STEREO VISION SYSTEM MODULE ON LOW-COST FPGAS FOR AUTONOMOUS MOBILE ROBOTS

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Connor Citron

August 2014

COMMITTEE MEMBERSHIP

TITLE:	Stereo Vision System Module on Low-Cost FPGAs for Autonomous Mobile Robots
AUTHOR:	Connor Citron
DATE SUBMITTED:	August 2014
COMMITTEE CHAIR:	Professor John Seng, Ph.D., Department of Computer Science
COMMITTEE MEMBER:	Professor Franz Kurfess, Ph.D., Department of Computer Science
COMMITTEE MEMBER:	Professor Chris Lupo, Ph.D., Department of Computer Science

Abstract

Stereo Vision System Module on Low-Cost FPGAs for Autonomous Mobile Robots

Connor Citron

Something, something, robots. that

ACKNOWLEDGMENTS

I would like to especially thank my parents and family for their love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Introducing ...

Chapter 2

Background

This chapter presents some general information on stereo vision that be useful for understanding the decision that were made in developing this stereo vision system.

2.1 Computer Stereo Vision

Computer vision is concerned with using computers to understand and use information that is within visual images [15]. There are many different types of computer vision, which range from using one image to multiple images to obtain information. One image is not enough to determine the three dimensional properties of the objects within the image.

Stereo vision uses multiple images of the same scene in order to construct a three dimensional representation of the objects in the images [12]. Comparing multiple images together for their similarities and differences allows for the depth to be obtained.

Binocular stereo [19] involves comparing a pair of images. These images are normally acquired simultaneously from a scene. By searching for corresponding pairs of pixels between the two images, depth information can be determined [19]. Pixel based comparisons can require substantial amount of computational power and time. Certain assumptions are made because of the computational resources

required. Camera calibration and epipolar lines [cite 14-14 and define better] are common assumptions. For example, two images of the same scene are 640 x 480 pixels in size. Each image therefore contains 307,200 pixels, which is over 600,000 pixels between the two images for one frame. For a real-time application, say 30 frames per second for example, that becomes over 18 million pixels between the two images that would need to be processed every second.

Computational requirements for real-time applications can be reduced in several ways. First, by lowering the number of pixels in the images reduces the number of pixel comparison per second. Images at a size of 320 x 240 pixels would require a quarter of the number of computations at the cost of losing some amount of detail in the images. Also, reducing the number of frames per second will decrease the amount of computing needed. Going much below 30 frames per second is noticeable to a person and can be annoying to observe a slow frame rate. A robot on the other hand, depending on its task and how fast its moving, might only need a few frames per second in order to function within a desired range. So image resolution could be more important than frames per second for a robot if details are more important than speed.

Figure 2.1 below represents a simplified illustration of binocular stereo vision. The two cameras are held at a known fixed distance from each other and are used to triangulate different the distance of objects in the images they create. The points U_L and U_R in the left and right images, respectively, are 2D represents the point P that is in 3D space. By comparing the offset of between U_L and U_R in the two images, it is possible to obtain the distance of point P away from the cameras [1].

The closer an object is to the stereo vision system, the greater the offset of corresponding pixels will be. If an object is too close to the system, it is possible

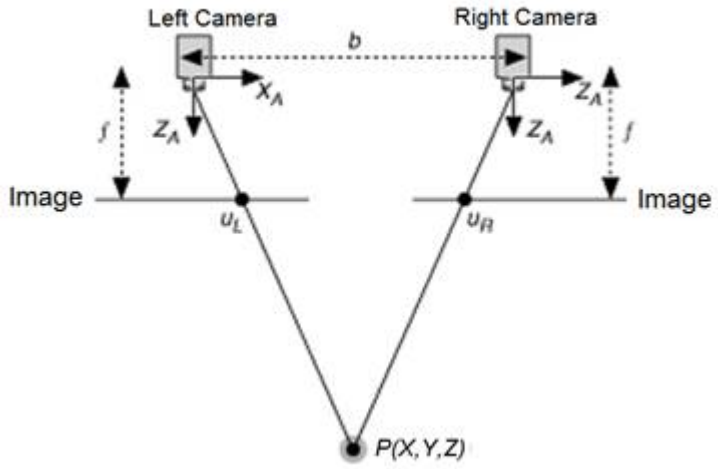


Figure 2.1: Simplified binocular stereo vision system [1].

for one camera to see part of an object that the other camera cannot. The farther an object is away from the stereo vision system, the smaller the offset of corresponding pixels will be. If an object is far enough away, it is possible for an object to be in almost the exact same location in both images. You can show this to yourself by holding a finger up close to your face, close one eye, and then alternate between which eye is open and which eye is closed. Your finger should appear to move a noticeable amount. Next, hold your finger as far away from you as you can and again alternate between which eye is open and which is closed. You should notice that your finger appears to move significantly less than it did when your finger was close to your face. That is how stereo vision works. The distance of an object is inversely proportional to the amount of offset between the two images.

2.1.1 Parallelism in Stereo Vision

Processing images for stereo vision allows for a high degree of parallelism. Locating the corresponding position of a pair of pixels is independent of finding another corresponding pair of pixels. This independent nature allows for the ability to process different parts of the same images at the same time, if there is hardware to support it.

Field Programmable Gate Arrays (FPGAs) allow for parallel processing to be implemented of the images. In Section 5 the amount of parallel processing used for the modular stereo vision system presented in this paper is discussed.

2.2 Stereo Vision Algorithms

Stereo vision algorithms can be placed into one of three different categories: pixel-based methods, area-based methods, and feature-based methods [10]. Pixel-based methods utilize pixel by pixel comparisons. They can produce dense disparity **(define!)** maps, but at the cost of higher computation complexity and higher noise sensitivity [10]. Area-based methods utilize block by block comparisons. They can also produce dense disparity maps and are less sensitive to noise, however, accuracy tends to be low in areas that are not smooth [10]. Feature-based methods utilize features, such as edges and lines for comparisons. They cannot produce dense disparity maps, but have a lower computational complexity and are insensitive to noise [10].

There are a lot of stereo vision algorithms out there [22]. In the taxonomy of [22], 20 different stereo vision algorithms were compared against each other using various reference images. Many algorithms are based on either the sum of

absolute differences (SAD) or correlation algorithms [18].

An algorithm that is similar to SAD is Sum of the Square Differences (SSD). Both of these algorithms produce similar results and contain around the same amount of error [10]. SAD was chosen over the other algorithms to implement because it is simpler to implement in hardware. SSD requires squaring the difference between corresponding pixels and summing it up. Since squaring a number is the number multiplied by itself, the number will be added to itself that many times to produce the squared value. This is a lot more over head, and more hardware, than just taking the absolute difference of the difference of each corresponding pair.

2.2.1 Sum of the Absolute Differences Algorithm

SAD is a pixel-based matching method [18]. Stereo vision uses this algorithm to compare a group of pixels called a window from one picture with a window in another picture. The SAD algorithm, shown in Equation 2.1 [18], takes the absolute difference between each pair of corresponding pixels and sums all of those values together to create a SAD value. One SAD value by itself does not give any useful information about those two corresponding windows. Several SAD values will be calculated from different candidate windows for each reference window. Out of the all the SAD values calculated for the reference window, the SAD value with the smallest value (all of them are positive because of the absolute part in the equation) is determined to contain the matching pixel. Figure 2.2 shows for one reference window, there are several candidate windows used. The line that

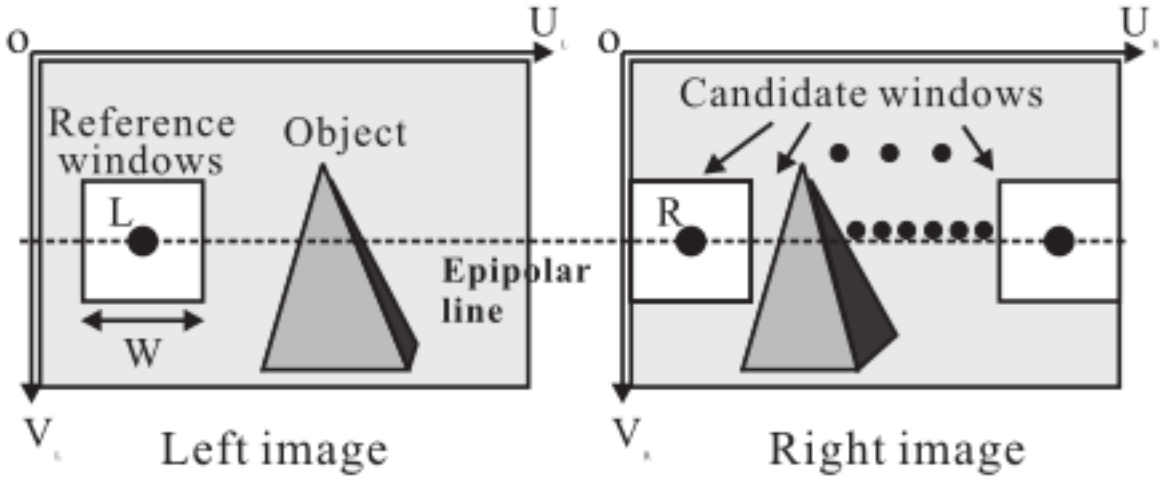


Figure 2.2: Searching for corresponding points between the two images [16].

the candidate windows move across are called epipolar lines.

$$\sum_{(i,j)\in W} |I_1(i,j) - I_2(x+i, y+j)| \tag{2.1}$$

In stereo vision, epipolar lines are created from the two cameras capturing images from the same scene. Figure 2.3 show the epipolar line that point X must be on in the corresponding images. This is useful because if the epipolar lines are known for both images, then it is possible to know the line that two corresponding points are on. It reduces the problem of finding the the same two points from a 2D area to a 1D line. Now, if the epipolar lines in both images are horizontal as they are in Fig. 2.2 as opposed to them being at a diagonal as they are in Fig. 2.3 then Eq. 2.1 reduces to Equation 2.2. For cameras that are not perfectly aligned, rectification is often used in order to align epipolar lines between images [17]. However, many stereo vision algorithms will assume that the epipolar lines are rectified to simplify the overall processing required.

