Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3595

Implementation and Hardware Accelerated Verification of a Connected Component Labeling Architecture

Benjamin Bässler

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. S. Simon
Supervisor:	M. Sc. Yousef Baroud
Commenced:	September 2, 2013
Completed:	March 4, 2014
CR-Classification:	B.5.0, I.4.6

Abstract

A newly developed algorithm has to be thoroughly verified to ensure correctness. Applying only a set of test cases is by no means sufficient to conclude that an implementation is error free. In the case the new implementation is a functionally equivalent solution for the same problem, one can run both implementations, observe the outputs and report discrepancies.

Verification based on software implementation is prohibitively costly in many cases pertinent to image processing. In this thesis, a comprehensive verification environment for FPGA architectures is used to run both implementations directly on a FPGA and make use of the massive parallel structure.

As a proof of concept for this verification methodology, a recently proposed image processing algorithm for Connected Component Labeling will be researched and validated by running it concurrently with a reference architecture. Though the reference architecture is not as fast, it is well established and based on the standard two pass algorithm.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ACK Acknowledgment

AMBA Advanced Microcontroller Bus Architecture

ARP Address Resolution Protocol

ASCII American Standard Code for Information Interchange

AXI Advanced eXtensible Interface

CCL Connected Component Labeling

CP Charge Pump

CRC Cyclic Redudancy Check

DCM Digital Clock Manager

DSP Digital Signal Processing

DUT Device Under Test

FCS Frame Check Sequence

FIFO First In First Out

FPGA Field Programmable Gate Array

GMII Gigabit Media Independent Interface

GUI Graphical User Interface

HDL Hardware Description Language

IC Integrated Circuit

ICANN Internet Corporation for Assigned Names and Numbers

IEEE Institute of Electrical and Electronics Engineers

IHL Internet Header Length

IP Internet Protocol

JSON JavaScript Object Notation

LF Low-pass Filter

LSB Least Significant Bit

LUT Look Up Table

MAC Media Access Control

MII Media Independent Interface

MMCM Mixed-Mode Clock Manager

MUT Module Under Test

NACK Not Acknowledgment

PBM Portable Bit Map

PCI Peripheral Component Interconnect

PAD Padding Bytes

PFD Phase Frequency Detector

PLL Phase Locked Loop

RAM Random Access Memory

RIPE Réseaux IP Européens Network Coordination Centre

RTL Register Transfer Level

RX Receive

SAT boolean satisfiability problem

SFD Start Frame Delimiter

TCP Transmission Control Protocol

TTL Time to Live

TX Transmit

UDP User Datagram Protocol

UML Unified Modeling Language

VCO Voltage Controlled Oscillator

VHSIC Very High Speed Integrated Circuit

VHDL VHSIC Hardware Description Language

1 Introduction

1.1 Motivation

Developing new algorithms always raises the question: Does the algorithm fulfill the requirements? And is the implementation of this newly developed algorithm correct? An erroneous algorithm or implementation could have dramatic impacts on the overall costs, especially if the new developed algorithm is part of a bigger system where an incorrectness leads to a failure of the entire system. If such an error is discovered in a late stage of development the identification of the part which raised the erroneous output requires time to fix and might lead to new problems such as degrading the overall performance of the system, limiting its usability as well as reliability and in extreme cases, for example in safety critical systems it might put human lives at risk. This problem can be prevented or at least reduced by a verification of the implementation before it is used in field.

In the past decades many different verification methods were researched and developed. In the 1960s the development of software as well as Integrated Circuit (IC)s were simple and testing was not that difficult and, hence, the functionality was in the center of development [Kan02]. Also back in the 1960s E. Moore predicted the doubling of components per chip every 18 months [M$^+$65]. With the fulfillment of his prediction the complexity of ICs and software have also risen exponentially. This made the former method of testing all possible inputs of an IC rapidly impossible.

A commonly used method for testing is the creation of a list with test cases. The requirements for such lists are often enough tests and scenarios that are likely to result in faulty behavior [Wie08]. This often leads to incomplete test list. Here, formal verification techniques can help to get a complete list of test scenarios. Yet, the problem is that the specification of the algorithm has to be translated into a mathematical model. It also has to use boolean satisfiability problem (SAT) solvers to generate this complete test list [FGP10]. The generation of the mathematical model is the main problem - this is not possible for every case and even if it is possible, the SAT problem which needs to be solved can be too big for the current computational power.

For formal verification the internal structure of an implementation is required. This is not always available and if a discrepancy between the algorithm and the specification exists this

verification method can not find it, which is always the case when it comes to image processing algorithms.

In case of developing systems for image processing the size of the image which is used for testing can be reduced. With reducing the image size by one pixel, of a binary input image processing system, the number of input combinations is reduced by half. If the image is sufficiently big, it is possible to find discrepancies between the algorithm and the specification. Especially if well researched and proved algorithms for the specification already exist. With such a relatively small image an exhaustive test with all possible input permutations can be run and the algorithm can be verified. In addition a run of the implementation not only verifies the algorithmically correctness, furthermore the complete development toolchain is included in the test.

1.2 Goal

As a proof of concept, for the last given verification scenario of the last section, this thesis has to research such a verification architecture for an image processing algorithm.

The architecture presented in this work has to verify the correctness of a recently developed improved single pass Connected Component Labeling (CCL) algorithm [KRW+12]. It has to check all possible input stimuli for a reasonable image size in an autonomous way. To verify the output of the Device Under Test (DUT) a well researched and proved two pass CCL algorithm has to be implemented and to be used as reference. Both CCL algorithm combined with a verifier for the comparison of the output data has to run on a Field Programmable Gate Array (FPGA).

An exhaustive test has to be run for a practical, in terms of runtime, image size. To cover errors which depend on the size of data types or other corner cases, different image sizes have to be tested. In addition a non exhaustive test should be done for a bigger image size than the one used in the exhaustive testing.

The discrepancies found between the both CCL architecture need to be reported to a PC and visualized in a way to help finding problems in the DUT implementation.

1.3 Outline

This thesis is structured in six chapters. A short overview of the content follows:

Chapter 2 – Background: The theoretical basis of validation, verification, and testing is described. Furthermore the CCL problem and solutions are explained.

Chapter 3 – Connected Component Labeling Hardware Reference Implementation: In this chapter the implementation of the two pass VHDL CCL implementation is discussed.

Chapter 4 – Hardware Verification Architecture: This chapter will describe the idea behind the whole Hardware Verification Architecture including its implementation.

Chapter 5 – Software Implementation: This chapter describes the software to collect and analyze the found errors from the hardware. As well as a software to apply some test images to the CCL before a test in hardware is done.

Chapter 6 – Results: In this chapter the results of this thesis can be found.

Chapter 7 – Conclusion This chapter summarizes the work of this thesis and gives some optimization ideas for future works.

2 Background

In this chapter the basic background for the following chapters are explained. The Section 2.1 gives a short introduction in verification methods. In Section 2.2 different methods for CCL will be presented.

2.1 Verification

Checking the correctness of a system is an important part of the development process. Undetected errors of a system used in field can in the worst case risk human life. To prevent such errors different methods for checking the fulfillment of the systems requirements developed - this is called verification[Gra10].

There are different methods for verification. For hardware verification the most common is simulation. A software simulates the behavior of a hardware model, for Register Transfer Level (RTL) mostly described in Hardware Description Language (HDL), on a computer. The simulator applies input stimuli to the model and computes the behavior of the complete modeled system. This stimuli can be applied by a testbench, which also evaluates the output.

The developer has to create the stimuli for the testbench, which is connected to the Device Under Test, and applies this stimuli. The structure of such a system can be found in Figure 2.1. The stimuli should test cases where the DUT can possibly generate wrong output data, this is a non nontrivial task [PF05]. It is likely that the developer forgets some important test cases in the list of tests.

One problem of the verification is there are nearly no metrics to measure the success of the verification. An often misinterpreted number as an indicator for a good test is the coverage [Wie08]. The coverage is a factor which gives the executed code statements in comparison to the overall statements in a simulation run. But even if the coverage is $100\,\%$ this has nothing to to with the correctness of the implementation.

A better way for verification is to do an exhaustive test instead of only applying some test cases. One approach for exhaustive testing is to use formal mathematical methods for generating test patterns. First a mathematical model of the implementation is created which is used in the second step to generate a list of test pattern. The generation of the test pattern can be done by

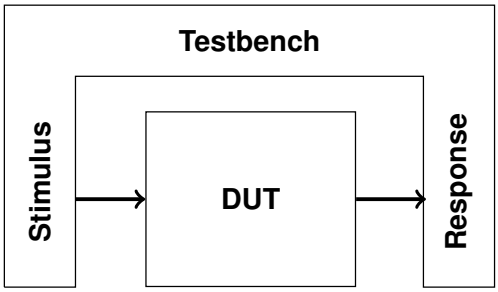


Figure 2.1: Structure of DUT and Testbench

using a SAT solver. A SAT solver checks a boolean expression for its satisfiability. The boolean expression $a \wedge b$ can be satisfied by $a = 1$ and $b = 1$ which leads to $1 \wedge 1 = 1$. With such a SAT solver all possible input values for a given output value can be calculated. Unfortunately this is not possible for systems with many inputs, outputs and internal states. In some cases it is even impossible to describe the mathematical model to use the SAT solver. This is for most image processing algorithms the case.

Another approach is to verify the output of all possible inputs. This is only possible for a small number of inputs and outputs, since for every additional input bit the needed number of tests will double. This leads to an exponential rising test time. In case of image processing algorithms it is in the most cases possible to reduce the size of the image which leads to lesser input values and reduces the test time. Structural errors in the implementation can also be found in the smaller image - as long as the image is not to small.

2.2 Connected Component Labeling

In computer vision image processing is done in multiple stages. The first step is to do a transformation of the image to get a better representation for image analysis. In this step the foreground and background is segmented. A common way to do segmentation of an image is by converting a gray scale or color image with a threshold to a black and white image. For the further processing steps it is mostly helpful to find foreground areas which belongs together. One common method for this is Connected Component Labeling [RP66]. In the book Machine Vision R. Jain et al. defines connected component as following.

Definition 2.2.1 (Connected Component)
A set of pixels in which each pixel is connected to all other pixels is called a connected component. [JKS95]

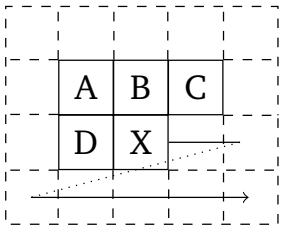


Figure 2.2: Neighborhood of the actual processed pixel X

Definition 2.2.2 (Connected Component Labeling)
A component labeling algorithm finds all connected components in an image and assigns an unique label to all points in the same component. [JKS95]

The labeled components can be used to extract features out of the image like areas of interest, number of areas, size of areas. This informations can be used for further processing steps for example object recognition. It can also be used to transfer only areas of interest in a image for further processing. This is especially for todays image sensor generation helpful. For Example the LUPA3000a 3 MPixel and 485 FPS CMOS Image Sensor has a throughput of over 13 Gbit/s [ON 12]. With only sending selected areas to the next processing step the saved bandwidth can reduce costs and complexity.

A pixel is connected to another if one of the neighbors have the same color. Each pixel has eight neighbors are directly below, above, diagonal above, diagonal below, left, and right of the current pixel. Most CCL algorithms are only for processing binary images - black and white. There are random access and pixel stream CCL algorithms [Bai11]. The random access implementations needs a buffered image for processing. On the contrary pixel stream directly process an incoming stream of pixels. Further only binary pixel stream CCL algorithms are discussed.

In terms of complexity the simplest is a multiple-pass algorithm. The incoming pixel is labeled with a zero if it is a background pixel. Otherwise the already labeled neighbors above and to the left are checked for labels and the smallest non-zero label is propagated to the new one. If none of the neighbors are labeled a new label is chosen. Figure 2.2 shows X and its neighbors with labels A, B, C, and D.

If an U-shaped structure is processed in the first row of the U both vertical lines are marked with different labels. But somewhere later in the picture one or multiple pixels merge the labels. An example image with labels after the first pass can be found in Figure 2.3.

Unfortunately the upper (now wrong) labels are written. Therefore we need an inverse run; starting with the last pixel and go from the right bottom corner to the left and then next is the

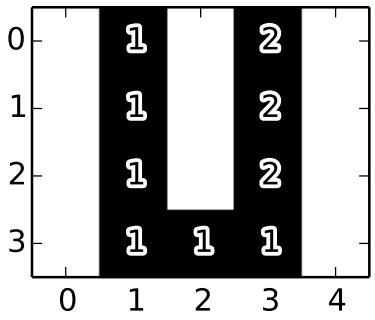


Figure 2.3: Labels of U shaped structure after the initial pass

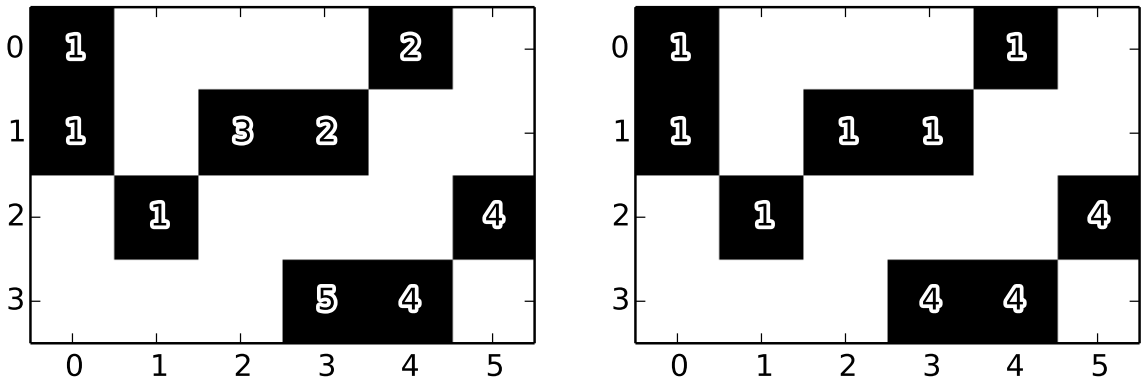


Figure 2.4: CCL example image after first-pass (left) and the final labels (right)

right pixel of the second last row and so on. Now the information is propagated back to the top of the image and all labels of the U are correct.

What happens if there is a W-shaped structure? In the first row are three different labels for the same component. After the first reverse run the label from the left V-shaped part of the W has the same labels. To get the label distributed over the complete component an additional forward and backward run is necessary. The algorithm stops only when a complete run passes with no labels altered. With this realization of the CCL the runtime of the algorithm depends strongly on the complexity of the input image.

2.2.1 Connected Component Labeling Two-Pass Algorithm

The idea behind the two-pass algorithm is to use the knowledge of the first run to do all label correction in a second run.

3	2
3	1
5	4

1	1
2	1
3	1
4	4
5	4

Table 2.1: Equivalence (left) and lookup table (right) of CCL example Figure 2.4.

If in the first run a foreground pixel with its neighbors having different labels is encountered, the smallest label is propagated to the current pixel and these different labels are written to an equivalence table.

In Figure 2.4 an example image is shown after the first-pass and the final labels. When the pixel at the coordinate (3,1) is processed it is possible to store the information label 3 is equals to 2. The resulting table can be found in Table 2.1.

An entry in the merge table would only be necessary if B is not labeled. Otherwise the label would have already been propagated to pixel C. In this case pixel C and A or D or both are labeled. The label of C and A/D are written in the equivalence table. At the end of the first run a table with all equivalence mappings are created.

In the second pass the equivalence table can be used to assign the final labels. Unfortunately there can be multiple equivalence entries for one label. In the example is the information stored $3 \rightarrow 2$, $3 \rightarrow 1$ which leads to $3 \rightarrow 2 \rightarrow 1$ this chain of labels needs to be resolved. The resolving of the chains by always selecting the smallest label as output label leads to a lookup table. The resulting lookup table can be used in the second pass to map all labels to the final labels. For the example image the lookup table can be found in Table 2.1. The second pass uses this lookup table to generate the final labeling.

The runtime of this algorithm is now better predictable. Two times processing all pixel and the time for the lookup table generation.

2.2.2 Connected Component Labeling Single-Pass Algorithm

In the section above the described two-pass algorithm is already the minimum number of passes through the image. The author of [Bai11] suggests to modify the problem slightly. In most cases the labeled image is used to extract features from it. The number of different areas can be easily calculated in one pass. By simply counting every time a new label is generated and decrement the counter when a merge is done. The size of the area can also be counted.

Another often needed feature is the bounding box around a region. Therefore it is not required to know all the final labels. This can be done in a single pass. Instead of storing and merging

labels a single pass algorithm stores the coordinates of each new labeled area. When another pixel gets the same label the coordinates of the area are updated. On the point where the two pass algorithm had to merge labels the single pass merges the coordinates of the boxes.

2.2.3 Parallel Connected Component Labeling

In [KRW+12] M. Klaiber et al. describes a method to parallelize the bounding box extraction of a binary image. The idea is to divide the image in multiple slices. For each of these slices a labeling unit is used to extract the bounding boxes. Every box with at least one neighbor pixel on each side of the border is merged after processing the own slice. All labeled areas without a pixel on the slice border are output directly.

An implementation of this CCL is verified in terms of correctness in this work.

3 Connected Component Labeling Hardware Reference Implementation

Exhaustive verification of all possible inputs needs a generator of the input stimuli. And a generator for the correct expected output to check the output of the DUT. This generator is a different implementation of a CCL algorithm and called in the following CCL reference implementation (CCL-REF). In Figure 3.1 the structure of the hardware architecture is given. The **STIMULI GENERATOR** generates the input stimuli. This stimuli are passed to a pixel generator (**PX-GEN**) which creates the input signals compatible for the **CCL-DUT** interface and another **PX-GEN** for the **CCL-REF** interface. Both outputs are connected to the **COMPARATOR** which checks the output for discrepancies and report this.

In this chapter the **CCL-REF** is explained all other components are explained in Chapter 4.

The goal for the CCL reference implementation in hardware is to get a correct implementation. As lesser the complexity of the structure used to compute the connected components as likelier there are no (or at least lesser) implementation errors. The simplest method is a multiple-pass algorithm. Here depends the runtime on the complexity of the processed image. A detailed look on this algorithm is given in Section 2.2.1. The two-pass algorithm as the name implies processes the image two times. Additional time for processing is required for the lookup table generation after the first pass. For this table some memory is required but on the other hand it is not necessary to store all labels after the first pass. Only the input image needs to be buffered. This saves memory since the input image is binary and the labels needs more bits to be stored. A discussion about the complexity in terms of runtime and size is given in Section 6.1. The two-pass implementation is a trade-off in runtime and complexity.

