Evaluation of Data Transfer from FPGA to PC: Increasing Frame Rate by BLOB Detection

by
Petr Samarin,
Rainer Herpers, Kenneth B. Kent, and Timur Saitov

TR 12-217, June 13, 2012

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
http://www.cs.unb.ca

Abstract

This work presents a novel hardware/software codesign approach to blob detection. Camera images are preprocessed on a field-programmable gate array that separates foreground pixels from the background. The foreground pixels are transferred to the PC over a 100 Mbit/s Ethernet interface by using a custom protocol. The PC reconstructs the images from received foreground pixels and extracts centers of masses from every blob in the reconstructed image. Results of evaluation show that under the constraint of 100 Mbit/s the FPGA is able to transmit a large amount of blobs if their number is small enough. The Ethernet interface turns out to be the bottleneck of the overall system.

Contents

1 Introduction

One of the tasks in the MI6 project is to precisely estimate the position of the user holding a 6 degrees of freedom (DoF) device that emits several infrared light spots onto the screens of the Immersion Square. By knowing the positions of the light spots it is possible to estimate the position of the device. This task was addressed by [1], where a field programmable gate array (FPGA) was used to detect the light spots, to compute their centers and to send this information to a computer, where a precise pose estimation could take place. However, the 6-DoF device for which the solution has been developed does not exist in practice, which makes it impossible to test the devised solution in the real world.

To address this limitation, a new version of the Immersion Square is planned. In the new version, as illustrated in Figure 1, the points will be produced by stationary devices positioned behind the projection screens, and the new 6-DoF interface will have an on-board camera that will send the data over a wireless interface to a PC. The PC will match the points and estimate the camera pose.

The new version brings a new challenge—raw camera images cannot be sent fast enough to the PC. To illustrate the problem, assume that the device sends 8-bit grayscale images with the resolution of 640 x 480 over a wireless interface. The transmission rate of a wireless interface

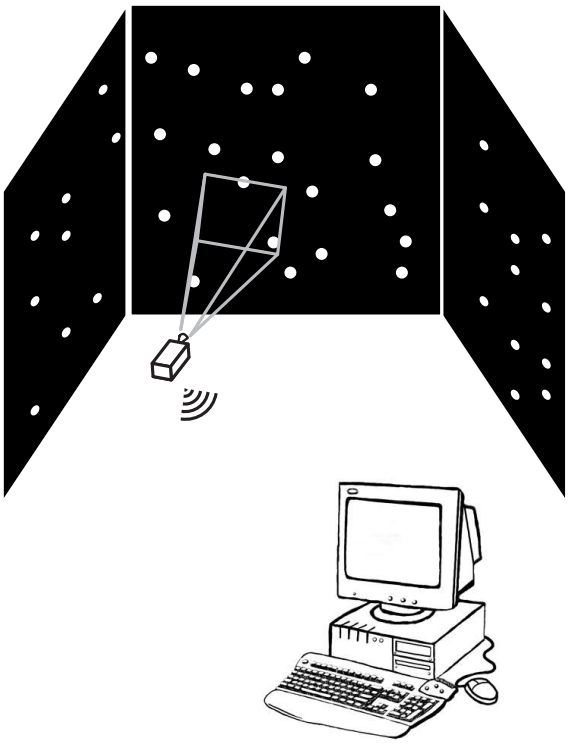

Figure 1: New version of the Immersion Square.

is usually around 50 Mbits/s, which is the maximum achievable transmission rate under perfect conditions. Under real world conditions, the transmission rate will be considerably less. The maximum amount of images that can be sent per second is equal to: $\frac{50000000}{640 \cdot 480 \cdot 8} \approx 20$ frames per second. Hence, the system will not be able to respond to user commands quickly enough.

This suggests that some preprocessing has to take place on the mobile device before data is sent to the host PC. The preprocessing can be done in several ways. One way is to only send data necessary for estimation of camera pose—centers of the light spots. If an image contains fewer light spots than there are pixel, the amount of data to be sent can be reduced considerably. Another way to reduce the amount of data, is to compress camera images. In this case, more intense image processing will take place on the computer because the PC needs to decompressed the images, search for laser spots in the decompressed images and calculate coordinates of their centers. Both approaches are explored in this project.

This R&D project develops and evaluates two ways to preprocess and transfer image data to a host PC by using an FPGA. The first method splits the task of blob detection between FPGA and PC. The FPGA separates foreground pixels from the background pixels and sends them to the computer, while the computer calculates the centers of the light spots. The second method performs single pass blob detection on FPGA and sends the centers of the laser spots to the host PC. Evaluation results show that, depending on the image, both methods constitute a trade-off of frame rate, precision and flexibility. A codesign solution is more flexible, but suffers from a low frame rate when the amount of foreground pixels is high. A pure FPGA solution allows high frame rates, but is less flexible and less precise than the codesign solution. Both systems have their advantages and drawbacks, and their use should be carefully considered based on the task at hand.

This report is organized as follows. The next section provides an overview of existing approaches to blob identification and blob analysis on FPGAs. Their advantages and disadvantages are briefly discussed. Section 3 presents two approaches developed in this project and the custom protocol for data transfer. Sections 4 and 5 describe the hardware and software tools used in this project, respectively. Section 6 shows that the approaches work as expected by presenting a verification strategy and the results of verification. Section 7 describes evaluation strategy and the experimental setup used for carrying out the evaluation. Section 8 presents the results of evaluation and discusses their implications. Section 9 concludes this report with a summary and gives directions for future work.

2 Background

Blob detection is the process of grouping together similar pixels based on some property [2]. This property can be a simple threshold that is used to group pixels whose pixels values exceed it, or pixels for which the pixel values are in a specific range. The property can also be more complex, such as for example, distance transform [3], that labels each pixel by its smallest distance to regions of interest. Blob detection enables a higher-level image processing than it is possible to achieve by only dealing with pixels.

Blob detection, is a widely studied topic, and many approaches to blob detection exist in the literature [4]. However, when surveying approaches that explore blob detection on FPGAs, only few approaches are efficient enough outperform blob detection on general-purpose processors. These approaches usually have one property in common—they perform blob detection in a single pass. Single pass, as opposed to multiple passes, means that the image is processed on the fly without storing it in a frame buffer.

[5] presents a single-pass real time algorithm for blob detection. The algorithm has been impemented and evaluated on an FPGA. The basic idea of this approach is to assign same labels to foreground pixels that exceed a fixed threshold and are adjacent to each other. In this approach, camera images are buffered in an external memory, which would lead to a bottleneck, if the memory is read out sequentially. However, the approach avoids this bottleneck by reading 32 pixels at once and by using run-length encoding in order to process all 32 pixels in parallel. The approach also covers situations in which some merges occur, i.e., when two runs are both adjacent to a run currently processed. In this case the runs are merged into a single run, and the label of the other run is freed. However this approach does not discuss the situation where multiple merges occur for the same label. The resulting labels are used for computation of three features: area, bounding box and center of gravity. These particular features can be implemented with almost no extra effort.

[6] introduced a design of a single pass connected component labeling [7] on an FPGA that has been implemented shortly after in [8]. This approach labels all foreground pixels based on a similar approach that was pursued by [5], however, this paper addresses the problem that arises when one label is relabeled several times in the same line. This problem is handled by exploiting the fact that each frame has horizontal and vertical blanking periods. Thus, the labels that are relabeled several times in a line, are found and corrected during the blanking periods. The labels themselves are only used for feature extraction. Several features that can be extracted from labels acquired from connected component labeling are reviewed in [2].

3 Approach

Two approaches have been developed in this project. The first approach splits the task of blob detection between FPGA and PC. The FPGA is used to filter out the regions of interest that are sent to the computer, and the computer is used to perform blob detection and computation of blob centers. The second approach performs blob detection on the FPGA and sends the centers of laser spots to the computer. A custom protocol has been developed for communication between the FPGA and PC over Ethernet.

3.1 Shared Blob Detection on FPGA and PC

One drawback of blob detection performed purely on an FPGA is the lack of flexibility in the way features of objects are extracted. For example, during the estimation of centers of light spots, it might happen that the spots are blurry because the light source has been moved rapidly. In this case, several passes over the image might be necessary. However, this requires random access to the image, which means that such approach cannot be realized on an FPGA without reducing the frame rate of the system. Often times, a more flexible way to extract features from objects is desirable. For this reason, a hardware/software codesign to blob detection has been developed. A fundamental insight of this approach is that instead of processing the image on the FPGA, it is possible to perform only coarse-grained processing on the FPGA and to send important parts of the image to the copmuter for doing heavier processing. This can be achieved by compressing the image on the FPGA, and letting the host PC decompress the

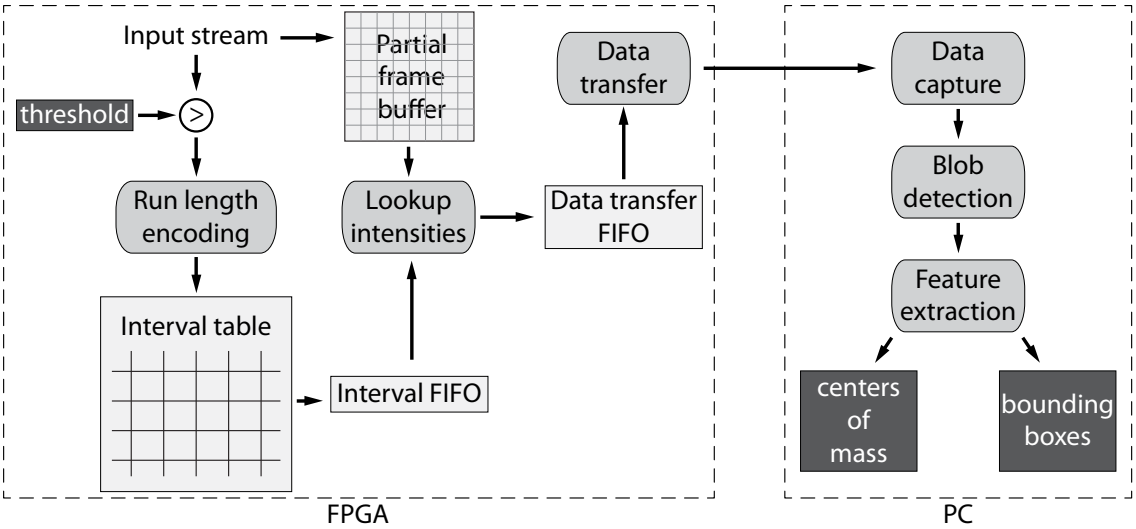

Figure 2: Identification of regions of interest—a high-level design overview. Camera image is partially stored in a frame buffer, that is realized in the block RAM. The interval table is stored in the registers.

image and handle the task of blob identification and analysis. A very simple way to compress the images is to transfer only the foreground pixels and to discard the background pixels.

Figure 2 shows a high level view of the approach. Camera images arrive in a raster scan order on the FPGA. A small part of the image is stored in a frame buffer for later look up. First, foreground pixels are separated from the background pixels by using simple thresholding, in which the intensity of each pixel is compared to a predefined threshold. Pixels whose intensities exceed the threshold are forwarded to a run length encoder that packs neighboring foreground pixels into intervals. The intervals are inserted into the interval table, where each line corresponds to a line of the camera image. After receiving a complete line, the interval table is used to look up the intensities of each foreground pixel in the interval. Subsequently, the intensities are sent to the computer by following a custom data transfer protocol. The PC performs blob detection on received foreground pixels and extracts features from every detected blob. This section describes design and implementation of this approach.

3.1.1 Two Thresholding Variants

Regions of interest are identified by applying a variant of thresholding. In simple thresholding, all pixels whose intensities exceed a predefined threshold are retained, and all other pixels discarded. However, in some applications, simple thresholding can result in a cutoff of

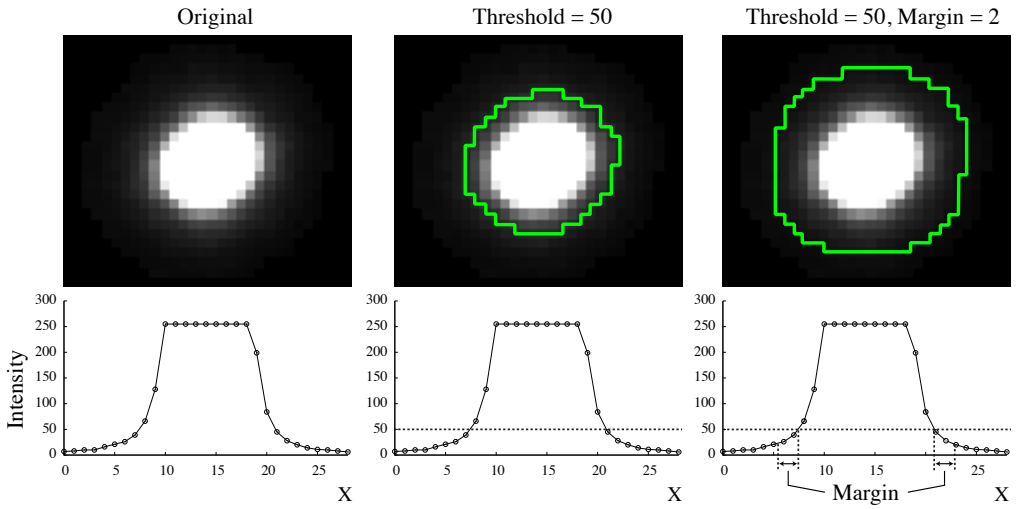

Figure 3: Two variants of thresholding used for identification of regions of interest. On the left is the original image. In the center, simple thresholding is applied. On the right, the border produced by application of a simple threshold is extended by a margin. The border is marked in green. Pixels inside the border comprise the foreground. The upper subfigures show the original image and overlayed borders. The lower subfigures show intensities of pixels obtained from the line that cuts through the center of the light spot along the X-axis.

important pixels on the edge of a blob, which might compromise the precision of subsequent feature extraction. Figure 3 (up on the left) shows a typical light spot produced by a laser on an office wall (up). The lower part of Figure 3 plots the intensities of pixels obtained from the middle of the light spot.

Simple thresholding, as shown in the middle of Figure 3, with a threshold of 50, corresponds to drawing a line parallel to the X-axis and discarding all pixels that are below the line. If the threshold is set too high, pixels that are important for subsequent feature extraction will be cut off. However, if the threshold is set to a low value, too many unimportant pixels might be marked as foreground and negatively impact subsequent feature extraction. To overcome this problem, an additional margin extends the border obtained from applying simple thresholding. While the threshold dissects the image along the intensity axis, adding a margin extends the border along the X and Y-axes. The lower right part of Figure 3 shows the result of applying thresholding and margining on the intensities along the X axis.

3.1.2 Interval Table

Simple thresholding without a margin can be implemented in an FPGA by saving all pixels whose intensities exceed the threshold in a buffer and counting their number on the fly. After reaching a background pixel, or by reaching the end of a line, the number of the foreground pixels and the intensities can be transferred to the computer. However, when applying a threshold with a margin, data transfer has to be deferred until at least *margin* lines have been processed.

To implement thresholding with a fixed margin on an FPGA, an auxiliary abstraction called *interval table* has been developed. An interval table, as shown in Table 1, consists of several rows, each containing a fixed number of intervals. An interval is defined by its *left* and *right* coordinates (inclusively). Adjacent foreground pixels in a single line are grouped together into intervals. Rows of an interval table correspond to lines in an image. All rows are sorted by their *left* coordinates. If a line contains no foreground pixels, no entries will be written into the interval table. The size of the interval table is defined by the margin M and the number

Table 1: Abstract representation of an interval table.

