6.111 Project Report

Brian Axelrod, Amartya Shankha Biswas, Xinkun Nie [*]

Contents

1 Introduction **3**

2 Systems Design **4**

2.1 Filtering 5

2.2 Rectification 5

2.3 Census Transform 5

2.4 SGM Cost Calculator 5

3 Design Methodologies **6**

3.1 Standard Interfaces 6

3.1.1 AXI4 interfaces 6

3.1.2 AXI4-Stream Video 7

3.1.3 IP CONTROL 8

3.1.4 AXI4 Master 9

3.2 Block Diagrams 9

3.3 Verilog IPs 9

[*]MIT, Cambridge MA 02139. E-mail: baxelrod, asbiswas, xnie@mit.edu.

3.4 Vivado HLS 10

4 Memory Subsystem 11

4.1 Simple DMA 12

4.2 Axi Crossbar 13

4.3 Triple Buffer Controller 14

5 Camera Capture 14

5.1 Rectification 15

5.1.1 Getting the calibration parameters 15

5.1.2 Rectification in real-time 16

6 Pre-Processing and Feature Extraction 17

6.1 Gray-scale conversion 17

6.2 Windowed Operators 18

6.2.1 Line Buffer 18

6.2.2 Rolling Window 18

6.3 Census Transform 19

6.4 Gaussian Blur 20

7 Semi-Global Matching 20

7.1 Algorithm 21

7.2 Main Formula 21

7.3 Performance Analysis 23

7.3.1 Area Utilization 23

7.3.2 Latency and Throughput 23

7.4 Testing 24

8 Axi Compliant Modules and utilities 24

8.1 AxiVideo2VGA 25

8.2 Cam2AxiVideo 26

9 Conclusion 26

1 Introduction

Stereo vision is the process of extracting 3D depth information from multiple 2D images. This 3D information is important to many robotics applications ranging from autonomous cars to drones. Conventionally, two horizontally separated cameras are used to obtain two different perspectives on a scene. Because the cameras are separated, each feature in the scene appears at a different coordinate in both images. This difference between these coordinates is called the disparity and the depth of each point in the scene can be computed from its disparity. Computing the disparity at each point accurately and efficiently is quite difficult.

Algorithms for computing features between images are generally complex, memory inefficient and require random access to large portions of memory. The state of the art stereo matching algorithm is based on Semi Global Matching (SGM). This algorithm performs very well in in practice but is extremely memory and processing inefficient. This makes it difficult to process it on small computers that can fit on small robots like drones. Since FGPAs are fairly low power, an FPGA implementation of SGM would allow us to use SGM on small platforms such as drones.

SGM is not a natural streaming algorithm making it quite difficult to implement on an FPGA. Our goal was to develop and demonstrate an efficient implementation of SGM on an FPGA. This will require carefully redesigning the algorithm to fit an FPGA architecture. Finally we want to demonstrate our SGM implementation as part of a full stereo pipeline that can render 3d images.

Writing a complete stereo pipeline requires many diverse components ranging from filtering to a complicated memory architecture. Our goal was to demonstrate an entire working Stereo Vision system built around SGM.

Sections of this document were written according to which part of the system we worked on: Brian Axelrod was responsible for Sections 1, 2, 3, 4 and 9.

Amartya Shankha Biswas was responsible for sections 6 and 7.
Xinkun Nie was responsible for Sections 5 and 8

2 Systems Design

In order to be able to compute high quality disparity maps we must combine many complicated modules to compute SGM and pre and post processes our images. Our design decisions are primarily driven by the need to manage this complexity without sacrificing performance. Thus we establish a design pattern based on good software engineering patterns that have been adapted to the Vivado workflow. The main idea is that our design should be split into small manageable pieces that can be tested individually. We will leverage Vivado HLS and C++ test benches to quickly create thorough testbenches based on real data. We will also use standard streaming interfaces which will make it easy to replace modules and design tests. This will make it easy for us to understand exactly what we want to get out of a module and verify that it is correct. We will also use a softcore for running tests on the FPGA and running the state machine. This will allow us to use code that has been auto-generated by the Xilinx tools and avoid having to write and test more code.

Our design revolves around a pipeline for processing stereo images shown in Figure 1.

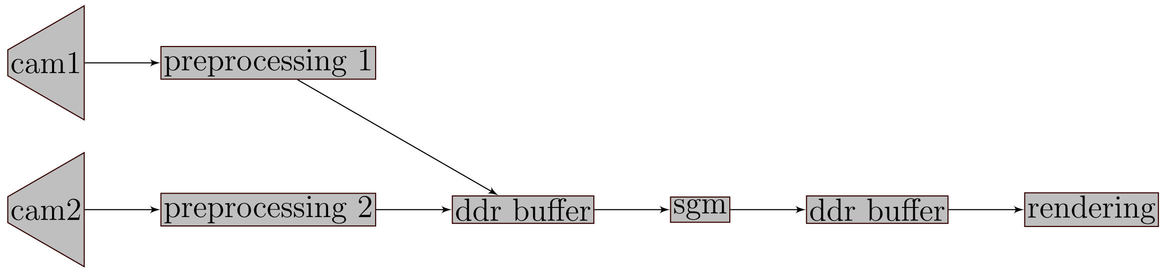


Figure 1: A high level overview of the design

The first part of the pipeline grabs frames from the cameras. It handles synchronization and passes on the results over an AXI stream that feeds into the preprocessing module. The preprocessing module applies the rectifications transformation, the Gaussian blur that mitigates the effect of noise, and applies a census transform to compute a value that describes the neighborhood of each pixel. The result is streamed into ddr memory through a Direct Memory Access (DMA). We then take the results and pass it through the SGM module twice, first in the forward direction and then in the reverse direction. Then the second part of the SGM module combines the information from these two runs to compute the disparity values and stores the results in ddr memory using a DMA. Then a rendering module reads the disparity values and renders them.

See the detailed block diagram in figure 2 for more information. Here's a list of modules in the detailed flowchart and a brief description of their purpose:

2.1 Filtering

In order to make our system more robust to noise we apply a standard technique in computer vision—applying a Gaussian blur. We apply a gaussian kernel to the image, essentially blurring it by making each pixel a weighted average of it's neighbors.

2.2 Rectification

To handle a camera's intrinsic optical distortions and extrinsic rotation and translation shifts, we plan to rectify the incoming images. The basic premise of most stereo algorithms is to find corresponding patches along epipolar lines. In a perfect world, these epipolar lines would simply be horizontal lines. Optical distortion bend the epipolar lines, which will be made to align with the horizontal axis after rectification.

We rectify the images by first calibrating the cameras off-line to get a rectification matrix. The streamed frames would then be multiplied by this matrix to get a rectified image.

2.3 Census Transform

We use the Census Transform to compute the matching cost over all pixels, which is a term in the SGM cost function that needs to be globally optimized. We use a 5x5 window to get information around each pixel to perform the Census transform.

2.4 SGM Cost Calculator

The SGM algorithm finds the optimal disparity value for each pixel by minimizing over a global cost function. The algorithm iterates through the pixels in two passes.

In the first pass, the iterator moves from left to right, and top to bottom in the frame. Only the line above the current line and the current line need to be stored in the DDR memory. For each pixel, we look at the pixel above it, right left to it, above and left to it, and above and right to it.