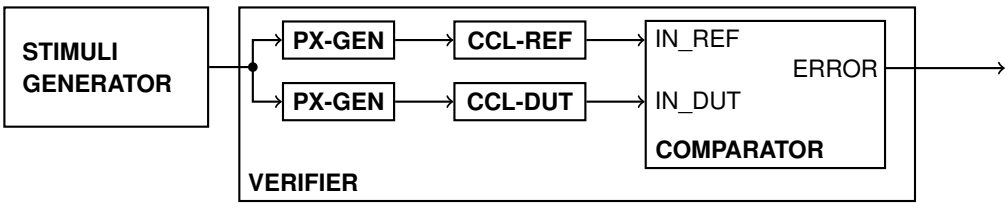


Figure 3.1: Overview of Hardware Implementation

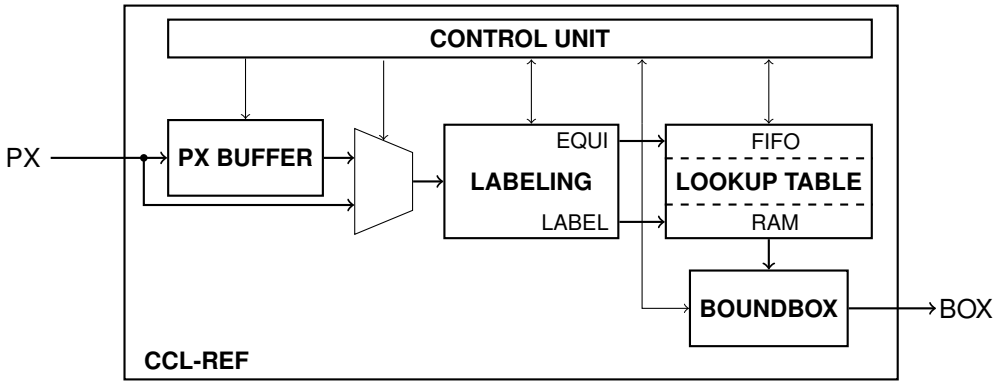


Figure 3.2: Overview of CCL Hardware Reference

The structure of the hardware implementation is given in Figure 3.2. Images are passed as pixel streams to to the labeling unit. At the first run each pixel is labeled by the **LABELING** entity with an initial label. The detailed description of this process can be found in Section 3.1. But as mentioned in Section 2.2.1 there can be **U** shaped structures in the image for which two labels need to be merged to one. The data of the LABEL output are discarded in the first run. Only the label merging information is passed through the EQUI output to the **LOOKUP TABLE**. After the image is processed once the **LOOKUP TABLE** merges all the equivalent labels to a lookup table. This process is described in Section 3.2. The next step is to process the image again with the labeling unit. Therefore the **PX BUFFER** is required to store the whole image in the first pass. Now the output of the labels are passed through the **LOOKUP TABLE** to replace the merged labels. In Figure 3.3 the states of the state machine are visualized. The condition last_lbl='1' after the state LBL P1 and LBL P2 means wait until the last pixel is labeled. To switch from the state WAIT LU to LBL P2 it is necessary that the lookup table generation is done. This is signaled by lu_ready. After the second pass there is a PRERST state (prepare reset). It is required to wait one clock to get the final label mapped by the lookup table which needs one extra clock cycle. In the next clock cycle (indicated by the condition clk) all entities: **LABELING**, **PX BUFFER**, and **LOOKUP TABLE** are reseted. One additional clock cycle later the entity is ready to process a new image.

As last step the bounding boxes around areas with the same label are calculated by the **BOUNDBOX** entity. This is explained in Section 3.3. The bounding box gives the start and the end coordinates of the areas with the same labels.

The maximum size of the image can be set to all entities as a generic. In addition there are inputs on every entity to set the size for each image in the range of 1 to the maximum image size defined as generic. The size, clock, and reset inputs are not shown in any figure to get a better clarity.

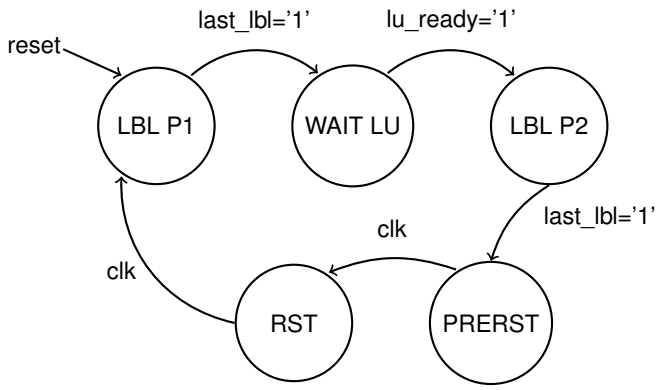


Figure 3.3: State machine of CCL entity

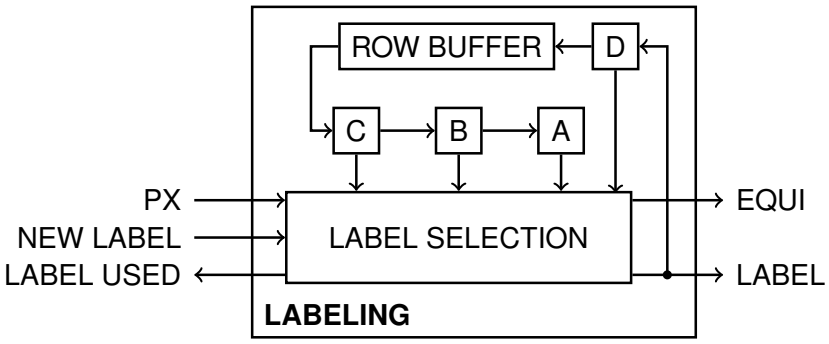


Figure 3.4: Overview of labeling entity

3.1 Labeling

In Section 2.2.1 it is described how the CCL two-pass algorithm works. This implementation only labels binary images. If the pixel has the same color like the background the label zero is written to the output. Otherwise the actual pixel is foreground and it is necessary to check the labels of the already processed pixels in the neighborhood. As shown in Figure 2.2 the labels are named: A, B, C and, D. The storage of these neighbor labels are implemented as a kind of shift register. The current assigned label is shifted in the register D then in a row buffer; a shift register with a size to store two labels lesser than the image width. Apparently, the out shifted label of the row buffer is C, which is stored in the next clock cycle in register B and after this in A. Figure 3.4 visualize the structure of registers.

To decide which label the current foreground pixel will get all necessary informations are stored in the registers. If all neighbors are unlabeled the LABEL SELECTION choose a new label number. This new number is provided by the input NEW LABEL. Then the unit asserts LABEL

Listing 3.1 EQUI Output Generation in Case of Merging

```
1  equi_valid_out <= '0';
2  if px_in /= C_BACKGROUND and b = C_UNLABELED then
3    --possibly merging required
4    if ((a /= C_UNLABELED) or (b /= C_UNLABELED)) and (c /= C_UNLABELED) then
5      --merge
6      equi_valid_out <= '1';
7      equi_out(1)    <= c;
8      equi_out(0)    <= a;
9      if a = C_UNLABELED then
10       equi_out(0) <= d;
11     end if;
12   end if; -- merge
13 end if; -- foreground
```

USED to confirm that the label has been used. Otherwise the neighbor label with the smallest number is chosen.

To solve the issue with areas labeled in previous rows differently and in a following line it turns out to be the same region; the **LOOKUP TABLE** needs the information of the equivalent labels. The EQUI output passes this information as a 2-tuple to the **LOOKUP TABLE**. In Listing 3.1 the determination of the output values is given.

The size of a label register needs the capability to store the biggest possible label. The worst case image in terms of the maximum number of different labels has alternating foreground and background pixels in the even rows. The odd rows complete background - no labels are merged. Certainly even and odd rows can be swapped. In this case, one gets a new label for every foreground pixel. From this it follows that $\left\lceil \log_2 \left(\left\lceil \frac{IMG_w}{2} \right\rceil \times \left\lceil \frac{IMG_h}{2} \right\rceil \right) \right\rceil$ bits are required to store one label. IMG_w means the width of the image - IMG_h height of image.

3.2 Lookup Table and Equivalence Mapping

The **LOOKUP TABLE** entity has to generate, with the equivalence tuples of the first pass, a table to map all incoming labels of the second pass to the final labels. The table is realized as a Random Access Memory (RAM) with a word width of the maximum label size and with the ability to store each possible label in a different row. This is necessary to represent the lookup table for worst case image. Where for each row the address is the same than the value.

To initialize the rows in the lookup table it is required to know how many labels are used in the first labeling run. This is done by a label counter which is incremented in a handshaking fashion. The LABEL USED signal gets high every time a new label is used. The current label counter goes through the NEW LABEL output to the LABELING entity. With this approach the

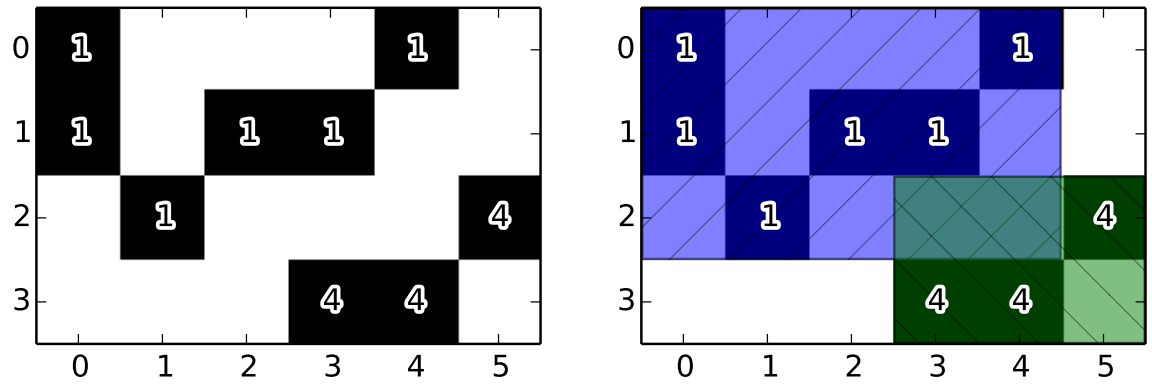


Figure 3.5: Labels after second-pass and bounding boxes

RAM is initialized while the first pass is in progress. The received equivalence tuples are stored in First In First Out (FIFO)-Buffer; only called FIFO.

After the first pass is completed, the FIFOs equivalence tuples are read out and the RAM entries are updated accordingly. An update is done when either the RAM address or the value is equivalent to the first value of the FIFO tuple. In this case the RAM entry is replaced by the second FIFO tuple. Only if the new value is smaller than the old one. The replaced value and the new value is added as a new tuple to the FIFO. This is repeated until the FIFO is empty. At this point all equivalent labels mapped to the smallest common label.

The CONTROL UNIT gets a ready signal and triggers the second pass.

3.3 Bounding Box Calculation

Last step is to get the diagonal start and end point of the minimum surrounding rectangle around an area with all the pixels having the same label. The labels of the second labeling pass are used to calculate the boxes. An example of the labels and the resulting boxes can be seen in Figure 3.5. There are two different labels and hence two boxes. The result of the bounding box calculation will be (0,0),(4,2) and (3,2),(5,3).

The computation of the boxes is done in a serial way. All labels after the second pass goes into the **BOUNDBOX** entity. This stores the start and end coordinate of first received labels to a RAM. The address is the number of the label. To get the coordinate the has a counter which keeps the number of received labels. The image width is also required. With this information a counter tracks the coordinate. The algorithm to update the ram can be found in Algorithm 3.1.

Algorithmus 3.1 Bounding Box Calculation

```
if line_valid[label] then
    if pos_x < RAM[label].start_x then
        RAM[label].start_x ← pos_x
    end if
    if pos_x > RAM[label].end_x then
        RAM[label].end_x ← pos_x
    end if
    if pos_y > RAM[label].end_y then
        RAM[label].end_y ← pos_y
    end if
else
    RAM[label].start ← (pos_x, pos_y)
    RAM[label].end ← (pos_x, pos_y)
    line_valid[label] ← True
end if
```

The output of the coordinates are done if for at least one row no update happened, or in the last row if the current x coordinate is greater than the x coordinate of the end coordinate. Applying the algorithm, the delay for the output of the boxes is at most one image width.

The only open question is how big should this RAM be? It is necessary to store for every possible label a box. Therefore we need $\lceil IMG_w/2 \rceil \times \lceil IMG_h/2 \rceil$ lines. Same reason as for the biggest possible number of the label. The size of one line depends on the size of the coordinate tuple. A x-coordinate needs to store between 1 and the image width. Same for the y-coordinate but as maximum the image height. Putting all this together the number of bit per line is:

$$2 \times \left(\log_2\left(IMG_w\right) + \log_2\left(IMG_h\right)\right) \left\lceil \log_2\left(\left\lceil \frac{IMG_w}{2} \right\rceil \times \left\lceil \frac{IMG_h}{2} \right\rceil\right) \right\rceil$$

3.4 Resource Usage and Timing

All hardware implementations are done in VHSIC Hardware Description Language (VHDL) and optimized for FPGA synthesis.

The used FPGA is a Xilinx Virtex-6 LX-240 (xc6vlx240t-ff1759-2). This FPGA has $37\,680$ Slices. Each Slice persists of four six input Look Up Table (LUT)s and eight flip-flops. There are also Digital Signal Processing (DSP) Slices, Block RAMs, clock management modules, PCI Express interfaces, and Ethernet Media Access Control (MAC)s. More informations about this FPGA can be found in the Virtex-6 family overview [Xil12b]. In Table 3.1 a resource usage and timing overview of every entity is given. The total number of used LUTs for one complete CCL unit

Entity	Max Clock	Registers	Logic LUTs	Memory LUTs
LABELING	313 MHz	52	151	8
LOOKUP TABLE	250 MHz	116	225	8
BOUNDBOX	330 MHz	42	117	24
CCL with **BOUNDBOX**	250 MHz	233	569	42

Table 3.1: Usage of Resources and Timing Results after Synthesis for a 6 x 6 Image

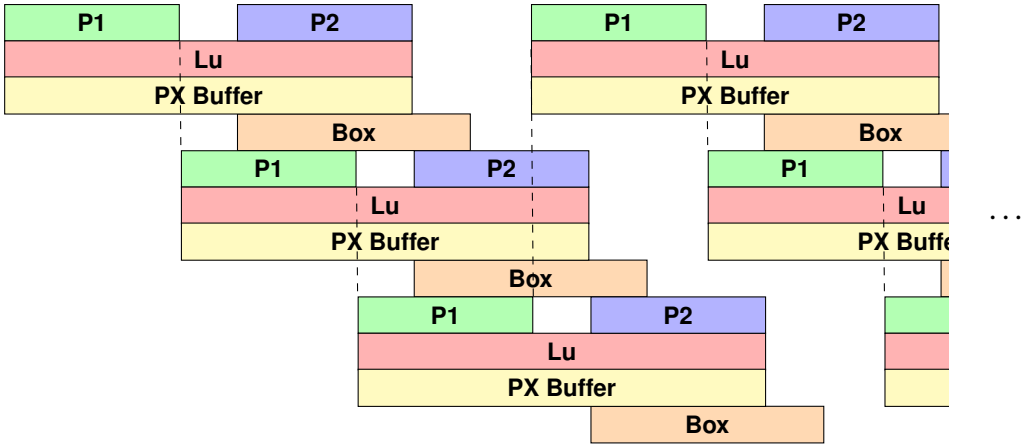


Figure 3.6: Component usage of pipelined CCL

including the bounding box calculation is $569 + 42 = 611$. Since every Slice has four LUTs a total of 153 Slices are occupied. This are $0.41\,\%$ of all slices.

3.4.1 Optimization

With the former described architecture it is not possible to write each input image after another. It is required to wait until the whole image is processed before the processing of the next image can start. This issue can be solved by simply instantiate three times the complete CCL architecture, and use the inputs alternating.

A much cleverer way to do this is replicating only the required components and build a fully pipelined architecture. Figure 3.6 shows the utilization of the required components to process a continuous input stream.

The box marked with “P1” and “P2” are the first and second labeling pass which requires a **LABELING** unit. In the figure it can be seen, all the “P1”s are not overlapping. This means one **LABELING** unit is enough to process all first passes, same with “P2” for the second pass. The

Unit	Number	Registers	Logic LUTs	Memory LUTs
LABELING	2	52	151	8
LOOKUP TABLE	3	116	225	8
BOUNDBOX	2	42	117	24
PX BUFFER	3	58	2	12
Total	-	710	1217	124

Table 3.2: Resource usage of pipelined CCL implementation for a 6 x 6 Image

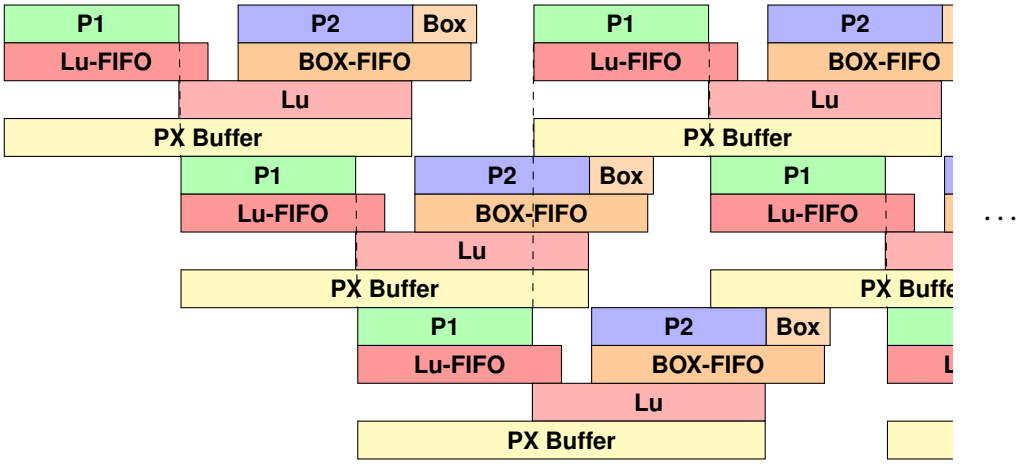


Figure 3.7: Component usage of improved pipelined CCL

"Lu" labeled boxes needs **LOOKUP TABLE** units for processing. Three of them are overlapping - three **LOOKUP TABLE** units are required. Also three of the "PX Buffer" labeled boxes are overlapping - three pixel buffers are necessary. And finally two **BOUNDBOX** units are required. The Table 3.2 gives a total resource usage of 1217 logic and 124 memory LUTs and 710 registers. The synthesis tool implemented all memory for optimization reasons in LUTs.

Is there more space for optimization? A closer look to the usage of the **LOOKUP TABLE** shows, while the pass one runs it is only used as a storage, the lookup table creation starts when the first pass is done. It is possible to add a FIFO to the output of the first pass **LABELING** unit. With two of the FIFOs it is possible to do the **LOOKUP TABLE** processing of the equivalence table after the first pass is done. Now only one **LOOKUP TABLE** unit is required. The same can be done for the bounding box calculation. The Component usage of this improved pipelined architecture can be found in Figure 3.7. An overview of the FPGA utilization can be found in Table 3.3

Unit	Number	Registers	Logic LUTs	Memory LUTs
LABELING	2	52	151	8
LOOKUP TABLE	2	116	225	8
Lu FIFO	2	32	33	16
BOUNDBOX	1	42	117	24
Box FIFO	2	40	62	24
PX BUFFER	3	58	2	12
Total	-	696	1065	172

Table 3.3: Resource usage of improved pipelined CCL implementation for a 6 x 6 Image

This improved architecture needs 7.76% lesser of the FPGA LUTs than the first pipelined version. In comparison to the naive three times replicated architecture it saves 32.5% of the LUTs.

4 Hardware Verification Architecture

Verification of hardware means to do a check of the implementation against the requirements [Gra10]. The goal here is to do this check autonomously and report any found discrepancy. Verification is often associated with formal mathematical way. But this formal verification methods has a weakness. They verify that the specification fits to the code but neither it proofs the correctness of the translation process to hardware or machine code nor that the specification fulfills the requirement. The idea here is to do an exhaustive test of the DUT (a single pass CCL). And check for all possible input images all outputs. This only works for small images. A simulation of the DUT for a 4×4 image will take for all 2^{16} input stimuli round about 30 minutes (with ModelSim and an Intel i7-3770 processor), an image with a dimension of 6×6 will already take 2^{20} times longer - nearly 60 years.

For this reason the verification is done in hardware on a FPGA. The runtime for the 6×6 exhaustive test is for the implemented architecture 14 minutes. An additional advantage is the translation of the verification architecture uses the same synthesis tool as the final DUT. With this it is also possible to find errors in this tool.