	0	1	...	N
0	left, right	...	...	...
...	...	...	...	...
y	...	...	...	...
...	...	...	...	...
2*M + 1	...	...	...	...

of intervals N. The margin defines the depth of the table, whereas the number of intervals defines its width.

The approach for identification of regions of interest works by raster-scanning an image and gradually inserting new entries into the table. The table must be maintained in a way that no intervals overlap, which can be done by merging overlapping interval. This is accomplished by keeping pointers to the interval that is adjacent to the current X-coordinate. When a new interval is ready to be inserted, it needs to be checked whether it overlaps with some existing interval of those already stored in the interval table. If this is the case, the old interval will be merged with the new one by taking the minimum of their left limits and the maximum of their right limits to produce a new interval. This algorithm can be run in parallel for every line stored in the interval table.

Sometimes, a new interval overlaps with several intervals stored in the interval table. In this case, all the overlapping intervals have to be merged into just one interval. Since the pixels arrive in raster-scan order, it is easy to remember how many intervals overlap with the currently active interval and update them when the current interval is finished. This can be achieved by again finding the minimum and maximum of the last overlapping interval and the new interval that is ready to be inserted. However, this time all overlapping intervals must be merged. This can be done by shifting the whole line one time for each overlapping interval.

When end of line is reached, the whole table is shifted up, and the most upper interval list is ready to be sent. For this purpose, it is copied into a temporary register, from where the data transfer circuit takes the information. When the whole table is shifted up, an empty interval list is inserted into the table from the bottom. There are several special cases in the beginning and at the end of each frame. In the beginning, the algorithm should wait for the first $margin$ lines to be processed, before shifting up the interval table. And $margin$ lines before the end, the algorithm should stop updating the interval table, otherwise it will overwrite the data for the next frame. After shifting up the most upper list of the interval table, the intervals stored in the list can be used to extract regions of interest for the particular line represented by the list. Since the information about the foreground pixels in a line becomes apparent only M lines after their encounter, at least M lines of the image have to be stored in a frame buffer. Pixel values of all the intervals in the most upper list can be extracted by knowing *left* and *right* and the current Y-coordinate. This information is enough to calculate an address in the frame buffer, where the corresponding pixel values can be found.

In VHDL, the interval table has been realized as an array of arrays. Each entry in the array is 21 bits wide—10 bits for the left and right coordinates, respectively, and one additional bit to denote that the interval is active. Figure 4 shows the high-level overview of how this approach is implemented. The interval table finite state machine (FSM) shown in the center of the figure manages the interval table based on the pixel values that it receives from the camera, as it has been described in the previous paragraphs. The most upper interval list of the interval table is

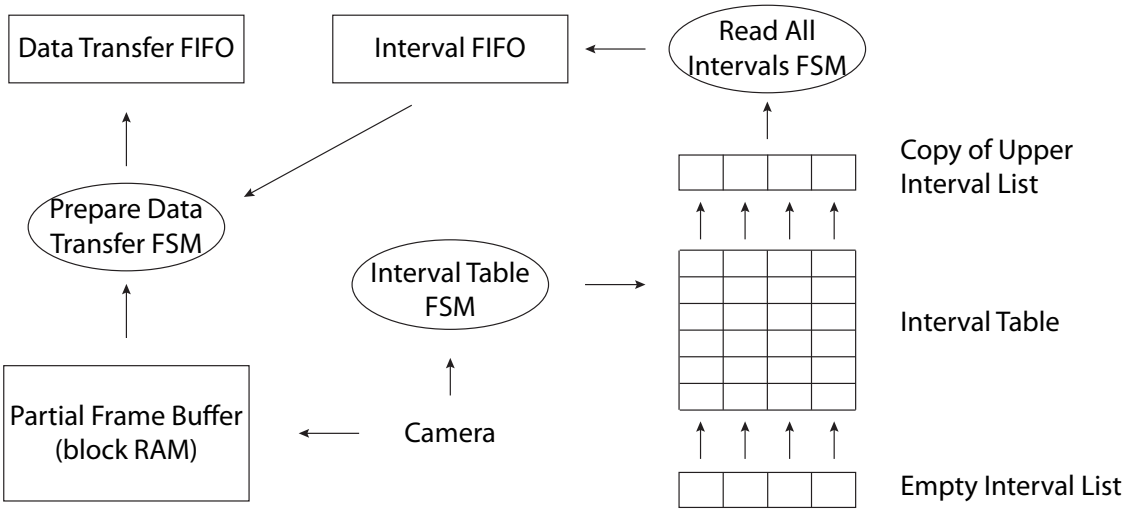

Figure 4: Interaction of the interval table with the other parts of the system.

constantly read out by the other FSM that, upon finding an active interval, writes it into the interval FIFO. The interval FIFO is constantly polled by a third FSM that uses the intervals for retrieval of pixel values from the partial frame buffer. The Y-coordinate of the interval, together with the *left* and *right* coordinates, as well as all pixel values of the interval are written into the data transfer FIFO.

The data transfer FIFO is 16 bits wide, such that the Ethernet controller can send data without interpretation. However, the Ethernet controller needs to know how large the piece of data is, before it sends it. Therefore, this value is written before the coordinates of an interval and the corresponding pixel values. This is because there is a limit on how many bytes can be sent in one Ethernet packet. Another reason for providing the number of bytes to be written, is that starting a new packet with a new interval is safer than distributing an interval over different Ethernet packages. If an error occurs during transmission, it is possible that the host PC completely looses track of what the data means.

The advantage of this approach is that the addresses of pixels of interest can be immediately calculated. Another advantage is that the size of an interval is known beforehand, and can be written right away into the Ethernet transmitting buffer. It is necessary to transmit the size of intervals because otherwise the PC cannot know what in the data stream is a pixel value and what is the coordinate. The computer needs to know how many pixels are there before it starts reading them.

The drawbacks of this approach are that if there are more foreground runs than there is space allocated for the intervals, it will result in a unwanted overwrite of the interval in the interval list, and the pointer will be reset to the zeroth interval, which will corrupt the whole interval table. This will force the Ethernet controller to send the same data several times.

Another drawback of this approach is that it requires a large amount of logic elements, because of the interval updating circuitry that needs to be placed beforehand for every possible interval.

3.1.3 Bitmap Approach

Instead of storing intervals, an alternative approach is to store for each pixel in the image, one bit that indicates whether the pixel is a foreground or a background pixel. This approach is better in terms of resources required since no shifting is involved—only the pointer is incremented and a window written. The main obstacle of this approach is the amount of memory required. This results in a sliding window approach that slides over the registers and updates a set of registers for each foreground pixel. It is possible to reduce the amount of memory by realizing that only a small portion of the actual bitmap needs to be updated on arrival of every foreground pixel.

This approach is expensive in terms of required logic elements, because any of the 6 neighboring pixels may be updated at any time. Because the circuit has to be realized beforehand, a large amount of logic elements has to be dedicated to the update circuitry. The advantage of this approach is that it allows a more flexible blob handling, because the image can be processed in a wider range of ways on the host PC. In addition, developing algorithms on a general-purpose computer takes less time, is more flexible, and easy to adapt to similar tasks. It is easier to add new features, or change the approach, because of the relatively short amount of time required to develop a program for a PC. However, this approach works as long as the ratio between foreground and background pixels is small enough. In the evaluation section, one of the goals is to find out the exact ratio between foreground and background pixels that allows reliable image transmission without data loss.

3.1.4 Blob Detection on PC

A simple approach to blob detection on the computer has been realized. Its underlying idea is to examine every pixel that exceeds a fixed threshold and determine whether any of its 8 neighbors also exceed the threshold. Such pixels are grouped together. After the whole frame is received and processed in such way, the centers of mass are calculated by finding the average coordinate weighted by their corresponding intensity values. This is done for each group of neighboring pixels. The resulting centers of mass are visualized on the screen, which allows us to quickly determine whether the FPGA sends correct data or not. This approach is not optimized to take advantage of structure of incoming data and was only used for visualization purposes. However, such optimization is planned for the future.

3.2 Data Transfer

The DE2-70 board provides different ways to communicate with the host computer. It has a RS-232 serial port, a USB 1.0 connection, 100Mbps Ethernet, infrared IRdA, and allows the connection of additional hardware over two GPIO expansion connections. Data throughput, latency, and compatibility with a wide range of devices are the most important criteria when choosing an interface. Ethernet is the fastest connection on the board that is available from the start. It is possible to achieve higher data throughput by connecting expansion boards to the two GPIO connectors available on the board.

However, the idea behind this project is to have a wireless communication between the mobile camera and the host PC. And the usual transfer rate of a commonly used wireless communication, such as 802.11a and 802.11g is 54Mbps. The speed is the maximum achievable speed, that is only a theoretical value and will be slower in real world usage. The main advantage of using Ethernet is the ability to connect to a wide variety of devices with no extra development effort.

It is also possible to use the USB connection and attach a USB adapter that transmits data over WLAN, however, there is a wide variety of devices that all use different ways to interact with the host device. In case of a USB wireless device, a driver on top of the existing USB driver needs to be written in order to communicate with the USB device. To achieve a result that can be used with a wide range of devices, it is more advisable to use the Ethernet interface. In the next sections, the custom protocol for communication between the FPGA board and host PC developed in this project is described in detail.

3.2.1 DM9000A Ethernet Interface

The Ethernet interface of the DE2-70 board is a 10/100 Mbps Ethernet MAC controller that supports full-duplex mode that allows the interface to send and receive data at the same time. The chip has a 16 KByte SRAM buffer for sending and receiving packets, where 3 KByte are devoted to packet transmission, and 13 KByte to packet reception.

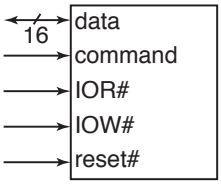

Figure 5: General processor interface of DM9000A Ethernet controller. Adapted from [9]

As shown in Figure 5, the DM9000A chip has a bidirectional 16-bit data port, a command pin, two active-low pins for reading and writing the internal registers, and one active-low

reset. By setting INDEX or DATA command, it is possible to choose an address or write data, respectively. When reading, the *nRD* signal has to be set low, and the output data pins have to be set to high impedance ('Z' in VHDL).

Before the Ethernet interface can send data, it has to be initialized. The internal PHY chip that is responsible for interaction with the physical layer, has to be powered up, then the interface has to be set and reset several times. Subsequently, some internal registers should be set to facilitate the interaction with the interface. For example, it is possible to enable pointer auto return function, which automatically resets the read/write address pointer of the RX/TX buffer when the pointer reaches the maximum possible address.

Before writing into the transmit FIFO, the automatic memory management (AMM) should be started. This will automatically increment the pointer to the address in transmitting buffer. It is also possible to write into the FIFO without automatic memory management, by manually changing the TX/FIFO pointer, but AMM is more efficient, because it allows us to write one word every 2 clock cycles, as opposed to one word every 4 clock cycles. This overhead results from the necessity of manual pointer increment, that requires one write into the index port and one write into the data port.

3.2.2 Ethernet Driver

In order to send data with the Ethernet interface, a driver circuit has been written in VHDL to accommodate the needs of this project. The driver circuit initializes Ethernet interface and provides means to send packets. A state machine has been developed to handle Ethernet initialization and to send different data types. To facilitate the initialization phase, a high-level programming language [10] has been used to produce portions of VHDL code. The instructions necessary for initializing the Ethernet interface have been written in a high-level programming language that was then translated into VHDL source code. Instructions are saved in the look-up tables of the FPGA. It is a primitive form of assembler, where the instructions are fetched, interpreted and executed. Following instructions have been implemented:

- Write INDEX
- Write DATA
- Wait(ms)

Without this abstraction, writing the initialization sequence in VHDL would result in a large state machine with many states, that had to be written manually. This would make the code difficult to change and maintain. However, software-generated code is more flexible.

After initialization, the interface is ready to send and receive packets. Figure 6 shows a high-level finite state machine that controls data transmission. In order to send a packet, first the data must written into the transmitting buffer of the Ethernet interface. After writing the buffer, the size of the packet must be written to a special register. Then, transmission

request command has to be issued. Before sending a new packet, the driver has to wait for confirmation that the packet has been sent successfully, otherwise packet transmission might result in corrupted packets. The finite state machine for data transfer is connected to a FIFO with the data to be sent. This data has been written in a way that the FSM does not need to interpret it. The only information that it extracts for its own use is the number of bytes of the type of data. From this number, the FSM knows how many times it should read from the FIFO before next piece of data appears in the FIFO. Packet reception has not been covered, since it was not needed in this project.

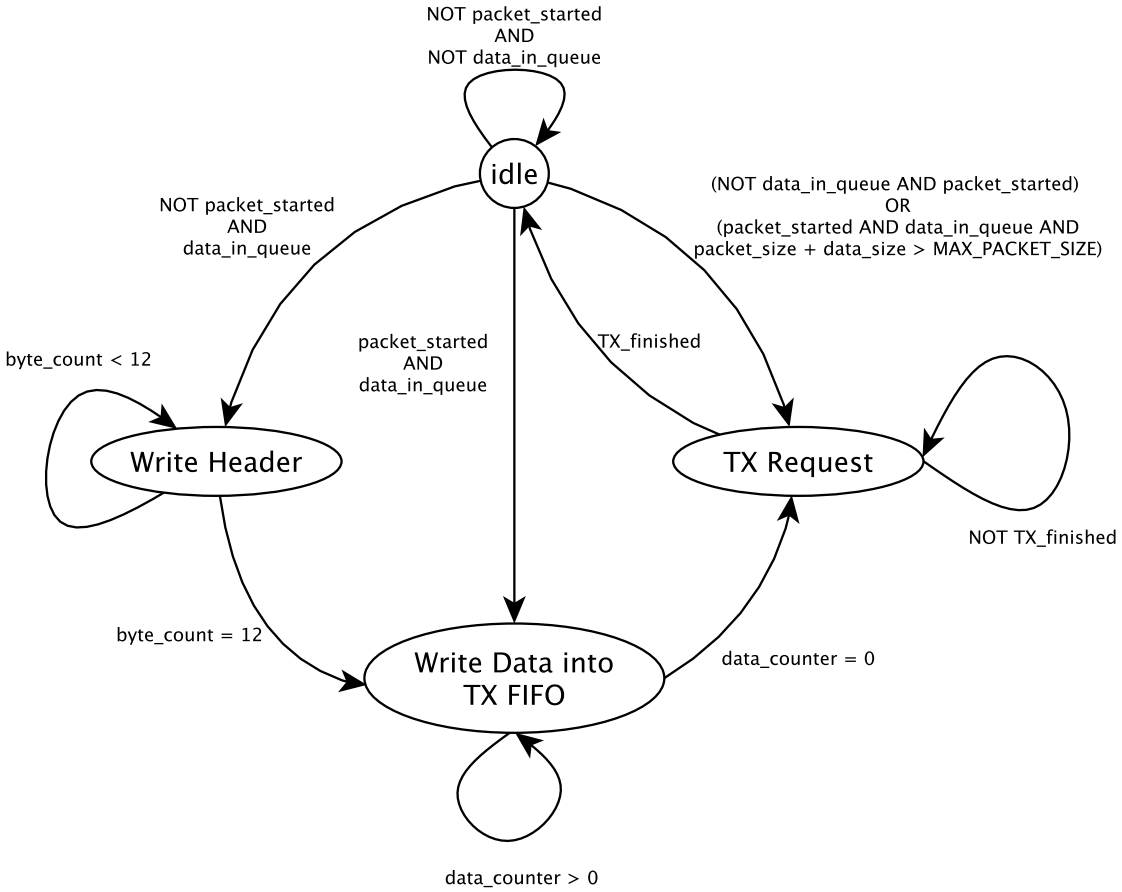

Figure 6: Finite state machine for data transfer.