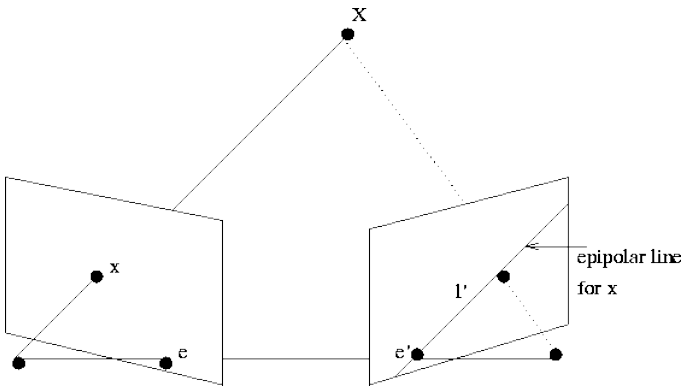
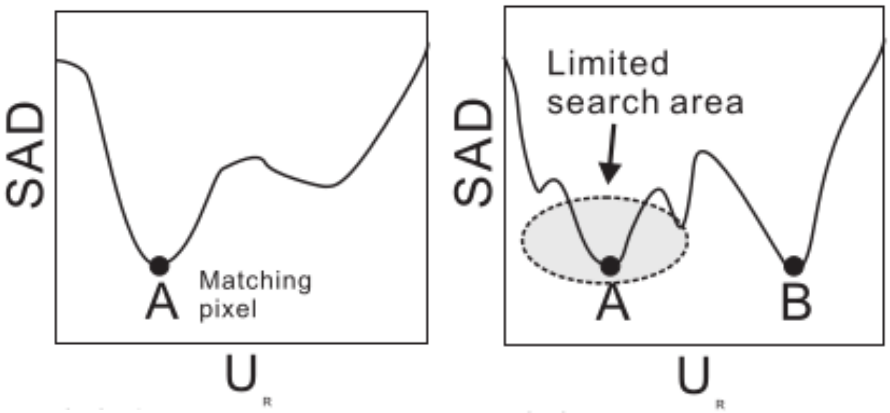


Figure 2.3: The epipolar line that point X is on for both images [8].

$$\sum_{(i,j)\in W} |I_1(i,j) - I_2(x+i,j)| \tag{2.2}$$

The disparity is the amount of offset between two corresponding pixels. The disparity range is the range that the candidate window will move through the image and is represented by the value 'x' in Eq. 2.2. It corresponds to the amount of SAD values that will be calculated. Figure 2.4 shows two types of SAD search methods. Fig. 2.4a selects the overall SAD value with the lowest value to be the matching pixel. However, Fig. 2.4b limits the search region to a specific area. This helps to avoid issues of similar looking areas that are not near the reference window from being falsely identified as matching. The downside to this is that if an object gets too close, meaning it would have high disparity values, and if the search region is not large enough, then the objects distance will be miss classified. It is important to determine a window size and a search region that are not too small and are not too big.

For example, Figure 2.5 shows a template window (Figure 2.5a) from one image and the search area (Figure 2.5b) in the other window. The disparity range is three, or zero to two. There are three 3x3 windows within the search



(a) Global SAD search (b) Local SAD search

Figure 2.4: The SAD between a reference window and several candidate windows [16].

region in Fig. 2.5b. From left to right the three search windows have the center pixel as 4, 6, and 5, respectively.

Comparing corresponding pixels in the template window with the first search window (let's call it S0) gives the absolute differences for all nine pixels going from left to right and top to bottom of 8, 1, 1, 2, 1, 0, 1, 2, and 2. So the SAD value for S0 of 18 is obtained by adding up all nine of those values. The SAD value for the second search window (S1) is 6 and the last search window (S2) is 13. The template window has the smallest difference between S1, which means that the center pixel in S1 is determined to be the corresponding pixel for the center pixel in the template window.

The disparity value is 1 (how far the matching search window was shifted to the left). The disparity value is used to create a disparity map. Each disparity value in the disparity map is at the same relative location that the center pixel of its corresponding template window is located.

1	2	3
4	5	6
7	8	9

(a) Template Window

9	1	2	4	5
2	4	6	5	3
8	6	7	8	7

(b) Search Region

Figure 2.5: Template (reference) window and search (candidate) window.

Chapter 3

Related Works

There are several different ways to implement a stereo vision system. Many stereo vision systems are implemented on field-programmable gate arrays (FPGAs). FPGAs allow for parallelization when processing images. Systems that use FPGAs generally can achieve a high frames per second on a decent or good image quality, but most of these systems are expensive.

FPGA Design and Implementation of a Real-Time Stereo Vision System [18] uses an Altera Stratix IV GX DE4 FPGA board to process the right and left images that come from the cameras that were attached to it. [18] uses the Sum of Absolute Differences (SAD) algorithm to compute distances. This system allows for real time speeds up to 15 frames per second at an image resolution of 1280x1024. However, the Altera Stratix IV GX DE4 FPGA board costs over $4,000, [4] which makes the system impractical for non-high budget projects.

Improved Real-time Correlation-based FPGA Stereo Vision System [14] uses a Xilinx Virtex-5 board to process images. [14] uses a correlation-based algorithm, which is based on the Census Transform, to obtain the depth in images. The algorithm is fast, but there are some inherent weaknesses to it. This system can run at 70 frames per second for images at a resolution of 512x512. Unfortunately, the Xilinx Virtex-5 board costs more than $1,000, [5] which is still quite expensive.

Low-Cost Stereo Vision on an FPGA [20] uses a Xilinx Spartan-3 XC3S2000

board. [20] uses the Census Transform algorithm for image processing. This allows images with a resolution of 320x240 to be processed at 150 frames per second. The total hardware for the low-cost prototype used in [20] costs just over $1,000, which is a bit too pricy for a lot of projects.

An Embedded Stereo Vision Module For Industrial Vehicles Automation [11] uses a Xilinx Spartan-3A-DSP FGPA board. [11] uses an Extended Kalman Filter (EKF) based visual simultaneous localization and mapping (SLAM) algorithm. The accuracy of this system directly varied with speed and distance of detected object. The Xilinx Spartan-3A-DSP FGPA board is around $600, [7] which is fairly expensive still.

Several commercial stereo vision systems exist presently [10]. Most of them are quite capable of producing good quality depth maps of their surroundings. However, the cost of these products can be relatively expensive, especially from a club or hobbyist standpoint. The Bumblebee2 [3] from Point Gray is able to produce disparity maps at a rate of 48 frames per second for an image size of 640x480, but it costs somewhere around $1,000 or so. Having been involved with the Cal Poly Robotics Club for 6 years and seen the budgets each project in the club gets, $1,000 would be most of their budget for the year. That kind of money could be better spent elsewhere on the project.

During the course of this thesis, a stereo vision surveillance application paper [21] was published that used the Digilent Atlys board [2]. A stereo camera module, VmodCAM [6], can be purchased with the Atlys board and was also used. The Atlys board is relatively cheap, at least by the standards presented thus far, at $230 for academic use. With the VmodCam included, the price goes around $350, which is still significantly cheaper than the other FPGA boards presented from other papers. This is why the Atlys board was selected for use in

this thesis (the selection was independent of the surveillance paper). The surveillance paper used the AD Census Transform to calculate distance. Their board output the disparity map data over HDMI to a monitor. The output image is rather noisy, but it is very easy for a human to understand what is in the image.

Chapter 4

Implementation

This chapter presents the implementation and architecture of this stereo vision system.

4.1 Architecture Overview

The stereo vision system module in this paper is composed of three main parts: SADs, minimum comparators, and a wrapper around the previous two that takes in image data and outputs disparity values.

The code for the following sections is located on github under: `https://github.com/cccitron/mastersThesis`.

4.1.1 Sum of the Absolute Differences Architecture

Two versions of the SAD algorithm have been implemented in this paper. The first uses a 9x9 window and the other one uses a 7x7 window. Figure 4.1 shows the top level entity of how the SAD implementation used. Both versions have a clocked input (clk_I) and a one bit data input (data_I) to notify the algorithm to begin calculating the SAD value. The tempalte_window_I and search_window_I between the two versions differ in the sense that the number of bytes, 49 or 81, sent to the sadAlgorithm entity are different. The data_O signals when the calculation is complete and ready that the algorithm is ready for the next set of

input. The calculated SAD value is sent out of the entity through sad_O.

There is a slight variation between the standard SAD algorithm and how it is implemented in this stereo vision system. Instead of subtracting two pixel values and then taking the absolute difference between them, the implementation in this paper first finds which corresponding pixel has a greater value and then sends the two pixels to the subtracter based on that. See APPENDIX (INSERT NEEDED) for the code used. The subtracter then takes the greater value and subtracts from it the lesser value and returns the difference. This difference, since it will always be greater than or equal to zero, will always be equal to the absolute difference of the two corresponding pixels. This process was implemented to reduce the complexity of using signed values and allowed for the number of bytes used for logic in the algorithm to be reduced by one bit.

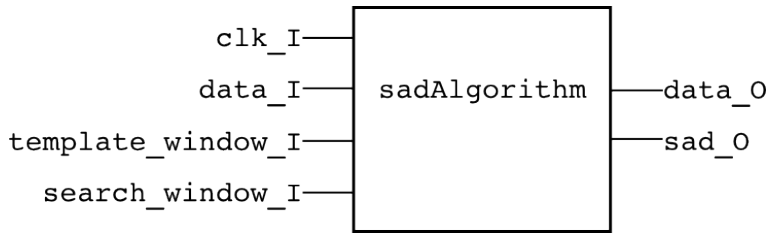


Figure 4.1: The top level SAD algorithm implementation.

4.1.1.1 State Diagram

Inside the sadAlgorithm entity from Fig. 4.1, the state machine from Figure 4.2 controls the SAD algorithm. The state machine begins at SO and initializes all the values used in it to 0. It then proceeds to S1 where the state machine remains on standby until data_I becomes '1'. In S2, the counter begins from 0, the subtraction between respective pixel values begins, and on the next clock

cycle, the state will be S3. While in S3, the counter is incremented by 1 every clock cycle. S3 is where the SAD algorithm is performed. After the counter is equal to windowSize (7 for the 7x7 and 81 for the 9x9, see the sections below for details), the SAD calculation is complete. The state machine sets data_O to 1 to notify the SAD wrapper that the calculation is complete and it then moves to S1 and waits for new input.

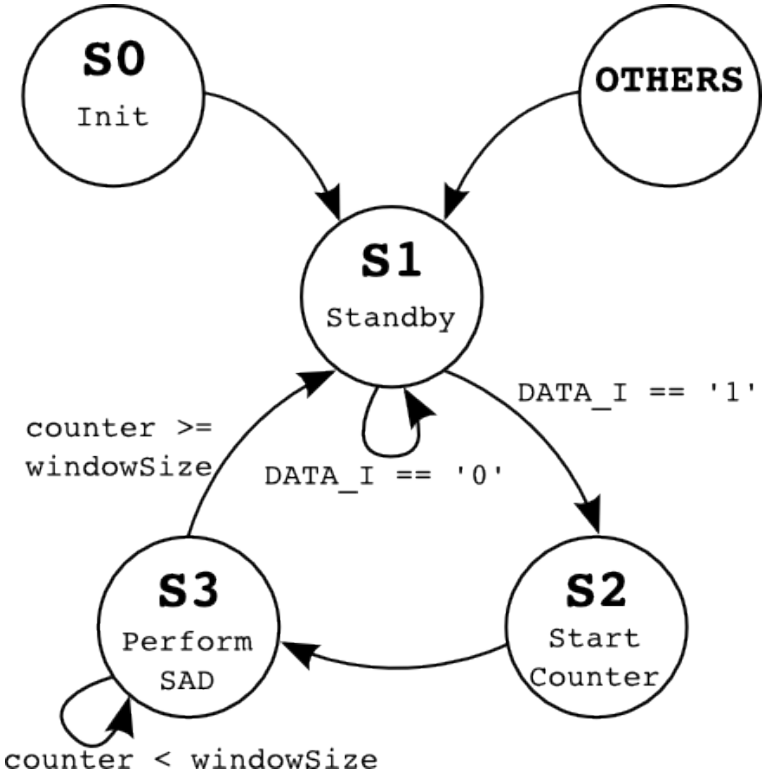


Figure 4.2: The state machine for implementing the SAD algorithm.