To do this automatically the input data needs to be generated automatically by the verification hardware. The simplest means to generate the input data is to use a counter. All the stimuli

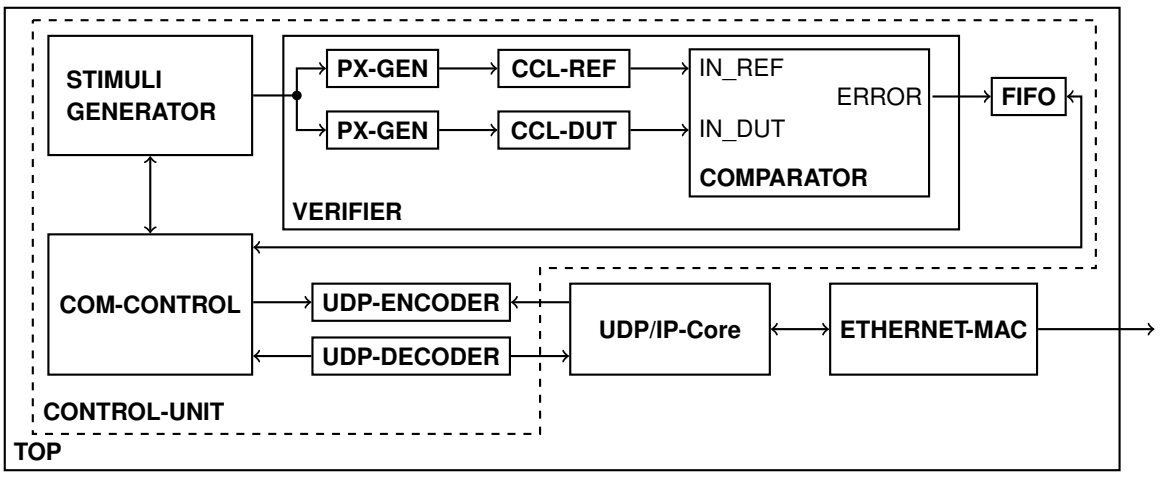


Figure 4.1: Overview of complete Hardware Verification Architecture

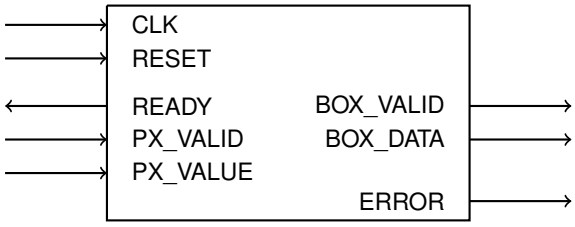


Figure 4.2: Inputs and outputs of the CCL DUT

are applied to the **CCL-DUT** (more details in Section 4.1) and, in parallel, to the reference implementation described in Chapter 3. Now the outputs of both CCL architectures needs to be compared. The problem here is the order and the delay of the outputs can differ. Therefore a comparison unit is necessary. This is explained in Section 4.2. We assured that the reset of both architectures is implemented correctly. Otherwise data of a former image can influence the output of a later image. Without this, all possible permutations of the input order needs to be tested. This are even for small images to many input combinations to test them all in an appropriate time.

The pixel generation for the CCL implementations is done by the **PX-GEN** process of the **VERIFIER** entity. Each of them gets the same stimulus from the **STIMULI GENERATOR** and generates the pixel data according to the interface specifications of the CCL implementations.

If the comparator detects any discrepancy this needs to be reported. Failed stimuli and errors are stored in a FIFO and can be transmitted to a PC over User Datagram Protocol (UDP). The UDP communication is handled by the **COM-CONTROL** state machine. Which also gives the ability to read some status informations and set the start and end stimulus for the **STIMULI-GENERATOR**. This is a simple counter. To encode and decode the UDP messages there are two entities **UDP-ENCODER** and **UDP-DECODER**. All of them are instantiated in the **CONTROL-UNIT** entity. That is connected to the **UDP/IP-Core**.

The software on the PC side is described in Chapter 5. Its corespondent communication part of the hardware verification architecture is explained in Section 4.4.

4.1 Device Under Test

The DUT is a single pass CCL unit for a parallel implementation. It is used in the verification as a black box. The only information of the DUT required is the interface specification. A visualization of this can be found in Figure 4.2. It can be clocked with a maximum speed of $143\,\text{MHz}$ for an image size of 6×6 pixels. The PX_VALUE is the binary pixel input. Pixel data can only be assigned to the DUT if the READY output is high.

On the output side the BOX_VALID is high if the BOX_DATA bits are valid. The BOX_DATA is the binary coded bounding box. Its size depends on the image size and can be calculated as followed: $2\lceil\log_2(IMG_w)\rceil + 2\lceil\log_2(IMG_h)\rceil$ (IMG_w is the image width and IMG_h the image height).

The error output is an error vector used to report internal errors like unexpected overflows. This error is sent to the PC with the stimulus which raised the error and can be used to verify that some assertions are true for all tested input pattern.

4.2 Comparator

To check the correctness of both CCL implementations the outputs has to be compared. The output of the found boxes of both implementations can be in different order. Every output can only appear once. Therefore all detected boxes of **CCL-REF** and **CCL-DUT** need to be stored while the image is processed. In Section 4.2.1 the complexity of different approaches in terms of runtime is analyzed. Section 4.2.2 gives a look at the chosen implementation.

4.2.1 Runtime Analysis

The comparison can be done in different ways. First the amount of outputs is compared if they are not equal the error detection can report a discrepancy and no more testing is required. Otherwise some how all elements of both CCL units needs to be compared. With the assumption the outputs are in the same order the comparison can be done by checking the first output of the DUT with the first output of the reference; next the second of both and so on. When all outputs compared and no discrepancies are detected the comparison is successful. As soon as one value is different the comparison can be stopped and the error reported. To use this approach the values needs to be sorted before the comparison can be done.

Another approach for unsorted inputs is to check the first output of the DUT with all outputs of the reference. If no one of the compared values are equal both outputs are different. Otherwise the same is done for the second output of the DUT. This is repeated for all DUT outputs.

The question is which of the two methods are faster?

The outputs are integer values (represented as STD_LOGIC_VECTOR). Therefore only sorting of integers need to be done. For sorting the data the Selection sort algorithm is simple and easy to implement. Both, best and worst case sorting performance is $\mathcal{O}(n^2)$ [Knu98]. After sorting the data needs to be read once again to do the comparison; $\mathcal{O}(n)$. The total runtime is dominated by the sorting runtime $\mathcal{O}(n^2)$. A more efficient sorting algorithm is heap sort. Its implementation in hardware is not trivial. But the runtime is only $\mathcal{O}(n \log n)$ [Knu98].

Without sorting and a direct comparison both outputs of the implementations can be seen as two list; further called left and right list. In the worst case it is necessary to compare the first value of the left list with all values on the right list; n operations. Then the found element is removed form the right list. Next element of the left list has to been compared to $n-1$ elements of the right list. The total number of comparisons is:

$$n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2} = \frac{n^2+n}{2} \Rightarrow \mathcal{O}\left(n^2\right)$$

For small values of n the runtime difference of both methods are not that big. An image with a size of 6×6 can have in the worst case 9 boxes; see Section 3.3 for more details. The difference of both implementations is round about a factor of two. Bigger factors than two are ignored by the asymptotic Landau notation and the sorting is necessary for both lists of output values. For this and the lesser implementation complexity of the second method without sorting it is the choice we implemented int the comparator. This approach has another advantage over the sorting approach. One of the lists can be implemented as a FIFO the second list needs to be a RAM. The FIFO can be implemented asynchronously and combine to different clock domains. This gives an easy possibility to run one of the CCL implementation with a different clock speed. In Section 4.2.2 an implementation with this approach is presented.

A third option for a small amount of values to compare is sorting while inserting. In this case the lists are represented as one big register. Adding a new value to the list needs to check on which position the new value needs to be to get a sorted list after inserting. The inserting itself is done by shifting all values bigger than the new value to the right and write the new one into the gap. This can only work if the critical path of the multiplexer and comparators is short enough to fulfill the timing requirements. The benefit here is the sorting is done while the both other approaches are idle. The output of the result can be done in one clock cycle by only comparing the both registers. But this implementation needs more space on the FPGA.

4.2.2 Implementation

In Figure 4.3 the internal structure of the comparator is depicted. This implementation stores the output of the **CCL-REF** in a RAM. The input address is incremented every time valid data are on the IN_REF input. The output of the **CCL-DUT** is stored in a FIFO. After the last output is written LAST_IN signal starts the comparison. In the first clock the IN ADR **ADR CNT** value is compared with the **FIFO** FILL LEVEL a difference in this signal ends the comparison with an error. Two different errors can be reported in this case. If the **CCL-DUT** has written more words to the output than the **CCL-REF**, or the other way around, where the same amount of output words the comparison is done as explained in Section 4.2.1 for the approach without sorting. The **CONTROL UNIT** is a state machine which controls the comparison. In case the output at

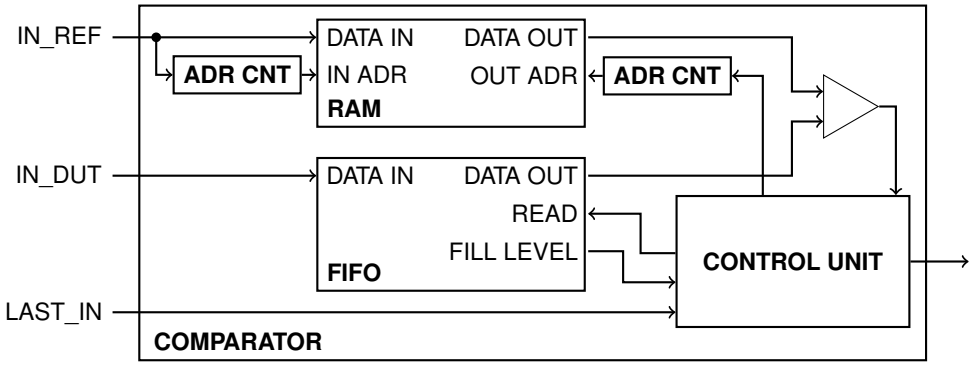


Figure 4.3: Structure of the Comparator

Code	Error Type
0	NO ERROR
1	#**CCL-DUT** < #**CCL-REF**
2	#**CCL-DUT** > #**CCL-REF**
3	DIFFERENT OUTPUTS

Table 4.1: Error value encoding of Comparator

the **FIFO** is found in the **RAM** the address is marked as found in the **FOUND** register. At the end it reports no error if the **FIFO** is empty and all valid addresses of **FOUND** are marked, otherwise an error is reported. The coding of the errors is given in Table 4.1.

An implementation of the third option described in Section 4.2.1 (sorting while inserting) is also done the VHDL code can be found in Listing A.2.

Image Size	Max Values	Implementation	Max Speed	LUTs
8×6	12	Type 2	291 MHz	221
		Type 3	227 MHz	1828
9×5	15	Type 2	230 MHz	301
		Type 3	170 MHz	2430
6×6	9	Type 2	289 MHz	197
		Type 3	251 MHz	750
4×4	4	Type 2	343 MHz	110
		Type 3	355 MHz	156

Table 4.2: Performance of different Comparator implementations

Image Size	LUTs	FFs	BRAMs	Clocks per Image	Clocks per Pixel
4×4	5646	3180	5	72.3	4.52
6×6	6680	3649	5	123.7	3.44
21×2	7052	3827	5	169.7	4.04
9×5	7339	3876	5	140.9	3.13

Table 4.3: Speed and Utilization of Hardware Verification Architecture

An overview of required space and the maximum possible clock speed can be found in Table 4.2. It is interesting to see an image with a size of 8×6 pixel can have a maximum of 12 boxes to compare. On the other hand a 3 pixels smaller image with a size of 9×5 can have 3 more boxes. In the table, Type 2 is the implementation without sorting and Type 3 is the sorting while inserting.

4.3 Speedup

The runtime for different image sizes can be found in Table 4.3. To get a value for the comparison between different image sizes, the clocks per image and clocks per pixel is added. It is the average runtime to verify one image, respective one pixel, in the exhaustive test run. The used clock speed is $125\,\text{MHz}$. Every time the image size is increased by one pixel the runtime at least doubles since the number of stimuli are doubled. To reduce the runtime it is possible to use multiple **COMPARATOR** units and attach a pair of CCL architectures to each of them for each of them a pair of CCL architectures and run them in parallel; or reduce the runtime for one image.

In the next section the **VERIFIER**, which uses multiple instances to speedup the architecture, is explained in more details.

4.3.1 Verifier

The idea of the **VERIFIER** is to instantiate as many **COMPARATOR**s, **CCL-DUT**s, and **CCL-REF**s as the resource utilization of the FPGA allows. For that scheduling the inputs and outputs of the different entities is required in a way to get the lowest possible idle time of each. The CCL-REF for example can not output data before the first pass is done. This gives at least as many clock cycles as the image has pixels for the comparator to do another comparison. By adding a FIFO to the box output this time can even be increased since the amount of boxes is much lesser than the number of pixels.

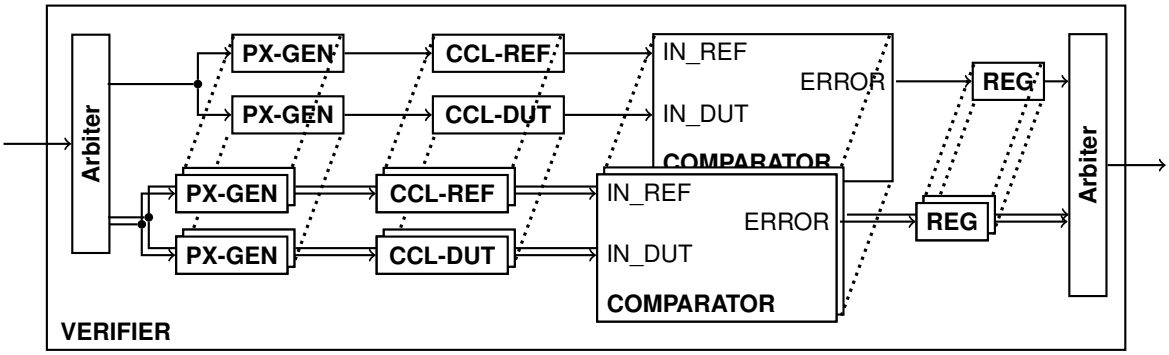


Figure 4.4: Multiple CCL entities for speedup

The most sophisticated scheduling of the components can be done by a full dynamical assignment of the components to each other with a scoreboard. It is a technology used for superscalar processors to optimize the order of the instruction execution to get a higher utilization of the components and prevent data hazards [HP12]. Therefore a table with all available components is created and for a fixed number of next instructions their required data and components are marked. As soon as all requirements are fulfilled the execution start. To use this method here, a fixed number of stimuli are queued and all required data to process are listed in a table. As soon as the necessary data are available and an entity for the next processing step is idle, it is assigned to this stimulus. The problem here is the scheduling gets really complicated and it is difficult to ensure that nothing could go wrong. This approach is not researched in detail here.

Another method is to get the **CCL-REF** and the **COMPARATOR** to a runtime with lesser variances between the worst and best case. With this it is possible to do a static scheduling of the components and increase the utilization. Therefore the **CCL-REF** is optimized to read a continuous stream of input pixels without the need of a reset or pause at the input. The output of the boxes is done in a more compressed way, by adding a FIFO to the output. See Section 3.4.1 for more details. An implementation with a generic image size influences the runtime of each component to highly to do a fixed scheduling.

In Figure 4.4 the most intuitive implementation of parallelization given. It uses the basic implementation of the **CCL-REF** and combines it with one **CCL-DUT** and one **COMPARATOR** in a fixed way. The number of instances can be defined by a VHDL generic. For the output of the errors an arbitration is implemented to write every clock cycle at most one error with its stimuli to the output. To prevent the lose of errors a register to the comparator output is added. Without this additional register the arbiter can only output one error in case two of the comparators are done in the same clock cycle. As long as there are lesser comparators than the minimum time required to process one image, no error report can be lost. The input

Image Size	Max Instances	occupied Slices	Clocks per Image	Clocks per Pixel
4x4	95	95 %	0.672	0.042
6x6	84	99 %	1.368	0.038
9x5	75	99 %	1.890	0.042

Table 4.4: Speed and Utilization of Hardware Verification Architecture with Multiple Instances

also needs an arbiter to select an idle CCL pair an assign the stimuli to this one. This input interface only works as long as the number of instances is smaller than the number of average clock cycles required to process a stimulus. Every clock cycle only one stimulus can be assign. If the number of instances is higher there are all the time some units idle. This bottleneck can be removed by simply partition the stimuli and add an additional **STIMULI GENERATOR**. But with the partitioning the architecture around the **VERIFIER** needs also to be changed. Without partitioning it is the same interface as before and the architecture needs no changes.

Table 4.4 gives an overview of this improved verification architecture.

4.4 Communication with PC

The detected errors should be reported to the PC without losing data. Therefore the communication with the PC should be reliable. In addition no special communication hardware should be used. One of the most used and tested interfaces is Ethernet. Today nearly every computer has at least one interface. On the operating system side the software stacks are tested and developed over many years. And nearly every programming language has an interface to communicate with this stack.

The used FPGA development board provides multiple Ethernet transceivers. Therefore Ethernet is used as communication protocol in combination with UDP over Internet Protocol (IP) (now on called UDP/IP). An introduction to Ethernet, IP, Transmission Control Protocol (TCP), and UDP is given in Section 5.2.1. UDP is used for this architecture since it reduces the complexity on the FPGA side. We did not use TCP as there is much overhead. For example it is necessary to store the state of a connection and do handshake to establish a connection. It is also required to store already transmitted data to resend it in case of a transmission error. This requires more memory on the FPGA side and is prohibitively expensive to implement in our case as it limits the replication of the comparison units drastically.

On the other hand with the usage of UDP the protection of the data integrity is not as good as with TCP. However, this issue is resolved as will be detailed in Section 5.2.2.

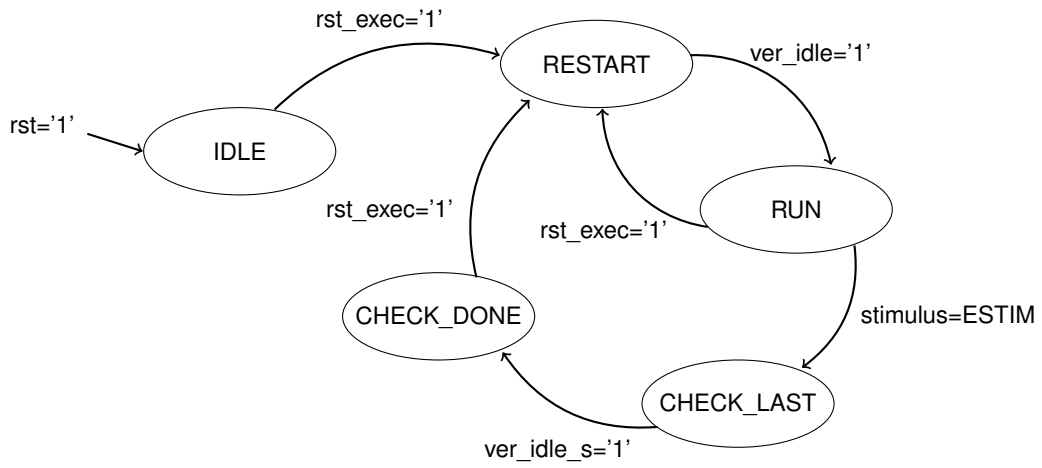


Figure 4.5: State machine for internal state of CONTROL-UNIT

In the following Section 4.4.2 the used Ethernet core on the FPGA is explained. Followed by the illustration of the UDP/IP-Core in Section 4.4.3.