When interacting with Ethernet interface, it is important to consider the timings for reading and writing. A failure to take the timings into account results in unpredictable behavior, which makes it very difficult to debug the driver. If the timings are ignored, it might happen that the driver is not properly initialized and does not start, or does not work properly. For example, after sending a packet, it is necessary to wait for completion before sending a new packet. An attempt to send a new packet while the previous packet is still being sent results in a corrupt

packet that can only be discarded by the receiving side. It might shift the data in a way that the preamble does not match with the predefined MAC addresses of source and destination systems.

3.2.3 Custom Data Transfer Protocol

A custom protocol has been developed for sending results of image processing to the host PC. The program that receives data on the host PC needs to know the meaning of each piece of data that it receives. Thus, extra data needs to be sent to denote the types of data sent over communication channel. One of the extras is to distinguish the packet from the board from all other packets. Even if the PC is only connected to the board, there are many packets sent over the Ethernet interface which have nothing to do with our task at hand. They are the standard packets that are usually sent when a cable connects two Ethernet devices (ARP requests, DHCP requests, etc.). In order to make the packets from the board distinguishable from all the others, a unique header is written in the beginning of every packet. The header consists of the MAC addresses sender and receiver. Sender MAC address was fixed to 00:11:22:33:44:55:66, and the receiver MAC address fixed to the MAC address of the Ethernet interface of the host PC. The maximum packet size has been chosen to be the standard Ethernet packet size of 1500 bytes.

As discussed in the previous sections, two basic designs have been developed in this project: a system that transmits regions of interest to the PC, and a system that performs blob detection on the FPGA and transmits the results. If the task of data transfer is approached in a generic way, it is possible to avoid developing two different circuits for sending data to the PC as well developing two different programs for the host PC that receives the data. This can be achieved by introducing different types of data. In this way, we only need one circuit that can handle different types of data, and only one program on the host PC, that, depending on the data received, either visualizes the centers of mass data received from the FPGA, or calculates the centers of mass from the received pixel values.

Different types of data can be sent from the board to host PC. A necessary data type is the end of frame, since the host PC can only guess when a piece of data belongs to the current frame or to the next one. It is possible to guess when the Y-coordinate is less than that of the previous blob, but it is not reliable. Thus, it is necessary to tell the host PC which packets belong to one frame and which packets belong to the next frame. This is accomplished by sending specific data that signals the end of frame, meaning that all the data that comes after it already belongs to the next frame. Two bytes have been allocated to denote the type of data. There are three types of data that can be sent over the Ethernet, which are as follows:

- End of frame: 0xC000
- Center of mass: 0x0C00

- Region of interest: 0x0300

Center of mass and regions of interest are mutually exclusive, because the system either sends the centers of mass, or the regions of interest, but never both together. However, the end of frame is used in both designs. Regions of interest are the parts of the image that are considered to be important for blob analysis. In this case, these are the pixels that exceed the threshold and are around the margin, as discussed in the previous sections. The protocol for sending regions of interest, is as follows: $<$ *type of data*, *y*, *left*, *right*, *data* $>$. *Type of data*, *y*-coordinate, *left* and *right* together require 46 bits and can be packed in three words. Subsequently, pixel values are written into the transmission FIFO packed together in 16 bits.

3.2.4 Packet Capture on Host PC

For capturing the packets sent from the board on the PC, the Berkeley Packet Filter was used [11]. It provides the ability to access raw packets received over a network interface. It is necessary to access raw Ethernet packets because the protocol developed in this work only uses the Ethernet layer. In order to read a packet from Ethernet interface, one of the free device files with the names /dev/bpf0, ... /dev/bpf10 must be opened first, and the resulting file descriptor must be bound to the Ethernet interface of the machine. Now the packets can be accessed. The packet filter can be instructed to only look for specific packets, e.g., all packets addressed to a machine with a specific MAC address. In our case, all packets sent from the board contain the MAC address of the host PC and the source MAC address of the board, which is fixed to 00:11:22:33:44:55. This allows us to set up a filter that provides our program only with packets sent from the board, and ignores all other packets.

As an alternative approach, the PCAP [12] library has been used in the beginning. The PCAP library provides similar ways to access Ethernet interface and to handle raw packets, however, the BPF was found easier to use. Since the host PC is the only connection to the board, it was not necessary to develop a protocol that could be sent on higher levels, such as IP, UDP, etc. For protocols on a higher level, such IP, UDP, etc., a more abstract library, e.g., sockets, would be a better choice.

3.3 Blob Detection on FPGA

In the beginning of this project, the goal was to perform blob identification on an FPGA and to send the dimensions of the bounding boxes and the pixel values enclosed by the bounding boxes. The approach has been developed until it was noted that no external memory controller was available for storing the whole image in an external memory of the DE2-70 board. At that moment, the project has focused on developing an approach that does not require an external memory, which resulted in the approach that was presented in the first part of this section. Here, we present the old approach that was put on hold in course of this project.

The approach is similar to the single-pass connected component labeling presented in [8]. The basic idea is to label each pixel in the frame with a number based 8-connectivity principle. The labels are only used to extract features of objects on the fly and are discarded as soon as the end of the frame is reached.

3.3.1 Single Pass Connected Component Labeling

Figure 7 illustrates the approach to single-pass connected component labeling that was developed in this project. At the core of the approach is a finite state machine that checks a long list of conditions in order to decide how to label foreground pixels.

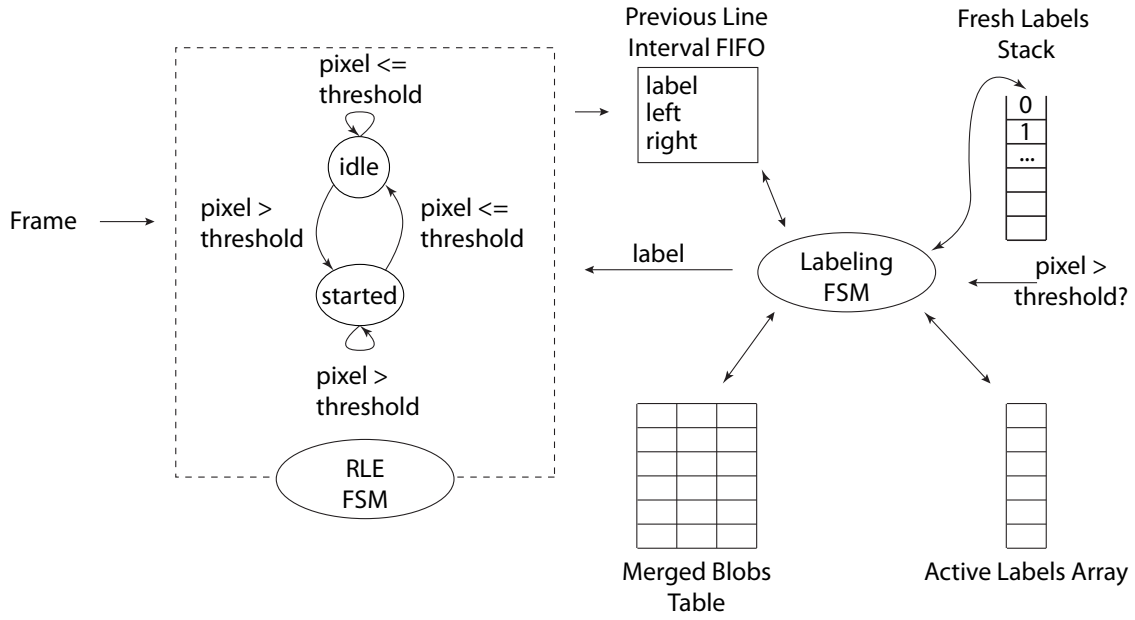

Figure 7: Connected component labeling.

Foreground pixels are determined by comparing their pixel values with a predefined threshold. If the pixel value exceeds the threshold, then it is a foreground pixel, otherwise, it is a background pixel. The source of labels is provided by a stack that is filled with decreasing numbers before blob detection is started. Whenever the main FSM needs a fresh label, it obtains it from the stack and pops the label, and in this way makes the stack ready for the next read.

Upon the assignment of a new label, two events happen at once: 1) the label is saved as active in the active labels array, which is done by using the label as address and writing a 1 under that address; 2) the label is provided to the run-length encoder FSM, that saves the run that just ended together with the provided label in a FIFO. The run (or interval) that is stored in the FIFO will be read in the next line when its boundaries are reached. This allows

the labeling FSM to find out whether a foreground pixel is adjacent to the run in the previous line. The FSM compares the left coordinate of the interval with X-coordinate of currently active pixel, and sends request to dequeue the FIFO in case the X-coordinate is larger.

In some situations, it is necessary to merge two existing objects, because the current run connects them with each other. In such cases, a table that maps labels to other labels is used to store information about merges. It might happen that a run is a neighbor to several runs. In this case, all runs are merged, and for every merged run, an entry is noted in its corresponding merge table. When merging a blob that has been merged in the same line already, the merge table is looked up first, and the correct blob is read from the merger table. If both blobs have been merged, one of them will be chosen to contain the new merged blob. [6] presented an approach where the merge table is kept in block RAM of the FPGA. Block RAM introduces a delay of one clock cycle for every read or write. Another constraint is that the content of the memory cannot be accessed all at once, and must be accessed in sequential manner. Keeping the merger table in block RAM does not allow us to check two entries at once, which is resolved in this paper by noting that all merges can be processed during the horizontal blanking interval of the camera. If the merge table is kept in the lookup tables of the FPGA, it is possible to randomly access the entries in the table without one clock cycle penalty, which is one of the differences of this approach to the approach developed in [6].

Before connected component labeling can be started, several initialization steps have to be performed. One of the steps is to completely empty the stack and then to fill it completely with distinct labels. If the stack is large, i.e., when a large number of blobs should be detected, then it takes long time to pop and fill the stack completely. This delay might harm subsequent labeling. To avoid this problem, two stacks should be used—while one stack is actively used by the blob-updating FSM, another stack can be safely emptied and filled with distinct labels.

An observation in hindsight is that run-length encoding is not necessary if the previous line is buffered in the block memory of the FPGA. This strategy is used in the approach by [8].

All runs identified in a line are pushed into a queue, and read on the next line. This is the core of connected component labeling. The queue provides the context that helps the algorithm decide whether a run belongs to an existing label or forms a new label.

3.3.2 Blob Analysis

Connected component labeling labels every foreground pixel with the number of the object to which it belongs. After labeling every foreground pixel, the objects can be analyzed and their features extracted. As discussed earlier, the underlying problem is to compute camera pose (position and orientation of the camera), which can be done by knowing the center coordinates of the blobs. Since the blobs are arranged in a predefined way, it is possible to calculate camera pose from knowing the center coordinates of the blobs. For this reason, the only feature that

needs to be extracted is the center of mass of each blob.

In order to find camera pose, it is necessary to know the coordinates of the blobs' centers. It is possible to find the center of gravity on the fly, while performing object labeling. This approach has the advantage that the image does not need to be stored in a memory for a second pass analysis. The center of mass can be computed by summing up the coordinates of each pixel multiplied by their respective pixel values and normalized by the accumulated pixel values in the end.

$$x_{COM} = \frac{\sum_i^n x_i \cdot I_i}{\sum_i^n I_i}, y_{COM} = \frac{\sum_i^n y_i \cdot I_i}{\sum_i^n I_i},$$

where n is the number of pixels that belong to an object, i is the pixel currently being processed, and I_i is the intensity value of pixel i. For each blob, only three numbers have to be stored: the accumulated sum for X-coordinates $\sum_i^n \mathrm{x}_i \cdot \mathrm{I}_i$, the accumulated sum for Y-coordinates $\sum_i^n \mathrm{y}_i \cdot \mathrm{I}_i$, and the accumulated intensity $\sum_i^n \mathrm{I}_i$. After the blob is complete, and no additional pixels have been joined to it for two subsequent lines, last step in computation of the center of mass can be performed by dividing the accumulated multiplication by the accumulated intensity values. When blobs are merged, the accumulated values and areas can be combined by adding them. If in two subsequent lines the blob has not been extended, it is considered as complete, and its features can be sent to the host PC. As discussed in [2], it is possible to compute additional features on the fly, however, in this project only the centers of mass are considered.

The synthesis tools do not support division other than by numbers that are a power of 2 because it is too complex to be placed on an FPGA. This means that the circuit for performing division has to be either developed manually or included into the design as an intellectual property (IP). Altera provides a division IP, where the user can set the bit-width of input and output signals. Only two such circuits are needed, since at any point in the X/Y coordinates, only one blob can be completed two lines before. The required bit-width of input and outputs of the circuit can be calculated by assuming the worst case. The division IP core provided by Altera allows the user to select the bit width of the input signals and the output latency. For 64 bits output (43 bits for representation of mantissa and 11 bits for the power) it introduces at least 10 clock cycles delay, however, the division can be pipelined.

4 Hardware

This section describes the most important hardware that was used in this project.

4.1 Altera DE2-70 Development Board

In this project, Altera DE2-70 board developed by Terasic was used [13]. The core of the board is a Cyclone II 2C70 FPGA device with 68,416 logic elements, 250 M4K RAM blocks, 150 embedded multipliers, and 4 PLLs. The board has several switches and buttons that can be used to interact with the board. It has many different connections that can serve as input and output.

4.2 Matrix Vision mvBlueCOUGAR-X 100

The mvBlueCOUGAR-X 100 is a camera with a CMOS sensor that delivers 10-bit grayscale images with resolution up to 752x480 pixels and allows frame rates up to 117 frames per second [14]. It is highly configurable over an Ethernet connection. It is connected to the DE2-70 board over one of the two GPIO expansion headers provided by the board.

5 Development Tools

Several development tools have been used in this project in order to write the code, simulate, debug, and load designs onto the DE2-70 board. The most important tools and their features are described in this section.

5.1 Quartus II Web Edition v11.1

Quartus is an IDE for design entry, synthesis, placement & routing, and programming of Altera's FPGAs [15]. In addition, it allows the users to include custom IP cores, to perform timing analysis, and to debug their design while it is running on an FPGA. In the following two sections, IP cores and Signal Tap Logic analyzer are described in more detail.

5.1.1 IP Cores

Some IPs have been used to accelerate system design. IP cores are designs that have been extensively test and verified. They can be used as building blocks in ones own design, thus, saving valuable development time, and allowing the programmers to concentrate on more important problems. IP cores used in this project are described in the next paragraphs.

Phase-Locked Loop (PLL) Cores PLLs are used for overclocking the available clock signals. An advantage of PLLs, as opposed to clocks derived from clock oscillators manually, is that PLLs send clock over dedicated clock circuitry. This allows the user to control the delay of the clock. In this project, PLLs were used to derive clock Ethernet and to test the performance of the developed systems with higher clock speeds.