In the second pass, the iterator moves from right to left, and bottom to top in the frame. Only the line below the current line and the current line need to be stored in the DDR memory. For each pixel, we look at the pixel below it, right to it, right below to it and left below to it.

We compute the cost associated with each disparity value for the current pixel.

3 Design Methodologies

Since our design is very complex and involved many components we needed to adopt practices which allowed to manage complexity and contain risk. It become very important that we were able to design our components individually and plug them in and expect that they work. We adopted several design methodologies to help us achieve these goals. We used standard interfaces and a mix of block diagrams, verilog and Vivado HLS.

3.1 Standard Interfaces

In order to ensure the various modules in our design worked together we decided that all our modules would use standard interfaces. The inputs and outputs would be clearly defined according to industry standards which would resolve any ambiguity as to the specifications of the inputs and outputs of the modules. We decided that all our modules would conform to the following rules (defined in greater detail below):

- All video inputs and outputs must be AXI4-Stream video compliant
- They must use the standard IP CONTROL control interface
- Modules that interact with memory must be compliant AXI4 masters
- All other inputs must correspond to configuration and must remain constant

3.1.1 AXI4 interfaces

ARM defines a set of standards known as AXI4. These are standards for on-chip communication meant to make it easy for various modules in an FPGA or chip design to share data. These standards are very frequently used in FPGA designs because it allows modules to be reusable from design to design, and greatly reduces integration time.

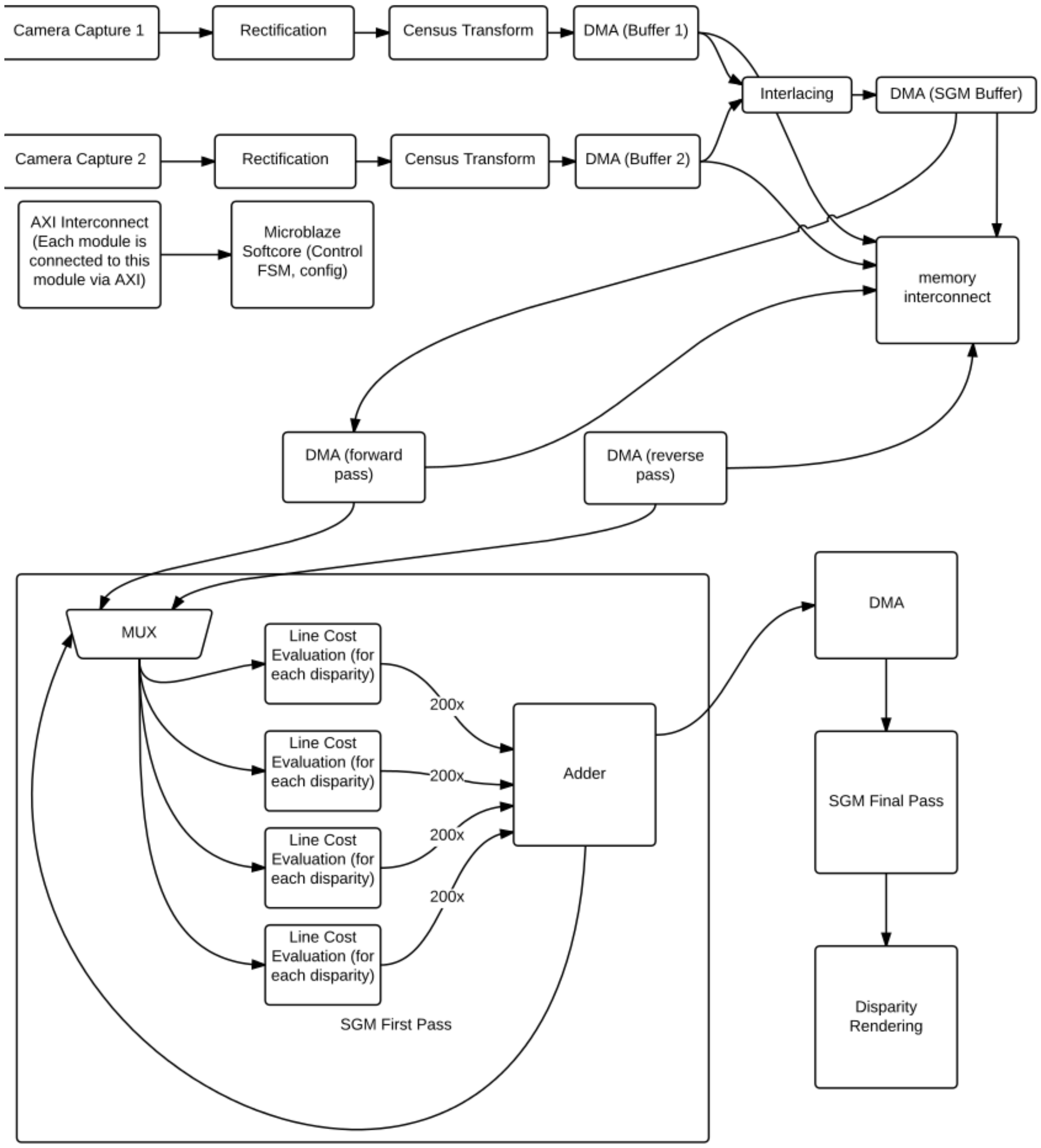


Figure 2: Detailed Block Diagram

3.1.2 AXI4-Stream Video

The AXI4-Stream Video interface is a slightly modified version of the AXI4 streaming interface. The AXI4 streaming interface is used for transmitting streams of data. The AXI4 streaming interface assumes that there is a master that is outputting data and a slave that is reading data. The master must provide a data bus, a valid signal and a last signal. The slave must provide a ready signal. When the master is ready to transfer a piece of the stream it pulls the valid signal high and sets the data register accordingly. If this is the last piece of the stream it also sets last to high. When the slave is ready to read the next piece of the stream it raises the ready signal. When both the ready and the valid signals are high the piece of the stream is consumed, i.e. the slave reads it, and the master moves on to prepare the next element in the stream. The timing diagram of AXI4-streams is shown in figure 3. Streaming interfaces are a very logical fit for FPGAs because they correspond to the inputs and outputs of streaming algorithms—algorithms which port very well to FPGAs.

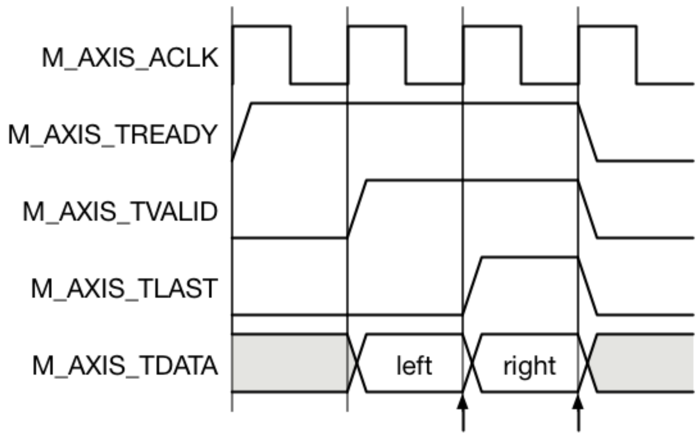


Figure 3: AXI4-Stream timing diagram. Image courtesy of `http://www.beyond-circuits.com/wordpress/wp-content/uploads/2015/04/tutorial18_axi4_timing4.png`

The AXI4-Stream Video interface is almost identical to the AXI4-streaming interface. In addition to the AXI4-Stream interface, the AXI4-Stream Video interface uses a user signal to indicate the start of the frame, and raises the line last value at the end of every line.