4.4.1 Communication Control

The **CONTROL-UNIT** coordinates the communication between the internal registers and the UDP/IP-Core. For the decoding of incoming UDP packages the **UDP-DECODER** unit is instantiated. It is a state machine which does the communication with the UDP/IP-Core and passes the message type, id, data length, and data specified in Section 5.2.2. The former serial stream data are written to registers. **UDP-ENCODER** does the same visa versa.

There are two state machines one is used to track the internal state see Figure 4.5. After an external reset the it is in the IDLE state. This is leaved when the rst_exec signal gets high which is controlled by the communication control state machine. In the RESTART state the error **FIFO**, and the **VERIFIER** with the CCL entities and the COMPARATOR are reset. The value of start_stimuli, a register which is controlled by UDP messages, is set to the **STIMULI GENERATOR**. After the **VERIFIER** signals idle and the reset is active for a given minimum time the RUN state is entered. A RUN is done after the current stimulus reaches the value of the end stimulus register. Then the state machine remains in the CHECK_LAST until all running verifications are done and all errors are stored in the error **FIFO**. The other way to leave the RUN state is by receiving a restart message over UDP.

The mentioned communication state machine switches between a no operation and one of the possible communication message states. All possible messages are explained in Section 5.2.2. In addition there are a state for a header error, in case a malformed header is received and

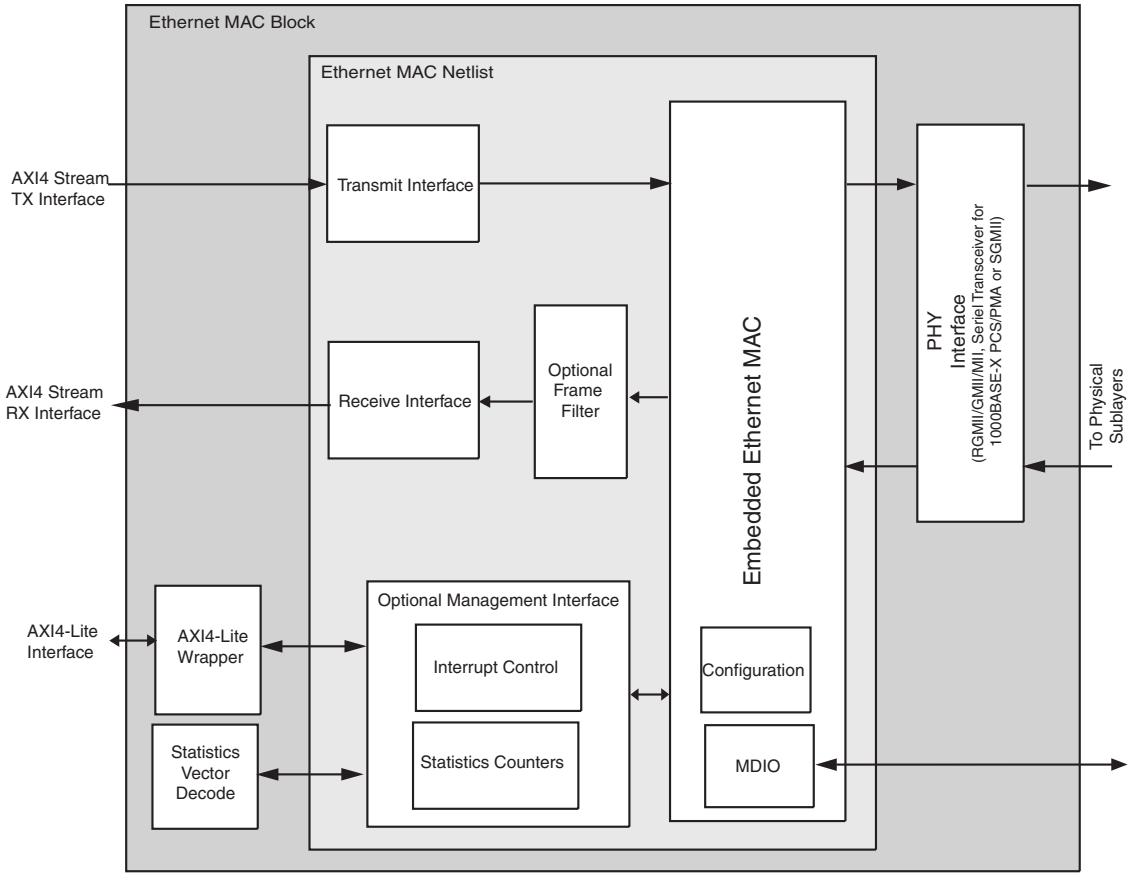


Figure 4.6: Block Diagram of Xilinx Virtex-6 Tri-Mode Media Access Control Core [Xil12a]

a state which is reached after receiving unknown message types. In every state except the NOP state a Transmit (TX) message is generated and transfered by the **UDP-ENCODER**. How the **UDP-ENCODER** transfers its message over Ethernet to the PC is explained in the following Section.

4.4.2 Ethernet MAC Core

The Ethernet core provides the interface between the Media Independent Interface (MII), an interface to the physical layer chip, and the data which are transmitted over Ethernet. More informations about this interface can be found in the IEEE 802.3 the standard which specifies Ethernet [IEE02].

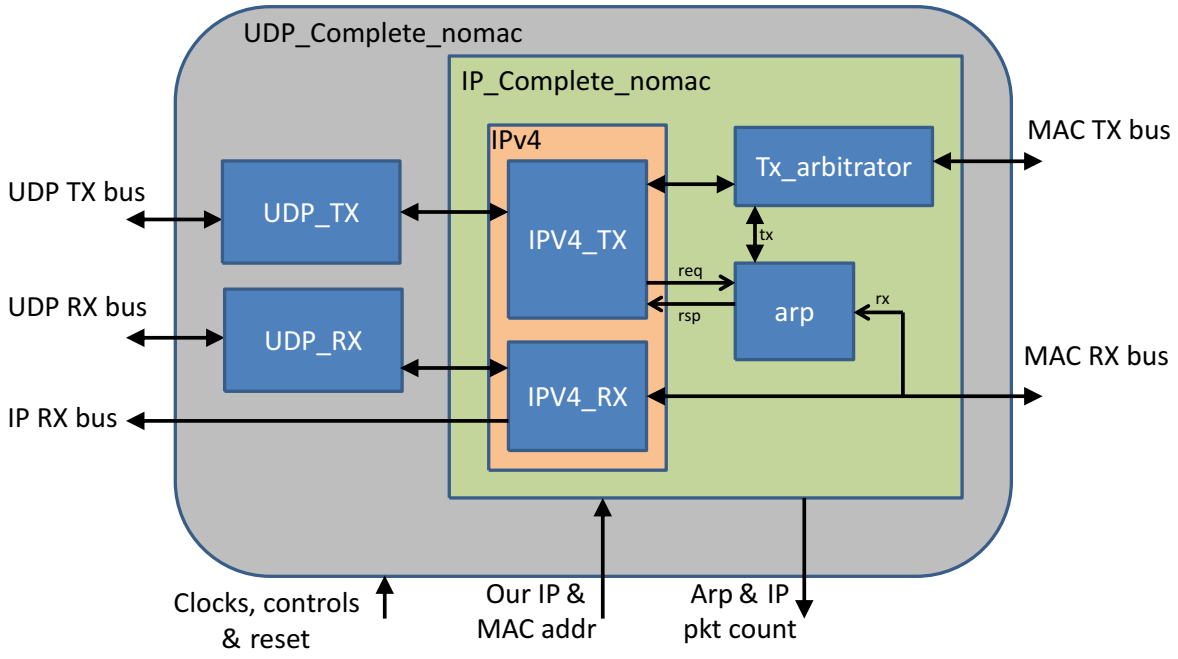


Figure 4.7: Block Diagram of Complete UDP/IP-Core [FF13]

Figure 4.6 shows a block diagram of the internal structure of this core. On the right side of the diagram is the (G)MII is connected on the development board High Tech HTG-V6-PCIE-L240-2 to a Marvell 88E111 10/100/1000 Gigabit Ethernet transceiver. The left side has a AXI4-Lite Interface it is not used by this architecture. It can provide statistic informations about the received and transmitted bytes as well as different error or quality counters.

The other interface is the AXI4-Stream TX/RX interface. It is connected to the IP core and is used to transfer the valid Ethernet data. The core gives the ability to configure the supported transmitting speeds. For the architecture the support of 10/100/1000 Mbits is enabled. The different clock speeds for the three transfer speeds are generated by the core. Only one 125 MHz clock speed is required.

AXI4-Lite and AXI4-Stream are part of the Advanced Microcontroller Bus Architecture (AMBA) protocol specification of ARM Holdings. The AXI4-Lite is specified in [axi13] and AXI4-Stream in [axi10].

4.4.3 UDP/IP Core

Decoding of the received Ethernet data is done by the UDP/IP core. The core is licensed under the LGPL and provided by Peter Fall and the FIXQRL project through `http://opencores.org`.

A platform for sharing open source hardware IP-Cores (IP means in this context not Internet Protocol it stands for Intellectual Property).

In Figure 4.7 internal structure is visualized. The MAC Receive (RX)/TX bus is the AXI4-Stream interface to the MAC core. There is a Address Resolution Protocol (ARP) entity which implements the IP address resolution specified in RFC 826 [Plu82]. It has to answer whenever an Ethernet broadcast message with a ARP discovery is received. A ARP discovery asks for the MAC-Address of the user with a specific IP-Address. It resolves the corresponding MAC-Address to a IP-Address and visa versa. To prevent sending ARP requests for every outgoing IP package a storage to remember the last 100 used IP-Addresses and their corresponding MAC-Addresses is implemented. An overview of the ARP entity is given in Figure 4.9. To give the ARP entity the ability to send messages the Tx_arbitrator switches between the TX data of the IPv4-Core and the ARP.

A receiving Ethernet package is decoded by the IPV4_RX entity. The decoded IP header and the RX AXI4-Stream is passed to the UDP_RX entity and to the **CONTROL-UNIT** entity. The **CONTROL-UNIT** can check the header information of the package like source IP-Address or if this is a broadcast message. The complete ipv4_rx_type passed on its declaration can be found in Listing A.3.

In the next step the UDP_RX entity decodes the UDP header. The in Listing A.4 defined record is passed to the **UDP-DECODER** entity. It includes the decoded header information like source IP-Address, source and destination port, length of the data part, and a valid flag. The **UDP-DECODER** uses the AXI4-Stream to read the data part of the UDP package.

The sending of a package is done in reverse order first the **UDP-ENCODER** transfers the header as a record (Listing A.5) to the UDP_TX entity. As soon as the IPV4_TX entity transmitted the IP header followed by the UDP header to the Tx_arbitrator the UDP_TX signals the **UDP-ENCODER** to start transmitting the data part as a AXI4-Stream.

4.5 Clock and Reset Generation

In the previous chapters the clock and the reset generation to all entities was ignored. The **ETHERNET-MAC** requires a reset for a specific amount of time. To fulfill this requirement a reset module is connected between an external reset, which can be triggered by a push button on the FPGA development board and the internal reset signal. The internal reset signal is distributed to all entities.

If the reset module gets an external reset or after the personalization of the FPGA it executes a reset for a given amount of clock cycles. This clock cycles can be defined by a generic. In case of an external reset counting starts after the release of the external reset button.

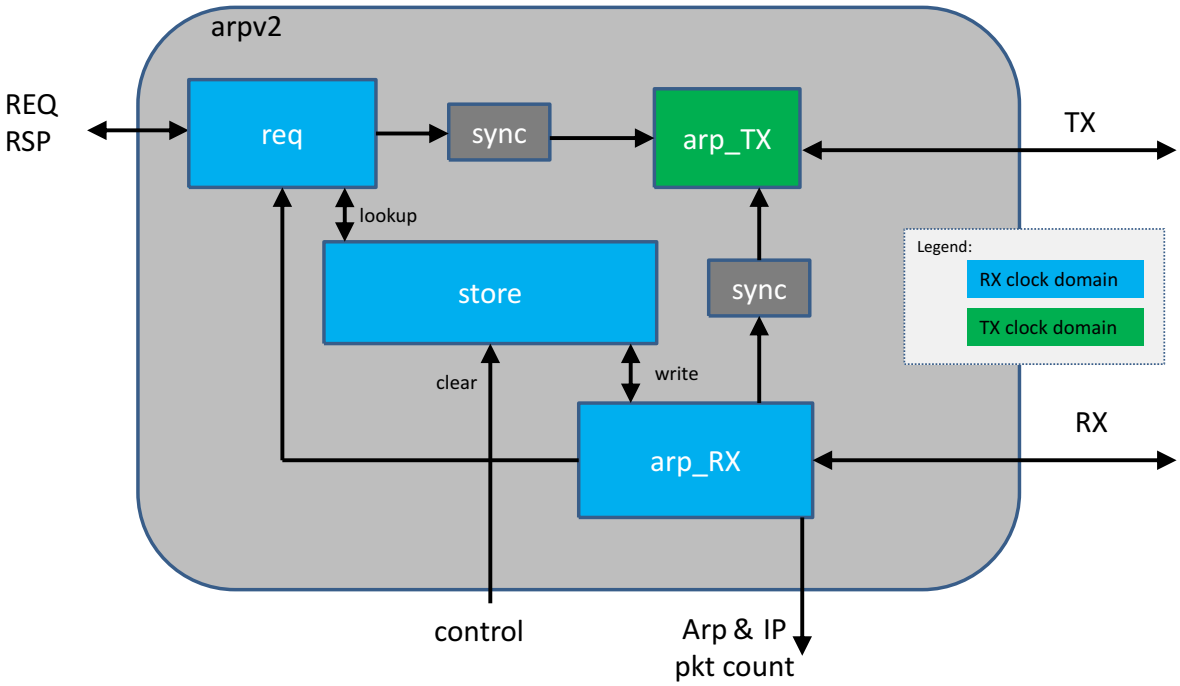


Figure 4.8: Block Diagram of the ARP-Core [FF13]

The required clock for the reset counter is the same used for all entities. The **ETHERNET-MAC** has special requirements for the clocking. It is necessary to provide a 125 MHz clock frequency which is required for the Gigabit Media Independent Interface (GMII) interface to the MAC transceiver, see Section 4.4.2. In addition it needs a 200 MHz clock for an internal counter. If one of the CCL architectures should run on a different speed a third clock frequency is required.

Unfortunately the FPGA development board only provides one clock cycle of 100 MHz. To get the different required clocks a Mixed-Mode Clock Manager (MMCM) which is described in the following section is used.

Currently the hardware verification architecture runs at 125 MHz clock speed and distributes this to all entities. Everything runs in the same clock domain with the exception of the named communication part.

4.5.1 Mixed-Mode Clock Manager

The used Virtex-6 FPGA provides two MMCMs. A block diagram in Figure 4.9 gives the internal structure of the MMCM.

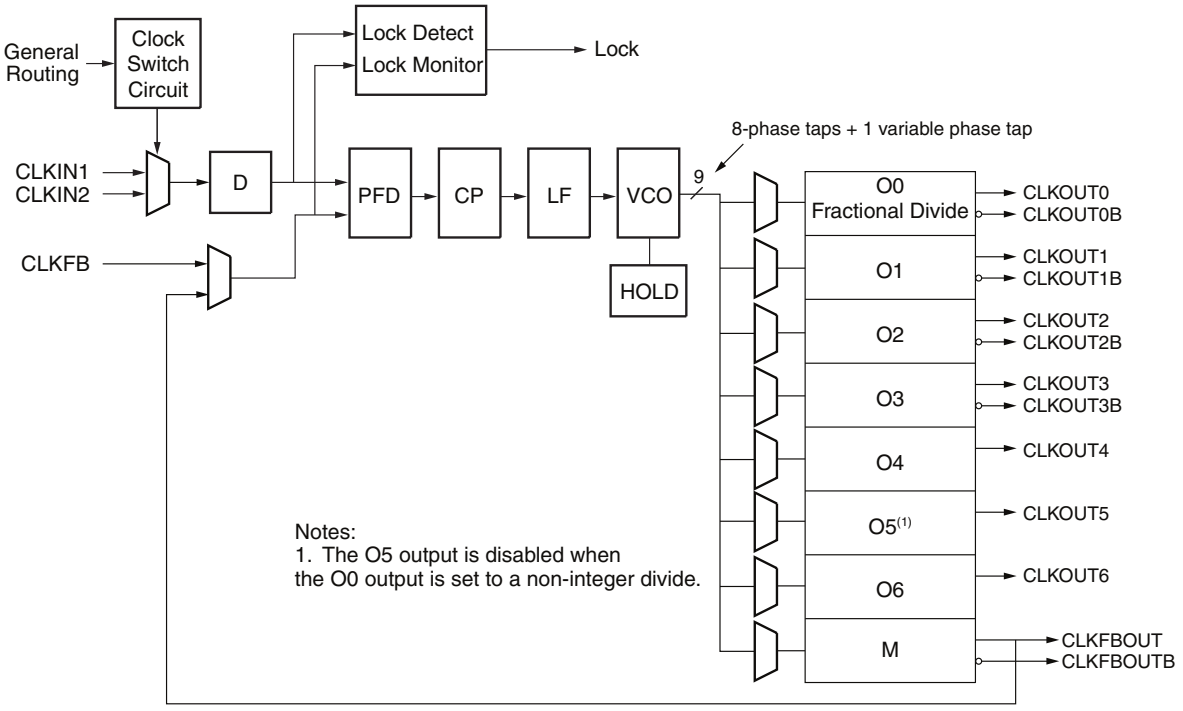


Figure 4.9: Block Diagram of Xilinx MMCM [Xil14]

In the used case the CLKIN1 input is connected to the boards 100 MHz oscillator, the CLKIN2 keeps open. To get the different clock speeds the incoming clock needs to be divided or multiplied or a combination of both to get the required output clock speed. To get the 125 MHz and 200 MHz out of the input clock a frequency multiplication of 10 and a division by 8 or 5 for the faster clock is possible. The division is done by setting the output counter in O0 to 8 and the O1 to 5. If the input clock of the output counters is 1000 MHz the O0 will output 125 MHz and the O1 200 MHz.

The required 1000 MHz is generated by a Phase Locked Loop (PLL). A PLL corrects the phase and frequency of two input signals. If the feedback loop of the PLL is divided by 10 the output frequency will be corrected to the same speed and phase like the 100 MHz input, the result is a frequency multiplied by 10. In the MMCM the M divider is used to do this division an use the CLKFBOUT as feed back signal.

The PLL is implemented with a Voltage Controlled Oscillator (VCO) which generates a frequency at the output depending of the input voltage. This input voltage is determined by a Phase Frequency Detector (PFD). Which identifies the phase shift of two input frequencies. With this phase shift information a Charge Pump (CP) is controlled to generate the input

voltage for a VCO. The Low-pass Filter (LF) is used to keep the change of the input in a certain range and ensure the stability of the PLL. More informations about stability can be found in H. Nyquist regeneration theory [Nyq32]. For further informations about PLL see [Ban06].

The Lock Detect/Lock Monitor is used to verify if the generated clock fulfill the defined requirements. While this Lock output is low the output frequencies of the MMCM not fulfill the requirements and the hole verification architecture keeps in reset state.

5 Software Implementation

The Hardware Verification Architecture explained in the previous chapter needs to report the error to a computer for further analysis. The software part is explained in this chapter. All software is written in Python. It consists of a CCL implementation which is explained in the following section. In the Section 5.1.1 an interface to Modelsim, a VHDL simulation software, is explained. This interface is used to compare the output of the VHDL CCL implementations with the Python implementation for some test images, more in Section 5.1.2. This is helpful to check for already known errors before a time consuming synthesis of the hardware architecture is started.