Block RAM Altera provides an IP core for using the block RAM of the FPGA. Designs that use block RAM leave more space for application logic. In many cases block memory Block RAM introduces one clock penalty every time when data is written or accessed. In cases when such penalty is not acceptable, FPGA registers should be used instead. In this project, Altera's block RAM core has been used to temporarily store the image during shared blob detection. Parts of the image were then sent to the host PC over the Ethernet.

Synchronous FIFO Synchronous FIFOs are used in several places in this project. For example, they are used to store adjacent runs of the previous line in connected component labeling. In the beginning, synchronous FIFO developed in [16] was used. However, this implementation uses the lookup tables of the FPGA instead of the block RAM. Later in the project, all designs were migrated for using Altera's own synchronous FIFO core that uses block RAM. The advantage of Pong Chu's FIFO is its flexibility that allows to use generic constants that can be changed

in one place, but when using IP cores provided by Altera, the dimensions of the FIFO have to be specified manually, every time when some parameters change. However, in the long term, as the design is getting clearer, this flexibility is not required, and most of the parameters are fixed.

Stack The stack is used in one place—it provides fresh labels in connected component analysis. The IP core was derived from Pong Chu's synchronous FIFO (it is offered as an exercise to the reader in his book) [16]. The disadvantage of this approach is that it does not use block RAM. In order to overcome this disadvantage, the implementation was extended to use block RAM by using memory IP described earlier.

5.1.2 Signal Tap Logic Analyzer

Altera's Signal Tap Logic Analyzer was used extensively during hardware debugging. It provides the ability to get real data from the FPGA while the system is running. The user can specify which signals should be recorded and the conditions that trigger data recording. The data is saved in the block RAM of the FPGA and fetched over the JTAG USB connection.

5.2 GHDL and GTKWave

GHDL is an open source tool for simulation of VHDL testbenches [17]. VHDL code is converted into a C program that is then compiled and executed on the computer. During execution, all changes of the signals can be saved into a *.vcd* file. Testbenches developed for GHDL usually work in ModelSIM (a commercial HDL simulator) as well. GHDL was used to allow rapid development, testing, and debugging of an approach. Most of the approaches were first developed and tested by using GHDL, and ported to Quartus in later development stages. In contrast to the Signal Tap Logic analyzer, it is possible to save data from all the signals, because no memory limit exists. For the same reason, GHDL does not require any triggers.

In order to visualize the waveform files produced by GHDL, GTKWave was used. GTKWave is an open source tool for wave visualization [18] that supports many different wave formats. It allows the user to inspect all signals that exist in a design.

5.3 Wireshark

Wireshark is an open source tool for analysis of Ethernet packets. In course of this project, it has been used extensively to verify correctness of the custom data transfer protocol and traffic analysis during evaluation. The user has the ability to inspect every received packet and see how many bytes are sent in each protocol layer. Figure 8 shows a screenshot from the program as it has received some packets from the MATRIX VISION camera.

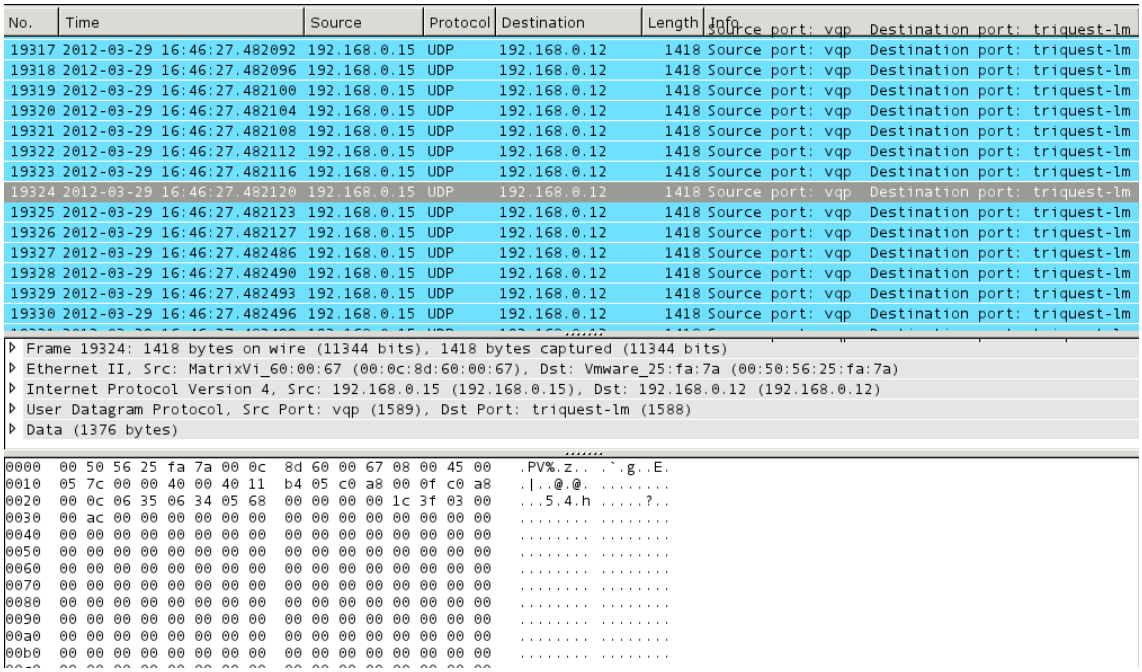

Figure 8: Screenshot of Wireshark that has captured packets from the MV camera.

6 Verification

It is necessary to verify that the developed systems work as expected. Verification was done during development, verifying each module individually, then their combinations, and finally, full systems. The developed systems are verified in simulation and hardware. This section provides detailed information about verification of final systems in simulation and in hardware, and omits the details for smaller modules.

6.1 Verification in Simulation

Most of the modules developed in this project have been verified in simulation before their verification in hardware has been done. This has several advantages as opposed to verification in hardware. The main advantage is that a simulation gives a full overview of what the system does. It is possible to see the changes of every signal in the design, as well as to print out intermediate values in the console or into a file. The file can be further processed with a programming language to check correctness of simulated design or to visualize the results. Another advantage is that the simulation allows faster development, because it takes less time to simulate a system than to place & route it on an FPGA. These properties make simulation a better tool to test and debug the system in the early development stages.

6.1.1 Regions of Interest Identification

Figure 9 shows how the approach to identification of regions of interest (RoI) is verified. A testbench written in VHDL reads test images from the hard disk and directs pixel values to the module that identifies regions of interest. Subsequently, the module separates foreground pixels from the background by using a threshold, and makes the results available to the testbench, which saves them in a custom format file. The custom format file encodes the foreground

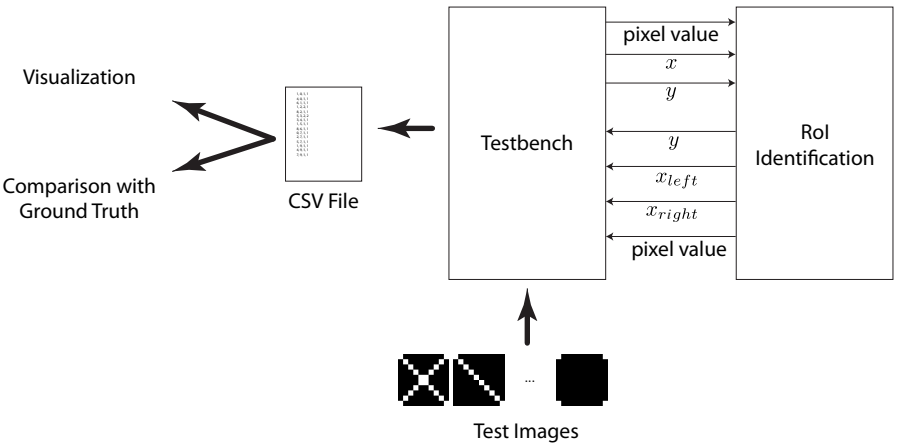

Figure 9: Verification of regions of interest (RoI) identification in simulation.

pixels by using intervals and pixel values and is of the form:

$$y, \text{start}, \text{end}, \text{pixelValue}_{\text{start}}, \ldots, \text{pixelValue}_{\text{end}}$$

The file is used to visualize foreground pixels and visually verify the correctness of results. Test images are either produced by hand or generated by a computer, and are saved as PNG files. Subsequently, they are converted into custom format bitmaps by using a high-level programming language. The custom format represents each line of the image by tab-separated grayscale values. These custom bitmap files are read by the VHDL testbench with help of a standard library for text I/O: *textio.vhd* from the *std* library. The testbench reads pixel values from the bitmap file and sends them over the simulated wires to the design under verification, as if it were a camera.

After RoI identification is finished, the testbench module saves the results on the hard disk in a file. For this purpose, an additional VHDL library called *txt_util.vhdl* (available from [19]) is used. It facilitates printing messages onto the screen and into files. In this way, the results can be saved in a file and used for comparison with the ground truth or for visual inspection. Figure 10 shows 6 test images and the results of applying RoI identification with margin equal to 1 (upper part) and 2 (lower part).

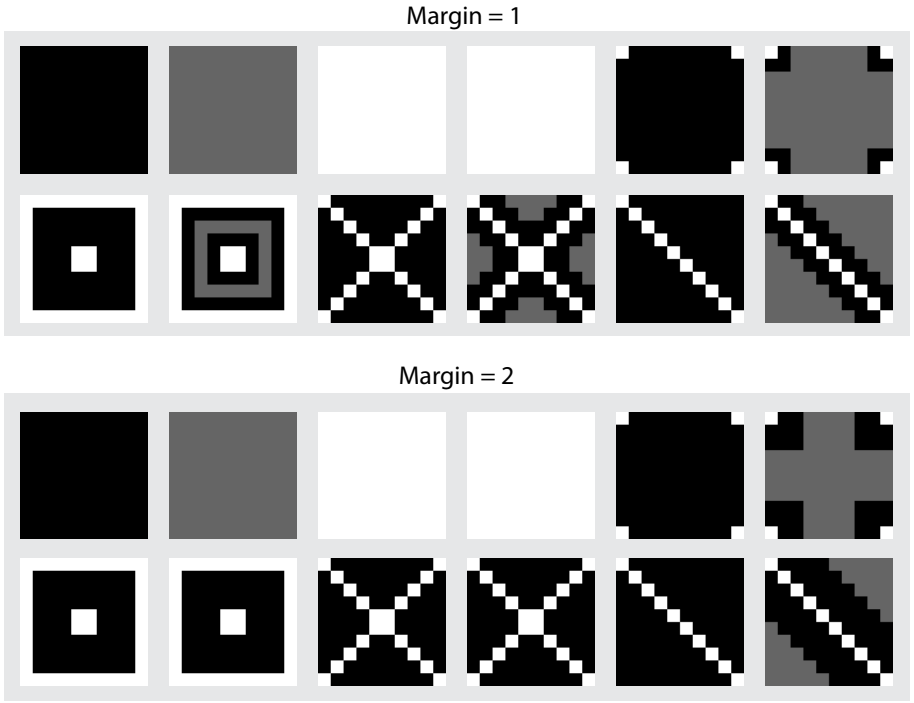

Figure 10: Test images and results of identification of the regions of interest. All images are 10x10 pixels. Two different margins have been used. Background pixels are black, foreground pixels are white, pixels that are thrown away in the process are dark gray.

6.1.2 Connected Component Labeling

Connected component labeling implementation is verified in simulation as shown in Figure 11. The verification strategy is similar to the strategy used to verify RoI identification. The basic idea is to let the testbench read test files and to provides pixel values to the connected component labeling module. Finally, the testbench prints the results of CCL into a file upon completion. In our implementation, the results of connected component labeling are only used to compute the features on the fly and discarded right away. Thus, it is not possible to acquire the labels for each foreground pixel from the CCL module, but only the labels for the extracted features. For this reason, in order to verify correctness of the CCL module, the bounding box feature has

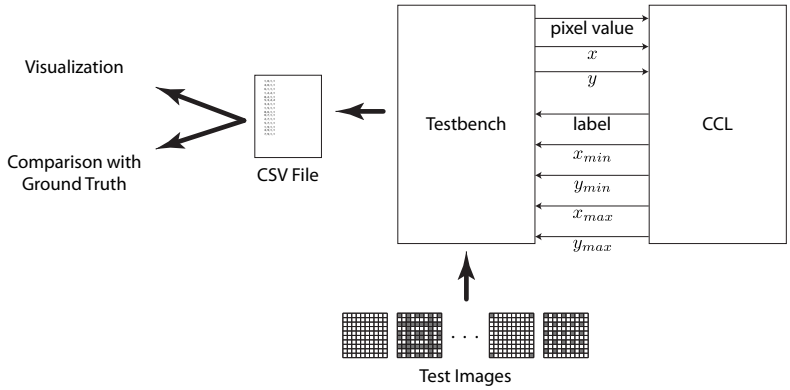

Figure 11: Verification of connected component labeling in simulation.

been used. For each feature, a label is associated. By using the label-feature pairs, it is possible to infer the labels for each pixel in the image. The labels and the dimensions of the bounding boxes are saved as comma-separated (CSV) values on the hard disk. The CSV file is used for verification between the original image and the results of CCL. Verification of correctness is done through visual inspection.

To test our implementation of connected component labeling, several test images have been developed. Some of these test images are shown in Figure 12. (a)—(c) show images that test for the simple cases. (d) shows how the algorithm deals with a simple merge. (e) shows that the algorithm is symmetric, i.e., it does not matter from which direction the image is raster-scanned (if the image is turned by 90°, 180°, or 270°), the algorithm always recognizes connected objects correctly. In this test case the label “3” is missing. This works as expected, because the CCL version developed in this project has not yet been optimized to recycle unused labels after merges. This issue will be addressed in future work. (f) shows that the algorithm can detect the maximum number of blobs in a 10x10 image. In images of this size, only 25 blobs can be drawn.