3.1.3 IP CONTROL

Many of our modules need to know when to start and be able to signal when they are done or able to accept new inputs. In order to standardize this we adopted the standard control interface used in Vivado HLS modules. Each modules would have a start input telling it when it should be active, and would have outputs corresponding to signal when the module finished processing the current set of inputs, when the module is ready to accept new inputs, and when the module is idle and waiting for new inputs. The modules must conform to the timing diagram given in figure **??**.

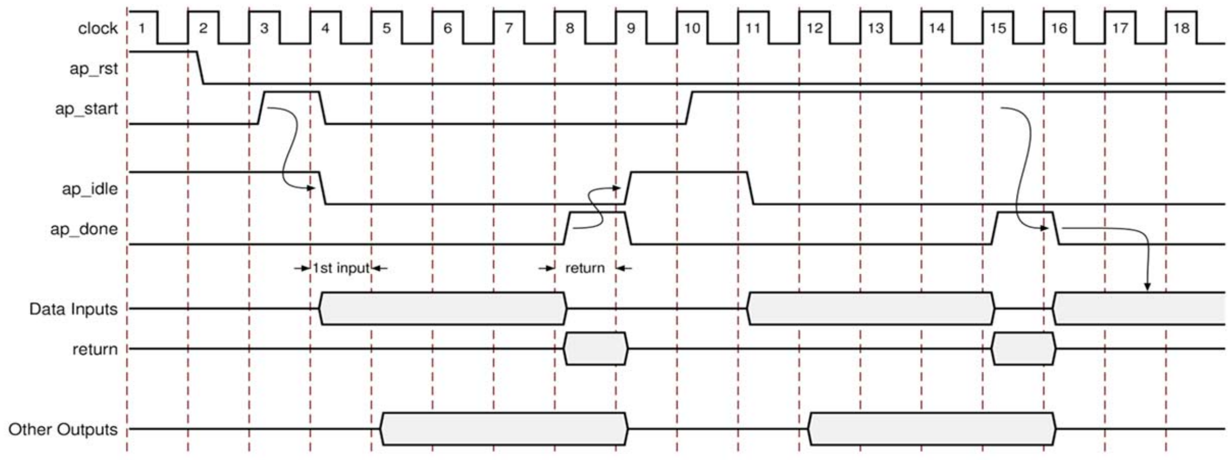


Figure 4: IP CONTROL timing diagram. Image courtesy of Xilinx UG902, figure 2-34

3.1.4 AXI4 Master

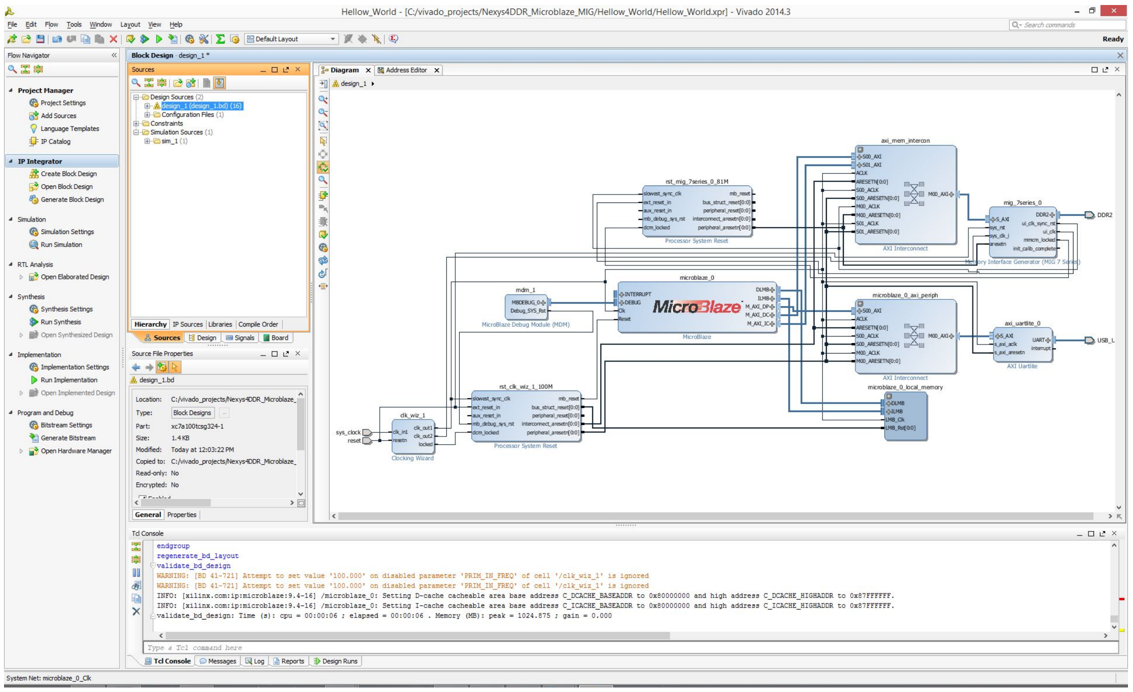
The most complicated interface used in our design was the full AXI4 interface. The AXI4 interface was used to communicate to the MIG and contains over forty signals, putting outside of the scope of the writeup. The full specification can be found on the ARM website.

3.2 Block Diagrams

Our design involved using many interfaces with a lot of inputs and outputs. If we consider just our 6 DMAs we already have more than 240 lines to connect. Connecting each of these inputs and outputs in human-written verilog is extremely time consuming and error prone. In order to avoid this source of error and make our design easy to ready we decided to use Xilinx block diagrams whenever connecting modules with complicated interfaces. In a block diagram each module shows up as a block connected to other blocks with wires. The key feature of block diagrams is that wires can be grouped together. In figure 5 all 42 wires corresponding to the S00_AXI port are all grouped together and displayed as one line. Block diagrams generate verilog which is later synthesized by Vivado and can be used in normal verilog designs.

3.3 Verilog IPs

Block diagrams do not always make sense. While it is easier to connect modules in block diagrams it is much more difficult to express complicated logic. As a result, we decided that



```
Location:   C:/vivado_projects/Nexys4DDR_Microblaze_
Type:       Block Designs
Part:       xc7a100tcsg324-1
Size:       1.4 KB
Modified:   Today at 12:03:22 PM
Copied to:  C:/vivado_projects/Nexys4DDR_Microblaze_
Read-only:  No
Encrypted:  No
```

```
endgroup
regenerate_bd_layout
validate_bd_design
WARNING: [BD 41-721] Attempt to set value '100.000' on disabled parameter 'PRIM_IN_FREQ' of cell '/clk_wiz_1' is ignored
WARNING: [BD 41-721] Attempt to set value '100.000' on disabled parameter 'PRIM_IN_FREQ' of cell '/clk_wiz_1' is ignored
INFO: [xilinx.com:ip:microblaze:9.4-16] /microblaze_0: Setting D-cache cacheable area base address C_DCACHE_BASEADDR to 0x80000000 and high address C_DCACHE_HIGHADDR to 0x87FFFFFF.
INFO: [xilinx.com:ip:microblaze:9.4-16] /microblaze_0: Setting I-cache cacheable area base address C_ICACHE_BASEADDR to 0x80000000 and high address C_ICACHE_HIGHADDR to 0x87FFFFFF.
validate_bd_design: Time (s): cpu = 00:00:06 ; elapsed = 00:00:06 . Memory (MB): peak = 1024.875 ; gain = 0.000
```

Figure 5: A simple block diagram in Vivado. Image courtesy of `https://reference.digilentinc.com/_media/vivado:mig_37.jpg`

most of our individual modules would be written in Verilog and we would use the Vivado tools to generate "blocks" based on our verilog. This allowed us to use the best of both worlds—the expressiveness of verilog and the maintainability of block diagrams. Examples of modules generated this way include the axi2vga module and the camera2axi module.