In Section 5.2 the software to control the verification architecture with an introduction to Ethernet is presented. This software consist of two parts one is for writing error log files (Section 5.2.4). The other visualizes the logged errors (Section 5.2.6). The splitting of this software is done to get a light wight console tool for data logging.

5.1 Python CCL Implementation

The implementation of the CCL is also done as a two pass algorithm. It would be for sure also possible to use a library which does a bounding box calculation. There is for example a Python binding to the OpenCV library, a simple-to-use computer vision infrastructure [BK08] which can provide the bounding boxes around a connected area. The benefit of doing the implementation manual is to get the labels before and after the first-pass of the CCL. This information is used for testing the different VHDL REF components (Section 5.1.1).

The implementation uses the NumPy library to store the image as an matrix and manipulate it. NumPy, part of the SciPy package, is a library which uses bindings to well tested and optimized C and Fortran numeric libraries, more information about this library can be found in [Oli06]. The CCL implementation reads in an image from a file using Python Imaging Library (PIL)[1] and converts the image to binary with a given threshold. In the next step it executes the first labeling pass as explained in Section 2.2.1 and stores the output labels in a matrix. Followed

[1] http://www.pythonware.com/products/pil/

by the lookup table generation with the stored equivalence array of the first pass. Finally the second run only replaces the stored labels of the first run with the values of the lookup table.

To visualize the results of the CCL run Matplotlib, a Python package (part of SciPy) for 2D plotting that generates production quality graphs [Tos09], is used. This library gives the ability to integrate the plots directly into different GUIs, supported frameworks are GTK+, Qt4, and wxWidgets. In Figure 5.2 a screenshot of the plot is given. The plot on the left side shows the first run with its corresponding labels. On the right plot the labels of the second run are written to the image. The two different boxes are marked with different colors and patterns.

5.1.1 Interface to Modelsim Simulator

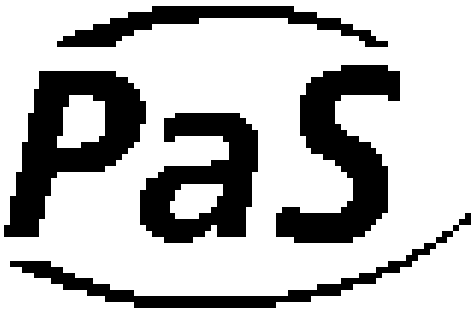
The information from the Python CCL implementation is now used to check if the VHDL implementations get the same results for the same image. There are two problems to solve. It has to be possible, to get the test image into a VHDL simulation and get the results of the simulation back to Python. The output of the data is achieved by just printing them in the simulation to the console. Therefore a testbench is used which instantiates the Module Under Test (MUT) and writes the output data of the module to console. This output data can be read by the Python program.

The data input is a little bit more difficult. To get data into the VHDL testbench the indirection over a file is used. This gives the additional advantage, that a written file can be read again if the developer wants to inspect the simulation with the error image in detail. If the data are passed directly to the input console of the testbench the Python program would be necessary for every simulation run.

To read the input image file a library for parsing Portable Bit Map (PBM) files has been implemented in VHDL. PBM is used by the Netpbm[2] project as an exchange format between filters. It supports two modes a RAW and ASCII mode which can also be modified in a text editor [BB05].

An example image with its ASCII coded PBM file can be found in Figure 5.1. In the first line of the Code the “P1” is the magic number to identify the file as a PBM ASCII coded image. For the binary coded version the magic number is “P4”. The next line is a comment, it is initialized by a “#” they can only appear before the image data starts. The two numbers in the third line specifies the image dimension. They are separated by a whitespace. After the next newline the data part of the image starts. Every pixel is coded as one ASCII coded number. It is as intuitive as possible a zero is white and one is a black pixel. Whitespace characters between the pixels are ignored. In the RAW mode every pixel has one bit, eight of them are packed as one byte.

[2] http://netpbm.sourceforge.net/



(a) A PBM image

```
P1
# CREATOR: GIMP PNM Filter Version 1.1
40 27
0000000000000111111111110000000000000000
0000000000111111111111111110000000000000
0000000111110000000000000001110000000000
0000011111000000000000000000001100000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0001111111000000000000000011111110000000
0001111111100000000000000111111110000000
0011110011110000000000000111100000000000
0011100001110000000000000111000000000000
0011100001110111111111000011100000000000
0011100011100111111111000011110000000000
0011100111100000000110000111111000000000
0111111111000000000110000001111100000000
0111111110000011111110000000111110000000
0111000000001111111110000000011110000000
0111000000001110001110000000011110000000
0111000000001100001110000000011100000000
0110000000001100011110011111111100000000
0110000000001111111100011111111000000010
0000000000000111000000001111110000001100
0000000000000000000000000000000000010000
0111000000000000000000000000000011100000
0011110000000000000000000000001100000000
0000011110000000000000000000111000000000
0000000011111111111111111111110000000000
0000000000001111111111110000000000000000
```

(b) ASCII coded PBM image file

Figure 5.1: A PBM image with its ASCII coded file

The last byte is padded with zeros or ones. The VHDL library supports both modes the ASCII and RAW mode. The python PIL library can read and write these PBM images.

5.1.2 Testing of VHDL Implementation against Software

With the help of the image reader library, three testbenches which outputs the labels after the first pass, another after the second pass, and a third one output the bounding boxes of the **CCL-REF** has been implemented. For the **CCL-DUT** also a testbench which output the bounding boxes has been written.

The testbenches with the ability to read an image file has been combined with the python CCL to, what we call, a verifier program. It parses the output of the Modelsim simulation and compares it with the python results. Two different verification modes are implemented. One does a simulation for a specific file and shows the result as a plot (see Figure 5.2). On the left side the Pyhton result and on the right side the VHDL simulation result. The second mode checks all images in a directory automatically and reports at the end which images leads to errors and how many errors occurred. This mode is helpful when checking all already known errors after changing something in the VHDL implementation.

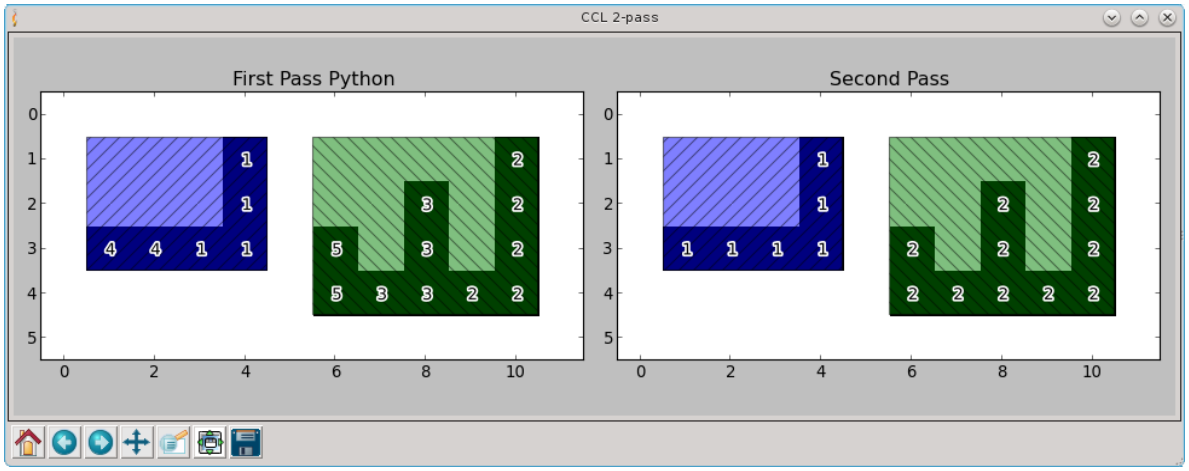


Figure 5.2: Screenshot of the Python two-pass run visualization

5.2 Hardware Data Analysis

In this section the data analysis of the hardware is described. First the communication interface with a necessary introduction to Ethernet followed by the communication protocol is given. In Section 5.2.3 the software to control the verification hardware and for analyzing the found errors is described.

5.2.1 Communication

Ethernet

Ethernet is a method for communication with other devices over a defined hardware and software. It is standardized by the Institute of Electrical and Electronics Engineers (IEEE) in [IEE02]. In this standard different physical transport layers like transmitting data over twisted pair copper or optical fibers and different transfer speeds 10/100/1000/10000 Mbit/s are defined.

Every client has its own 48 bit address to send data to one specific receiver. This address is called MAC-Address. On the software side the IEEE-802.3 specifies how the data format transmitted over the hardware has to look like. The basic MAC frame is depicted in Figure 5.3. It starts with a preamble of seven 0x55 followed by the Start Frame Delimiter (SFD).

Next, the Ethernet-Frame starts with the Destination MAC-Address. If the first byte of this address is odd, the packet is sent as a broadcast message to all clients. The second last bit of the

55	55	55	55	55	55	55	D5	00	AF	FE	EA	BE	EF	00	AD	D0	CA	FF	E0	08	00	Data 0-1500 Byte		00	00	00	00
Preamble							SFD	Destination MAC-Address						Source MAC-Address						Type			PAD	CRC-Checksum			

Figure 5.3: Example of a minimum Ethernet-Frame

MSB is to select between global and local addresses. A global address is unique and assigned by the manufacturer of a device. The local address can be assigned freely and requires no registration. Then the Source MAC-Address is sent followed by the Type-Field. In the example frame it is 0x8000 this stands for IP version 4. Now the data part with a length between 0 and 1500 bytes are followed. The Padding Bytes (PAD) are used to bring the Ethernet-Frame to its minimal size of 64 bytes; counting starts at the Destination MAC-Address. The last four bytes are the Frame Check Sequence (FCS) it is a 32-bit Cyclic Redudancy Check (CRC)-Checksum over the complete Ethernet-Frame starting with the Destination MAC-Address. If the FCS is zero it is ignored.

Internet Protocol

The Internet Protocol (IP) or in particular IPv4 is specified in [Pos81]. It is the base for the Internet communication and one of the most widely used Ethernet-Protocol. A IP packet starts with the version followed by the Internet Header Length (IHL). It is necessary since there are different optional fields in the header. The next 8 bits are the Type of Service it is used to optimize routing and give some packages a higher transport priority than other. Next 16 bits are used to transmit the total length of packet including the header length. The next fields are Identification, Flags, and Fragment Offset are used to restore the order of fragmented packages. Time to Live (TTL) limits the maximum number of hops a package can go. Every time a package passes one receiver which is not the final destination the TTL is decremented. If it is zero the package is dropped.

The protocol type is specified by the next 8 bits for example 0x06 is TCP, and 0x11 is UDP. Followed by a 16 bit checksum over the header. The last two values are the Source IP-Address and the Destination Address, both are 32 bit long.

The addresses is normally written in 4 groups of decimal numbers (e.g. 192.168.0.1). To divide all the possible addresses into different networks a mask is also assigned to each client. This means the client only accepts incoming connections from source addresses of the same masked network. If the sender has for example the address 192.168.0.1 and the receiver has 192.168.0.5 and the mask is 255.255.255.0; the masked addresses of both are 192.168.0.0 and the packet is accepted. If the sender has the address 192.168.1.1 the masked address of

the sender and receiver are different and the packet is dropped. Instead of writing the mask as 255.255.255.0 it is common to write only /24; means 24 ones.

There are different kinds of addresses the private addresses are 10.0.0.0/8, 192.168.0.0/16, and 172.16.0.0/12. This addresses can be chosen freely but they are not routed through the Internet. Addresses for the Internet are distributed by the Internet Corporation for Assigned Names and Numbers (ICANN) to local distribution organizations. The Réseaux IP Européens Network Coordination Centre (RIPE) announced [RIPE] in the end of 2012 they began to allocate the last 2^{24} addresses for Europe, the Middle East and parts of Central Asia.

In 1998 IPv6 was standardized [DH98] to resolve the issue of outrunning addresses. The major introduction are addresses with a size of 128 bits. But the details are not further discussed here. It is not used in this work.

Transmission Control Protocol

TCP is first described in [CDS74]. It is a reliable communication protocol. It can recover from lost, damaged, duplicated, or data delivered out of order. A TCP packet starts with a source and destination port. Next to that, a 32 bit sequence number is added to check the data order. Followed by the Acknowledgment (ACK) number. It is only used if it is a ACK package. Then a 4 bit data offset value is next. The value is the size of the header in 32 bit words. After 3 reserved bits another 9 bits are used as different Flags. The next 16 bits give the receive window seize; the number of bytes which can be received. The second last header information is a 16 bit checksum over the header and the data. Before the data part starts a 16 bit urgent pointer is transmitted. Followed by the data.

A new TCP communication needs to negotiated with the receiver side. All send packets needs to be acknowledged by the receiver. Otherwise the data are resend or other recovery strategies are done. For this it is necessary to buffer the already send data for some time.

User Datagram Protocol

User Datagram Protocol is a state less protocol described in [Pos80]. It is much simpler than TCP to implement. There is no flow control, no recovery of any data, or resend of data. The header has only a source and destination port. Like TCP this are 16 bit numbers. Followed by the size of the data and header as a 16 bit value. A 16 bit checksum over the header and data is optional - disabled if it is set to zero. As last part are the data transmitted.

User Datagram Protocol is used for low delay data transfer, e.g., live video, audio, or telephony streams.

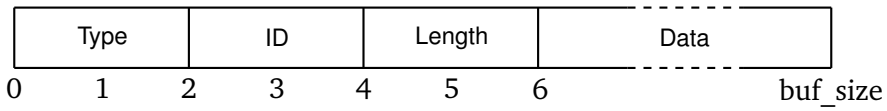


Figure 5.4: Verification Architecture UDP-Package

Type	Message Name
1	ACK
2	Not Acknowledgment (NACK)
3	Hardware Configuration
4	Current State
5	Start Stimulus
6	End Stimulus
7	Restart
8	Error Counter
9	Errors Stored
10	Errors Dropped
11	Read Next Error
12	Error Received

Table 5.1: Message types of Verification Architecture

5.2.2 Architecture Communication Protocol

For the communication with the PC UDP/IP over Ethernet is used. As mentioned earlier the UDP protocol does not grantees that a package is transmitted correctly or even arrives its destination. The used protocol was designed to prevent the lose of informations while using UDP.

The package structure can be found in Figure 5.4. There is type field with a size of two bytes followed by a two byte ID and two bytes which specifies how many of the following data bytes are valid. The Data part is optional and the maximum length of the data field is the size of the UDP buffer on the FPGA (buf_size) minus 6 header bytes.

In Table 5.1 all defined message types are listed. The “ACK” and “NACK” is used to confirm or deny a successful received and processed message. The “Hardware Configuration” message is used to requests the defined generics of the FPGA. An example for such a request and response is given in Figure 5.5 The structure of the data part is given in Table 5.2. The ID is used to identify to which request a response corresponds.

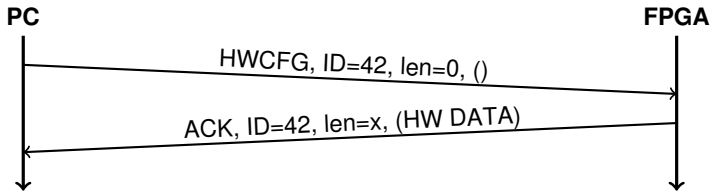


Figure 5.5: Hardware configuration request between FPGA and PC

Data	Size
Hardware-Version	4 Byte
MAX_DATA_LENGTH	2 Byte
len(MAX_IMG_WIDTH)	1 Byte
MAX_IMG_WIDTH	len(MAX_IMG_WIDTH) Byte
IMG_WIDTH	len(MAX_IMG_WIDTH) Byte
len(MAX_IMG_HEIGHT)	1 Byte
MAX_IMG_HEIGHT	len(MAX_IMG_HEIGHT) Byte
IMG_HEIGHT	len(MAX_IMG_HEIGHT) Byte
len(BOX_SIZE)	1 Byte
BOX_SIZE	len(BOX_SIZE) Byte
len(STIMULUS_SIZE)	1 Byte
STIMULUS_SIZE	len(STIMULUS_SIZE) Byte
len(INSTANCES)	1 Byte
INSTANCES	len(INSTANCES) Byte
CMP_ERR_TYPE_LENGTH	1 Byte
REF_ERR_TYPE_LENGTH	1 Byte
DUT_ERR_TYPE_LENGTH	1 Byte

Table 5.2: Data part of the response to “Hardware Configuration” request

To get the current state of the hardware the “Current State” message is used the response is the internal state of a run and the current value of the stimulus generator (Table 5.3).

A “Restart” command will start or restart a test run. As described in Section 4.4.1 the start and end stimulus register is used to set the stimulus generator. This values can be written with the “Start Stimulus” and “End Stimulus” message.

The error count register can be read with the “Error Counter” message. The fill level of the error FIFO can be read with "Errors Stored". There are two ways on reacting to a full error FIFO. First, new values can just be dropped. The number of this dropped errors can be read

Data	Size
Status	1 Byte
CURRENT_STIMULUS	len(STIMULUS_SIZE) Byte

Table 5.3: Data part of the response to “Status Request”

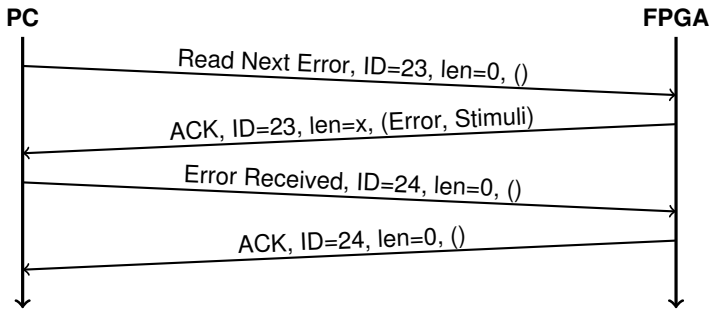


Figure 5.6: Messages to read error from FPGA

with a "Errors Dropped" message. The Second option is to set the verification architecture, with a generic, to stall in case of a full FIFO.

All messages can be lost while they are transfered to the FPGA or the ACK on the way back. In such a case the PC just resend the message and will get the data or write the data again. This is not an issue for the most commands but what happens if a “Read Next Error” message got lost? If the architecture drops the error after sending it over Ethernet this stimulus with an error will be lost. For this reason we use an additional message to tell the hardware architecture the sent error is received and can be dropped. This is done with the “Error Received” message. The messages to read one error is depicted in Figure 5.6.

There are now four messages in total which can possibly be lost. If the first “Read Next Error” or its corresponding ACK message is lost it is not a problem to send it again. If the ACK response to the “Error Received” is missed it is unknown whether the error is already dropped, depending on which message is lost. In this case the “Read Next Error” is sent again and checks if a new error is received or a retransmission of the last one has happened. Now the “Error Received” can be sent again. To check the correctness of the received error and stimuli it is possible to transmit it with the “Error Received” back to the FPGA which has to send a NACK if they do not match.

5.2.3 Verification Control Software

To run a test and get all the reported errors a software that reads the errors from the FPGA and evaluates it is required. This software is split into two parts one only communicates with the FPGA and writes the errors to the hard disk; described in the following section. The other software, we call Log File Analyzer reads the logged errors and gives the user a graphical representation of the files for evaluation. The Log File Inspector is explained in Section 5.2.6.

Splitting the software to two parts ha the advantage to run the data logger without a Graphical User Interface (GUI) on a different computer. With the usage of a terminal multiplexer, like screen[3] or tmux[4], a test run can be supervised remotely which is particularly helpful in long batch runs.

Data Logger

A class overview of the data logger can be found in the Unified Modeling Language (UML) class diagram Figure 5.7. For clarity reasons not all methods and variables of the classes are listed in the diagram. The user interacts with the FileLogger class and can run a test with a specified start and end stimulus or just print the configuration parameters of the FPGA. The software is written in Python 3. One advantage of Python 3 is the integer type is unbounded in the size [PW09]. This is particularly helpful to store a stimulus without caring if it fits into the type, since the type is always adjustable to the required size.

