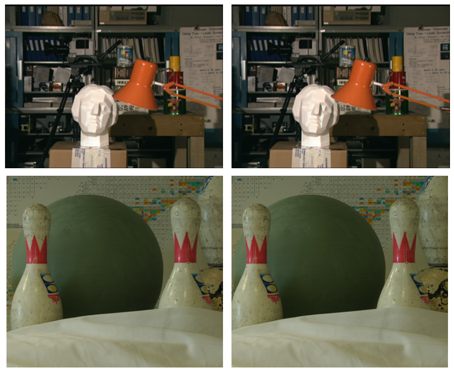
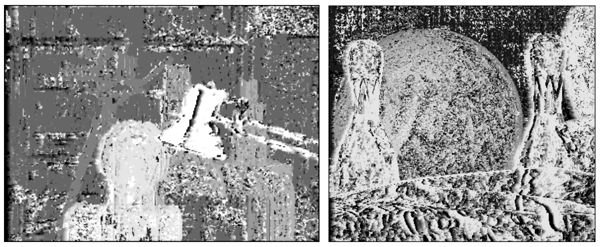


Fig. 1. "Left" and "Right" images (above) and their depth map outputs (below)



II. FPGA Communication (Remi)

A. Pixel and Frame Deconstruction and Reconstruction

Due to limited PMOD ports, our project required a primary and secondary FPGA, which communicated using SPI protocol (II-B). The pixel size and frame dimensions were limited by the interaction between boards, and based on calculations and tests in software [1], it was determined that 640×360 video frames with 8-bit luminance pixel data was sufficient and necessary to create live depth mapping.

B. SPI Protocol

It was originally calculated that to achieve a video rate of 25 fps, the bits would have to be sent at a rate of 46.08 Mbps ($(640 \cdot 360)$ pixels/frame$\cdot 8$ bits/pixel$\cdot 25$ frames/second). However, after initial testing, we discovered that, due to the nonuniform pattern of incoming camera pixels, a buffer was also necessary to prevent new camera data from overwriting the data being sent between boards. After reconstruction, the camera frames were sent into BRAM. A specialized counter was then implemented to guarantee requests for a new pixel to send would not overwrite the current pixel being processed.

The SPI controller was set to $16.\overline{6}$ MHz and parallelized the 8-bit pixels onto four wires. There was an additional wire that would go high on the last pixel of a frame for storage purposes later in the pipeline.

While the final product used 320×180 video, the SPI modules successfully sent and received 640×360 and 1280×720 video between the secondary and primary FPGAs. The larger frames required the BRAM buffer between camera and SPI controller to be switched out for a DRAM buffer. At $16.\overline{6}$ MHz, the SPI controller was able to send the desired video over the FPGAs for the 320×180 and 640×360 videos (25 frames/second $\cdot$ $(640 \cdot 360)$ pixels/frame $\cdot$ 8 bits/pixels $\cdot$ 0.25 cycles/bit = 11.52 MHz $< 16.\overline{6}$ MHz). But for the 1280×720 video, the SPI clock limited the the frame rate to around 9 frames per second ($16.\overline{6} \times 10^6$ cycles/second $\cdot$ 4 bits/cycle $\cdot \frac{1}{8}$ pixel/bit $\cdot \frac{1}{1280 \cdot 720}$ frames/pixels ≈ 9 frames/second), so there was more visible tearing and artifacts on the HDMI output.