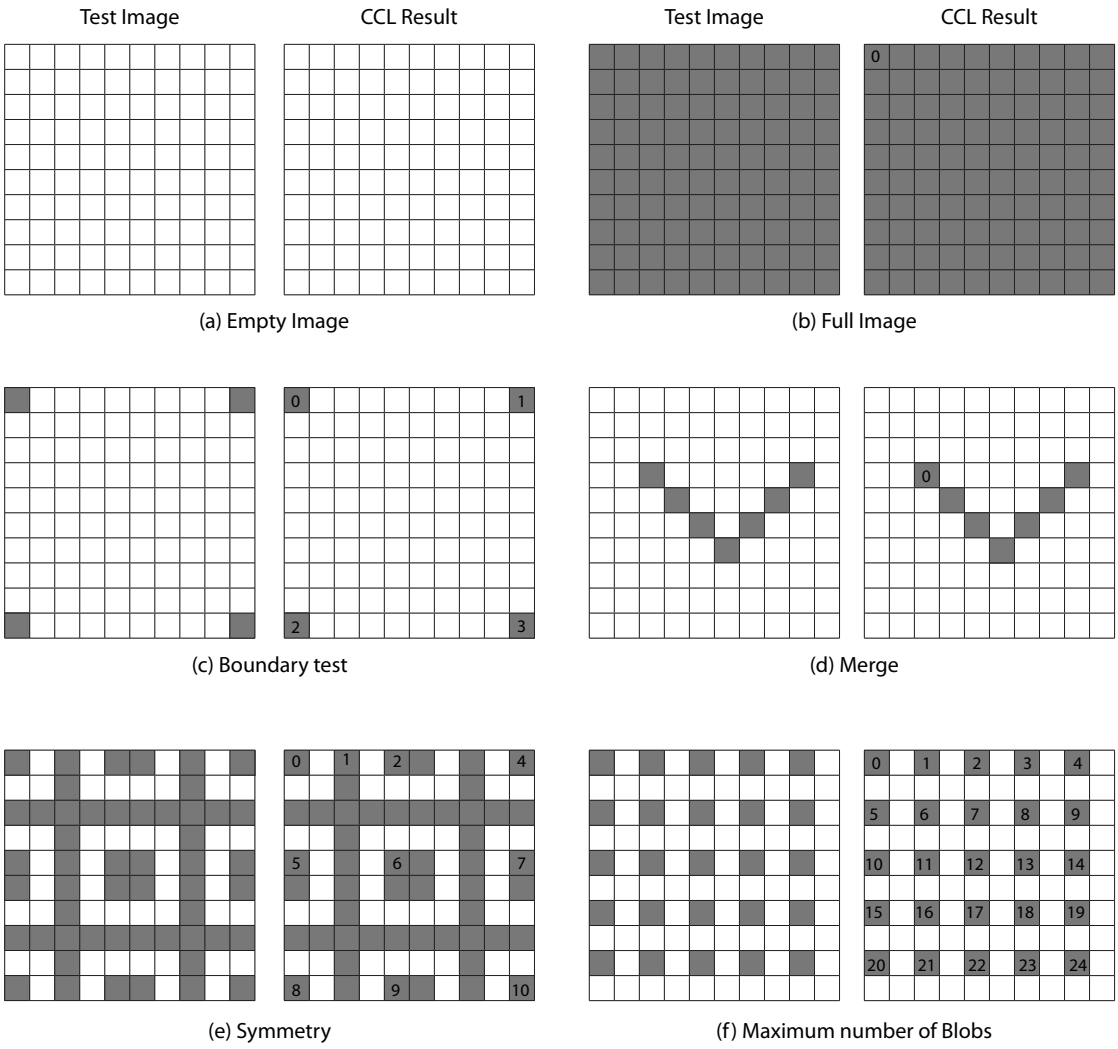

Figure 12: Test images for connected component labeling. Foreground pixels are dark and background pixels are bright (in real images it is vice versa).

6.2 Verification in Hardware

Simulation is a powerful tool for quickly prototyping, testing and debugging a design. However, the problem of simulation is that it does not guarantee that a simulated design will work in the same way when it is placed in real hardware. Simple simulation does not consider delays in circuits. In addition, in many systems, setup and hold timings are an important issue. If they are not considered correctly, the simulation might be successful, but tests in hardware might produce wrong results, or show wrong behavior for some small percentage of time.

Another problem arises when we try to simulate circuits that interface with external hardware, such as, e.g., external memory, serial port, Ethernet, etc. In order to test circuits that interface with external hardware, we also need to write circuits that behave like the external

hardware. For example, in case of the Ethernet interface, it is necessary to read and write some of its internal registers. This requires that the simulated Ethernet circuit behaves in the same way as the real hardware, otherwise the simulation is not valid. Another difficulty with simulating Ethernet controller is that it requires a complex initialization sequence, where some of the parameters are ambiguously described in the manual. In this case, it is impossible to find out which initialization sequence is the right one just by simulating the controller, and a verification in actual hardware is necessary. An additional problem arises when an IP core from Altera is used by the design. The cores cannot be simulated by using GHDL, as opposed to ModelSim. However, ModelSim is not used in this project. In order to overcome these problems, the system needs to be put to test in real hardware. This section describes the approach to verification in hardware and presents the proof of correctness for implemented designs.

6.2.1 Regions of Interest Identification

In order to verify correctness of the approach to regions of interest identification in hardware, the design is loaded onto the DE2-70 board and provided with real time camera data. Verification setup is shown in Figure 13. The blobs are produced by a USB-powered laser that can generate either a single large light spot or up to 121 small light spots when a special lens is mounted to its head.

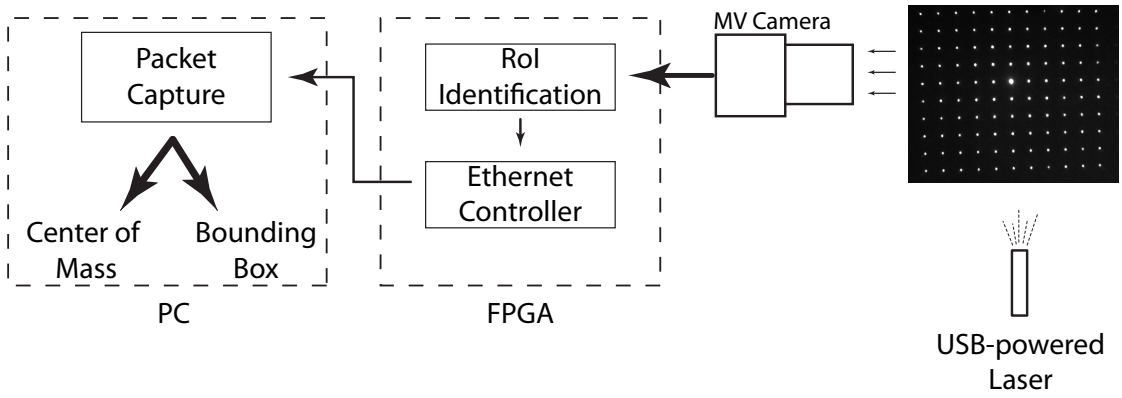

Figure 13: Verification of regions of interest identification in hardware.

The laser spots are projected onto a wall. Initially, the laser was used in combination with a smaller version of the Immersion Square developed by Timur Saitov. However, the light was absorbed by the semi-transparent walls of the smaller Immersion Square, and the resulting spots were too small. In the later stages of the project, the laser spots were simply projected on a wall, which produce larger spots. A MATRIX VISION camera mounted to the DE2-70 board captures the images and forwards them to the FPGA. The foreground pixels plus some small margin are extracted from the images and then transferred to the PC by using

the Ethernet controller of the DE2-70 board. In the next step, the computer reconstructs the images from received regions, detects blobs and extracts the center of mass for each blob. The results are visualized on the screen, and the user can capture any image by pressing a button. Image capture on the PC results in three images. The first image is reconstructed from the data that is received from the DE2-70 board. The second image is the result of blob detection with overlayed bounding boxes and centers of mass drawn for each blob—this is the image that is visualized for the user. The third image is a magnified-by-5 version of the second image with scaled bounding boxes and centers of mass—it is used for visual verification of the correctness of center of mass computation. The image was magnified by using the nearest neighbor algorithm, in order to preserve the edges.

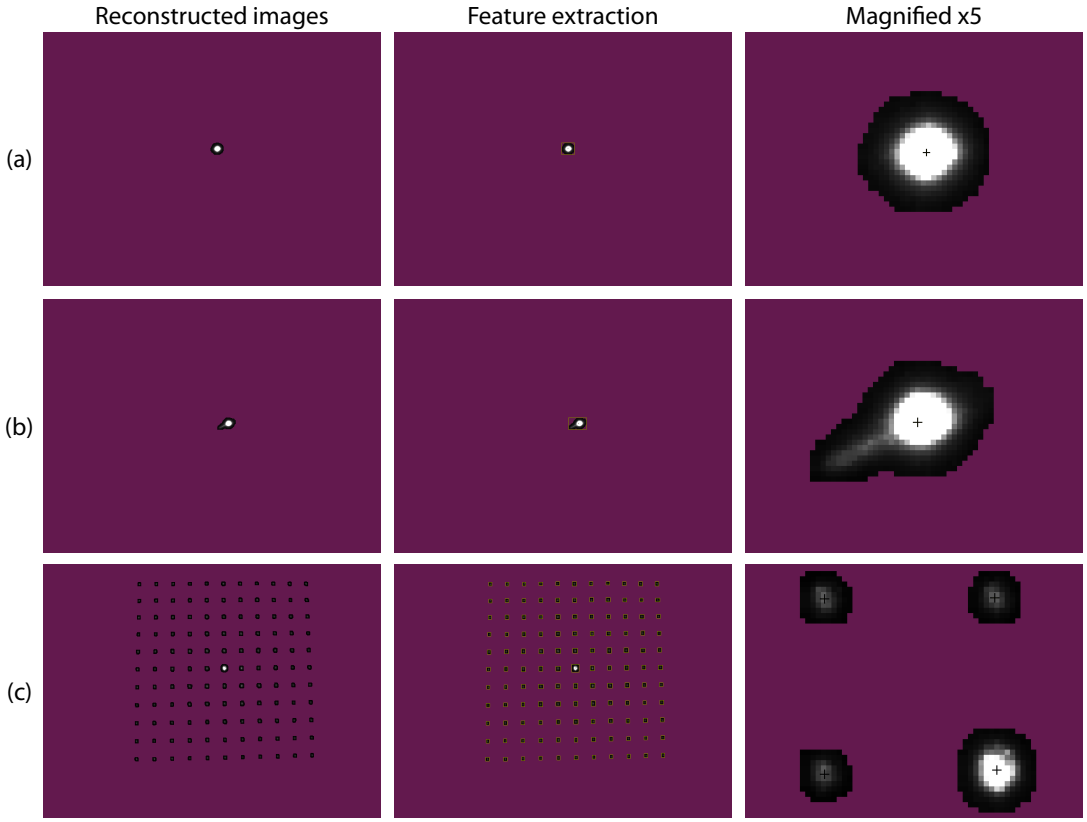

Figure 14: Blob detection of light spots produced by a USB-powered laser. All background pixels that are not sent are shown in pink. Row (a) is an example of a stationary laser spot; row (b) shows an example of what happens when the laser is moved quickly; row (c) shows 121 blobs. Reconstructed images (on the left and in the center) have resolution of 640x480 pixels with 8 bit grayscale pixel values. Thresholds on FPGA and PC are both equal to 15. The margin for RoI identification is equal to 2. The exposure time is 100 μs. Images in the center show the results of applying blob detection on the PC in real time. Images on the right are zoomed-in versions of received images magnified by a factor of 5.

Verification is done by moving the laser around and capturing the images reconstructed on the computer. Captured images are visually verified for correctness. Visual verification is valid because of the margin around the foreground pixels. The margin allows us to see if threshold worked correctly, and also it allows us to verify whether the margin itself is correct. In case that margin is equal to zero, it is not possible to verify whether the implementation works properly in this setup, because even if a pixel on the edge of the transferred region is equal to a threshold, we cannot be sure whether or not the neighboring pixel that was not sent, is less than the threshold. In current implementation, the margin is a hard-coded constant that cannot be changed during the run time. In simulation, it can be changed without any noticeable effect on simulation time. However, in order to change the margin in the hardware implementation, the whole design has to be recompiled, which takes approximately 40 minutes in cases where the width of the interval table is equal to 20. Thus, verification and evaluation are carried out with a fixed margin equal to 2. In order to reduce the amount of blurred light spots or even remove the blur all together, the exposure time of the camera is fixed to 100 microseconds.

Figure 14 shows several images reconstructed on the computer from the snippets received over the Ethernet connection. First, only one laser spot is tested. Rows (a) and (b) show reconstructed images of a single light spot, where in (b) it is moved quickly, whereas in (a) it is not. The center of mass in (b) is slightly moved towards the tail because of the simple nature of the applied algorithm. However, this blur can be reduced by decreasing the exposure time even more (exposure time of MV camera can be reduced to the minimum of 10 μs) and by increasing the thresholds. This will move the center of mass closer to the actual center of the light spot at the time of its capture. Row (c) is an example of all the possible laser spots.

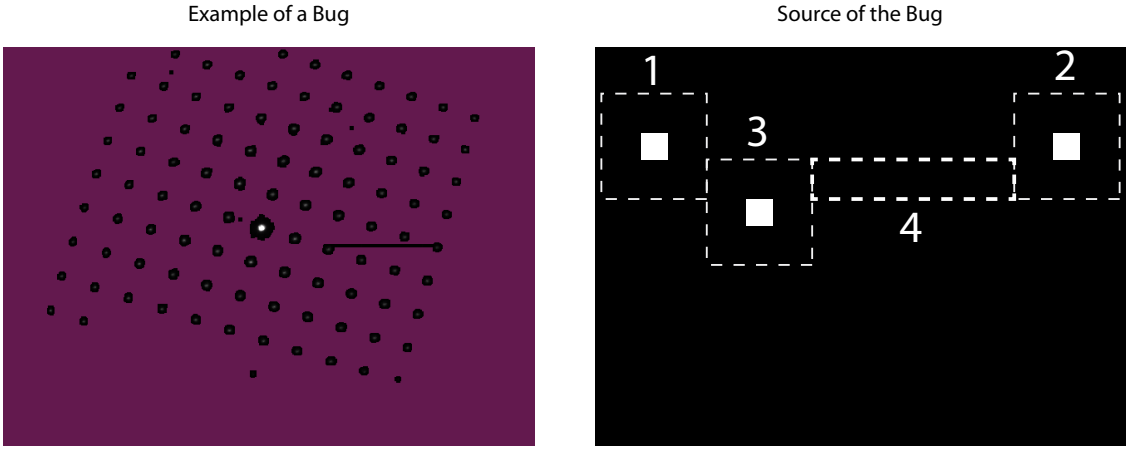

Figure 15: A bug and its source.

As illustrated in Figure 15, a bug exist in current implementation. When the bug occurs, extra region is sent to the PC, even though no foreground pixels are nearby. It was not caught during the simulation because the implementation was not tested exhaustively, and only the

exposure to real data made the bug apparent. However, the source of the bug has been found, though not yet addressed. The bug is reproducible and happens if a new interval is inserted into the interval table between two already existing intervals. To reproduce the bug, the new interval should touch the edge of the left interval but not touch nor overlap with the interval on the right. This results in a merge of all three intervals.

The right part of the figure shows such a scenario. Region 1 is inserted into the interval table as soon as the bright pixel at its core is encountered. In the same line, there is another bright pixel that extends the interval table by adding region 2. In the next two lines, no new intervals are added. In the fourth line, another bright pixel is encountered and region 3 inserted into the interval table. However, because of the bug, region 4 is also inserted into the table, forcing transmission of unnecessary data. Unfortunately, there was not enough time to fix this bug.

6.2.2 Ethernet Controller and Packet Capture

Several factors are important when verifying correctness of an Ethernet controller. Verification of the Ethernet controller has to make sure that all data sent from the DE2-70 evaluation board is received on the other end. This can be done by sending known test data from the FPGA to the computer and comparing the received data with the known ground truth. Thus, each packet contained a 384 bit number that started with 0 in the first packet and was incremented by 1 in each subsequent packet. In order to distinguish the test packets from any other packet, a header comprised of source and destination MAC addresses of the board and PC, respectively, is inserted before each number. All together, each packet is 60 bytes large.

On the computer, each packet received over the Ethernet interface is filtered according to the known source and destination MAC addresses. In the next step, the 384 bit number is extracted from the packet and compared with the previously received number. If the newly received number is not greater than previous number by exactly one (0 is a special case), then it means that either the controller has a bug, or there is a collision in the network. However, since the Ethernet interface supports full duplex mode, it can send and receive data at the same time, which makes a collision highly unlikely. Thus, the only source of error can come from the Ethernet interface. This test was run for 30 minutes, and all received numbers were correct.