[3] https://www.gnu.org/software/screen/
[4] http://tmux.sourceforge.net/

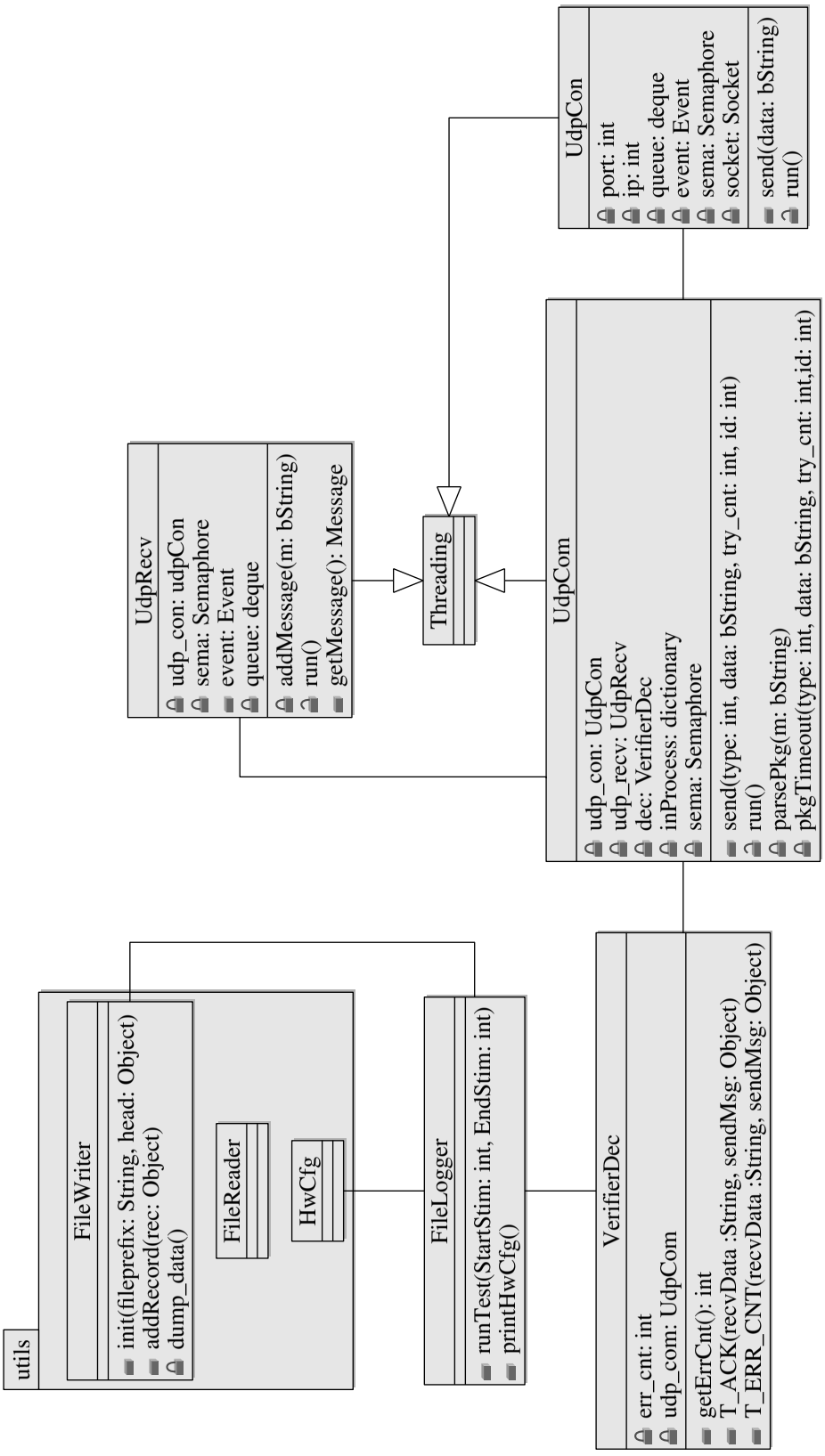


Figure 5.7: UML class diagram of the Data Logger

When the user calls the runTest method of the FileLogger the hardware configuration is fetched from the FPGA. This is required to know the size of all the data types like stimulus length, size of the error vector, ... and parse them correctly. Then the start and end stimulus for the test is written to the FPGA and the restart signal is send.

As soon as the status of the hardware goes to running, the start time is stored to write every minute a status of the current progress and the expected time when the test will be done. The main loop of the program checks for found errors and reads them. The errors are written to a file by the FileWriter.

The FileWriter writes to a file with a specified file name prefix like “file_2014_02_14-12_01”. It also adds header informations such as hardware configuration to the start of every file. To prevent the lose of data after a specified number of write operations a new file is written. To find the associated files of one run, the file is always named with its prefix followed by a incrementing number. The data are written as JavaScript Object Notation (JSON) to the file. JSON is an open standard and gives the ability to read this files with nearly every programming language in an easy way. In addition the file are also human-readable.

When the hardware state switches to idle and all errors are read the FileLogger writes an info file with the number of errors, number of dropped errors the start and end time of the run as well as the start and end stimulus.

The communication consists of multiple classes the VerifierDec is the class the FileLogger interacts with. The UdpCom uses the UdpCon, which is a generalization of the Threading class, to open an UDP socket for sending data. To prevent problems when more threads want to access the send queue at the same time it is protected by a semaphore. To prevent locking events are used to trigger the start of data transmissions. With this implementation the sending of messages will block the main process as short as possible.

The receiving of incoming messages is handled by the UdpRecv class. Here is also threading and events are used to prevent blocking. To prevent a corruption of the receive queue a semaphore handles the access.

The threaded run method of UdpCom waits for an event of the receive queue and decodes the data, id and message type. Additionally this thread checks if an answer to a former send message has already arrived. If a given timeout has exceeded the message is resent. If the resend fails for a given number of times, an exception is thrown. To know which message is send and not answered the sent method writes all messages to an dictionary, something similar to a hash table, as index the message id is used. With this information the received ACK or NACK can be matched to the original send message and type. This is used to directly call after a message is received the processing method of the VerifierDec. For example if an ACK message of an ERRCNT message is received the T_ERR_CNT method is called and the data part of the ACK and the original send message is passed. The access of the dictionary

also requires a semaphore to be thread save. Here is also minimal blocking time of the main process in focus of the design.

5.2.4 Presentation of Error Logs

The logged error data are now available as multiple files with stimuli and its corresponding error. With this kind of data, finding the reason of occurred error is still not so trivial. To support the developer a GUI has been implemented.

The GUI uses the Qt 4 framework. This gives the ability to run the software on any supported platform and run it without changes [Sum07]. Supported platforms are for example Windows, Linux, Mac OS X, and systems using X11 or Wayland.

The software consists of three main parts. The first part is for getting an overview of the logged test runs with some basic information, like how many errors are found, the number of recorded errors, runtime, . . . , this part is described in the next section.

The different runs can be analyzed in a more detailed way either with the Error Inspector, to get detailed informations about every logged error, or with the error HeatMap which gives a more global look over all errors occurred over a complete run. Both described in the following sections in detail.

An overview of the internal software structure can be found in the class diagram in Figure 5.8. Again, for simplicity many details are omitted from the diagram. Some of the Qt inheritance classes are duplicated to reduce the interconnections between the classes.

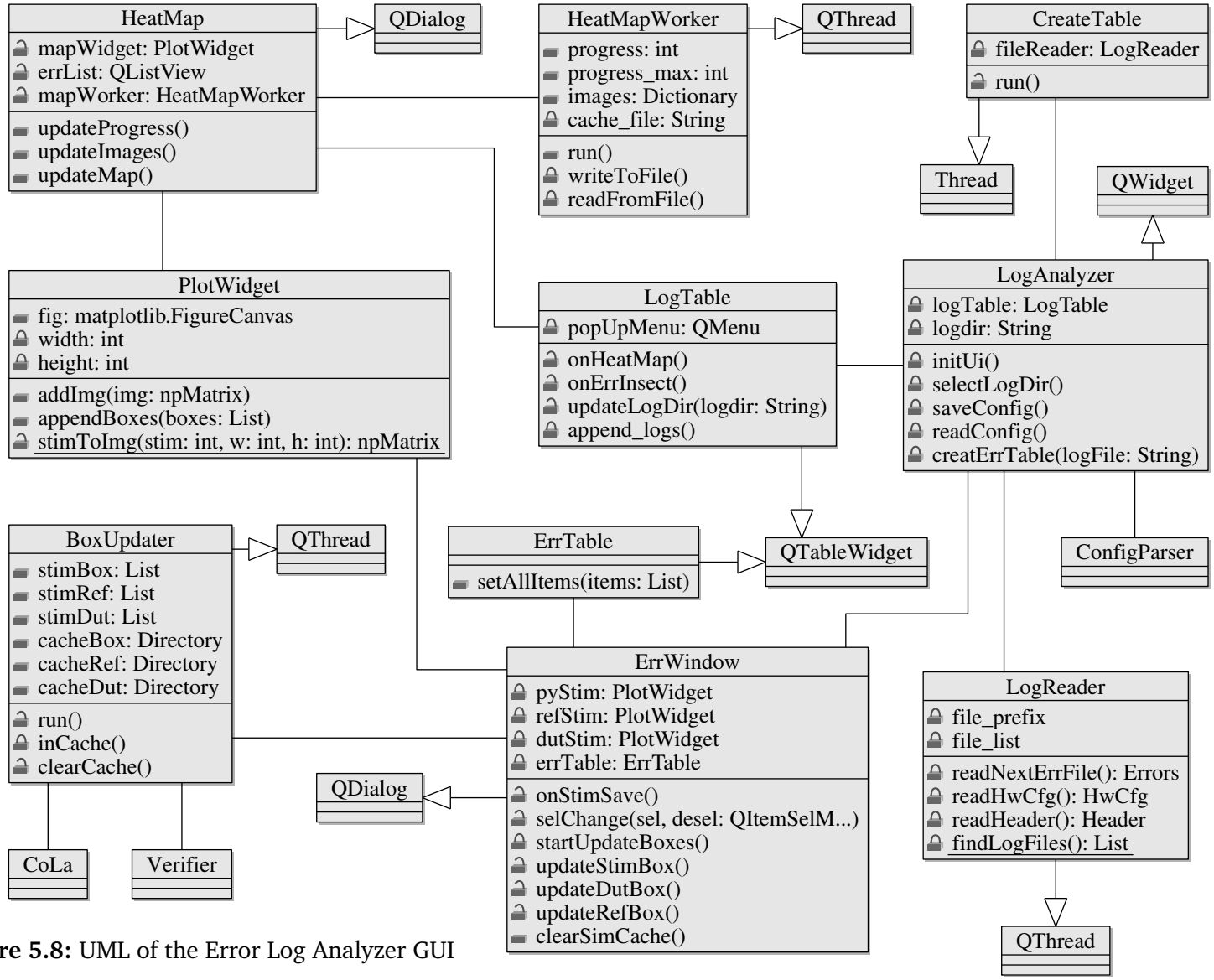
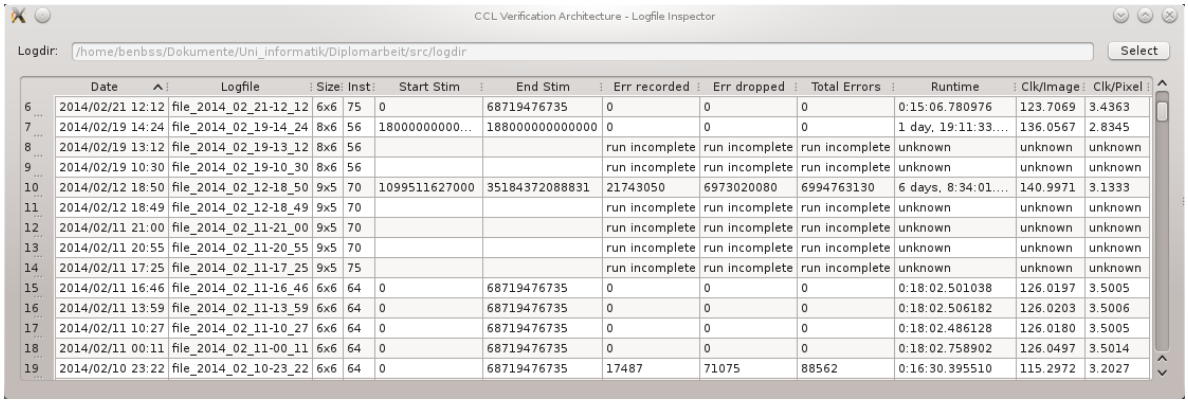


Figure 5.8: UML of the Error Log Analyzer GUI



	Date	Logfile	Size	Inst	Start Stim	End Stim	Err recorded	Err dropped	Total Errors	Runtime	Clk/Image	Clk/Pixel
6	2014/02/21 12:12	file_2014_02_21-12_12	6x6	75	0	68719476735	0	0	0	0:15:06.780976	123.7069	3.4363
7	2014/02/19 14:24	file_2014_02_19-14_24	8x6	56	18000000000...	188000000000000	0	0	0	1 day, 19:11:33....	136.0567	2.8345
8	2014/02/19 13:12	file_2014_02_19-13_12	8x6	56			run incomplete	run incomplete	run incomplete	unknown	unknown	unknown
9	2014/02/19 10:30	file_2014_02_19-10_30	8x6	56			run incomplete	run incomplete	run incomplete	unknown	unknown	unknown
10	2014/02/12 18:50	file_2014_02_12-18_50	9x5	70	1099511627000	35184372088831	21743050	6973020080	6994763130	6 days, 8:34:01....	140.9971	3.1333
11	2014/02/12 18:49	file_2014_02_12-18_49	9x5	70			run incomplete	run incomplete	run incomplete	unknown	unknown	unknown
12	2014/02/11 21:00	file_2014_02_11-21_00	9x5	70			run incomplete	run incomplete	run incomplete	unknown	unknown	unknown
13	2014/02/11 20:55	file_2014_02_11-20_55	9x5	70			run incomplete	run incomplete	run incomplete	unknown	unknown	unknown
14	2014/02/11 17:25	file_2014_02_11-17_25	9x5	75			run incomplete	run incomplete	run incomplete	unknown	unknown	unknown
15	2014/02/11 16:46	file_2014_02_11-16_46	6x6	64	0	68719476735	0	0	0	0:18:02.501038	126.0197	3.5005
16	2014/02/11 13:59	file_2014_02_11-13_59	6x6	64	0	68719476735	0	0	0	0:18:02.506182	126.0203	3.5006
17	2014/02/11 10:27	file_2014_02_11-10_27	6x6	64	0	68719476735	0	0	0	0:18:02.486128	126.0180	3.5005
18	2014/02/11 00:11	file_2014_02_11-00_11	6x6	64	0	68719476735	0	0	0	0:18:02.758902	126.0497	3.5014
19	2014/02/10 23:22	file_2014_02_10-23_22	6x6	64	0	68719476735	17487	71075	88562	0:16:30.395510	115.2972	3.2027

Figure 5.9: Screenshot of the Log File Inspector

5.2.5 Log File Inspector

The program starts with an instantiation of the LogAnalyzer class. A screenshot of the GUI can be found in Figure 5.9. It lists all test runs found in the log directory, specified in the top text field. The data in the list are the date of the run, log file prefix, configured image size, instances of the verifier, start and end stimulus, the number of recorded, total, and dropped errors, runtime and clock cycles required to process one image respective pixel of one verification instance in average over the complete run.

To give the user a good experience while using the software changed configuration parameters are stored to a configuration file. The interaction with the log files is handled by the LogReader class. The header information of a log file is sent to the LogTable object which is the table in the screenshot.

A run of the list can be selected and a detailed analysis with the Error Inspector or the Error Heat Map can be started.

5.2.6 Error Inspector

In Figure 5.10 a screenshot of the error inspector is presented. On the left is a list with the errors and its stimuli. The error are split in four columns. Each for a different error source, the comparator, DUT and REF as explained in Section 4.1. This is the decoded representation of the error vector in the forth column. The error vector is decoded with the error size information from the hardware verification architecture and the human readable translation in the HwCfg class. The last column has the stimulus in decimal and hexadecimal notation in it.

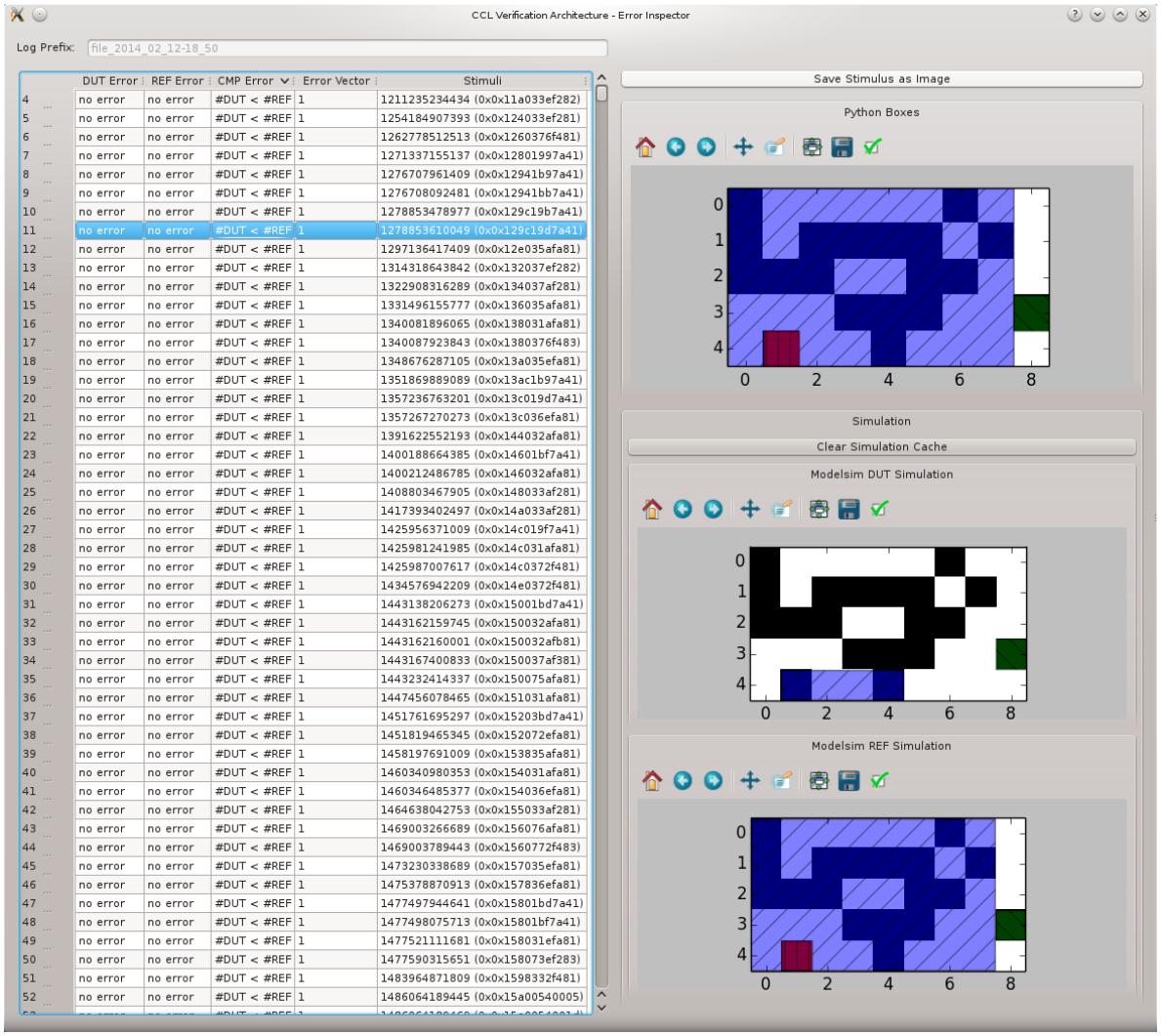


Figure 5.10: Screenshot of the Error Log Analyzer

On the right side of the window three images of the currently selected stimuli can be found. The top image shows the resulting boxes of the Python implementation (presented in Section 5.1). The middle image shows the output boxes of a ModelSim simulation of the DUT VHDL code for the selected stimulus. The last image is the result of the REF VHDL simulation. One could also save images of the currently selected stimulus to a file for further investigation.