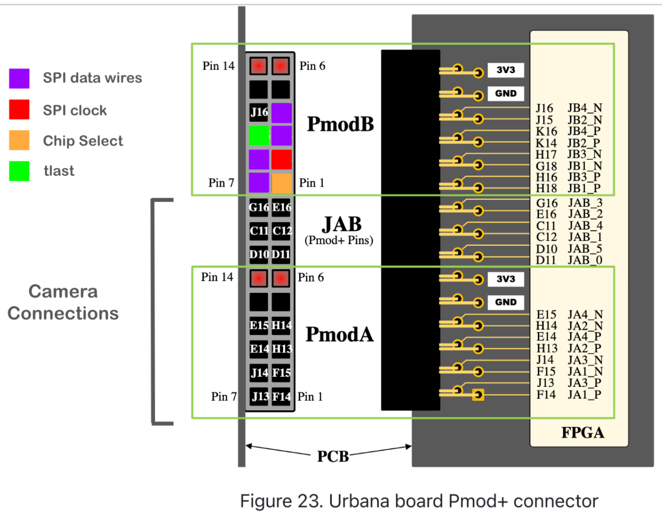


Fig. 2. PMOD SPI connections as specified by the modified XDC constraints file

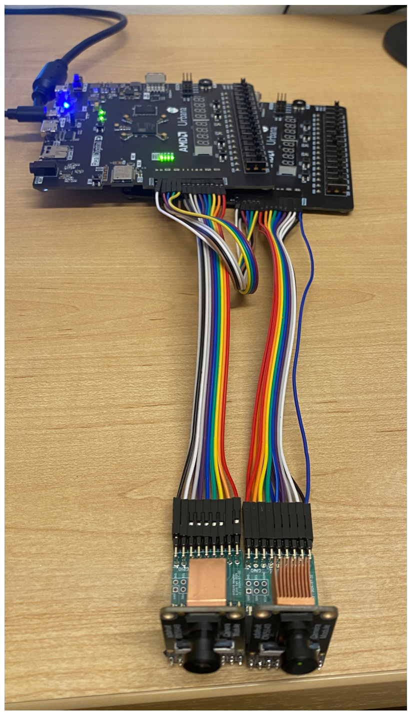


Fig. 3. FPGA setup and PMOD wiring between boards

III. Memory (Veloria)

A. On-Chip Memory Breakdown

We planned to use DRAM for a majority of the memory of our project, which guaranteed enough storage to save multiple higher resolution camera and depth video frames. In terms of BRAM, it would have originally only been utilized by the primary FPGA in the depth calculation and FIFOs. The left and right camera's line buffer modules maintain four BRAMs each (4), similar to the convolution lab. There would have also been be six FIFOs into and out of the traffic generator (5), which would require an additional six BRAMs.

However, we ended up using BRAM to store the video frames because of issues with interfacing with DRAM and utilized around two-thirds of the board's BRAM blocks (see section VI). For the primary FPGA, there were three frame buffers—one for the SPI camera, one for the onboard camera, and a final buffer for the SAD output. The total memory required for these frame buffers was 1382.4 kbits (320×180 pixels/frame $\cdot$ 8 bits/pixel $\cdot$ 3 frames), which was less than half of the available BRAM resources on the board.

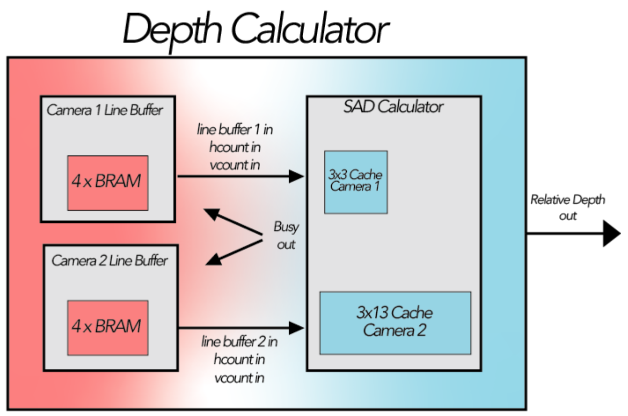


Fig. 4. Communication between the 2 camera inputs and the SAD Calculator

B. DRAM and Traffic Generator

For the bulk memory storage of camera and screen data, we are using the ISSI IS43TR16640CL-125JBL Off-Chip DRAM, a one Gigabit DDR chip, or double data rate chip. In order to interface with the chip, we are using Xilinx's MIG (Memory Interface Generator) IP, which has 16 parallel line connecting it to the DRAM. With the default MIG configuration, the UI clock interfacing with DRAM is run at 81.25 MHz, 4 times slower than the DRAM clock. Taking the DDR into consideration, we can send data 8 times along the 16 wires, or 128 bits of data, in one cycle.

The system requires 720,000 write requests to store the two images (2 frames $\cdot$ (640 $\cdot$ 360) pixels/frame $\cdot$ 8 bits/pixel $\cdot$ 25 frames/second / (128 bits/request)), or 360,000 requests per FIFO. The depth map calculations must be done on 25 frames per second, so it will require 720,000 read requests and 360,000 write requests to access the camera data then store the depth results back into DRAM (see previous calculation). Finally, to output the video at 60 fps, the last FIFO must make 864,000 read requests per second ((640 $\cdot$ 360) pixels/frame $\cdot$ 8 bits/pixel $\cdot$ 60 frames/second / (128 bits/request)). The total requests are far below the MIG's maximum of one request on every cycle of its 81.25 MHz clock, so our memory should be fine.