3.4 Vivado HLS

While verilog is quite capable capturing basic logic it lacks advanced features for generating complicated hardware programmatically—it relays on the programmer to build all the hardware. This makes seemingly straightforward hardware such as adder trees that compute the sum of many variables very time intensive to construct. Since SGM is a complicated algorithm we decided to use Vivado High Level Synthesis (HLS) to generate verilog for our most complicated modules.

In general we implemented streaming algorithms in Vivado HLS. In order to generate one of these complicated modules we would first design a streaming algorithm. We would then write a C++ implementation of this algorithm that closely mirrors how we would write it in verilog. We then annotate our C++ code with special keywords that instruct the Vivado HLS tools how to convert our C++ code to verilog. We then write testbenches and run RTL simulation to verify that the generated code behaves as expected.

4 Memory Subsystem

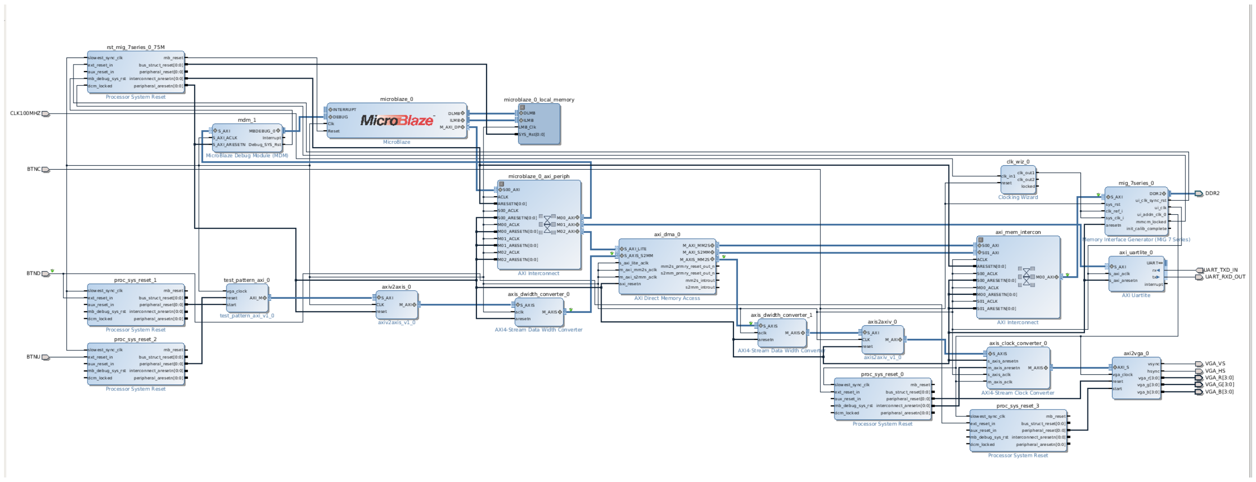


Figure 6: The block diagram of our memory subsystem using primarily Xilinx IPs. This didn't work due to issues with the MIG.

Our original design (shown in figure **??** relied on using Xilinx IPs to process much of the memory subsystem. This IPs rely on using a microblaze to configure the settings of the IP, and thus can only realistically be used in a block diagram setting. However, we were not able to generate a working memory interface generator (MIG) within our Xilinx block diagram, even when copying over all the settings from Weston's sample MIG. We were surprised that this was an issue, since in the past Brian Axelrod had always used a vendor configured MIG and never had any issues. Unfortunately there is no project file for a vendor configured MIG for the Nexys 4 DDR board. Furthermore the Digilinc board files do not work with the provided constraint file. Our development was greatly complicated by the fact that some resources provided by digilinc did not work as it became unclear as to which resources we could rely upon.

Our project failed primarily because we dedicated too much time and resources to getting the block diagram MIG. We spent a very large amount of time debugging the generated MIGs with integrated logic analyzers, testbenches, Xilinx memory tests, and our own custom memory test. The friday before the project was due we decided to use a Nexys4 Board with cellular RAM instead of DDR ram since cellular RAM is easier to interface with. We quickly discovered that the Digilinc provided board files and constraints file were again inconsistent. While we did attempt to make the two consistent, we decided that this was not likely to lead to a working configuration in a short period of time. At that point we decided to do everything ourselves and use a modified version of the MIG in Weston's non-block diagram project.

A diagram of our custom memory subsystem can be found in figure **??**. The memory subsystem consists of a direct memory access (DMA) which reads and writes streams to and from memory, an AXI crossbar which serves as an arbitrator allowing many DMAs to read/write from a single MIG, a controller which coordinates the various DMAs and the MIG itself which provides an interface to the DDR memory.

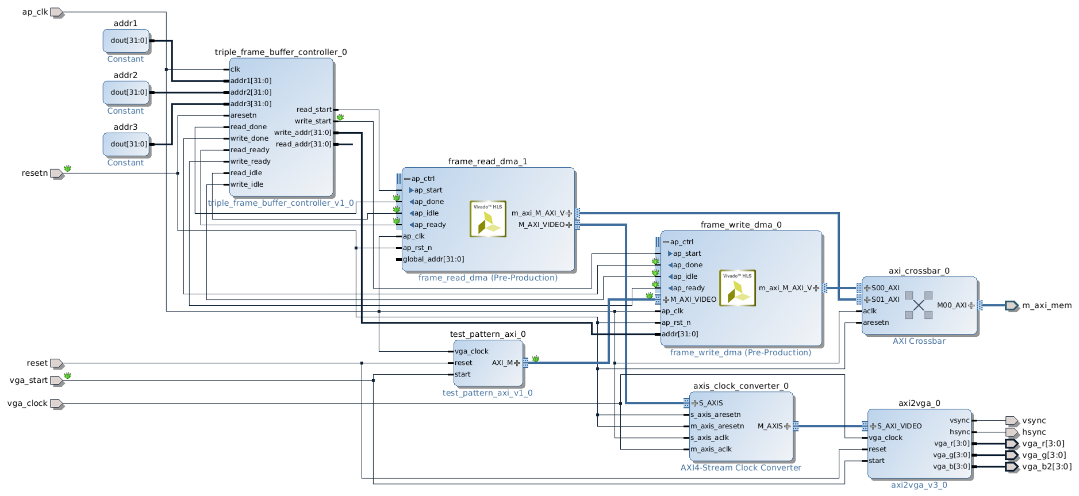


Figure 7: The block diagram of our custom memory subsystem with a three triple buffer

4.1 Simple DMA

Our direct memory access module (shown in in figures 8, 9) was designed to be simple to debug and thus provides significantly less functionality than the Xilinx DMAs. It is designed only to read frames or write frames from a configurable address in memory. They are controlled with a start port, and provided status information in terms of an an idle, done and ready signal. They speak to memory as an AXI4 master and comply to the AXI4 specifications provided by ARM. They read and write compliant AXI4 video streams which are used by the remaining modules in our design.