4.1.1.2 9x9 Window

The 9x9 window implementation operated with 4 pixels being processed in parallel to get their disparity values. Each pixel has 16 SAD operations occurring in parallel. With 64 SAD operations happening in parallel, each SAD calculation

needed to process their windows with a higher degree of serialization in order to reduce space to fit on the Atlys board [2]. Figure 4.3 shows a simplified version of this process. Each clock cycle, for 81 cycles, the difference between corresponding pixels is calculated. Beginning one clock cycle after the differences begin to be calculated, so there is a value, sub, to use. The sum_out is added to itself and sub. This process also occurs 81 times, one addition each clock cycle. Every clock cycle, a new "absolute" difference value is added to the sum_out. The state machine in Fig. 4.2 stops the calculation for sum_out after the full SAD value has been summed up.

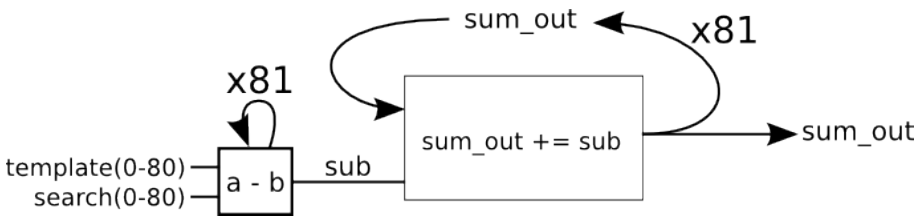


Figure 4.3: Architecture overview of the SAD algorithm with the 9x9 window implementation.

Figure 4.4 illustrates the pipeline used for the 9x9 window version. It takes 81 clock cycles to take all of the differences between all 81 pairs of pixel values. After the first difference is calculated, the differences can then begin to be summed up. The summing also takes 81 clock cycles and ends one cycle after the last difference is calculated. This results in a time of 82 clock cycles.

The code for the 9x9 window implementation can be found on github:
`https://github.com/cccitron/mastersThesis/tree/master/makestuff/libs/libfpgalink-20120621/hdl/fx2/vhdl/sad_simple_reg_9x9`

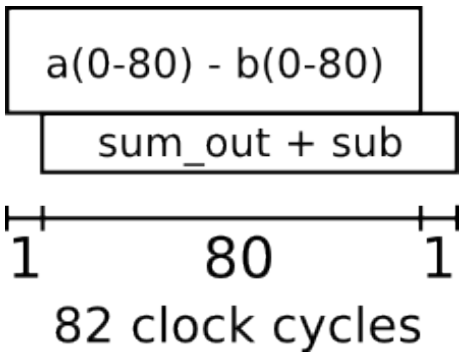


Figure 4.4: Pipeline architecture of the SAD algorithm with the 9x9 window implementation.

4.1.1.3 7x7 Window

The 7x7 window implementation operated with 2 pixels being processed in parallel to get their disparity values. Each pixel has 16 SAD operations occurring in parallel. With only 32 SAD operations happening in parallel, as opposed to 64 that were done in parallel in Section 4.1.1.2 and with a window size that has used 32 pixels for each window in each SAD calculation, the process could utilize a higher degree of parallelization, which takes up more space on the board than the more serial version from Sec. 4.1.1.2. Figure 4.5 shows a simplified version of this process. Each clock cycle, for 7 cycles, the difference between corresponding pixels is calculated. Beginning one clock cycle after the differences begin to be calculated, so there is a value, called sub, to use, the sum_out is incremented by sub. This process also occurs 7 times, one addition each clock cycle. Every clock cycle, a new "absolute" difference value is added to the sum_out. The state machine in Fig. 4.2 stops the calculation for sum_out after the full SAD value has been summed up.

The main difference between this implementation and the 9x9 window imple-

mentation from Sec. 4.1.1.2 is that the difference between corresponding pixels is parallelized to calculate 7 of them at once. The blue dotted square in Fig. 4.5 represents all 7 of the subtraction calculations occur 7 times in the SAD calculation. Instead of requiring 49 clock cycles to take all the differences, it now only takes 7 clock cycles. Accordingly, each clock cycles, all 7 differences, in the sub array, are added to sum_out.

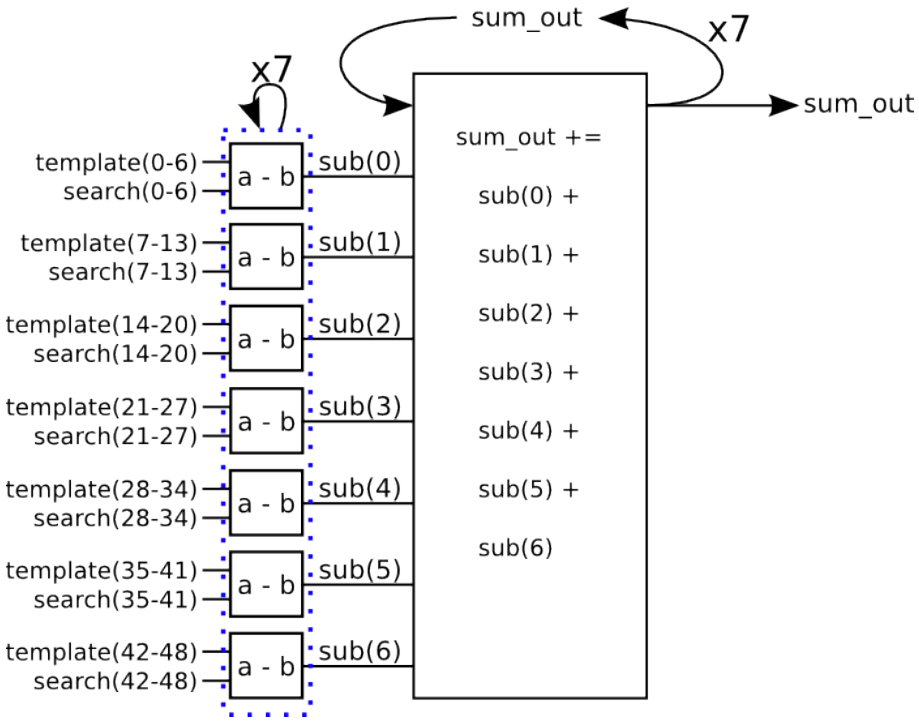


Figure 4.5: Architecture overview of the SAD algorithm with the 7x7 window implementation.

Figure 4.6 shows the pipeline used for the 7x7 window version. It takes 7 clock cycles to take all of the differences between all 49 pairs of pixel values. After the first difference is calculated, the differences can then begin to be summed up. The summing also takes 7 clock cycles and ends one cycle after the last difference is calculated. This results in a time of 8 clock cycles.

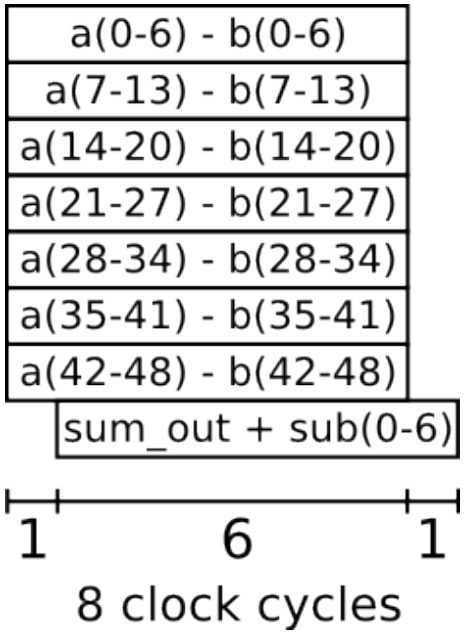


Figure 4.6: Pipeline architecture of the SAD algorithm with the 7x7 window implementation.

The code for the 7x7 window implementation can be found on github:
`https://github.com/cccitron/mastersThesis/tree/master/makestuff/libs/libfpgalink-20120621/hdl/fx2/vhdl/sad_parallel_7x7`

4.1.2 Minimum Comparator Architecture

The purpose of the minimum comparator is to find the lowest value of two input values and output the lowest value. The top level implementation of the minimum comparator is shown in Figure 4.7. The process is synchronous, noted by the clock clk_I. The index, pos0_I and pos1_I, of the SAD values, sad0_I and sad1_I, respectively, ranges from 0 to 15, which is the disparity range.

Appendix C shows the code for the minimum comparator. If sad1 is less than sad0, then sad1 and its index, pos1, are returned, otherwise sad0 and pos0 are

returned. Using a less than comparison takes up less hardware resources (CITE). This is useful because 15 minimum comparators (see Figure 4.8)are used for each pixel that is processed in parallel. So 30 minimum comparators are used for the 7x7 window implementation and 60 minimum comparators are used for the 9x9 window implementation. Constructing the minimum comparator in this way also accounts for cases where two SAD values are equal to each other. The SAD value with the lower index is always assigned to the sad0_I input and the higher indexed SAD values comes in through the sad1_I input. Therefore, if two values are equal, the SAD value with a lower index, and a lower disparity, will be returned.

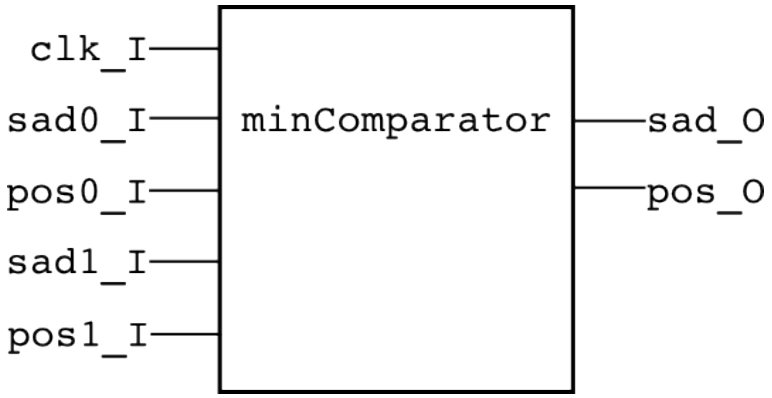


Figure 4.7: The top level minimum comparator implementation.