We updated the traffic generator logic from a previous lab to take in the requests from six, rather than two, FIFOs. It would have sent different data to the read and write inputs of the MIG based on the state, i.e. reading or writing from one of the cameras or getting data to or from the SAD module.

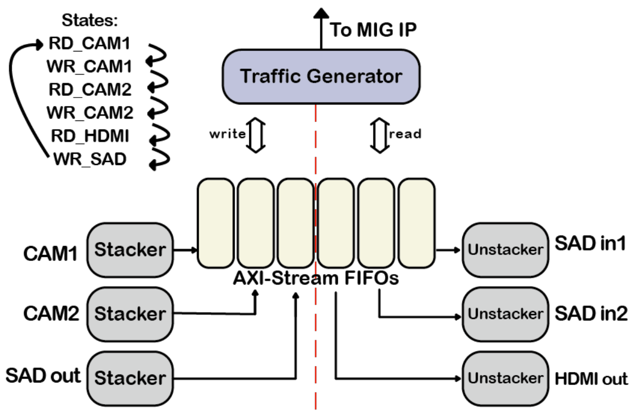


Fig. 5. Traffic Generator Block Diagram with 6 FIFOs & State Cycle

These FIFOs are fed into through stacker and unstacker modules to accumulate the data into larger 'chunks'. Through our debugging, we actually caught an error in the originally given stacker.sv file, correcting it to improve synchronization with a reset that correctly reflected the number of stored chunks and the tlast signal.

C. Components for Memory Access

The traffic generator instantiates three event counter modules in order to separately track the three sets of data stored in different locations / at different addresses in memory (Camera 1 input, Camera 2 input, and SAD Depth output). In each state, the data is passed through AXI-Stream FIFOs, which helps handle crossing the three main clock domains, and the state is looped-through round robin style to alternate reading and writing from data evenly.

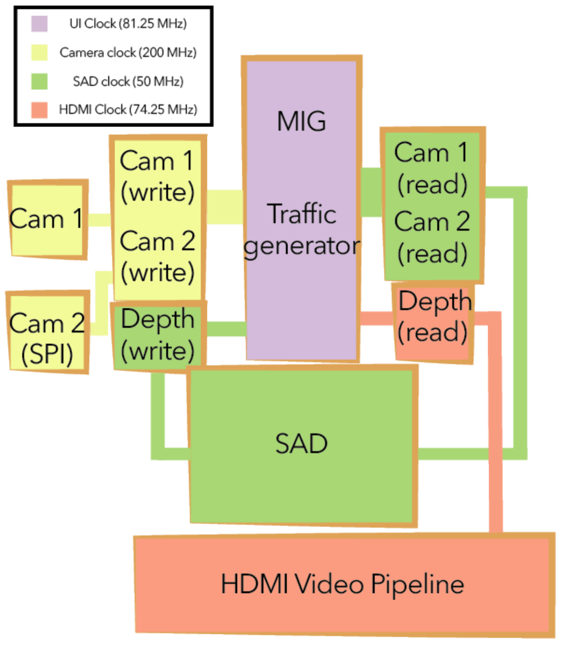


Fig. 6. Clock speeds / domains across the system

In order to input or retrieve data from Distributed RAM, the traffic cycles through a series of states that switches between reading and writing 3 sections of memory, each representative of a pair of inputs and output accessing the same memory locations. For one frame of 640x360 pixels, 1,843,200 bits are sent per frame, requiring 14,400 read requests of 128-bit messages (230,400 pixels · 8 bits/pixel ÷ 128 bits/message).