6.2.3 Connected Component Labeling

The approach to verification of connected component labeling (CCL) is shown in figure Figure 16. Test images are sent to the DE2-70 board, where connected component labeling is applied. The results are sent back to the PC, where they are saved in a comma-separated values (CSV) file. The CSV file is then used to draw the results of CCL over the original image and verify correctness through visual inspection.

Communication between board and computer is accomplished by using the RS-232 port of

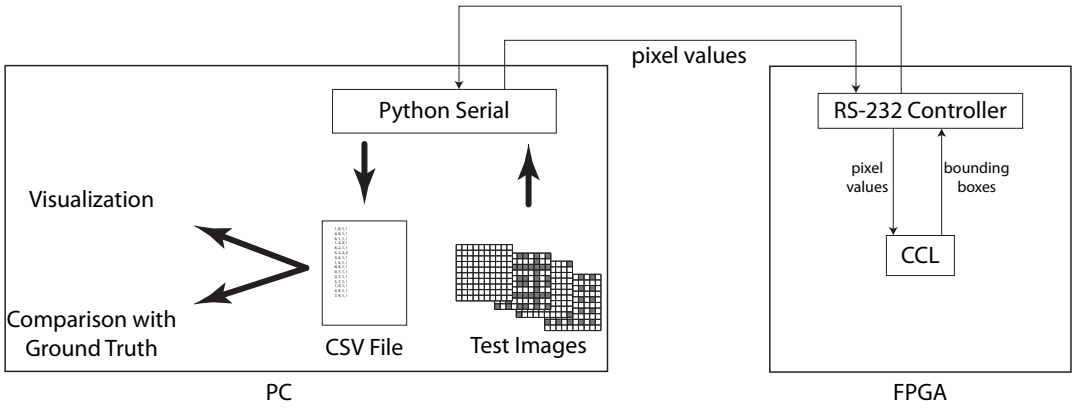

Figure 16: Verification of connected component labeling in hardware.

the board. On the PC, Python serial library is used. It provides easy means to interact with serial devices. The images are sent to the board one pixel at a time, which is convenient, because the pixel values are represented by 8-bit numbers. On the FPGA, a serial port controller is adapted from [16, chapter 7]. The controller receives pixel values and sends them to the single pass connected component labeling module. As opposed to verification in simulation, during verification in hardware the labels are not sent to the host PC—the correctness of CCL is verified by checking the correctness of the bounding boxes. The underlying assumption of this approach is that if connected component labeling does not work correctly, then the extracted bounding boxes will be wrong as well. And conversely, in case that connected component labeling is wrong, then the bounding boxes will not be correct.

Figure 17 shows some test images and the test images with overlayed bounding boxes received as result of connected component labeling. Parts (a) and (b) illustrate two simple cases, in which no merges occur. These are the typical images for the task at hand. However, in order to test the robustness of the developed approach, more complex figures are used. (c) and (d) introduce shapes that test the ability of our approach to deal with shapes that require several merges. For example, the shape in (d) starts with several unconnected vertical lines, but as the raster scan proceeds, a long horizontal line connects them all, forcing a merge of all the vertical lines. (e) and (f) test the approach for its ability to deal with images where objects are inside of other objects.

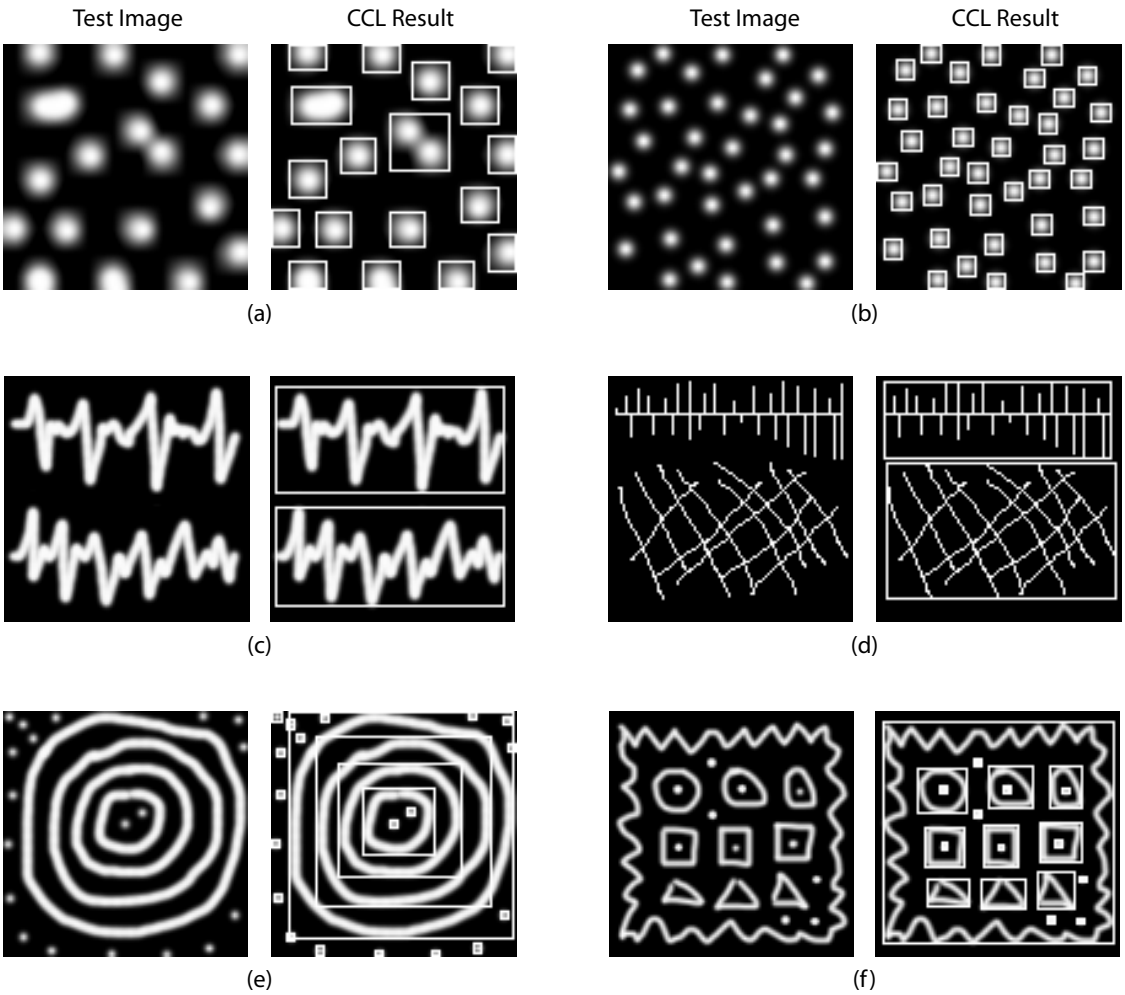

Figure 17: Verification of connected component labeling in hardware. All test images are 100x100 pixels. Bounding boxes are denoted by white rectangles. The foreground pixels are all pixels that exceed a predefined threshold (in this case, threshold is equal to 50).

7 Evaluation

This section introduces the criteria for evaluation, describes the evaluation strategy and presents experimental setup in which evaluation is carried out. To determine the advantages and drawbacks of developed approach, several evaluation criteria have been selected.

7.1 Criteria for Evaluation

Scalability Scalability determines how well the system behaves under increasing load. Scalability can be analyzed independently for the data transferring circuit and the circuit that identifies regions of interest. In data transfer, scalability is directly influenced by the amount of foreground pixels in an image. The more foreground pixels exist in an image, the more network traffic will be produced in the process of transferring the foreground pixels to the host PC. However, the amount of foreground pixels can be further divided into two parameters: the *size* of blobs, and the *number* of blobs in the image. These two parameters give us a precise control over the amount of foreground pixels, and allow us to measure scalability of the developed approach in terms of traffic. On the other hand, the module that identifies regions of interest depends on the parameters used during its synthesis: the maximal *number of intervals* in the interval table and the *size of the margin*. Together with the number of logic elements available on the Cyclone II FPGA, these parameters define how many blobs per line the system can process overall.

Precision Scalability, however, says nothing about the quality of the results. But in some tasks it is important to know how precise is the system. As in the case of camera pose estimation from centers of blobs, center of mass calculation should be very precise, because even a small deviation can cause large errors in camera pose estimation. Thus, the second criterion for evaluation is precision. The purpose is to measure how precise are the two approaches and which parameters influence this criterion. Since there was no time to fully implement connected component labeling on the FPGA, the precision of the second approach could not be evaluated.

7.2 Evaluation Strategy

The evaluation strategy in this project is to analyze how the system behaves under varying parameters that influence chosen evaluation criteria, that were described in the previous section. By varying the number of blobs in the image and their sizes, we can find out the practical limits of our approach that apply for the 100 Mbit/s Ethernet interface fo the DE2-70 board. And by varying the size of the interval table, and analyzing how many intervals can fit into FPGA, the theoretical limits of the system can be analyzed.

The size of the interval table can be changed in the VHDL code, and the size of the resulting design can be gathered by letting Quartus compile the code. However, it is more difficult to control the size and the number of blobs in an image, because the camera is directly connected to the FPGA. There are several ways to overcome this problem. The first way is to manipulate camera image by using light sources. However, in this approach, it is difficult to precisely set the number of foreground pixels, because the camera is not calibrated, noisy, and any kind of external light source is difficult to control precisely. The second approach is to send a test image to the FPGA and to load it into the external memory of DE2-70 board. Then the image can be constantly read from the memory and provided to the system under test. However, this approach increases the number of logic elements required to test the design, and changes the placement, i.e., a part of the FPGA is devoted to logic that is not necessary in for the final design and might even change its placement, which will result in inaccurate evaluation. Another problem with this approach is that it is not possible to test different frame rates without recompiling the design. To overcome these limitations, two DE2-70 boards are used. The next section describes the experimental setup, in which evaluation of the approaches developed in this project was carried out.

7.3 Experimental Setup

This section describes the experimental setup used for evaluation of the approaches developed in this project, as depicted in Figure 18.

7.3.1 Two DE2-70 Boards: Camera Emulator

For evaluation, the original board with the MV camera is connected to a second DE2-70 board. However, now the board with the camera is used to provide real and synthetic images to the second board, where the identification of regions of interest takes place. In the initial approach to emulate the camera, camera timings were captured with Signal Tap Logic Analyzer. Then, a controller was written to behave in the same way. However, this approach turned out to be inflexible when the evaluation was started. The problem is that there is no flexible way to change the frame rate to a desirable value during the run time. Hence, the emulator has been reworked to use the synchronization timings from the original camera. This allows the user to change the frame rate of the camera over the provided control panel, so that the synthetic images can sent with the same timings of the real camera. This approach is more flexible and allows us to set any frame rate that the real camera can provide. In addition, by flipping a switch, the user can choose between the original camera or synthetic images stored on the same board.

In actual real-world application, such scenario introduces an additional delay between the camera and the FPGA on which the actual image processing takes place. This comes from the

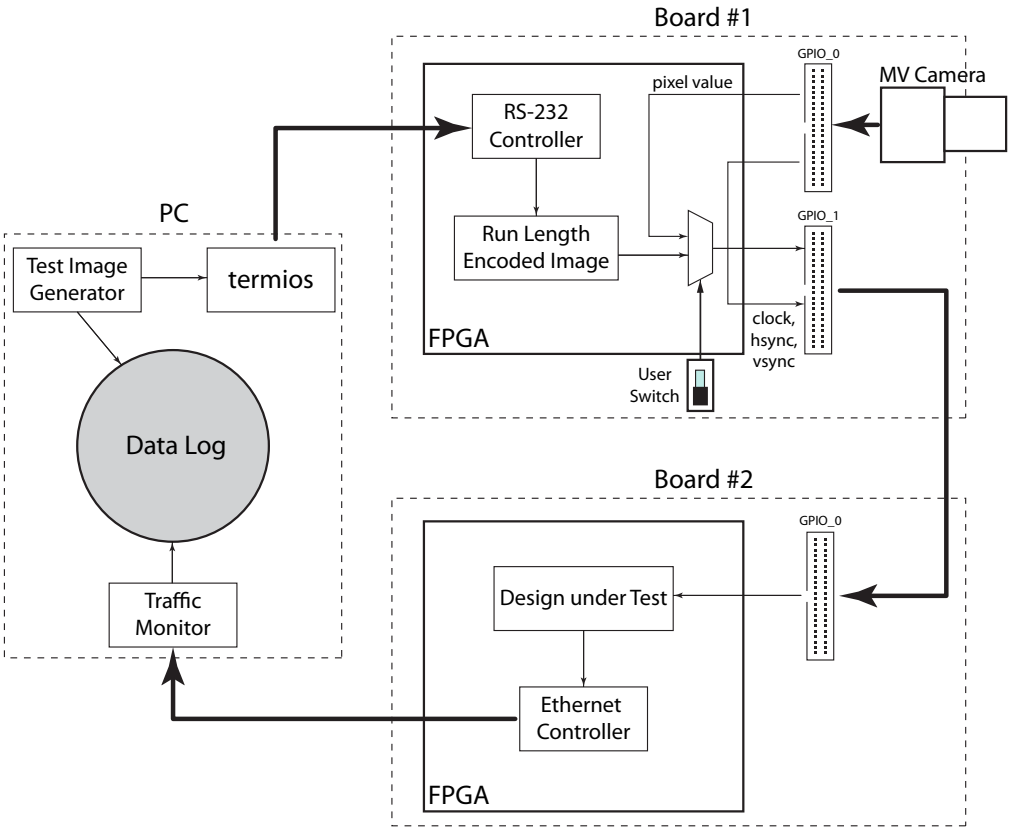

Figure 18: Experimental setup.

fact that the data from the camera takes an indirect route to the second board by first going to the FPGA of board 1, then over the GPIO cable and then from the GPIO pins to the FPGA on board 2. However, the delay plays no direct role in evaluation of scalability of the approach developed in this project.

7.3.2 Test Images Generator

In order to evaluate the approaches developed in this project in a systematic way, the amount of foreground pixels must be controllable. However, it is difficult to control this amount when using the camera, because it requires camera calibration and exact positioning. These problems can be avoided if we use synthetic images instead. Synthetic images enable control over the amount of foreground pixels. For this purpose, a test image generator has been developed.

The generator is written as a command line tool that accepts two arguments: size of blobs, and number of blobs. It tries to generate an image with these parameters and upon successful generation saves the image in PNG format under a unique name on the hard disk. In addition, it copies the name of generated image into the clipboard. In the next step, the main program compresses the image by using run-length encoding and send the image to the DE2-70 board

over the serial port. Figure 19 shows some sample test images generated for evaluation.