When multiple minimum comparators are put together to create a tree, as shown in Fig. 4.8, it is possible to quickly determine which SAD value is the lowest. This process is used to ultimately find the index of the lowest SAD value out of the 16 SAD values calculated for a pixel. A normal serial comparison of 16 values, would take 15 comparisons, or 15 clock cycles, if one comparison were to happen each clock cycle. By having 15 comparators, the number of SAD values needed for comparisons can be reduced by half each clock cycle. Using a tree of comparators drops the comparison time from 15 clock cycles to only 4 clock

cycles. Almost a 4 times speed up.

4.1.3 SAD Wrapper

The SAD wrapper is the entity that encompasses the SAD algorithms and minimum comparators. It receives the template image data from templ_I and receives the search image data from search_I. The write_t_I and write_s_I notify the wrapper when new data is actively being sent for the template and search images, respectively. The h2fReady_I and f2hReady_I are used to communicate when data is being sent to or from the host, the computer, from or to the FPGA. The sw_I allows the 8 switches on Atlys board to be connected to data that is within the wrapper to be displayed on the 8 LEDs, led_O. The outputs templ_O for template image region, search_O for search image region, sad_O for the SAD values calculated from the current template and search image regions, and disp_O that were found from the SAD values are sent out so they can be read, if desired. In the current implementations, templ_O, search_O, and sad_O are used for debugging while disp_O is used to obtain a depth map that is created from the pair of images. See Section 4.2.1 for how the data is transferred to the computer.

4.1.4 Top Level

The implementation of the SAD wrapper and its internal entities has been designed such that it should be able to work with any FPGA that is large enough to hold it. Figure 4.10 shows the SAD wrapper inside a top level entity. The top level gives the SAD wrapper image data and the SAD wrapper gives the top level disparity values. Those values are then transmitted to the computer, see Sec. 4.2.1. The implementation in Fig. 4.10 represents the 9x9 window imple-

mentation. For the 7x7 window implementation, there are only two SAD and two minimum comparators, as opposed to 4 each, due to the higher level of parallelization used within the SAD algorithms.

4.2 FPGA and Computer Communication

4.2.1 FPGALink

FPGALink [13] was used to facilitate communications between the computer (host) and the FPGA (Atlys board) over USB. An overview of how the FPGALink works between the host and FPGA is shown in Figure 4.11. The FPGALink has two possible communication modules to choose from, FX2 and EPP. According to [13], FX2 has an observed throughput around 26 MB/s, while EEP has a observed throughput of around 1.26 MB/s. FX2 was used due to its higher throughput.

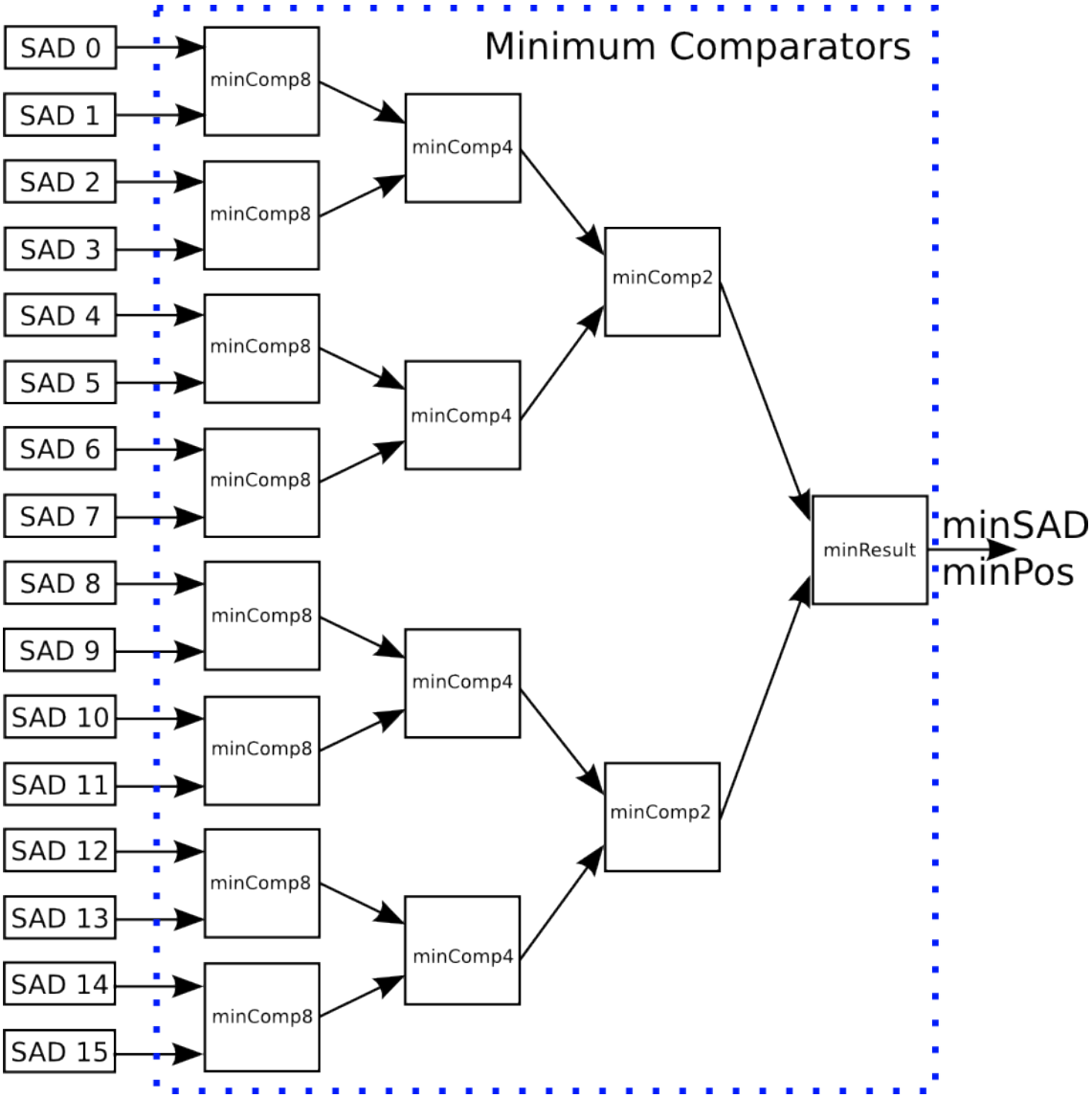


Figure 4.8: The minimum comparator tree designed to quickly find the minimum value and corresponding index out of the 16 SAD values that are calculated for one pixel.

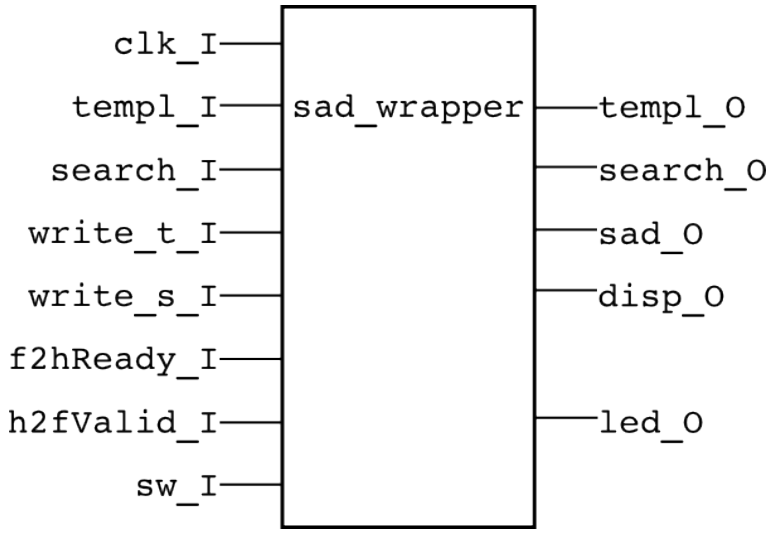


Figure 4.9: The SAD wrapper that encompasses the SAD algorithm and minimum comparator. It interacts with the top level.

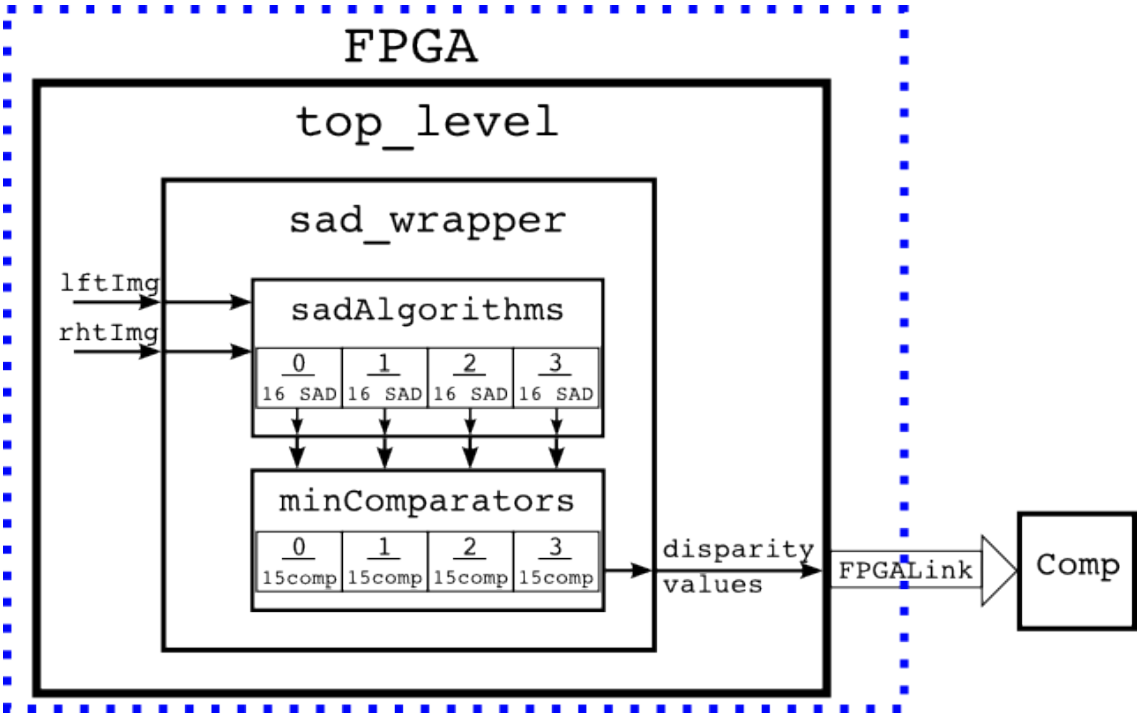


Figure 4.10: The overview of the structure used for implementing the 9x9 window. The 7x7 window has two less SAD and minComp each.