D. Implementation

In the top level, there is an instance of the Xilinx MIG IP to communicate with the DDR off-chip DRAM, the inputs and outputs of which are managed by the traffic generator, utilizing AXI Communication protocol with ready/valid signal handshakes to interact with the MIG and FIFOs. The state variable within the traffic generator selects between read and write FIFOs, and the input/output wires link to the 6 FIFOs. Unless a FIFO fills up, one frame (14400 · 128-bit messages) are sent every cycle. These input FIFOs are loaded by stackers that help assemble the packets of data received while a different bus is reading or writing data to the DRAM. Likewise, the outputs are put through unstacker modules to feed into subsequent stages when requested. However, our final implementation forgoes the DRAM and traffic generator because of outputting issues. As stated previously, instead, the camera frames are stored in separate BRAM buffers, the SAD module reads from both buffers and stores its output into another BRAM buffer, and the video path reads from the SAD buffer before outputting the HDMI video data (10).

IV. Depth Map Construction (Remi)

A. SAD Algorithm

If set up properly, the two cameras will have matching y positions. To determine a point in the image's true position $p = (x, y, z)$, the point's depth, z, must be calculated (7). Variables f and b are constant while x_l is chosen and x_r must be determined. To calculate the corresponding pixel (x_r, y) from the right image, the block matching sum of absolute differences (SAD) algorithm will be used (1). The SAD algorithm calculates the absolute difference between the left pixel and the right pixel shifted by a varying horizontal offset, sums the absolute differences for a window centered around (x, y) of size $k \times k$, then selects the horizontal offset that produces the smallest $SAD(x, y)$ for a given (x, y) (2).

There are two general formulas for calculating the depths. The true depth, $z = f(\frac{b}{best\_offset})$ (3), requires dividing by the determined offset. However, our project will utilize the disparity—the inverse of depth—to calculate the relative depth (4). The relative depth multiplies the max pixel size (8 bits means a max pixel size of 255) by the proportion of the best offset to the max offset, which we have chosen to be 10.

$$SAD(x, y, offset) = \sum_{i=-\lfloor \frac{k}{2} \rfloor}^{\lfloor \frac{k}{2} \rfloor} \sum_{j=-\lfloor \frac{k}{2} \rfloor}^{\lfloor \frac{k}{2} \rfloor} |I_l[x+i, y+j] - I_r[x+i-offset, y+j]| \tag{1}$$

$$best\_offset(x,y) = \underset{n \in [0, max\_offset]}{arg\,min} (SAD(x,y,n)) \quad (2)$$

$$depth(x,y) = f * \frac{b}{best\_offset} \quad (3)$$

$$relative\_depth(x,y) = 255 * \frac{best\_offset}{max\_offset} \quad (4)$$

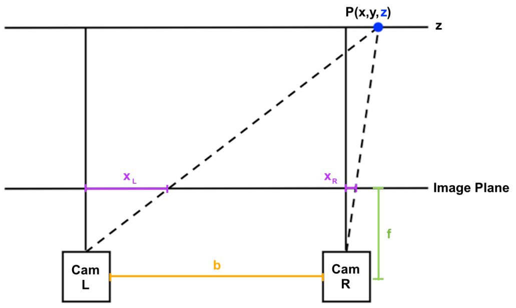


Fig. 7. f is the cameras' focal length, b is the baseline distance between the cameras, and z is our unknown depth. [2]

B. Implementation

The depth calculation was implemented with a multi-cycle, convolution-inspired module.

The tests and modules used a 3×3 kernel for the calculations, so the data input were two line buffers of three pixels each. The line buffers corresponded to the left and right camera frames, respectively. The left camera data input was then stored in a 3×3 cache while the right camera data input was stored in a 3×13 cache. Depending on the current offset of the SAD calculation, different 3×3 blocks from the right cache were compared with the left 3×3 block.