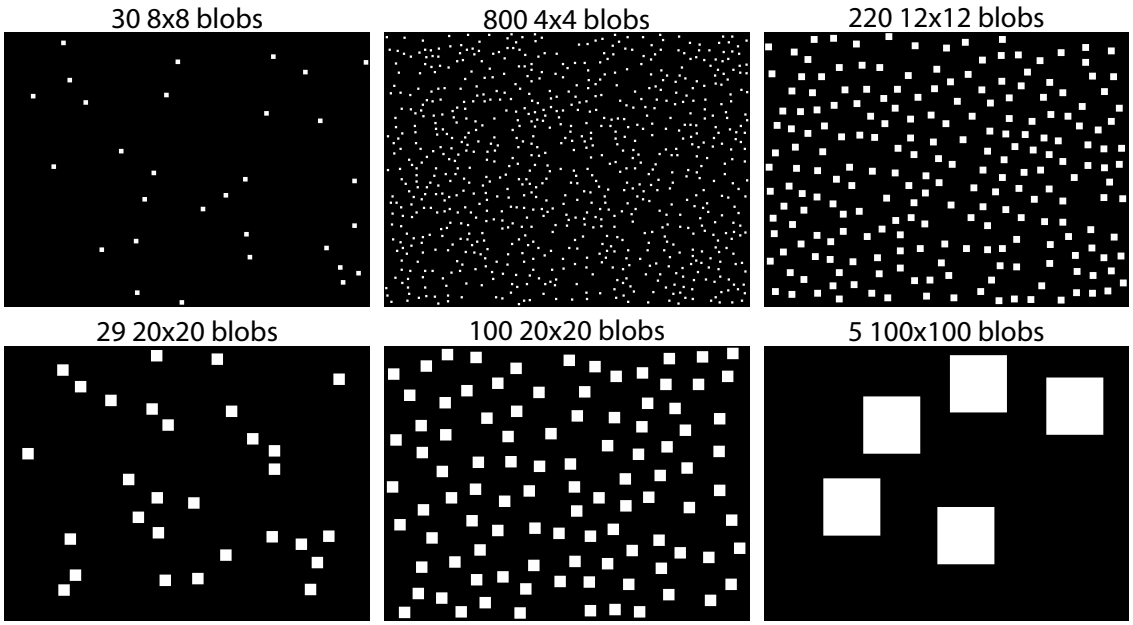

Figure 19: Sample test images. All images are 8 bit grayscale with resolution of 640x480.

7.3.3 Run-Length Encoder and Decoder

In order to send pixel values of a test image from one board to another, the first board needs some means to store the image. However, the Cyclone II FPGA of the DE2-70 evaluation board does not have enough lookup tables (LUTs) to store the whole image. In addition to the LUTs, the FPGA has 250 M4K memory blocks that are distributed across the FPGA. The block RAM provides storage space for 1,024,000 bits. However, storing a 640x480x8 bits image requires 2,457,600 bits, which means that the block RAM of the FPGA is able to store only 200 lines of the image, which is 42%.

There are several ways to handle this problem: one option is to use an external memory that is capable of storing the whole image; another option is to compress the image and thus reduce the amount of required storage space. In this project, the second approach has been chosen for a number of reasons. The main reason against the first option is that it requires integration of two additional modules: external memory controller, and a faster communication channel. A faster communication channel is necessary because it takes around 30 seconds to send an uncompressed 640x480x8 bits image to the board over the serial interface. Performing evaluation in this manner would substantially increase evaluation time. Thus, a faster interface would have been necessary. Instead of integrating and testing two additional controllers, it was decided to compress transmitted images using a simple compression algorithm. A compressed

image can be transferred quickly and stored completely in the block RAM of the FPGA.

The only limitation is that the compression algorithm should be simple enough to be implemented in a shorter amount of time than it would take to develop Ethernet reception circuit (or a USB transmission circuit) and to connect it to a memory IP core. A very simple compression algorithm—run-length encoding (RLE)—has already been developed earlier in order to speed up connected component labeling. Lossless run-length encoding for grayscale images represents an image as a series of runs each containing two numbers—a pixel value and the length of the run. Run-length encoding is implemented on PC and on FPGA.

On the PC, RLE speeds up image transfer. Each test image is compressed by encoding groups of identical pixel values as single runs represented by two 8-bit numbers: the first number is the pixel value; the second number is the length of the run [20]. Since a run-length cannot be zero, the second number allows us to encode runs of up to 256 pixels. RLE has limitations for images with many different shades of gray, which might produce images with a larger size than the original image. However, RLE is ideal in terms of implementation complexity and compression ratio for test images that only have two different pixel values, as described in the previous section.

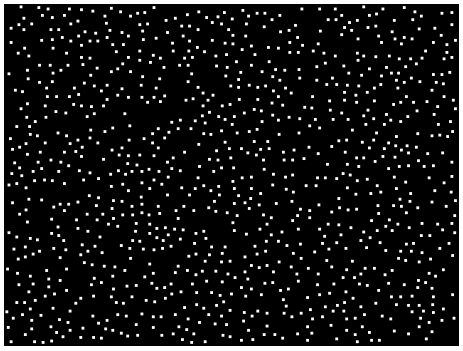

Figure 20: A 640x480 test image that contains 1000 4x4 rectangles. This image can be represented by 8114 runs.

On the FPGA, RLE is used to store compressed test images that are received over the serial interface. Run-length encoding allows us to save the whole image in the block RAM of the FPGA. All 250 M4K memory blocks are allocated to storage of the test images, which provides enough memory to store $\frac{1024000}{2 \cdot 8} = 64000$ runs. For comparison, one of the most complex test images used in evaluation, as shown in Figure 20, can be represented by 8114 runs, which can be sent over the serial port in $\frac{2*8*8114\ bits}{115200\ \frac{bits}{s}} \approx 1.13$ seconds, assuming a baud rate of 115200 bits/s.

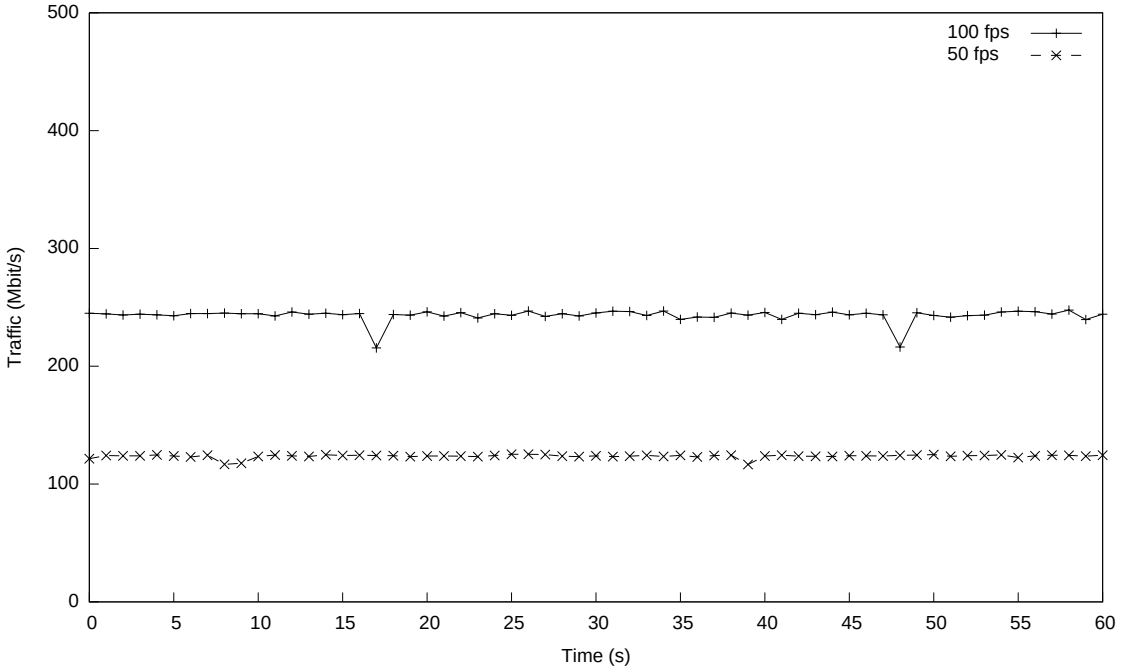

Figure 21: Traffic and mean average traffic generated by the mvBlueCOUGAR-X 100 camera running for a minute at 100 and 50 frames per second and resolution of 640x480 pixels with 8 bit grayscale pixel values. The average traffic generated at 100 fps is 243.273 Mbit/s; at 50 fps the average traffic is 123.645 Mbit/s.

7.3.4 Traffic Monitor

Another tool necessary for evaluation of the shared approach is the traffic monitor. There are several traffic monitors available, for example, Wireshark, that was used during debugging, can also be used to analyze packets and to store them on the hard disk. However, it cannot be automated easily to compute average traffic given a start and stop command by another program. By writing our own program, we have the ability to save the amount of traffic in a file without the necessity of human interference during evaluation.

The traffic monitor is written by using the Berkeley Packet Filter library described earlier in this report. However, in contrast to the packet capturer described in previous sections, where this library was used to filter and analyze the packets according to the developed protocol, this time the library is used only to compute the amount of bits received every second. This amount is averaged over the number of samples at the end of data capture and written into a file. It runs in the so-called *promiscuous mode* that allows it to capture all packets, even those that are not targeted at the capturing interface. This allows us to use the traffic monitor to capture all traffic that runs over a chosen network interface.

Figure 21 shows the traffic produced by the MATRIX VISION (MV) camera running at

100 fps and 50 fps. By sending pixel values alone, the resulting traffic should be equal to $\frac{640*480*8*100bits}{1000000} = 245.76$ Mbit/s for 100 frames per second and $\frac{640*480*8*50bits}{1000000} = 122.88$ Mbit/s for 50 frames per second. The actually measured values close to the theoretical values. Since the camera also sends the MAC addresses of source and destination and supports the UDP protocol as well, the actual measured traffic should be slightly larger. This suggests that the data is not sent in raw format, but is compressed with a simple algorithm, which keeps the traffic around the theoretical value.

8 Results and Discussion

The approach developed in this project has been systematically evaluated according to the strategy introduced in the previous section. This section presents results of evaluation. The first part analyzes the traffic produced by the developed approach and how it behaves under different parameters. The second part shows how many logic elements the design uses when the interval table is varied in size. This section is concluded with a prediction of the performance of connected component analysis—the approach that was only partially developed in this project.

8.1 Traffic Analysis

The parameter space spanned by the number of blobs in an image and the size of blobs, has been sampled both, randomly and in a systematic way. Initially it was sampled randomly, however, many times the image generator was not able to generate an image because the blobs could not fit in. Then, the sampling was carried out in a more systematic way. Figure 22 shows the dependency between the number of blobs, selected blob sizes and the traffic that results from sending regions of interest to the host computer. For comparison, average traffic of the MATRIX VISION camera computed in the previous section is included in the figure. 22

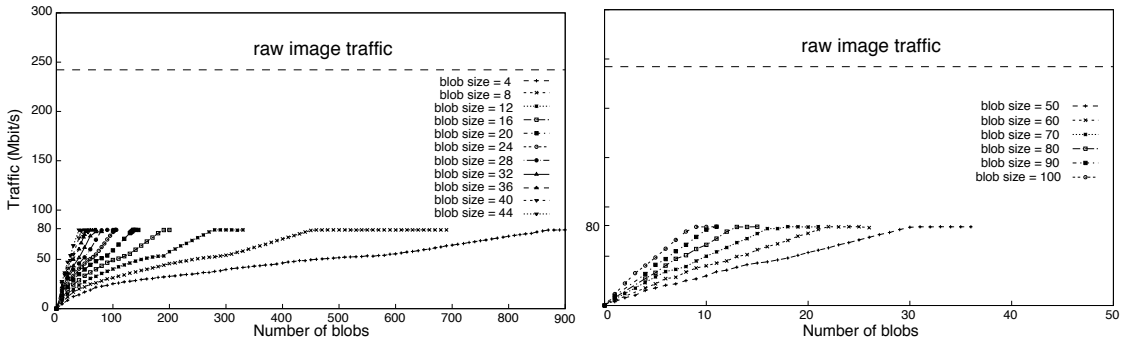

Figure 22: Average traffic for blob sizes of 4x4 to 44x44 with step size of 4x4 (on the left) and for blob sizes of 50x50 to 100x50 with step size of 10x10 (on the right). All images have resolution of 640x480 pixels with 8-bit grayscale pixel values. The margin of the interval table was fixed to 2. Note that the X-axis in the right figure is not the same as in the left figure.

is a zoomed in version that also shows the expected theoretical values that are computed as follows:

$$\frac{(W^2 + W * 3) * N * f * 8\text{bits}}{1000000},$$

where N - is the number of blobs in the image, W - is the size of the blobs plus the margin, and f - the frame rate. The number in parenthesis is the number of bytes sent for each blob. 3 extra bytes are sent for each run. This value is multiplied by the number of blobs then by

the frame rate f and, finally, converted to bits by multiplying the result by 8. In the end, the result is divided by $1 * 10^6$ to convert the number to Mbit/s. This calculation is only valid if the images are generated in a way that the neighboring blobs never touch each other even after adding the margin. In addition, this calculation assumes that all blobs and their margins are completely in the image. However, this calculation underestimates the actual traffic by a small number, because it does not consider that in the beginning of each package 12 bytes are sent to denote the MAC addresses sender and receiver. From the formula we can derive that traffic increases *linearly* with the increasing number of blobs, but *quadratically* when their sizes are increased.

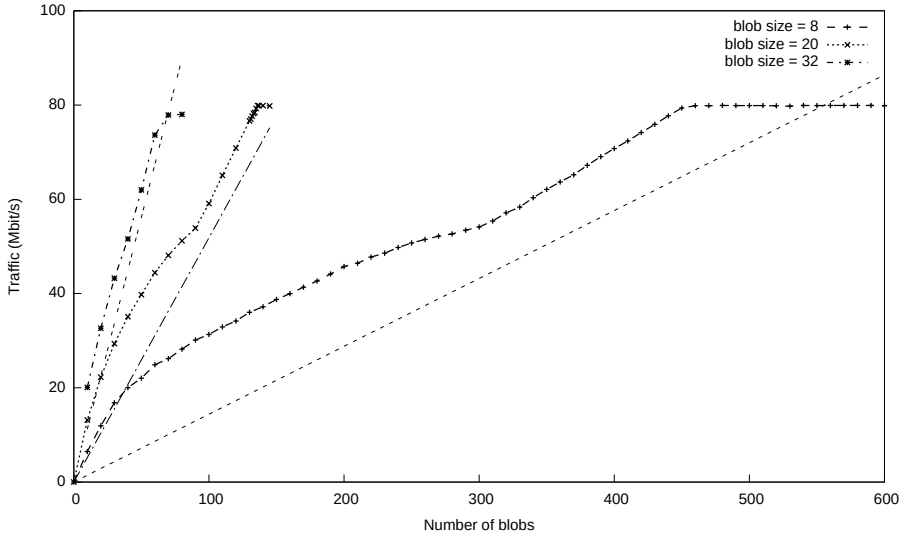

Figure 23: Average captured traffic and theoretical traffic (shown by the straight lines) for three different blob sizes.

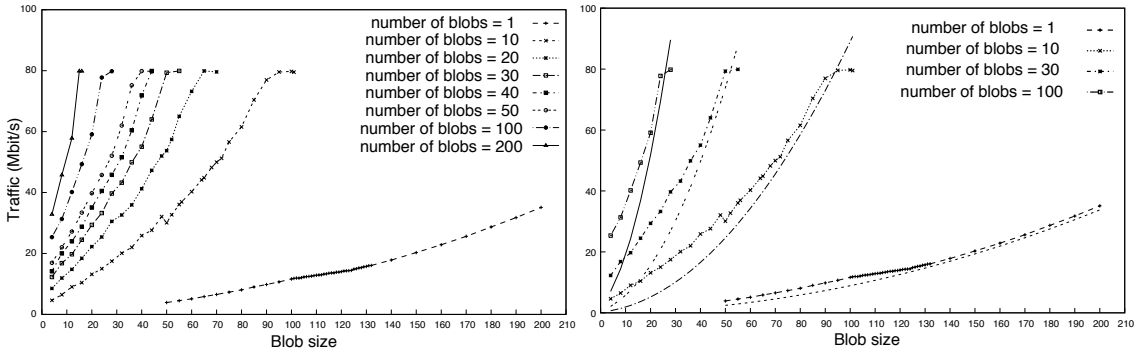

Figure 24: Average traffic for selected blob numbers (on the left), and also their theoretically computed values which are shown by smooth lines without dots (on the right). The parameter space has not been sampled uniformly, which explains a large concentration of data points for the range between 100 and 130.