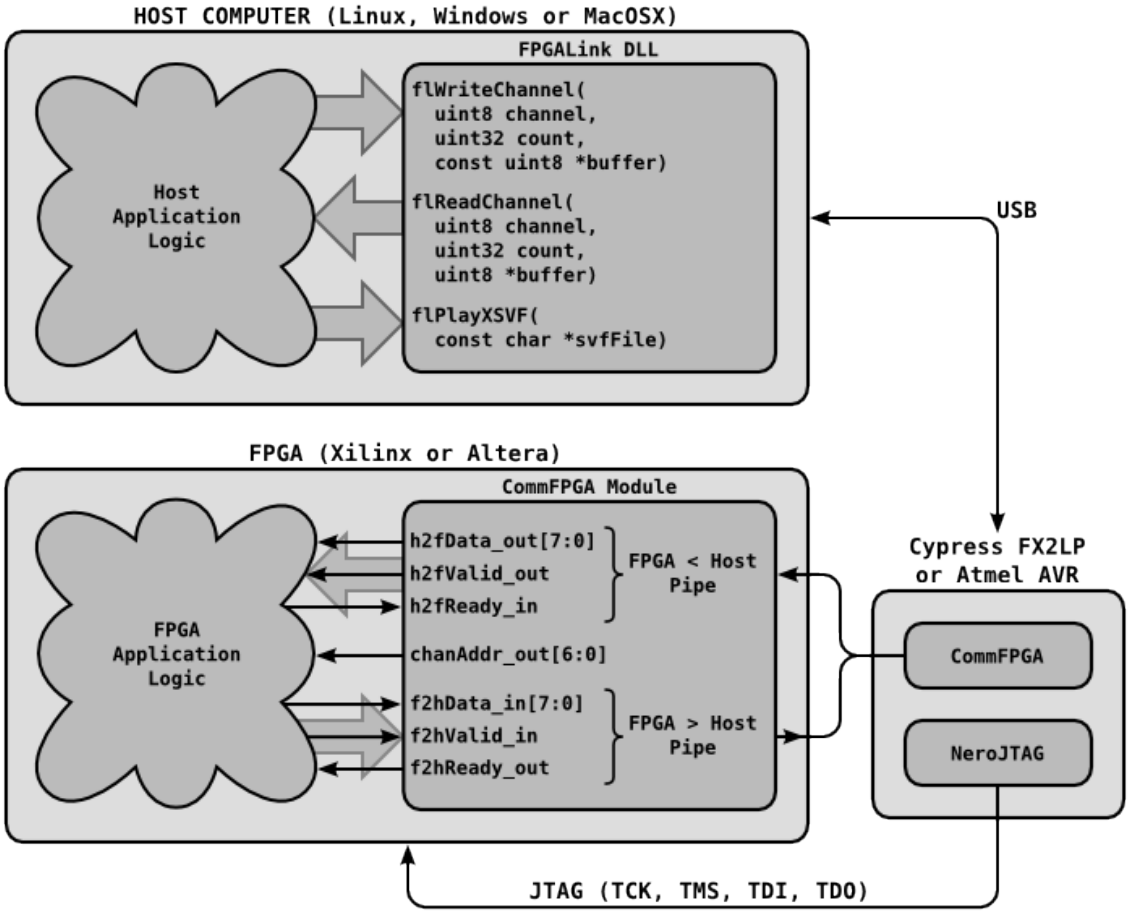


Figure 4.11: Overview of FPGALink communications between host computer and FPGA [13].

Chapter 5

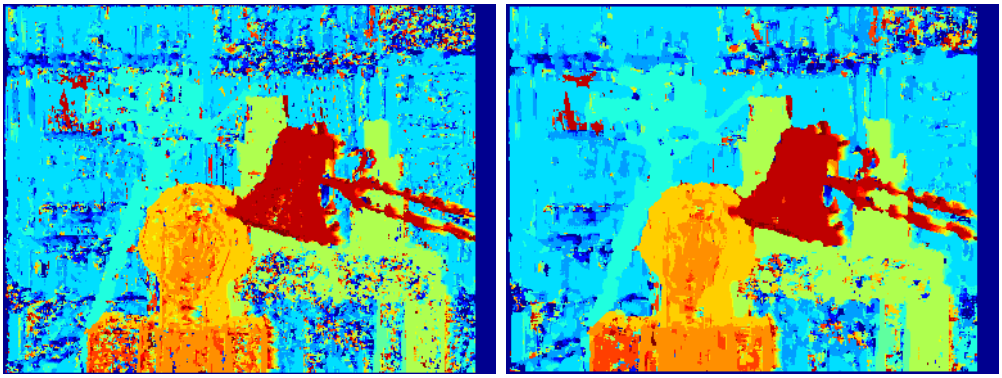
Experiments and Results

5.1 Window Size Selection

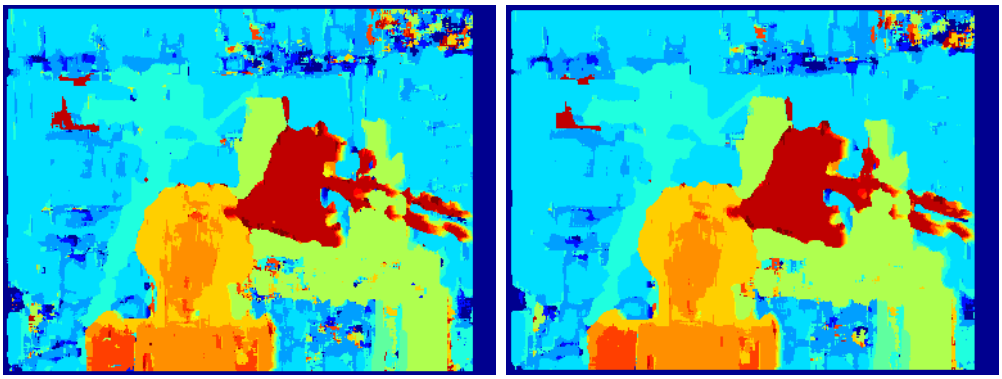
The size of the window (i.e. 9x9 pixels) affects the quality of the disparity map (see Figure 5.1) and the number of computations required to create the disparity map. The 3x3 window size in Figure 5.1a will be processed the fastest out of the window sizes shown since each SAD calculation only has 9 pairs of pixels compared to say the 13x13 window, or 169 pairs of pixels in Figure 5.1f. The 13x13 window has the least amount of noise, in its disparity map, but it also loses some detail as shown by the lamp in the foreground of the image compared to the other images. Also, as the window size gets larger, more resources are needed on the FPGA board. The 7x7 and 9x9 window sizes were used because they both were reasonably compromising on the amount of noise in the disparity maps and the amount of hardware resources needed for implementation.

5.2 Resource Utilization on FPGA

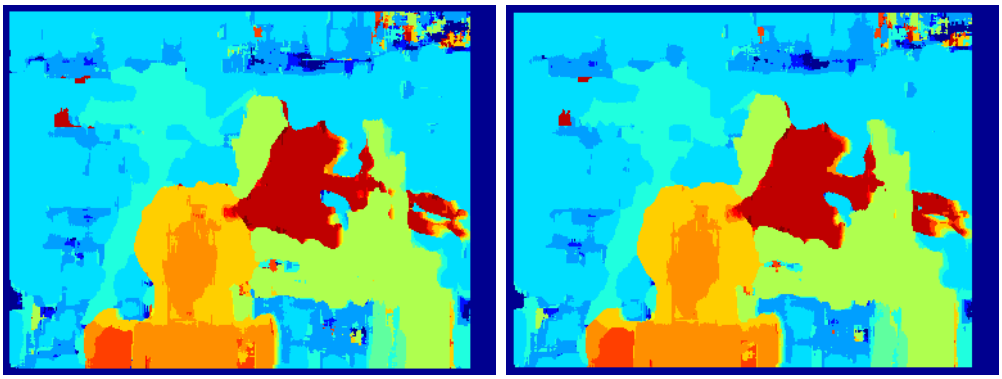
See Table E.1 in Appendix E. The 7x7 window implementation actually uses more resources on the FPGA board than the 9x9 window implementation due to the amount of parallel calculations used in the SAD algorithm. There is still plenty of space on the board for other types of top level designs that could use this SAD module.



(a) SAD 3x3 Window Disparity Map (b) SAD 5x5 Window Disparity Map



(c) SAD 7x7 Window Disparity Map (d) SAD 9x9 Window Disparity Map



(e) SAD 11x11 Window Disparity Map (f) SAD 13x13 Window Disparity Map

Figure 5.1: Window size comparisons for disparity maps [9] of the Tskukuba image pair [23].

5.3 Testbench Simulation

See Figure F.1 and Figure F.2 in Appendix F for the testbench simulations for the 9x9 window and 7x7 window implementations, respectively.

The h2fvalid_i value near the top of the figure is high when image data is being written to the SAD wrapper. The first section that is high is the initial rows being sent to the wrapper. The smaller following sections that are high are the next row being sent to the wrapper. The wrapper has been designed to allow both the template image data and the search image data to be sent to the wrapper at the same time, thus reducing the amount of time taken to get all necessary data into the wrapper. When the signal f2hready_i goes high, it means that the disparity values are being sent out of the wrapper.

5.3.1 9x9 Window Implementation Runtime

Based on the testbench simulation in Fig. F.1 the theoretical frames per second can be inferred for different image sizes. The simulation assumes the 100 MHz clock on the FPGA is used, and thus each clock cycle is 10 ns long.

According to the testbench simulation, for a 9x9 window with 4 pixels being processed in parallel, the first section includes the initial image data given to the SAD wrapper up to the when the the corresponding disparity values come out, takes 3.35 us. After that, a constant cycle is produced that has the SAD wrapper take in the next row and produces the next disparity values, which takes 1.22 us. A 640x480 image has 307,200 pixels, which will produce a disparity map of 617x472, which is 291,224 pixels. Disparity values are not produced for pixels that either the windows cannot fit on or there is not enough room for the 16

Image	Image Width	Image Height	Sec/frame	Frames/sec
VmodCAM	640	480	0.0888	11.26
Tsukuba	384	288	0.0308	32.43
Venus	434	383	0.0470	21.27

Table 5.1: 9x9 window for theoretical runtime for the FPGA board for different image sizes.

SAD values to be calculated for pixel. Since 4 pixels are processed in parallel, 291,224 pixels is divided by 4 pixels/iteration, giving 72,806 iterations. So, 72,806 iterations times 1.22 us/iteration plus 3.35 us (initial section, hence minus one on number of iterations) gives 88,826.67 us or approximately 0.0888 seconds per frame. Therefore, an image size of 640x480 can be processed at around 11.26 frames per second. Table 5.1 shows the theoretical frame rate for the image sizes used in this chapter.

5.3.2 7x7 Window Implementation Runtime

Based on the testbench simulation in Fig. F.2 the theoretical frames per second (fps) can be inferred for different image sizes. The simulation assumes the 100 MHz clock on the FPGA is used, and thus each clock cycle is 10 ns long.