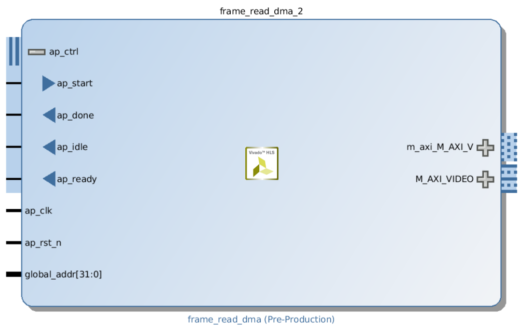


Figure 8: The block diagram of our custom memory subsystem

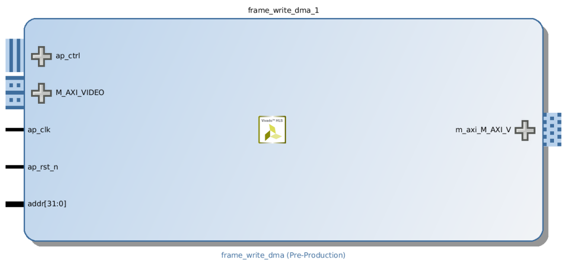


Figure 9: The block diagram of our custom memory subsystem

4.2 Axi Crossbar

Since our design necessitated using many DMAs which share a single MIG we needed a module which ”shares” access to the MIG in a safe manner. This module was responsible for arbitration, i.e. sharing the single MIG between the many DMAs. This modules allows us to use as many DMAs as we want—a significant advantage over Weston’s reference design.

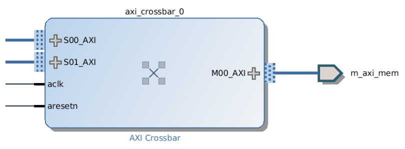


Figure 10: AXI4 Crossbar

4.3 Triple Buffer Controller

Rendering often requires a memory structure known as a triple buffer. A VGA display must be rendered at a fixed rate, whereas the input image often becomes available at a different rate. This can lead to a phenomena known as tearing where the image displayed on the screen does not correspond to a single frame. The standard solution for this problem is the use of a triple buffer, which contains three ”slots” for frames. One of these frames is always being written to, one is always being read from and on frame is kept as a reserve to allow the the input channel to store its results in memory without overwriting the previous frame. As input the triple buffer module takes the addresses of the three frames, the status signals of a write and a read DMA. A triple buffer has outputs corresponding to the control lines of a read and write DMA. It also tells the DMAs which addresses in memory they should be reading and writing. A rendering of our triple buffer is shown in figure 11.

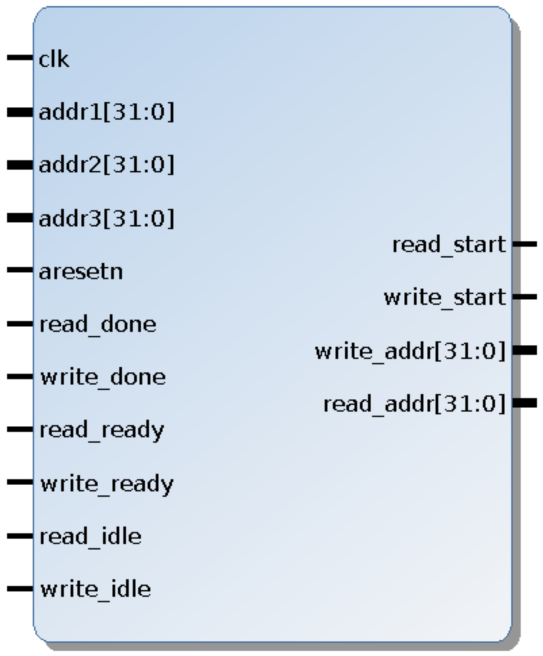


Figure 11: Triple Buffer Controller attached to read and write DMAs

5 Camera Capture

The camera capture module is based off Lab Assistant Weston’s module to output the camera data. The difference between his module and our need is that we need to have two cameras, both of which need to be captured. The first camera is connected to the JA and JB ports on the Nexys4 board, and the second camera is connected to the JC and JD ports on the board. Both cameras share the same clock output, because there is only one input port on the FPGA that can handle clock signals. Both cameras are driven by the same input clock.

We have successfully been able to switch between the two camera captures using a switch on the Nexys4 board.

5.1 Rectification

5.1.1 Getting the calibration parameters

In order to perform rectification of the image in real-time, calibration parameters are needed for the rectification task. We achieve this by running a Matlab script

(http://www.vision.caltech.edu)

to generate the calibration parameters.

In order to get an image, we decided to store one frame of the image in a microSD card for off-line computation. I spent approximately two weeks on this part of the project. After much help from Lab Assistant Jono, I was able to read and write to a microSD card. I had some trouble reading the microSD card information on a computer, because the microSD card is not formatted, and only has raw data. Eventually, I was able to display the microSD card information in a hex editor on my computer.

I also had trouble writing different values to neighboring bytes to the microSD card. The microSD card can be written to 512 bytes at a time (after asserting the write signal high for one clock cycle). To be able to write each individual byte, the "ready for next byte" signal out of the microSD card controller needs to go high before the writing happens. I did not realize that there is no specification on how long the "ready for next byte" signal keeps HIGH. It turned out I needed to catch its rising clock edge and update the din register (which keeps the data to write to the microSD card).

The other issue I encountered is that I couldn't seem to be able to write to the first block of 512 bytes to the microSD card. When I tried to write an entire camera frame worth of data (640 x 480 x 2 bytes), the first block of 512 bytes couldn't be written to. The issue turned out to have to do with the non-blocking assignment. In clock cycle 1, wr signal is low, ready signal is high, and then we do write HIGH to wr, and change the state register to a writing state, which is a state in which we write to the microSD card. The wr signal doesn't go high until the end of current cycle, so the ready signal doesn't see wr has been turned HIGH until the next clock cycle. The ready signal can only go low after a clock cycle's delay. Since I increment the address to the next block of 512 bytes for the microSD card by checking to see if the ready signal is HIGH or LOW. Having such a delay had the effect of skipping an entire block of memory.

I wrote a script in Python to generate an image from the raw byte data in the microSD card. The image we had captured looks like a corrupted image, for reasons I haven't found. After spending so much time to get the microSD card to work, we eventually ran out of time to capture a proper frame.

In retrospect, to capture a frame, I could have only used a grayscale of the image and capture that in BRAM. If we did it that way, we could needed to export the image to a serial connection or a microSD card, because the Matlab code needs to run offline on a computer to process the captured frame.

5.1.2 Rectification in real-time

I wrote a script in C++ that given the parameters, projects each pixel from the original image to a new pixel location in the rectified image. More accurately, it finds the matching pixel (which is usually a pixel location in fractions) and its surrounding neighbors, with its respective weight. The code involved a lot of arithmetic, understanding of the Matlab script, and translating it into C++. See appendix for the code used in this section. This code was used in Vivado High Level Synthesis (HLS) to perform rectification in real time.

6 Pre-Processing and Feature Extraction

We now use the rectified images to perform SGM (Section 7). The two incoming streams of rectified images are converted to gray-scale, low-pass filtered (Gaussian Blur) and Census transformed before being streamed into SGM (Figure 12).