In a test run the number of found errors can be high. In such a case the reading of all errors can take very long. To keep the GUI active the processing of the data is done in a different thread. After processing all log files the ErrWindow visualizes the processed data.

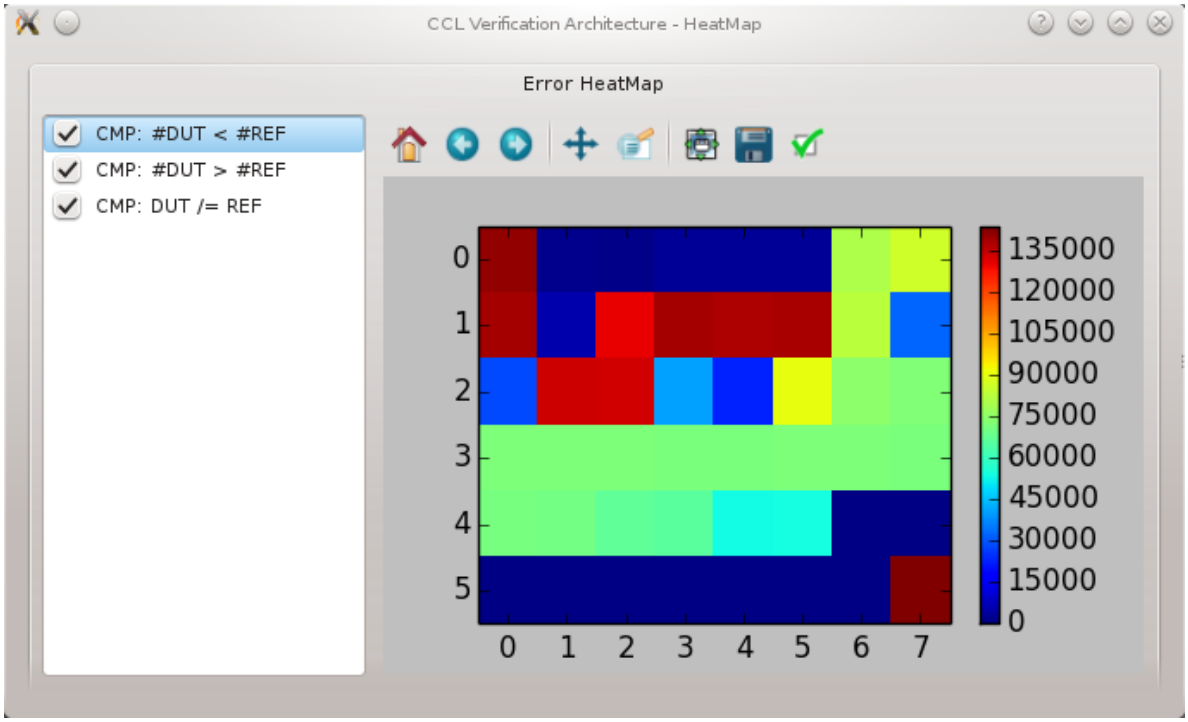


Figure 5.11: Screenshot of the Error Heat Map

The GUI uses threading to run the ModelSim simulation (see Section 5.1.1) to keep very thing reactive. As soon as the results are there the threads uses events to signal the ErrWindow a update is required. The image is created by the PlotWidget which uses the matplotlib library.

To give the user a smooth GUI the results of ModelSim simulations are cached. To clear the cache the “Clear Simulation Cache” button flushes the cache. This is useful if changes on the VHDL code are done and the new resulting boxes should be shown.

Unfortunately it can happen that a test run reports many million errors. This number of errors can be a problem when trying to sort the table. To solve this some pre-filtering can be implemented. At the moment only a given number of errors are shown in the table.

The ModelSim simulation runs with the currently compiled VHDL code. If the code is compiled for a different image dimension than the current analyzed error had the shown data are wrong.

5.2.7 Error Heat Map

Another screenshot in Figure 5.11 shows a HeatMap. The idea of the heat map is to count how often a certain black pixel is involved in an erroneous result. In other words if the stimulus image is represented as a matrix, a one represents a black pixel and zero a white, all stimuli matrices of the run which leaded to an error are summed up. The resulting matrix is visualized as a heat map.

Pixels which are often in an error stimulus lead to hot - red - pixel positions which are never in an error stimulus to cold - blue - pixel. The summation is done for each error type to a different matrix. On the left of the screenshot is a list of the occurred errors. Each entry in this list represents one of these matrices. The matrices of all selected errors are summed up and visualized on the right image. The colorbar on the right gives a mapping between the number of occurred errors and the color.

The HeatMap is implemented by the HeatMap class, another QDialog. It reads in all errors of a verification run a with the help of the LogReader. To prevent the freezing of the GUI the HeatMapWorker does the processing is done by the HeatMapWorker in a different thread. After all data processed the resulting matrices stored to a cache file on the hard disk. If the cache file for a test run already exists the processing is skipped and the cache file is read.

For the visualization of the heat map the PlotWidget class uses matplotlib. Changing the selected error types leads to an update of the heat map which only use the corresponding matrices to generate the heat map.

6 Results

This chapter presents the results of the proposed architecture. In Section 6.1 an analysis of the FPGA usage and the speed of the exhaustive test depending on the image dimension is done. Followed, by the found discrepancies between the DUT and the reference architecture, in Section 6.2.

The synthesis for all test of the architecture was done with Xilinx ISE 13.4. As a timing constraint $125\,\text{MHz}$ was set for the global clock. This clock is required for the Ethernet communication. The synthesis optimization goal was set to speed and the optimization effort to high. The additional keep hierarchy flag was set.

The FPGA used is a Xilinx Virtex-6 LX-240 (xc6vlx240t-ff1759-2).

6.1 FPGA Performance Analysis

This section is divided in two parts, Section 6.1.2 analyzes the utilization of the FPGA depending on the image size and two different comparator implementations. In the last section the runtime of the architecture for different image sizes and again for two different comparators is shown.

6.1.1 FPGA Utilization

To get an overview of the required resources six different image sizes are tested. In Figure 6.1 and Figure 6.2 the complete architecture has been synthesized. The former shows the results for type 2 comparator which does comparing without sorting (see Section 4.3.1), the type 3, does sorting while inserting and the comparison is done in one clock cycle (see Section 4.2.1). The three plotted values for each image size are the LUTs used as logic, as memory and the used flip-flops. The number of used block RAMs is not listed since it is for all tested sizes 5 out of 416. For this small image dimensions the amount of stored data makes no difference, since one block RAM cell can store 1024 words with a width of 36 bits.

Interestingly, an image of 9×5 pixels require more space on the FPGA than 8×6 (Figure 6.2). The reason is that there are 15 different labels in 9×5 while in 8×6 only 12 different different possible labels (see Section 3.1).

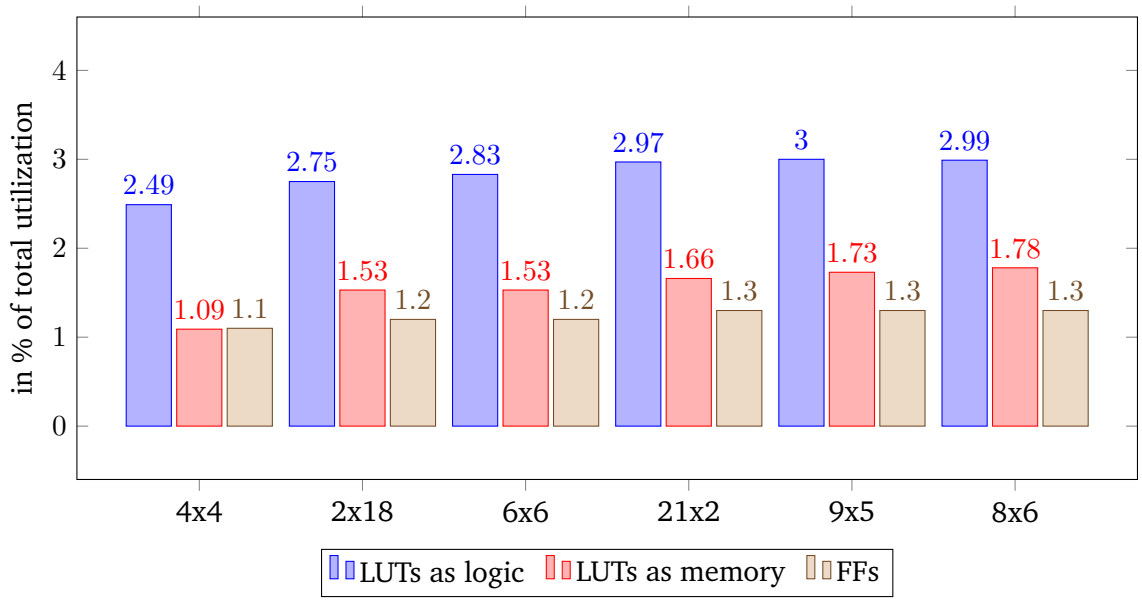


Figure 6.1: FPGA utilization versus image size with one verifier unit (comparator type 2)

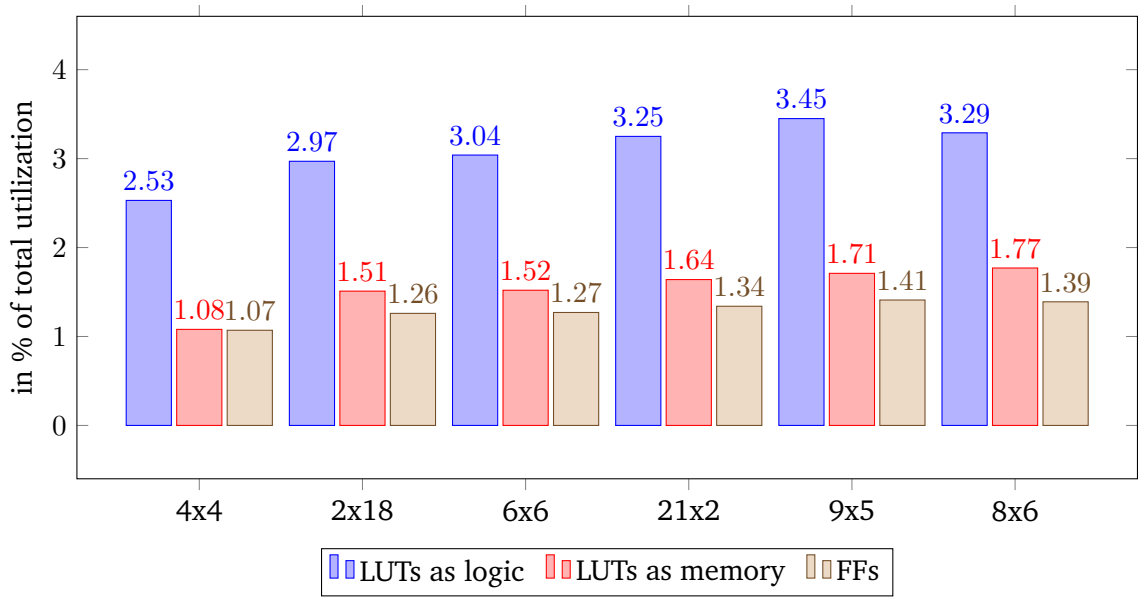


Figure 6.2: FPGA utilization versus image size with one verifier unit (comparator type 3)

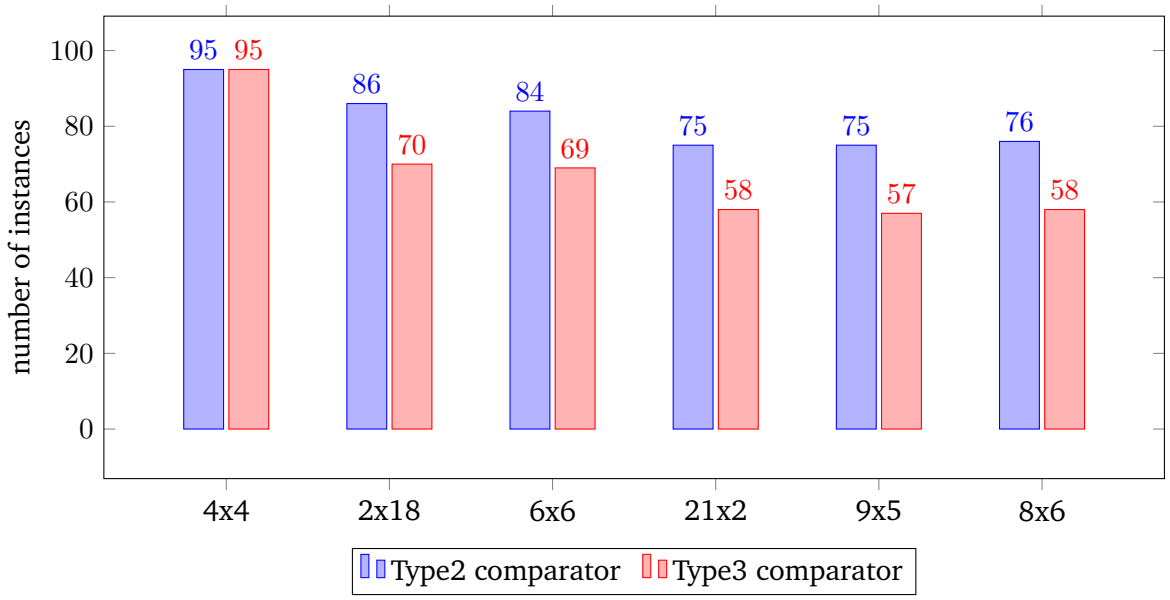


Figure 6.3: Maximum number of parallel verifier units on FPGA

The expected higher utilization of the type 3 comparator can also be seen. The reason behind the higher utilization of type 3 is that every additional label can lead to one more bounding box. For each additional bounding box one more comparator is required as well as the number of values to compare also increases by one. The type 2 comparator on the other hand requires only one comparator no matter how many values to compare.

In the next step the number of parallel instances are raised. Figure 6.3 depicts the maximum number of parallel verifier. This values are determined by many synthesis runs. Started with a first guess for the instances and if it failed another run with lesser instances was done or if it worked the number was raised. This was repeated until the maximum number was determined.

6.1.2 Runtime

In this section different exhaustive test runs are analyzed to determine the speed of the architecture. Table 6.1 gives the runtime for different exhaustive test runs with different images sizes. The introduced parallelization gave the ability to run an exhaustive test with a dimension of 9×5 in lesser than seven days instead of more than a year with only one instance.

In addition in Figure 6.4 a comparison of the hardware verification architecture and the simulation runtime with ModelSim for an exhaustive test is plotted. The values for the software

	Type 2		Type 3	
Image Size	1 instance	max instances	1 instance	max instances
4×4	38 ms	$525\,\mu s$	36 ms	$581\,\mu s$
6×6	18:48 h	00:13 h	18:58 h[a]	00:16 h
2×18	1 d 03:57 h[a]	00:19 h	1 d 02:50 h[a]	00:23 h
21×2	69 d 02:26 h[a]	22:06 h	61 d 22:22 h[a]	1 d 01:37 h
9×5	456 d 17:42 h[a]	6 d 12:35 h	-	-

Table 6.1: Runtime for exhaustive test, different image sizes, different comparators

[a] interpolated value

simulation are interpolated from a test run of a exhaustive test simulation for a 4×4 image, which took 30 minutes. To respect the different complexities of images the now introduced average clocks per pixel are used as a scaling factor.

The efficient comparison between the different image sizes in terms of component utilization of the architecture requires another metric. We use the average time, over a complete exhaustive test, required to process one pixel. This is calculated as follow:

$$\text{average clocks per pixel} = \frac{T_{exhaustive} \times f_{CLK}}{IMG_h \times IMG_w \times 2^{IMG_h \times IMG_w}}$$

This clocks per pixel value gives the ability to compare the efficiency of the architecture between different image dimensions. If the architecture would only do the two pass run without lookup table generation, bounding box calculation and comparison this value would be 2. Additional clock cycles are required to generate the lookup table and the bounding boxes, both of this values depends on the complexity of the image, as mentioned in Chapter 3. Figure 6.5 gives the average clocks per pixel value for different image sizes with one comparator instance.

The average clocks per pixel value is higher for smaller image sizes this is because of the delay for every register stage in the architecture, as lesser the number of pixels per image the higher the impact of this delay to the total runtime is. But this has no significance if the runtime of the exhaustive test runtime is considered. For instance, the 4×4 image takes lesser than one second.

Figure 6.6 depicts the exhaustive runtime with the clocks per pixel value for the maximum possible number of comparator instances. It shows that the type 2 comparator, which is slower for only one verifier instance, can now benefit from the higher parallelization as shown in Figure 6.3. This is because of the lower resource utilization compared to the type 3 comparator (see Section 4.2.2). In this case some of the additional verifier units are not always utilized. The

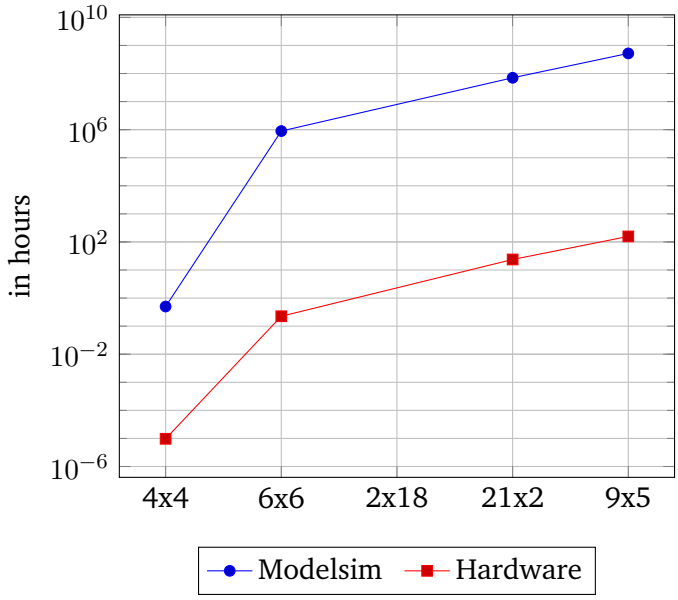


Figure 6.4: Simulation time vs hardware verification runtime

reason behind that is the assignment of the stimuli to the different verifiers and is explained in Section 4.3.1.

6.2 Detected Errors

This section is divided in two subsections the first shows the results of the exhaustive test runs and the second the results of the non exhaustive test runs for bigger image sizes.