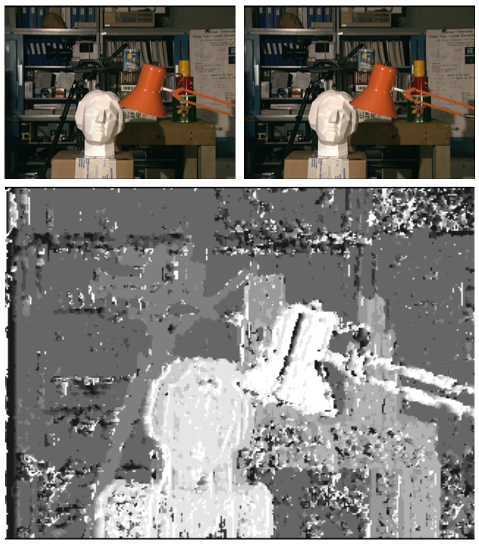


Fig. 8. Simulated depth map output using Tsukuba images

The original intent was to calculate each SAD block in one cycle on the 74.25 MHz pixel clock, but the indexing into the larger 3×13 cache caused timing issues. So instead, we clocked the SAD calculations at a generated 50 MHz clock, and each calculation was pipelined. The final implementation iterated from 0 to the max offset of 10, with each SAD block calculation taking two cycles. On the 22nd cycle, the relative depth was calculated and outputted. Figure 8 shows the simulated output from the depth map modules after inputting the two stereo images. Due to the significant decrease in throughput by increasing the cycles per calculation and decreasing the clock frequency, the resulting video did include unwanted artifacts from previous frames that were yet to be updated.

V. Video Pipeline

A. Outputs

The resulting depth map from a frame was stored in a BRAM buffer and accessed to enter the video pipeline. We wrote a simplified pipeline to account for the smaller frame and pixel size. The output was a 320×180 video stream using HDMI. We included logic in the buffer write ports to stop the storage of the camera data stream. By doing this, the FPGA would essentially capture a still depth map picture. There was also logic included to switch from the depth map video to the simultaneous stream from both cameras.

VI. Design Evaluation and Project Insights

The final implementation met timing requirements and did not run into resource utilization issues. The primary FPGA utilized 64% of the BRAM blocks. The secondary FPGA utilized 49.33% of the BRAM blocks, which could have been reduced had we not included a video path on the board for debugging purposes.

There were unexpected issues with the updated traffic generator that led us to use BRAM to store the frames, and the size limitation of BRAM meant that we had to reduce the stored video to 320×180 rather than the intended 640×360 dimension. Additionally, the lower throughput from the SAD calculations to meet timing affected the clarity of the video output, but the effects were not too noticeable.

The depth map output was also extremely sensitive to lighting. Running the SAD calculations in lab with the bright overhead lights resulted in a very noisy output. Softer and dimmer lights seemed to reduced noise within sections of the video that were of the same depth. Figure 9 shows the difference between a single bedside lamp being turned on and off. You can see that the wall in the top left corner was far more uniform, which was expected, when the lamp was off. There are many different approaches to calculating depth, and it is likely that a more complex algorithm like semi-global matching would have been more immune to lighting changes.

We managed to meet our commitment goals by 1) successfully communicating between the FPGAs and outputting simultaneous video and 2) building a functioning depth map on static frames. We partially met our regular goals by outputting video depth map for a 3x3 kernel with a max offset of 10 (although it was on 320×180 video instead of 640×360). We also partially met a stretch goal by increasing the resolution of the frames sent over from SPI from 640×360 to 1280×720.

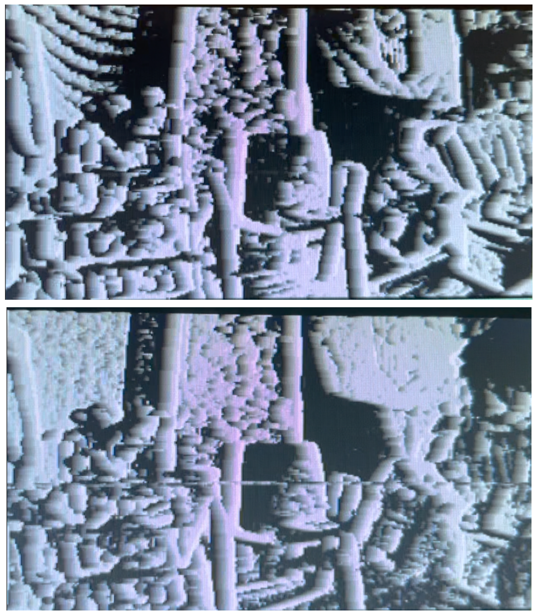


Fig. 9. Depth otuput of a dorm room with a lamp on (top). Depth output of same dorm with with lamp off (bottom)

VII. Source Code

[redacted]

Acknowledgment

Thank you to our TA, Kiran Vuksanaj, for the endless help and guidance throughout the project (and for putting up with the constant requests for various ROM files). Additional thanks to the rest of the 6.2050 staff that helped us during office hours and helped develop the lab code, which we utilized.