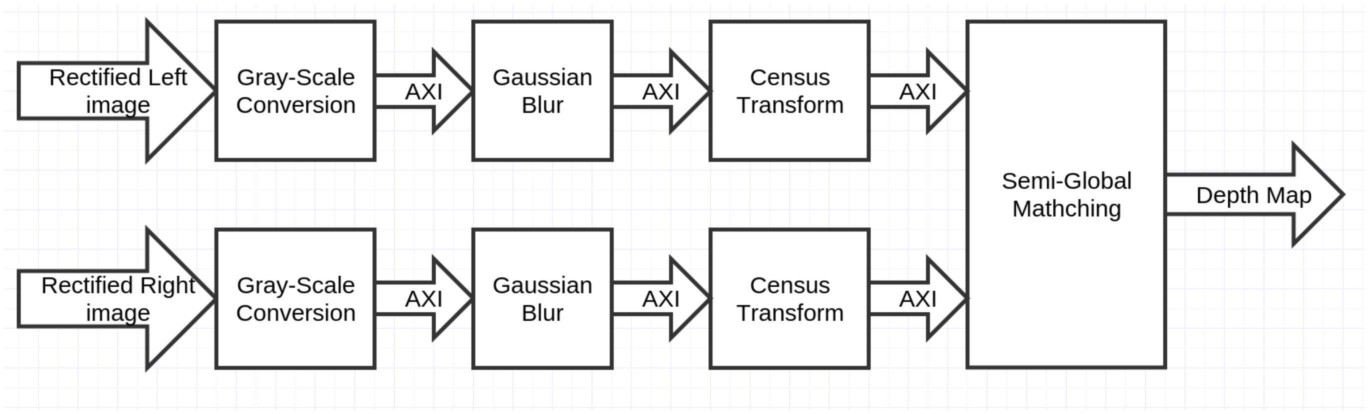


Figure 12: Data Flow

We get a stream of RGB pixels from the rectified images as input. First we convert both the images to gray-scale. This is because our feature descriptor only depends on intensity values.

Our first step before computing features is to low pass filter the image to reduce noise (Section 6.4). We can then compute features for each pixel and stream the features into the SGM module.

We use a *Rolling Window* to facilitate the convolution and feature transformations. This allows us to get good throughput by processing one pixel per clock cycle.

6.1 Gray-scale conversion

Our first step is to convert the incoming pixels to intensity values (gray-scale). The intensity value for a pixel is calculated from the RGB values as follows –

$$I = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B$$

The intensity values are then streamed into the next module to be low-pass filtered (Section 6.4).

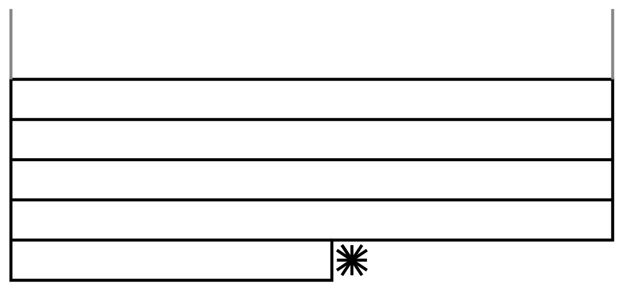


Figure 13: Line Buffer. The next pixel is $*$

6.2 Windowed Operators

We need to compute feature descriptors for our images. A feature descriptor of a pixel is just a description of its neighbourhood. We will use this description to match pixels between the left and right images. This is because two pixels are likely to be matched correctly if and only if they have similar neighborhoods. Since our module receive the pixels in a stream, we need to be able to maintain a neighbourhood for each pixel which is updated every clock cycle (as a new pixel streams in). Our feature descriptor uses a 5×5 window.

We also want to be able to compute one descriptor every clock cycle to maintain throughput. We achieve this by pipelining our computation.

A similar rolling window is used to perform a Gaussian Blur on the image.

6.2.1 Line Buffer

Our required window spans five columns. So, as the image streams in row by row, we always need to maintain a buffer of the last five rows of the image (Figure 13). We store these rows in five separate blocks of BRAM. These blocks are separate because we want to be able to read from all five rows concurrently.

When a new pixel on the current row streams in, we write it to the "*last*" block of BRAM. When we reach the end of a row, we start overwriting the oldest (lowest index) row still stored in BRAM. This way, we always maintain a buffer of the last five rows in BRAM.

6.2.2 Rolling Window

Now that we have buffer of the last five lines of the image, we want to have a rolling window that stores a 5×5 patch of the image. By *rolling*, we mean that every time a new pixel streams in, the window shifts to the right (Figure 14). This is performed by setting each

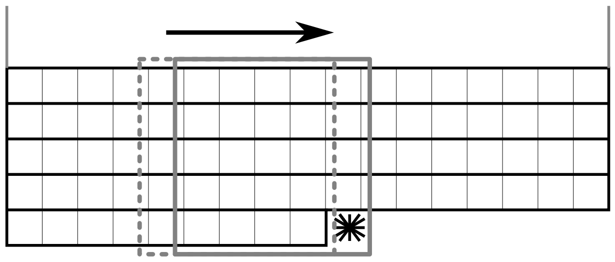


Figure 14: Window moves right. The next pixel is ∗

value in the window (except the rightmost column) equal to the value element to its right. The values in the rightmost column are simultaneously assigned values from the four blocks of BRAM (line buffer) and the incoming pixel.

After a row ends, the window shifts down and moves to the beginning of the next row. This is done by clearing the window and shifting the line buffer down

Since, these shifts happen every clock cycle, the window is implemented as a register array.

6.3 Census Transform

The Census Transform creates a feature descriptor for each pixel in the image. We use a 5×5 Census Transform. This creates a descriptor of the 5×5 pixel neighborhood of a pixel. Specifically, each pixel in the neighborhood is assigned a binary value which is 0 or 1 is the intensity of the pixel is greater or less than the intensity of the center pixel (Figure 15).

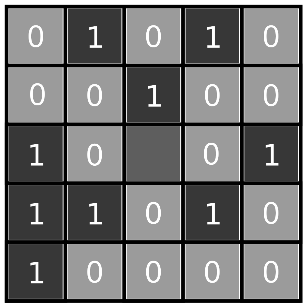


Figure 15: Census Transform Window. Pixels with intensity less than the center pixel get a value of 1 and pixels with intensity greater than the center pixel get a value of 0.

This set of 24 bits forms the census transform for the center pixel. So, each pixel produces a 24-bit descriptor.

So, we can now use the rolling window from Section 6.2.2 to calculate these 24-bit census features and them stream them into the *SGM* module.

6.4 Gaussian Blur

Before we compute the census features however, we want to minimize the amount of noise in the image. So, the first step is to low pass filter the images. We do this by using a Gaussian Filter which simply "*blurs*" the image.

Our Gaussian Filter works by convolving the image with a 5×5 kernel (Figure 16).

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figure 16: 5×5 Gaussian Kernel

Again, we can use the rolling window from Section 6.2.2 to convolve with the kernel, and stream the blurred image into the Census Transform module.

7 Semi-Global Matching

We want to reconstruct a 3D depth image from two stereo camera inputs using Semi-Global Matching. This involves matching corresponding pixels between the two images.

This gives us a **disparity** value D_p for each pixel p, where D_p is the difference in the position of the pixel across the two images. The 3D depth of each pixel can then be computed from it's disparity. Figure 17 shows a pair of stereo images and the depth map computed during RTL simulation.