According to the testbench simulation, for a 7x7 window with 2 pixels being processed in parallel, the first section includes the initial image data given to the SAD wrapper up to the when the the corresponding disparity values come out, takes 1.78 us. After that, a constant cycle is produced that has the SAD wrapper take in the next row and produces the next disparity values, which takes 0.42 us. A 640x480 image has 307,200 pixels, which will produce a disparity map of

Image	Image Width	Image Height	Sec/frame	Frames/sec
VmodCAM	640	480	0.0616	16.23
Tsukuba	384	288	0.0215	46.51
Venus	434	383	0.0327	30.58

Table 5.2: 7x7 window for theoretical runtime for the FPGA board for different image sizes.

619x474, which is 293,406 pixels. Disparity values are not produced for pixels that either the windows cannot fit on or there is not enough room for the 16 SAD values to be calculated for pixel. Since 2 pixels are processed in parallel, 293,406 pixels is divided by 2 pixels/iteration, giving 146,703 iterations. So, 146,703 iterations times 0.42 us/iteration plus 1.78 us (initial section, hence minus one on number of iterations) gives 61,617.04 us or approximately 0.0616 seconds per frame. Therefore, an image size of 640x480 can be processed at around 16.23 frames per second. Table 5.2 shows the theoretical frame rate for the image sizes used in this chapter.

5.4 Test Image Pairs

In this section, FPGA disparity maps are compared to disparity maps created using Python. The SAD algorithm implementation in Python is show in Appendix D. The Python SAD version is performed completely in serial, so 1 pixel at a time. It is run on a desktop that has a i7 CPU 950 at 3.07 GHz, 16 GB of RAM, and runs Ubuntu 64-bit. The images the Python version produced are also used to compare runtime of the algorithm.

5.4.1 Data Overflows

In VHDL, the code for the hardware to be generated is designed. The size of the data used for storing logic and variables is defined during the coding process. In the SAD algorithm, it is possible for the SAD value to become much larger than the individual pixel values. For example, the pixel values range from 0 to 255, or 2^8 bits, while some SAD value could be over 4,095, or need more than 2^{12}. Most SAD values were under 4,096, however, to account for those that were above it, the SAD algorithm use 2^{14} bits to account for any values from 0 to 16,383. Figure 5.2 shows what can happen when the data size allotted for the SAD algorithm is not large enough (i.e. only having 2^{10} bits of space). The data used is unsigned, so when it goes above the highest supported value, it goes back to 0 and continues from there.

Since most of the values were below 4,096, a measure was put in place in order to reduce the amount of bits needed during the minimum comparisons. If a SAD value was greater than 4,095, then 4,095 was returned for the calculated SAD because the greater the value, the less likely that search pixel is the correct corresponding one to the template pixel. In Figure 5.3 and Figure 5.4, the only real noticeable difference in the Python to FPGA comparisons is at the top of the images. The colors, warmer is closer and cooler is farther away, show that the top areas are thought to be closer than they actually are in the FPGA images. For a robot, it would be better to err on the side of thinking an object is closer than it actually is because the robot will be less prone to collide with the object. If a robot thought an object was farther away than it actually was, then the likelihood of collision would increase.

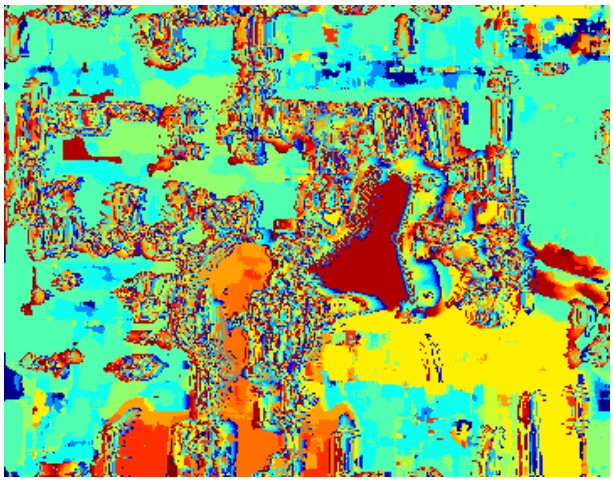
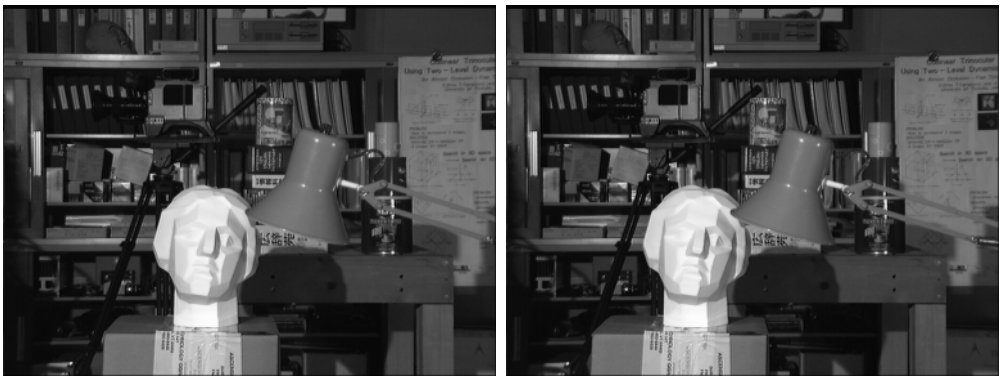


Figure 5.2: Data overflow for Tskukuba image pair [23].

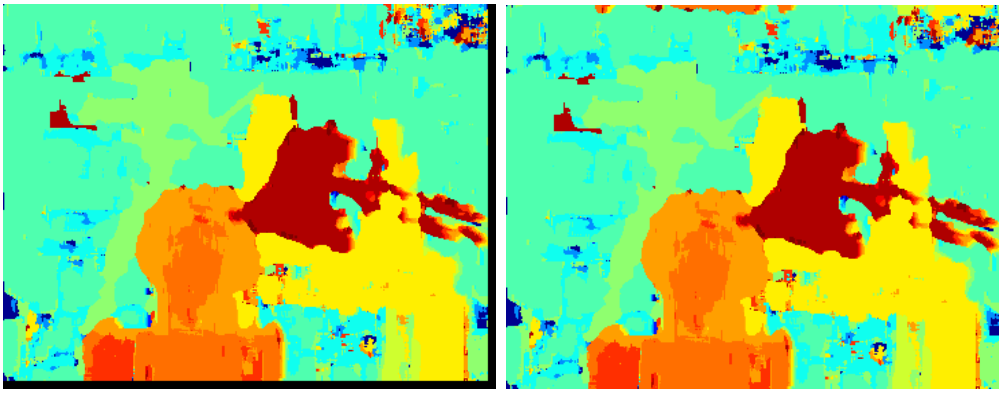
5.4.2 Tsukuba

In Figure 5.3a and Figure 5.3b, the Tsukuba image pair are shown. Figure 5.3 shows how the 7x7 window implementation is slightly noisier than the 9x9 window implementation. As discussed in Section 5.4.1, the only noticeable difference between the Python implementation and the FPGA implementation is at the top of the disparity maps. This is caused by the two images not having similar enough corresponding regions, which causes SAD values to be greater than normal.

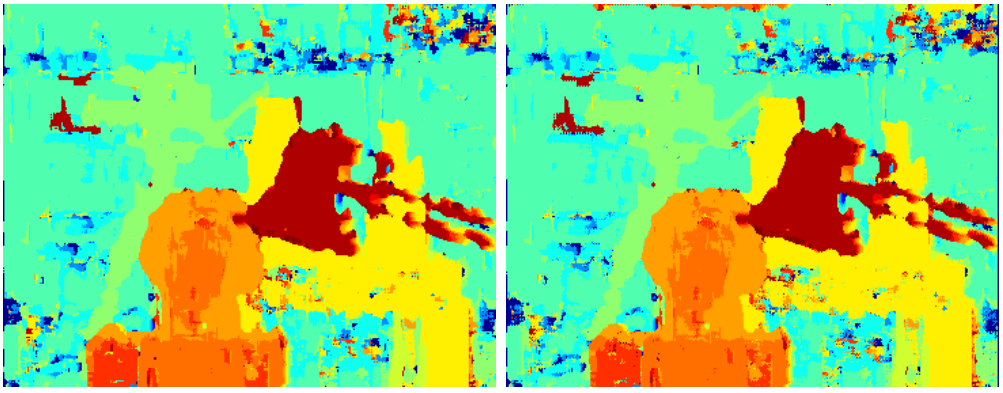
For Tsukuba, the FPGA version should have a theoretical runtime of 32.43 or 46.51 frames per second for the 9x9 and 7x7 window implementations, respectively, from Table 5.1 and Table 5.2. The serial Python implementation took 7.81 minutes for a 9x9 window with a disparity range of 16. The 7x7 window implementation with a disparity range of 16 in Python took 4.22 minutes to complete.



(a) Left Tsukuba Grayscale Image (b) Right Tsukuba Grayscale Image



(c) Python 9x9 Disparity Map (d) FPGA 9x9 Disparity Map



(e) Python 7x7 Disparity Map (f) FPGA 7x7 Disparity Map

Figure 5.3: Disparity map comparison of the Tskukuba image pair [23].

5.4.3 Venus

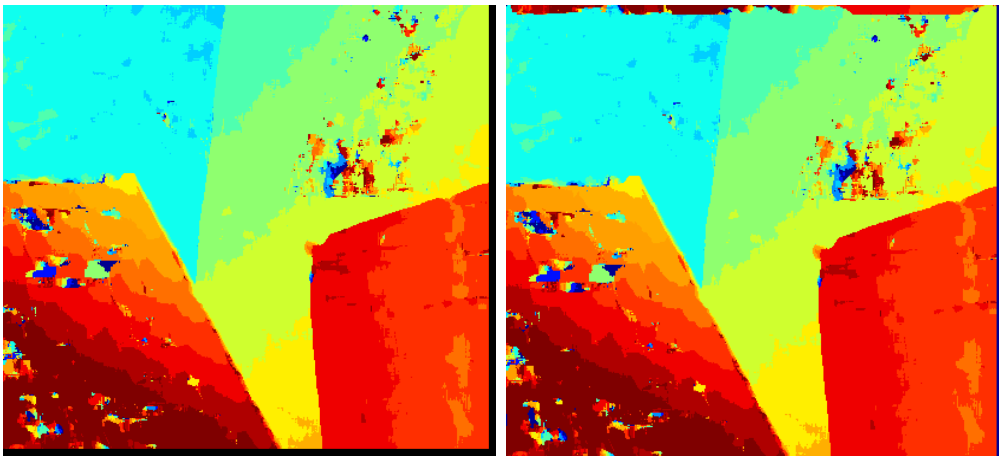
In Figure 5.4a and Figure 5.4b, the Venus image pair are shown. In the image pair, the newspaper articles are flat and slanted, relative to the cameras. This gradual slope, also present in the background, can be difficult for the SAD algorithm to deal with, however, the algorithm is still able to give a fairly accurate representation of the depth in the image. It also causes the gradient pattern shown in the disparity maps. The 7x7 window depth maps have more noise than the 9x9 window depth maps.