References

[1] D. Christian (2021) Simple SSD Stereo [Source Code]. https://github.com/davechristian/Simple-SSD-Stereo/tree/main

[2] J. Lambert, "Stereo and Disparity," johnwlambert.github.io. https://johnwlambert.github.io/stereo/

VIII. Block Diagram

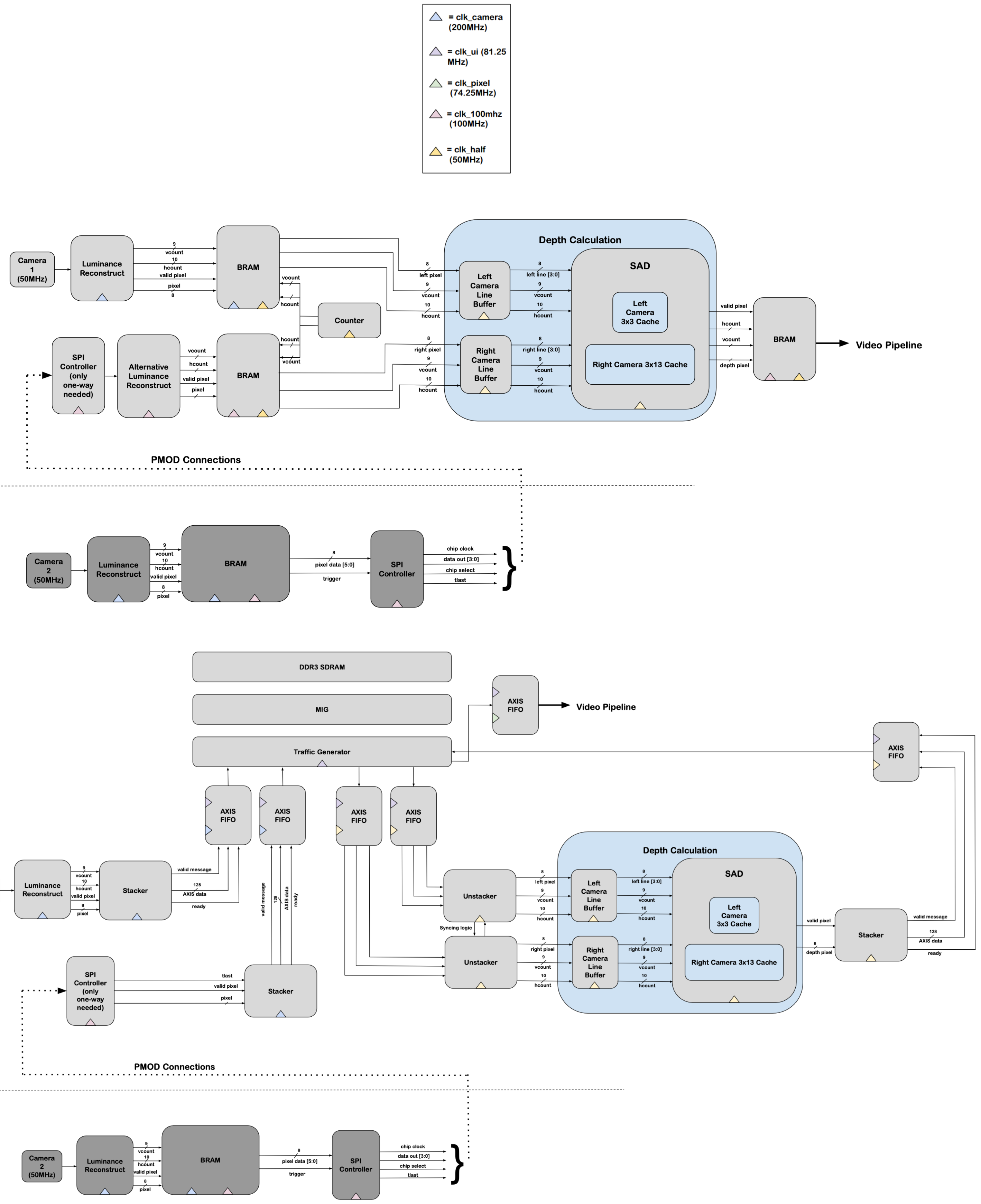


Fig. 10. Block diagram using BRAM for memory storage (top, current implementation) and block diagram using DRAM for memory storage (bottom, original implementation)