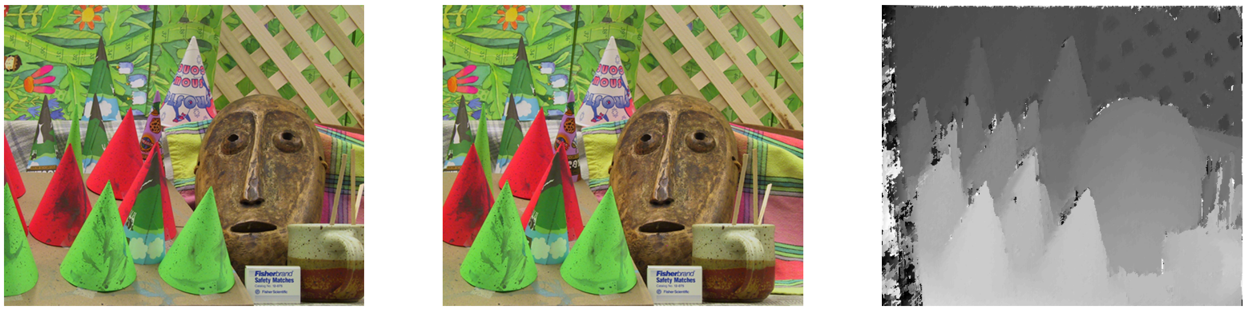


Figure 17: Left image, Right image and computed Depth Map

7.1 Algorithm

Semi-Global Matching uses dynamic programming to minimize a global cost function along the epipolar lines. Unlike other dynamic programming methods, it does not re-curse only along the epipolar lines. Instead we perform the minimization along four directions (Figure 18).

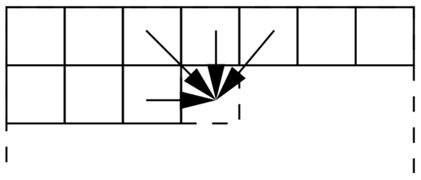


Figure 18: Dynamic Programming from four directions

7.2 Main Formula

We use the 5×5 Census Transform as a metric to assign cost values $C(p,d) = \|I_L(p) \oplus I_R(p-d)\|$ to each pixel p and disparity value d. Here I_L and I_R are the values of the Census Transform and the cost is calculated as the Hamming Distance i.e. we define how similar two pixels are as the number of positions at which their feature descriptors differ. Then we define the cost of each path ending at a pixel as $L_r(p,d)$. where d is the disparity value at pixel p, and r is one of the eight directions. $L_r(p,d)$ is computed according to the recurrence –

$$\begin{aligned} L_r(p,d) = C(p,d) + \min\{ & L_r(p-r,d), \\ & L_r(p-r,d-1) + P_1, \\ & L_r(p-r,d+1) + P_1, \\ & \min_i\{L_r(p-r,i) + P_2\}\} - \min_k\{L_r(p-r,k)\} \end{aligned}$$

In our design, we are using disparity values from 0..63 i.e. our disparity range is 64. We need to calculate the current pixel's value of L_r for each disparity value using the previous L_r values

The third term $(\min_i\{L_r(p-r,i)+P_2\}\}-\min_k\{L_r(p-r,k)\})$ is the most resource/computation intensive, but it is independent of the value of the value of d. We use a minimizer tree to calculate this value. Figure 19 shows a minimizer tree (with depth 3) which minimizes eight values. In the actual implementation, we are minimizing over all disparity values (64), and our minimizer tree has depth 6.

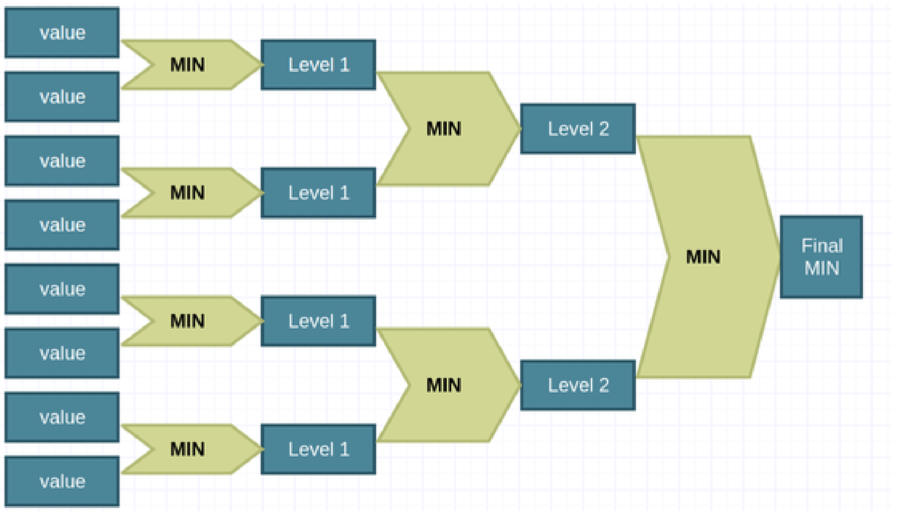


Figure 19: Minimizer Tree for eight values. Depth = 3.

We use a minimizer tree because it's easy to pipeline. The tree uses a large number of registers, but it can be pipelined at each level. So, the same minimizer tree can be used to minimize different sets of values every clock cycle. This also allows us to reduce our throughput by pushing through a different set of values for the next pixel every cycle.

The other terms in the expression are small minimizations which depend on the disparity values being computed. these are all computed in parallel and pipelined to improve throughput.

Finally, we perform the overall minimization over the four calculated values which gives us $L_r(p, d)$ for all disparity values and all directions for the current pixel.

After the L_r values are calculated, they are aggregated to find the overall cost, $S(p, d)$ value for the corresponding pixel.

$$S(p, d) = \sum_r L_r(p, d)$$

Then we use a final minimizer tree to find the disparity d for which the cost $S(p, d)$ is minimized. The disparity value gives us the calculated depth of the pixel. This is then streamed out to be rendered on the display.

The complete minimization has a latency of 14 cycles.

7.3 Performance Analysis

7.3.1 Area Utilization

We need to store the $L_r(p, d)$ for each pixel in the preceding line. The design uses a significant amount of BRAM to store all the $L_r(p, d)$ values. For a certain pixel, we need to access the L values for each disparity and each direction simultaneously. So, these are stored in separate blocks of BRAM.

We need to partition the L_r values to make efficient use of the BRAM. Since our computation has a latency of 15 cycles and one pxel s computed every cycle, we would be accessing two $L_r(p_1, d)$ and $L_r(p_2, d)$ from the previous row only when pixels p_1 and p_2 are in the same block of 14 columns (computation latency is 14 cycles).

So, we partition the L_r values for the previous row into 20 blocks this number needs to be a factor of the number of columns to prevent wraparound errors) in a cyclic manner (Figure 20).

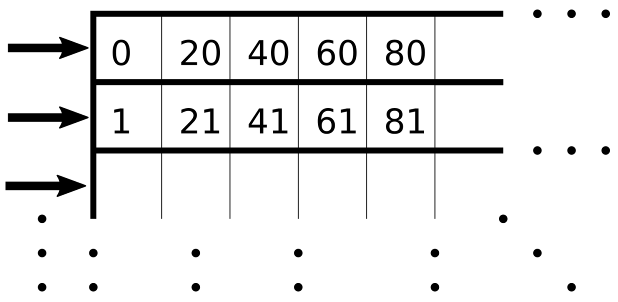


Figure 20: Partitioning L_r cyclically into BRAM. The arrows represent blocks that are never accessed simultaneously

This allows us to save overall BRAM usage. The overall design uses $\approx 50\%$ of the available BRAM on the Nexys 4 board.