For Venus, the FPGA version should have a theoretical runtime of 21.27 or 30.58 frames per second for the 9x9 and 7x7 window implementations, respectively, from Table 5.1 and Table 5.2. The serial Python implementation took 8.15 minutes for a 9x9 window with a disparity range of 16. The 7x7 window implementation with a disparity range of 16 in Python took 5.00 minutes to complete.

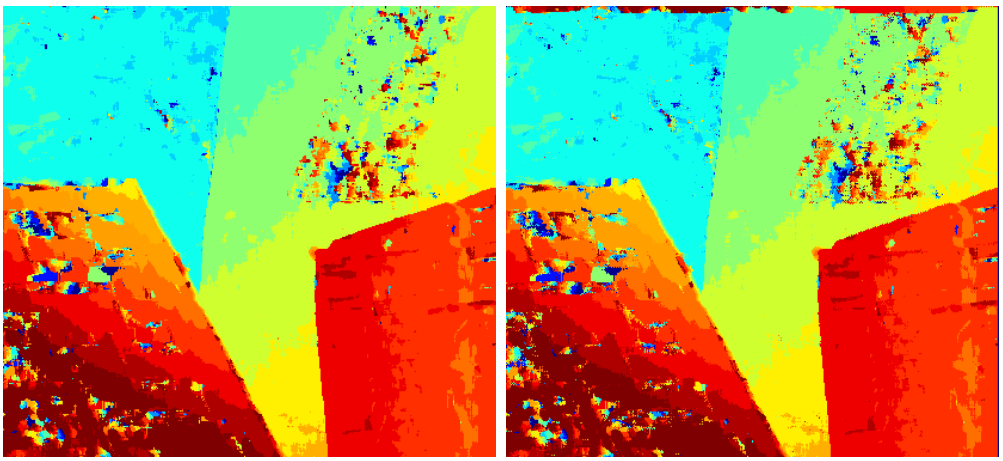
5.4.4 Cones

In Figure 5.5a and Figure 5.5b, the Venus image pair are shown. Figure 5.5 shows the issue of objects in an image pair being too close to the stereo cameras. The closer an objects is to the stereo cameras, the greater its disparity value will be. Using the SAD algorithm with a 9x9 window and a disparity range of 60 (as opposed to the range of 16 used on the FPGA board) produces the results in Figure 5.5c. When the disparity range is not high enough, the disparity map in Figure 5.5d is produced.

(a) Left Venus Grayscale Image (b) Right Venus Grayscale Image



(c) Python 9x9 Disparity Map (d) FPGA 9x9 Disparity Map



(e) Python 7x7 Disparity Map (f) FPGA 7x7 Disparity Map

Figure 5.4: Disparity map comparison of the Venus image pair [23].

(a) Left Cones Grayscale Image

(b) Right Cones Grayscale Image

(c) 9x9 at Disparity Range of 60 [9]

(d) 9x9 at Disparity Range of 16

Figure 5.5: Disparity map comparison of the Cones image pair [23].

Chapter 6

Conclusions

For image processing, the more operations that can be parallelized, the faster images can be processed. However, as parallelism is increased, the amount of hardware required also is increased. It could be possible to parallelize a SAD algorithm, or most image processing methods, to only take a few clock cycles to process the whole image (i.e. every SAD calculation for an image pair happening simultaneously). Unfortunately, the area required on an FPGA would be a lot more than what was implemented in this paper. The cost of an FPGA that could handle that amount of hardware would be very cost prohibitive and not something a club or hobbyist could readily use for a robotics project. There does come a point at which the frames per second of disparity maps produced exceeds the speed that the other parts of the robot can process, which is unnecessary cost. So the FPGA board only needs to be able to handle a SAD implementation up to a certain frame rate, which depends on the requirements of the application for the robot.

The smaller the image size, the higher the frame rate, as shown in Figure 6.1. Once the number of pixels in an image goes below 180,000 for the 7x7 window implementation or 140,000 for the 9x9 window implementation, the frame rate approaches 30 frames per second, which is good for humans. For robots, a frame rate of 10 should be sufficient for most tasks that do not require a higher frame rate. Both the 9x9 and 7x7 window implementations were shown be above 10

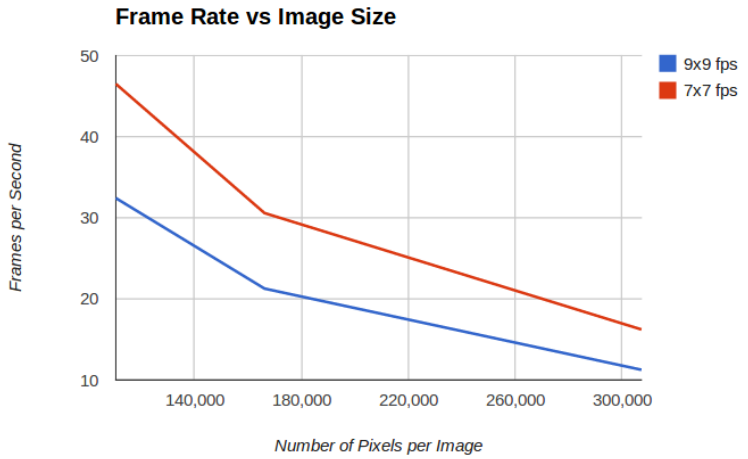


Figure 6.1: Frame rate comparison of different image sizes.

frames per second for an image size of 640x480.

Between the 9x9 window implementation and the 7x7 window implementation, unless a higher frame rate is needed, the 9x9 is better than the 7x7. While 7x7 has a higher frame rate, 9x9 produces a better quality disparity map with less noise and requires fewer hardware resources.

This modular implementation of the SAD algorithm has the potential to be used for FPGA implementations in autonomous mobile robotic applications.

Chapter 7

Future Work

The next steps are to get a fully functional stereo vision implementation on the Atlys board that uses the SAD module presented in this paper. The Atlys board has a 1 GB DDR RAM chip, which could be used to buffer the images from the VmodCAM stereo camera module [2]. The left and right images from the VmodCam could be buffered to the DDR RAM and then sections of the buffered images could be sent to the SAD module to obtain the disparity values. With the correct timing and buffering, both or one of the images and the disparity image could then be sent off board to a computer on a robot to use the image and depth data to navigate and interact with the world.

After a fully functional implementation on the Atlys board is working, a custom FPGA board could be designed and manufactured. The custom board only needs the functionalities of the Atlys board in order to: communicate with the computer, obtain images from the stereo cameras, buffer the images on the DDR RAM, and process everything else on the FPGA IC. A custom board without the extra peripherals on board has the potential to further reduce the cost of a stereo vision FPGA board. Also, the stereo cameras could be built into the board to reduce the cost of hardware needed for connections.

Furthermore, replacing the FPGA IC used on the Atlys board with one that has a clock frequency higher than 100 MHz or more space while keeping the cost

of the IC around the same as the Atlys board FPGA IC is way to speed up the SAD calculation time and increase the frame rate.

When all is said and done, having robots readily able to have better and less expensive “eyes” to perceive the world around them in greater depth will be pretty neat.

Bibliography

[1] 3d imaging with ni labview. http://www.ni.com/white-paper/14103/en/, August 2013.

[2] Atlys spartan-6 fpga development board. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,836&Prod=ATLYS&CFID=5602753&CFTOKEN=ff475ab97d889237-27A8B1C4-5056-0201-02F29D3CC5564ED6, July 2014.

[3] Bumblebee2. http://ww2.ptgrey.com/stereo-vision/bumblebee-2, July 2014.

[4] Digi-key. http://www.digikey.com/product-detail/en/DK-DEV-4SGX230N/544-2594-ND/2054809?cur=USD, July 2014.

[5] Digi-key. http://www.xilinx.com/products/boards_kits/virtex5.htm, July 2014.

[6] Vmodcam - stereo camera module. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,648,931&Prod=VMOD-CAM, July 2014.

[7] Xtremedsp starter platform spartan-3a dsp 1800a edition. http://www.xilinx.com/products/boards-and-kits/HW-SD1800A-DSP-SB-UNI-G.htm, July 2014.

[8] Epipolar geometry. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT10/node3.html, Accessed July 18, 2014.

[9] Siddhant Ahaja. Correlation based similarity measures-sum of absolute differences (sad). https://siddhantahuja.wordpress.com/2009/05/11/correlation-based-similarity-measures-in-area-based-stereo-matching-system May 2009.

[10] P. Ben-Tzvi and Xin Xu. An embedded feature-based stereo vision system for autonomous mobile robots. In *Robotic and Sensors Environments (ROSE), 2010 IEEE International Workshop on*, pages 1–6, Oct 2010.

[11] P. Ben-Tzvi and Xin Xu. An embedded feature-based stereo vision system for autonomous mobile robots. In *Robotic and Sensors Environments (ROSE), 2010 IEEE International Workshop on*, pages 1–6, Oct 2010.

[12] M.Z. Brown, D. Burschka, and G.D. Hager. Advances in computational stereo. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(8):993–1008, Aug 2003.

[13] Chris. Fpgalink: Easy usb to fpga communication. http://www.makestuff.eu/wordpress/software/fpgalink/, March 2011.

[14] Jingting Ding, Xin Du, Xinhuan Wang, and Jilin Liu. Improved real-time correlation-based fpga stereo vision system. In *Mechatronics and Automation (ICMA), 2010 International Conference on*, pages 104–108, Aug 2010.

[15] M. Gosta and M. Grgic. Accomplishments and challenges of computer stereo vision. In *ELMAR, 2010 PROCEEDINGS*, pages 57–64, Sept 2010.

[16] M. Hariyama, N. Yokoyama, M. Kameyama, and Y. Kobayashi. Fpga implementation of a stereo matching processor based on window-parallel-and-pixel-parallel architecture. In *Circuits and Systems, 2005. 48th Midwest Symposium on*, pages 1219–1222 Vol. 2, Aug 2005.

[17] Li He-xi, Wang Guo-rong, and Shi Yong-hua. Application of epipolar line rectification to the stereovision-based measurement of workpieces. In *Measuring Technology and Mechatronics Automation, 2009. ICMTMA '09. International Conference on*, volume 2, pages 758–762, April 2009.

[18] N. Isakova, S. Basak, and AC. Sonmez. Fpga design and implementation of a real-time stereo vision system. In *Innovations in Intelligent Systems and Applications (INISTA), 2012 International Symposium on*, pages 1–5, July 2012.

[19] Seunghun Jin, Junguk Cho, Xuan Dai Pham, Kyoung-Mu Lee, Sung-Kee Park, Munsang Kim, and J.W. Jeon. Fpga design and implementation of a real-time stereo vision system. *Circuits and Systems for Video Technology, IEEE Transactions on*, 20(1):15–26, Jan 2010.