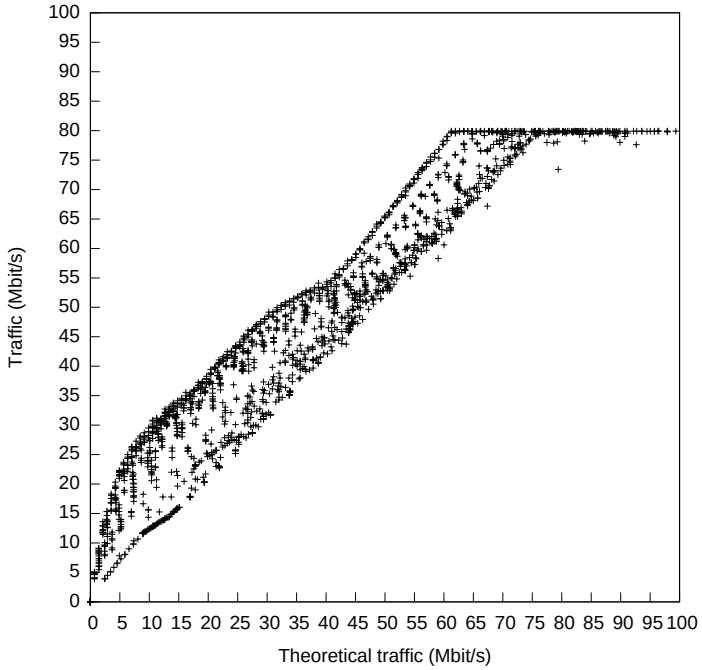

Figure 25: Relationship between actual and theoretical traffic. All entries in the data log are plotted here.

Figure 24 shows how traffic depends on the blob size for some fixed numbers of blobs. Several observations can be made based on these graphs. The first observation is that the maximum possible traffic of 100 Mbit/s is never reached, and the limit is around 80 Mbit/s. This is because the Ethernet controller has not yet been optimized. Reaching this limit results in data loss.

Another observation is that smaller blobs use more traffic when their number is small. The algorithm for data transfer works in a way that encourages early transmission if no additional data to be sent is waiting in a queue. In test images with few small blobs, it is likely that a blob is far away from other blobs, meaning that there is nothing else in the queue ready to be sent. This results in immediate packet transmission, where, in some cases, the header is larger than image data. For example, for a 4x4 blob with margin equal to 2, the algorithm has to transfer (4+2) bytes of pixel values, 4 bytes that represent coordinates and 2 bytes to denote that the data is a region of interest. All together, this makes 12 bytes to send one run. However, data transfer circuit starts every new packet with a fixed sequence of sender and destination MAC addresses, which make 12 bytes as well. In addition, the Ethernet interface adds several bytes to make it a standard size. In this case, it would add 36 bytes to make the package 60 bytes large. This explains the high traffic overhead for images with few small blobs. When the number of blobs increases, the chance of having a neighbor blob close by also increases.

Thus, it becomes more likely that after writing a region into the transmitting FIFO, the next

region is already waiting in the queue and will be written as well. This reduces the overhead for images with more blobs, and with larger blobs. The *early send* strategy reduces the delay between the time when a region is recognized and the time when it arrives at the host PC at the cost of traffic for small blobs in small numbers. However, when with increasing size of data to be sent, the overhead decreases, until it reaches a fixed point at around 52 Mbit/s (for blob size of 4), where the actual traffic is close to the theoretical traffic. The fixed point is reached faster by larger blobs. Figure 25 shows the relationship between the actual traffic and the theoretical traffic that is computed according to the formula introduced in the beginning of this section.

8.2 Usage of Logic Elements

Figure 26 shows the usage of logic elements (LEs) depending on the size of the interval table, which is defined by the margin and the maximal number of intervals in the table. Systems with margins 0 to 7 and the number of intervals varying from 10 to 120 have been synthesized by using Quartus.

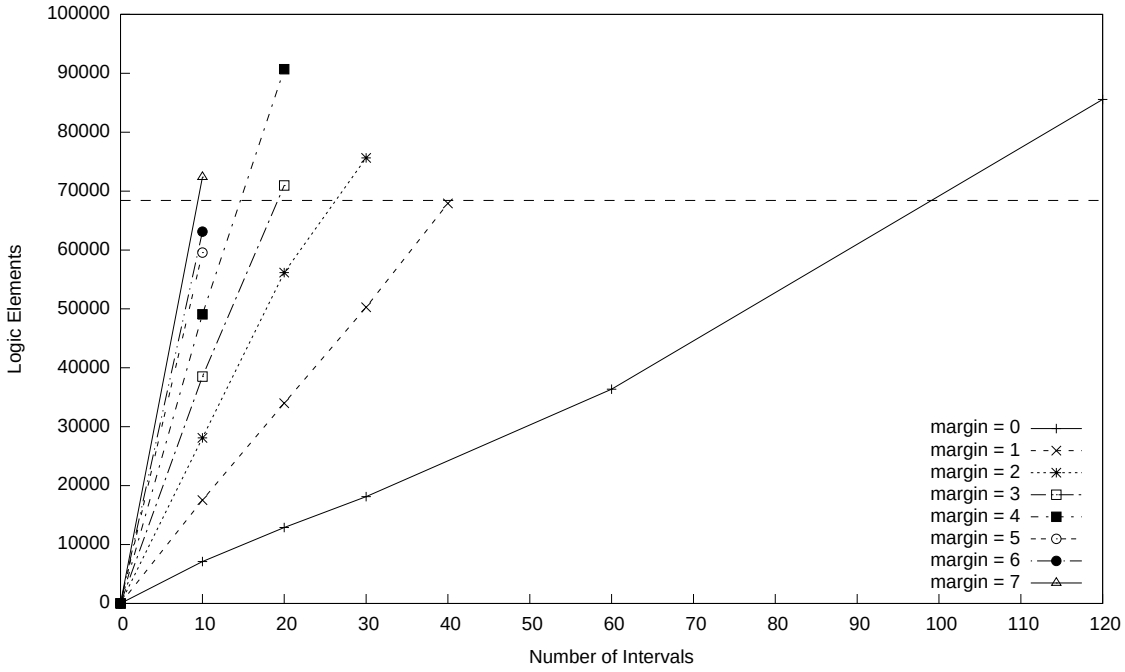

Figure 26: Usage of logic elements. The number of LEs was obtained by using the Quartus tool chain to synthesize the design. The Cyclone II FPGA provides 68,416 logic elements in total (denoted by the dashed horizontal line).

It can be observed that the size of circuit depends linearly on the number of intervals and the margin. The second fact becomes apparent after seeing that in the graph, when the number

of intervals is equal to 20, the distances between the LE usages for different margins are almost equidistant. It is interesting to note that in cases when margin is equal to 0, the approach simply performs run-length encoding. However, compared to run-length encoding, the usage of logic elements is considerably larger.

Table 2: Dedicated block RAM usage.

	Memory bits	Can store
Interval FIFO	94,208	≈ 4710 intervals
Partial frame buffer	400,000	≈ 78 lines
Data transfer FIFO	524,288	
Total	1,020,544	

Almost all of the available 1,024,000 bits of embedded block RAM are used by the design. Table 2 shows the block RAM usage of different modules. A relatively small part of block RAM is taken by the interval FIFO. The interval FIFO stores the coordinates of foreground pixels represented by intervals, each consisting of three numbers—y-coordinate, start, and end of the corresponding interval. Partial frame buffer stores the pixel values of a small part of the image in order to access them shortly after the intervals have been identified. Pixel values of each interval are gradually transferred into the data transfer FIFO, along with some additional information, such as the type of data, interval length, etc. The largest part of available block RAM is allocated to the data transfer FIFO.

In the worst case, an interval is as large as an image row (640 pixels), thus it takes *at most* 12.8 μs (640 clock cycles at 50MHz clock frequency) to fetch all of the interval's pixel values and to store them in the data transfer FIFO. However, it takes *at least* $\frac{640*8\ bits}{100{,}000{,}000\ bits/s} = 5.12 * 10^{-5} s = 51.2\ \mu s$ to send the same amount of data over the Ethernet interface. In reality, transmission will take longer, because of protocol overhead and driver inefficiencies mentined earlier. For this reason, the data transfer FIFO has to be the largest, in order to minimize data loss caused by a FIFO overrun.

8.3 Prediction of Performance of Connected Component Labeling

Unfortunately, there was not enough time to evaluate the approach that performs blob detection on FPGA. In future work a comparison between blob detection on FPGA and the algorithm evaluated in this project is planned. It will bring insight into the tradeoffs between precision, flexibility and development time of blob detecting algorithms.

Several advantages might arise after implementation of connected component labeling (CCL). First, by using CCL, images with the maximum number of blobs possible can be analyzed. This is inherent in the single-pass CCL, as shown by [8]. The advantage of this approach is that

only 2 division circuits are required. By sending only features, as opposed to regions with pixel values, the amount of possible blobs processed will be much larger than the one achievable by the algorithm presented in this project. The traffic will be proportional to the number of blobs in the image. This is due to the fact that for each blob, the system has to send a fixed amount of data—in this case the X and Y coordinates with finite precision.

As calculated earlier in this report, 64 bits are required to represent one floating point number. A blob is represented by its feature—the center of mass–which requires two 64-bit numbers. If the bandwidth of the system is limited to 100 Mbit/s, then a system will be able to send centers of masses of $\frac{100{,}000{,}000}{64*2*100\ \text{fps}} \approx 7812$ blobs. In contrast to hybrid blob detection, this approach does not depend on blob sizes nor their number.

9 Conclusions and Future Work

This work has presented an approach to shared blob detection, in which the images are preprocessed on the FPGA and sent to the computer, where the main processing takes place. The FPGA separates the foreground pixels from the background pixels and sends regions of interesting pixels plus some small margin to the computer. This approach has been extensively evaluated. Results of evaluation suggest that images containing a large number of blobs can be reliably sent to the computer, provided that the blobs are small enough.

At this point, three main directions for future work can be outlined. The first direction is to improve the Ethernet driver. In its current implementation, the Ethernet controller does not use its full potential. According to its data sheet, each operation needs the I/O signals be valid for 10 ns, which can be achieved by using a clock of 100 Mhz. However, in the current design, the clock has the frequency of 50 MHz, which results in the period of 20 ns. By using a 100 Mhz clock, it is possible to save 10 ns on every interaction with the Ethernet controller, compared to the current design.

Data transfer can be further optimized by introducing pipelining. For example, while the Ethernet controller waits for successful packet transmission, the next piece of data can be prefetched from the FIFO, such that packet transmission can be started right away. These two improvements will allow us to use the Ethernet controller to its full potential.

The protocol can be optimized by packing the data more efficiently. One inefficiency arises from the allocation of 16 bits to denote the type of data. However, if the type of data is represented by only 2 bits, it is possible to pack the data even tighter. Current implementation is generous with the amount of bits that are allocated for each piece of data. For example, the coordinates are represented by 10 bits, however, in a 640x480 image, we only need 10 bits for the X-coordinate, and only 9 bits for the Y-coordinate.

Another direction for future work originates from the insight that the approach developed in this project actually performs lossy compression. This fact becomes even more apparent when the margin is set to zero—this reduces our system to a run-length encoder. A lot of research has been done in the field of compression. An implementation of a state of the art compression algorithm on FPGA might perform better than the approach developed here.

A long-term research goal is to find out a good balance between development time, flexibility, and performance. FPGAs enable massive parallel processing, but developing applications on FPGAs requires a large amount of time. In contrast, developing applications for a general purpose processor requires less time, but the applications might not perform as good as on an FPGA. However, there is a space where both architectures can be used alongside of each other, similar to the approach developed in this project. The first step in this direction would be to carry out a fair comparison between the approach developed in this project and a state of the art approach to blob detection, such as, e.g., the single pass connected component labeling [8].

A fair comparison can be done by marking the time when camera pose estimation is completed, as well as by comparing precision of both approaches.

10 Aknowledgements

I would like to thank Rainer Herpers and Ken Kent for accepting me as their student and giving me the chance to work on this awesome project. I would also like to thank Timur Saitov for all the interesting discussions that we had. These discussions generated many great ideas, some of which can be found in this report. I especially want to thank my girlfriend, and soon-to-be wife Shima Shahi Irani for sticking with me till the end! :)

Bibliography

[1] A. Bochem, "Active tracking with accelerated image processing in hardware," Master's thesis, Bonn-Rhein-Sieg University of Applied Sciences, 2010.

[2] D. Bailey, *Blob Detection and Labelling*, ch. 11, pp. 343–375. Wiley-IEEE Press, 1st ed., 2011.

[3] R. Fabbri, L. D. F. Costa, J. C. Torelli, and O. M. Bruno, "2d euclidean distance transform algorithms: A comparative survey," *ACM Comput. Surv.*, vol. 40, pp. 2:1–2:44, Feb. 2008.

[4] S. Hinz, "Fast and subpixel precise blob detection and attribution," in *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, vol. 3, pp. III – 457–60, September 2005.

[5] J. Trein, A. T. Schwarzbacher, and B. Hoppe, "FPGA implementation of a single pass real-time blob analysis using run length encoding," in *MPC-Workshop*, February 2008.

[6] D. Bailey and C. Johnston, "Single pass connected components analysis," in *Proceedings of Image and Vision Computing New Zealand 2007*, pp. 282–287, December 2007.

[7] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *J. ACM*, vol. 13, pp. 471–494, October 1966.

[8] C. Johnston and D. Bailey, "FPGA implementation of a single pass connected components algorithm," in *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*, pp. 228 –231, jan. 2008.

[9] Davicom, *DM9000A 16 / 8 Bit Ethernet Controller with General Processor Interface Application Notes V1.20*, 2005.

[10] Racket, "The racket programmig language. http://racket-lang.org/," 2012.

[11] BSD, "FreeBSD kernel interfaces manual: Berkeley packet filter BPF(4)," 2010.

[12] PCAP, "Pcap—packet capture library," 2008.

[13] Terasic, *DE2-70 Development and Education Board: User Manual, version 1.08*. 2009.

[14] U. Lansche, *MATRIX VISION - mvBlueCOUGAR-X documentation, V1.0b24*. MATRIX VISION GmbH, 2012.

[15] Altera, *Quartus II Handbook Version 11.1*, 2011.

[16] P. P. Chu, *FPGA Prototyping by VHDL Examples*. Wiley Interscience, 2008.

[17] GHDL, "GHDL, available from http://ghdl.free.fr/," 2010.

[18] GTKWave, "GTKWave, available from http://gtkwave.sourceforge.net/," 2012.

[19] S. Doll, "VHDL txt_util library, available from http://www.stefanvhdl.com/vhdl/vhdl/txt_util.vhd," March 2012.

[20] D. Salomon and G. Motta, *Handbook of Data Compression*. Springer, Berlin, 5th ed., 2010.