7.3.2 Latency and Throughput

The following modules process the incoming image streams (Figure 12) –

- Gray-scale Conversion
- Gaussian Blur
- Census Transform

- SGM

Each module generates a stream, that is used by the the next module. The modules are connected by AXI (streaming) interfaces which allows different modules with different amounts of latency to work synchronously. The overall latency is the sum of all the individual latencies. This is however insignificant because we are processing $\approx 10^5$ pixels.

Each module processes one pixel every clock cycle. This is also the overall throughput. Assuming a conservative 10 nanosecond clock, this gives us a frame-rate greater than $100Hz$ which is faster than the VGA refresh rate (60Hz).

7.4 Testing

The sequence of modules was thoroughly tested using RTL Simulation. C code was used to generate the rectified input AXI streams.

All the separate modules (gray-scale, Gaussian filter, Census Transform, SGM) were tested by running RTL simulation. The output image stream was rendered using opencv.

After integration, the entire system was tested with five sets of rectified images and RTL simulation produced valid depth maps (Figure 17 and Figure 21).

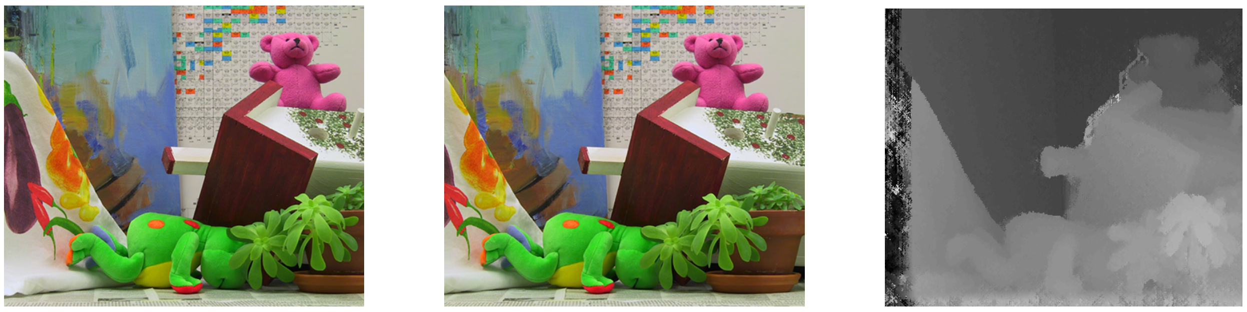


Figure 21: Left image, Right image and computed Depth Map

8 Axi Compliant Modules and utilities

Our design called for every module using our standard interfaces. For several modules this meant doing something that we had done previously, except for making it AXI compliant this time. This includes the AxiVideo2VGA module and the Cam2AxiVideo module. We also write conversion modules that allowed standard AXI4-Stream modules to interface with

AXI4-Stream Video modules. This would have allowed us to use Xilinx DMAs with our modules.

8.1 AxiVideo2VGA

This is a rendering module that reads from an AXIS4Video Stream and displays the stream to the VGA. The AXIS4Video Stream includes several data lines, including tuser (pulse signal of the start of a frame), tlast (pulse signal of the end of each line in a video), tdata(a data bus with configurable width), and tvalid (whether tdata is valid).

One complication we have encountered is using the AXIS4 Video Stream interface. The slave module that reads from the AXIS stream and writes to the VGA must be robust to the master module that produces the AXIS stream. The master module might have hiccups, such that the data will be misaligned when read from the slave module. Thus, the slave module must assert TREADY = LOW when TLAST from the master module is asserted HIGH. Basically, the slave module must wait until an entire line of a frame is read can it stop receiving. Otherwise, it is possible that the slave module stops reading, and the master module hasn't finish transmitting a line, which can make reading the next line corrupted by the previous line. This took several iterations and test benches for me to get it right. The module is therefore robust to input hiccups on the per line level of the video stream.

Another complication we have encountered is the robustness issue with regards to the per-frame hiccups. It is possible that the tuser signal is asserted HIGH in the AXI Master module when the Slave module is in the middle of rendering a frame. If we let the slave module keep rendering, the current frame would be reading the next frame, and the next frame would also get corrupted. This is a similar issue to the per-line robustness issue. I addressed it by keep TREADY=LOW when tuser is asserted high in the middle of a reading a frame.

This module took a long time to write and test, mostly because I was not aware of the importance of compiling to the standard AXI interface. Our initial spec did not compile to the standard interface. My teammates and I changed the spec for this module at least 4-5 times because we encountered new issues when we moved onto other parts of the project and needed to use this module to render. This module is also particularly difficult to test. Despite the fact that I have made testbenches for this module and the testbenches show that my code meets the spec and solve the two issues above, it is difficult to test it on hardware. I wrote a test pattern image generator that is AXI compliant and used it to test this module, which works fine. The success of this particular test, however, does not necessarily mean the module is flawless, because the test pattern is a static image and the test pattern generator behaves consistently (with no hiccups, etc.). It turns out that this module failed to render images properly when connected to Brian's module that reads an image from memory.

8.2 Cam2AxiVideo

This is the module that uses the camera output as the input, and outputs an AXI compliant output stream. I used Lab Assistant Weston's camera reader, which outputs a valid pixel value every other clock cycle, because a pixel value is 16-bit, and the camera output is 8-bit – which means it takes two clock cycles for each pixel to stream out valid data. Besides the camera data, the AXI outputs several AXI-specific data lines, including TUSER, TLAST, TVALID, which are asserted HIGH for one clock cycle at which each pixel's value has become valid and when the specific points in frames are reached (TUSER: start of frame, TLAST: end of line, TVALID: data is valid).

This module has also been tested with a testbench.

9 Conclusion

While our project failed it failed in a way that was surprising to me. The highest risk component, the memory subsystem was demonstrated working in hardware. The second highest risk component, SGM, was tested very rigorously in simulation. In fact our SGM implementation exceeded expectations and has performance comparable to the state of the art. The main factor behind our failure to deliver a complete working system is the failure of the AxiVideo2VGA module—a very simple module. It was not tested rigorously and was clearly not up to specification. Unfortunately this was discovered during integration and we did not have enough time to rewrite or fix the module before the deadline. However, if the MIG had been working as advertised we would have had sufficient time to address this issue.

Even though things did not work out as expected many things went surprisingly well. The systems design allowed each individual to work on his/her own with very clear specifications and goals. Integration time was also negligible (incredibly rare for an FPGA design of this complexity), and we were able to very quickly discover the failure point. We were able to build our own, working, highly performant, memory subsystem that is simple and easy to use. We were able to prevent a lot of issues by using good design practices.

A fair argument could be made that our failures had nontechnical causes. We failed to enforce discipline in testing the modules we wrote. While many modules were incredibly well tested and worked as expected, our design ended up failing because of an untested module. This of course could have been prevented if we had more time. We allocated too much time towards trying to get a MIG working in a block diagram. In hindsight these could have both been fixed with better project management. Our project was better suited for a four person team with three technical members and one manager that made sure that the team was disciplined in their testing and could push for a change of direction when a component did not seem likely to work.