[20] C. Murphy, D. Lindquist, AM. Rynning, Thomas Cecil, S. Leavitt, and M.L. Chang. Low-cost stereo vision on an fpga. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 333–334, April 2007.

[21] G. Rematska, K. Papadimitriou, and A Dollas. A low cost embedded real time 3d stereo matching system for surveillance applications. In *Bioinformatics and Bioengineering (BIBE), 2013 IEEE 13th International Conference on*, pages 1–6, Nov 2013.

[22] D. Scharstein, R. Szeliski, and R. Zabih. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Stereo and Multi-Baseline Vision, 2001. (SMBV 2001). Proceedings. IEEE Workshop on*, pages 131–140, 2001.

[23] Daniel Scharstein and Richard Szeliski. Stereo. `http://vision.middlebury.edu/stereo/`, April 2012.

Appendix A

Absolute Difference 9x9 Window Code Snippet

```
-- Assign greater value to more and smaller to less
IF (search_window(ndx) < template_window(ndx)) THEN
    more <= template_window(ndx);
    less <= search_window(ndx);
ELSE
    less <= template_window(ndx);
    more <= search_window(ndx);
END IF;

-- Subtraction IP CORE, sub = more - less
subber : subtr_core
   PORT MAP (
       a => more,
       b => less,
       s => sub
    );
```

Appendix B

Absolute Difference 7x7 Window Code Snippet

```
-- Assign greater value to more and smaller to less
-- Loop is unrolled in hardware, 7 assignments occur simultaneously
FOR i IN 0 TO 6 LOOP
    IF (search_window(ndx+(7*i)) < template_window(ndx+(7*i))) THEN
        more(i) <= template_window(ndx + (7*i));
        less(i) <= search_window(ndx + (7*i));
    ELSE
        less(i) <= template_window(ndx + (7*i));
        more(i) <= search_window(ndx + (7*i));
    END IF;
END LOOP;

-- Subtraction IP CORE, sub(i) = more(i) - less(i)
g_differ_10 : FOR i IN 0 TO 6 GENERATE
    i_subber : adder_10
        PORT MAP (
            a => more(i),
            b => less(i),
            s => sub(i)
        );
END GENERATE g_differ_10;
```

Appendix C

Minimum Comparator Code

```
-- Constantly assign inputs
sad0 <= sad0_I;
pos0 <= pos0_I;
sad1 <= sad1_I;
pos1 <= pos1_I;

-- Comparison
PROCESS(clk_I)
begin
    IF (RISING_EDGE(clk_I)) THEN
        IF (sad1 < sad0) THEN
            sad_out <= sad1;
            pos_out <= pos1;
        ELSE
            sad_out <= sad0;
            pos_out <= pos0;
        END IF;
    END IF;
END PROCESS;

-- Constantly assign outputs
sad_O <= sad_out;
pos_O <= pos_out;
```

Appendix D

Python3 Serial SAD Algorithm

```
# Parameters for 9x9 window size SAD Algorithm
winSize = 9
dispRange = 16
dispRow = 4
win = (int)(winSize/2)
ncol = (dispRow-1) + dispRange + (winSize-1)
dispH = height - (winSize-1)
dispW = width - (ncol - dispRow)

# Serial SAD Algorithm
for row in range(dispH):
    sadArray = numpy.zeros((dispW, dispRange), dtype = 'i')
    for i in range(dispW):
        for j in range(dispRange):
            for m in range(-win, (win+1)):
                for n in range(-win, (win+1)):
                    sadArray[i][j] += \
                        abs(templateBuff[row+m+win][n+i+win] - \
                        searchBuff[row+m+win][n+i+j+win])
    disparityArray = sadArray.argmin(axis=1)
    disparityAll[row] = disparityArray
```

The full code for the Python3 SAD Algorithm can be found on github:

`https://github.com/cccitron/mastersThesis/tree/master/pythonSAD`

Appendix E

Resource Utilization

Window Size	**Disparity Range**	**# of pixels processed in parallel**	**# of Slice Registers used out of 54,576**	**# of Slice LUTs used out of 27,288**	**# of occupied Slices out of 6,822**	**# of MUXCYs used out of 13,644**
7x7	16	2	8,465 (15%)	17,220 (63%)	5,640 (82%)	3,796 (27%)
9x9	16	4	8,136 (14%)	16,038 (58%)	4,719 (69%)	2,280 (16%)

Table E.1: Resource utilization on the FPGA Atlys board for both window implementations.

Appendix F

Testbench Simulations

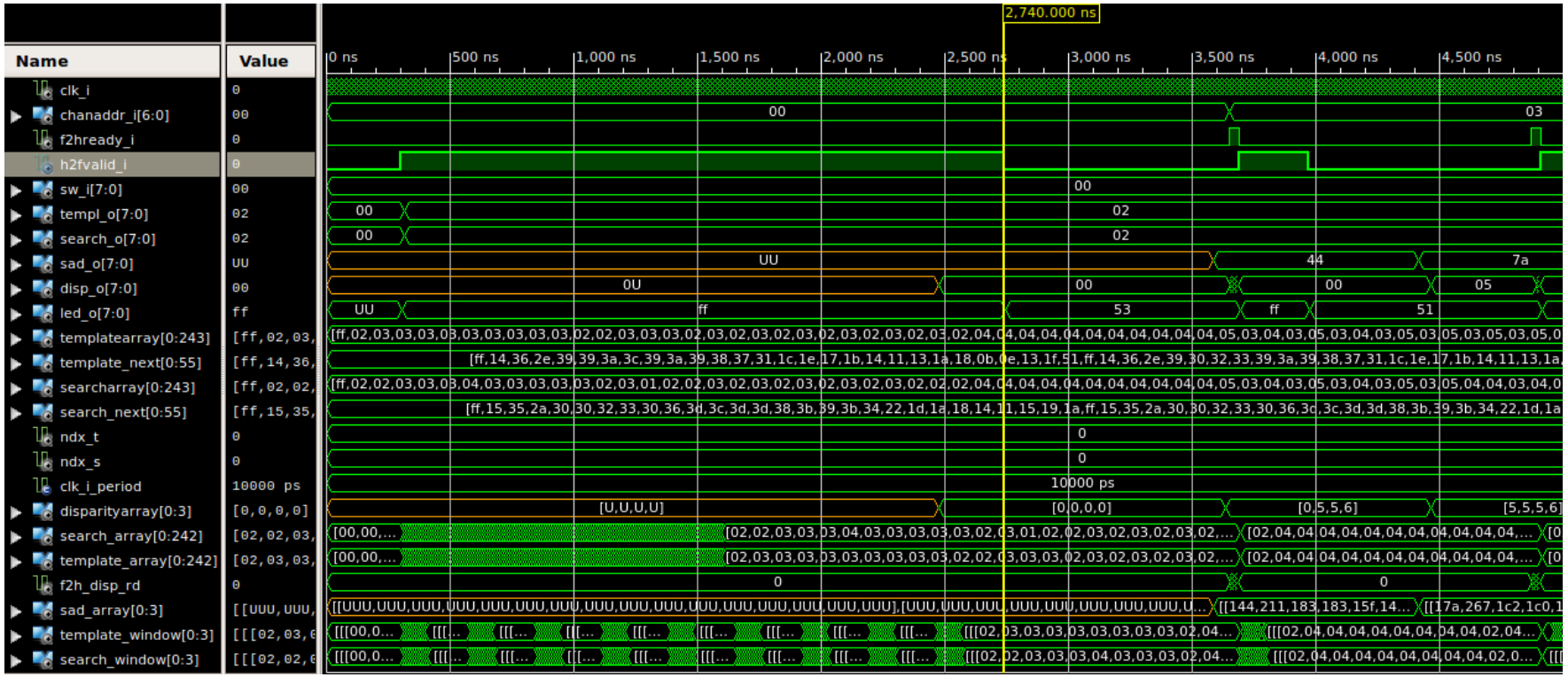


Figure F.1: Testbench simulation for the 9x9 window implementation.

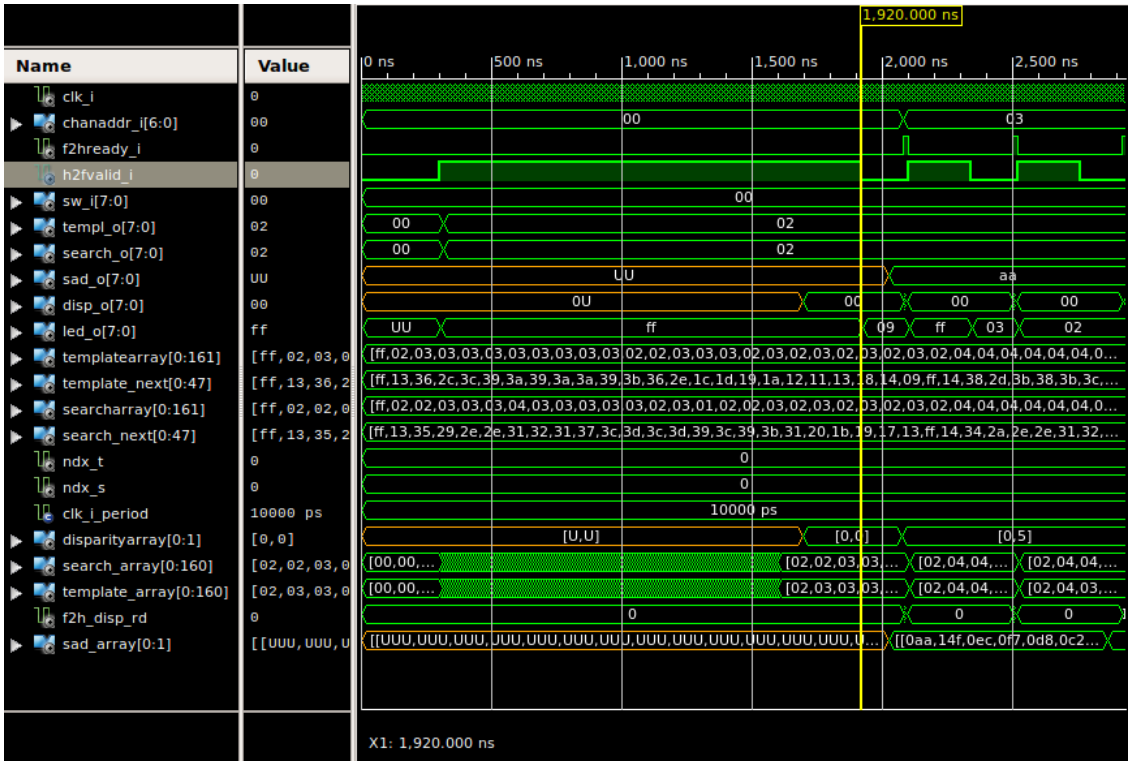


Figure F.2: Testbench simulation for the 7x7 window implementation.