6.2.1 Exhaustive Test

The exhaustive test runs has been carried out in two stages. First an image size which could be verified by the architecture in less than a day was selected. We have chosen 6×6 as it can be verified by one verifier instance in about 19 hours. After finding the first few discrepancies the stimulus, which raised the error, has been surveyed in DUT simulation and the bug which leads to this misbehavior was fixed. With this new revision another exhaustive test was started. The whole process has been repeated as long as the architecture detected errors. In Figure 6.7 the number of found errors of the last four DUT revisions can be found. The reason for the increased number of detected errors in the second plotted run is that a new feature was added to the architecture. This feature reports internal problems of the DUT to the

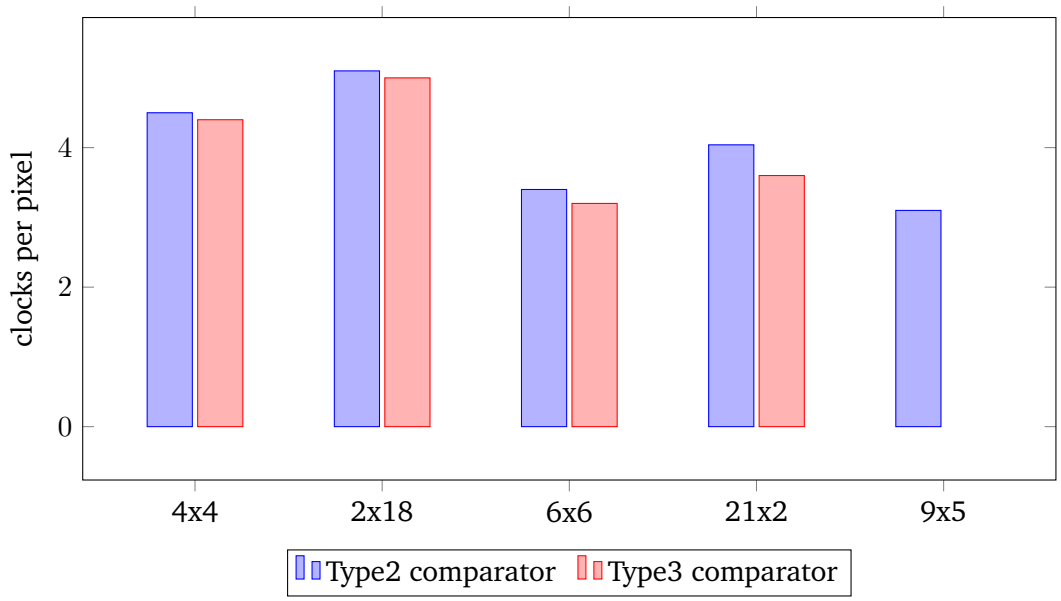


Figure 6.5: Average clock cycles per pixel for exhaustive test

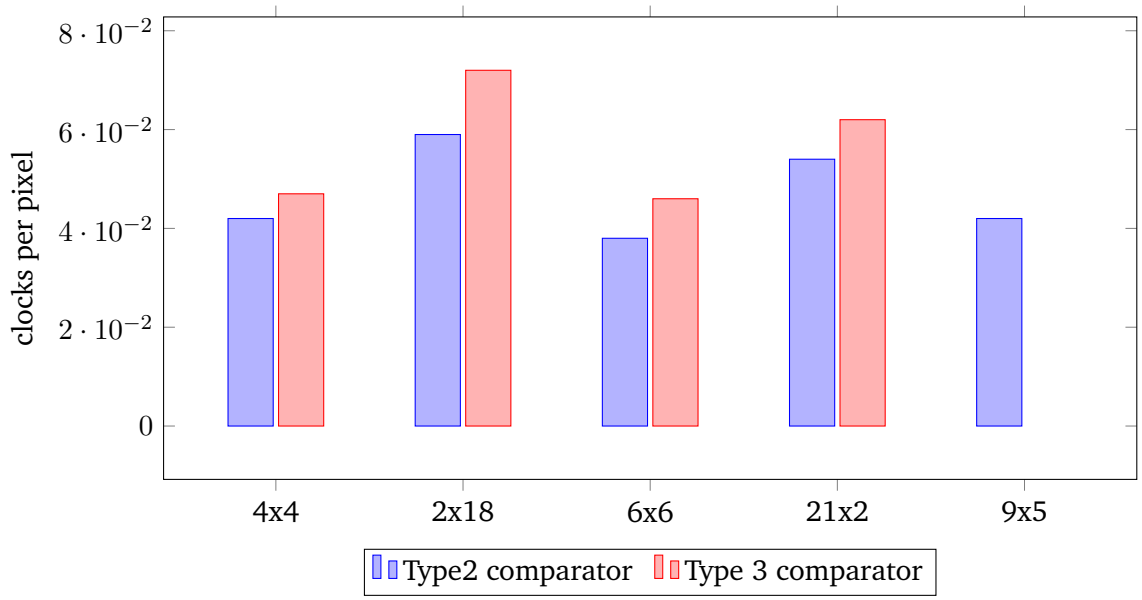


Figure 6.6: Average clock cycles per pixel for exhaustive test, maximum instances

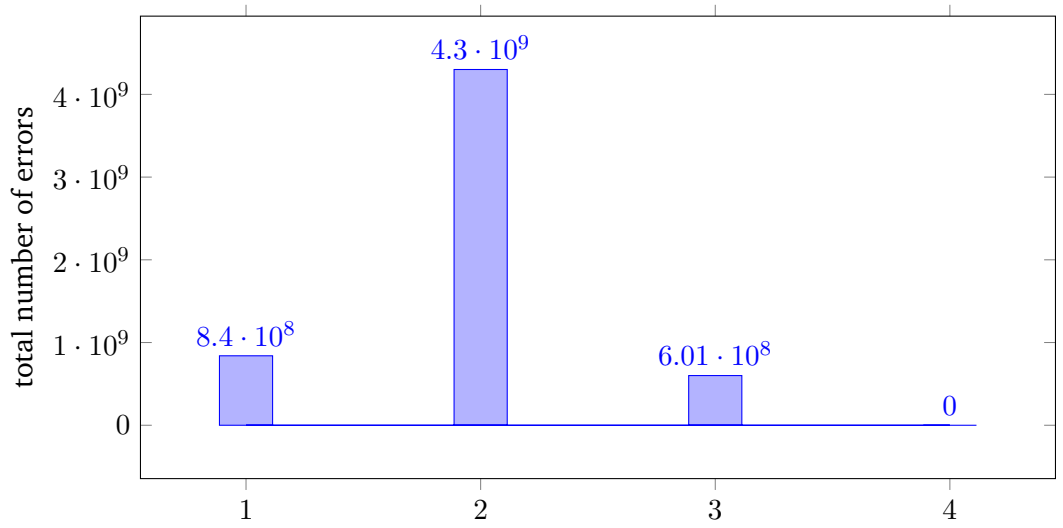


Figure 6.7: Found errors in different DUT revisions with image size 6×6

hardware verification architecture, such as overflows of the label counter. This internal errors not necessarily leads to a wrong output of the DUT. If for example the label counter overflows and no new label is assigned this has no effect on the output.

The refined revision that has passed test run four in Figure 6.7 with 0 error was eventually used to run exhaustive tests with different image sizes. The found errors are plotted as a percent of all possible input stimuli in the diagram in Figure 6.8.

6.2.2 Test of bigger Image Sizes

To test images of a bigger size than that used in the exhaustive test a test pattern generator has been implemented. Which takes from a list of predefined patterns a random number of patterns and places it on an empty image. In addition to this given patterns also random noise has been added. The code for the test pattern generator can be found in Listing A.7. The noise minimum and maximum noise factor for the generator can be configured. For each generated image a value in this range is randomly chosen. The same can be done for the number of test patterns.

The predefined patterns are known problematic structures for the CCL architecture. Figure 6.9 depicts one of the used patterns. This pattern for example leads to a lot of required label merges while processing.

A set of test images with a size of 63×61 has been generated. The images are divided in 10 different classes. Each class with a different range of noise and patterns. For each class 100

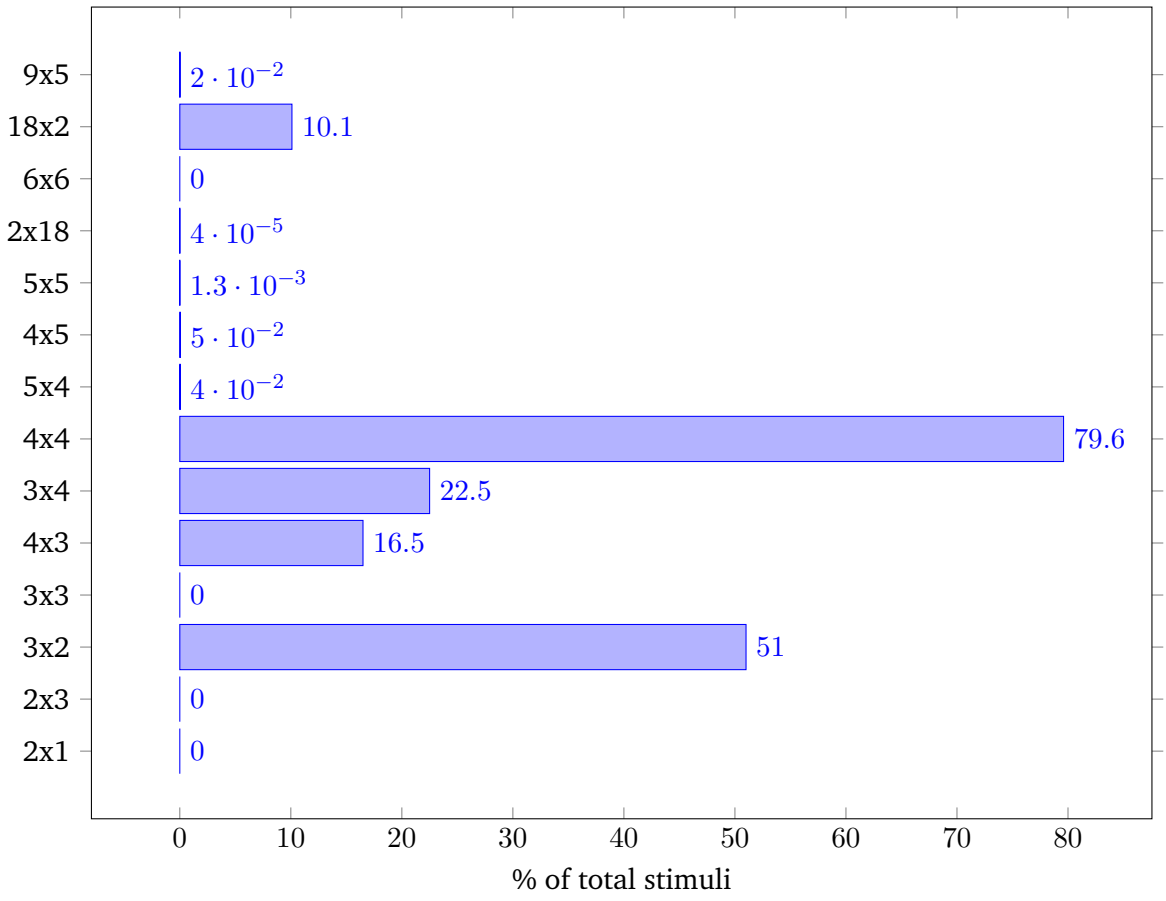


Figure 6.8: Found errors with different image size

images has been generated. In Figure 6.10 one of the generated images can be found. It has has been applied to the DUT in a VHDL simulation. No error was reported.

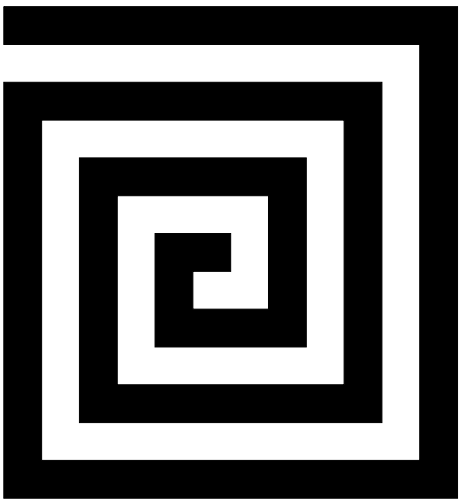


Figure 6.9: Example pattern for test image generator

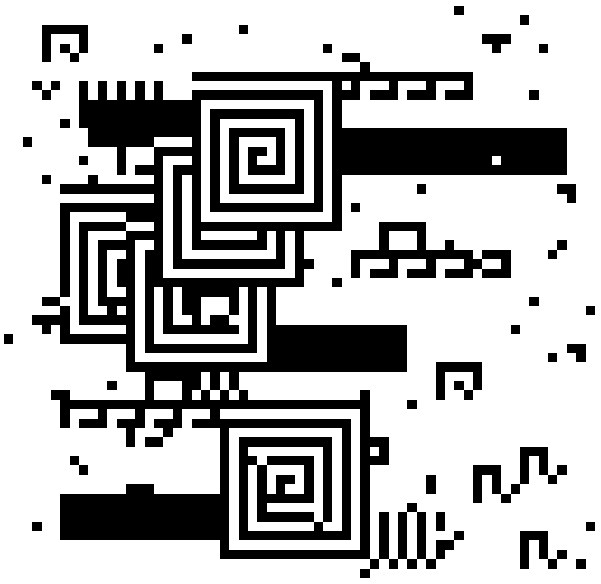


Figure 6.10: Example test image

7 Conclusion

In this work, a hardware architecture for autonomously exhaustive testing has been proposed and implemented. The architecture implements a two pass Connected Component Labeling (CCL) that runs in parallel with a new improved one pass CCL. Discrepancies are reported conveniently to a PC. Since the major concern was correctness, we have kept the design as simple and relinquished any complicated approach that, while marginally enhance the performance, might be difficult and time consuming to verify. For the automatic comparison of the output different approaches have been proposed and evaluated. Two of them have been implemented and tested. Both of them reports different classes of errors, like one CCL output more data than the other, same amount but different data and additional internal errors of the Device Under Test (DUT). Different speed and resource optimizations have been proposed.

For the communication with the PC a UDP/IP over Ethernet interface has been implemented on the Field Programmable Gate Array (FPGA). This made it straight forward to build an intuitive software using standard communication socket libraries. A software to evaluate the test runs of the hardware architecture has also been implemented, which gives the developer the following rich set of features:

- Logging of the found errors in the course of running the exhaustive test. No need to wait idle to the very end to view the errors.
- Different visualizations of the logged error with a GUI, like run configuration (image size, instances) and performance overview (runtime, clock cycles per pixel), error heat map, error inspector which compares the actual (erroneous) and expected output. This improves and speeds up the debugging process.
- An interface to the ModelSim simulator. Here, the developer can launch the simulation with the stimulus that generated the error directly from within the software.
- The possibility for running autonomous testing of the DUT and reference implementation with a given set of images. This is helpful to test images, which had in an earlier exhaustive test, exhibited erroneous results before a possibly time consuming exhaustive run takes place. This feature can also be used to do automated checks after committing a new revision to the development repository.

Exhaustive test runs for different image sizes have been performed and the found errors have been reported. The gain of using the verification architecture was tremendous as the number of errors for 6×6 images dropped from more than 800 million to zero over twelve subsequent revisions. Different image sizes have also been tested, the remaining errors are fixed at the moment.

For a non exhaustive test bigger image sizes than the one used in the exhaustive testing, a test image generator has been implemented. A set of this automatically generated images has been applied to the DUT - no errors were found.

The runtime and utilization of the proposed architecture has been researched and reported for different image sizes. Just as the performance of an additional comparator unit.

Future Work

To improve the verification time of the architecture different scheduling methods, for assigning the stimuli to different units, to achieve a higher utilization of the used components should be further researched. For example completely dynamical scheduling, which can use the additional advantages of the pipelined CCL reference architecture. This should at least reduce the exhaustive test runtime by half.

Another interesting approach to improve the architecture is to use different instantiated CCL reference architectures for different complex images. Each image size has a number of different complex images, which needs more memory or additional logic to be processed correctly. But in the proposed architecture every instance of the CCL can process the worst case image - which leads to a waste of resources. This becomes obvious after checking the number of worst case images. For example an image with the size of 9×5 pixels has only one worst case image with 15 differently labeled areas. One instance with the ability to process this worst case image is enough. It should be no problem for the instances without the ability to process the worst case image to detect when something goes wrong, for example a counter or FIFO overflow can easily trigger a rerun on the instance with the possibility to process all types of image complexities. This can also be done for the second worst image, since there are only a few images of this kind. The reduced instances should lead to more space on the FPGA which can be used for additional verification units.

Higher performance can also be achieved by using different clock domains for the CCL-DUT, reference and the comparator. The architecture developed in this thesis can be used as a basis for the verification of an improved architecture.

A Appendix

Listing A.1 Insert Procedure for Listing A.2

```
1   procedure insert_element(
2     signal el     : in T_BOX;
3     signal list   : inout T_BOX_RAM;
4     signal valids : inout unsigned
5   ) is
6     variable pos  : natural;
7   begin
8     pos := 0;
9     for i in list'range loop
10      if i < valids and list(i) < el then
11        pos := i+1;
12      end if;
13    end loop;
14    if pos = 0 then
15      list <= el & list(pos to list'high-1);
16    elsif pos > list'high then
17      list <= list(0 to pos-2) & el;
18    else
19      list <= list(0 to pos-1) & el & list(pos to list'high-1);
20    end if;
21    valids <= valids + 1;
22  end procedure;
```

Listing A.2 Comparator with Sorting on Insert

```
architecture comparator_arc of comparator is
  TYPE T_BOX_RAM is array (0 to G_MAX_BOXES - 1) of T_BOX;

  CONSTANT C_NO_ERROR    : unsigned(3 downto 0) := "0000";
  CONSTANT C_DUT_LESS    : unsigned(3 downto 0) := "0001";
  CONSTANT C_DUT_MORE    : unsigned(3 downto 0) := "0010";
  CONSTANT C_DUT_WRONG   : unsigned(3 downto 0) := "0011";

  signal box2_ram_s      : T_BOX_RAM;
  signal box2_valid_s    : unsigned(log2_ceil(G_MAX_BOXES+2)-1 downto 0);
  signal box1_ram_s      : T_BOX_RAM;
  signal box1_valid_s    : unsigned(log2_ceil(G_MAX_BOXES+2)-1 downto 0);
  signal last_box_in_d_s : std_logic; -- delayed last_box_in
  -- insert_element procedure in Appendix A.1
    valids <= valids + 1;
  end procedure;
begin
  -- this process does the inserting of new boxes and de comparation
  p_insert : process(clk_in, rst_in) is
  begin
    if rst_in = '1' then
      box1_valid_s    <= (others => '0'); -- reset value
      box2_valid_s    <= (others => '0'); -- reset value
      last_box_in_d_s <= '0';             -- reset value
      check_done_out  <= '0';             -- reset value
    elsif rising_edge(clk_in) then
      check_done_out  <= '0';             -- set default value
      last_box_in_d_s <= last_box_in;     -- delay signal for last_box_in
      if box1_valid_in = '1' then         -- new output1 to compare
        insert_element(box1_in, box1_ram_s, box1_valid_s);
      end if;
      if box2_valid_in = '1' then         -- new output2 to compare
        insert_element(box2_in, box2_ram_s, box2_valid_s);
      end if;
      if last_box_in_d_s = '1' then       -- generate output
        check_done_out <= '1';
        if box1_valid_s < box2_valid_s then
          error_code_out <= C_DUT_LESS;
        elsif box1_valid_s > box2_valid_s then
          error_code_out <= C_DUT_MORE;
        elsif box1_valid_s = 0 and box2_valid_s = 0 then
          error_code_out <= C_NO_ERROR;
        elsif box2_ram_s(0 to to_integer(box2_valid_s)-1) =
        box1_ram_s(0 to to_integer(box1_valid_s)-1) then
          error_code_out <= C_NO_ERROR;
        else
          error_code_out <= C_DUT_WRONG;
        end if;
        box1_valid_s <= (others => '0'); -- auto reset
        box2_valid_s <= (others => '0'); -- auto reset
      end if;
    end if; --clk
  end process p_insert;
end architecture;
```

Listing A.3 IPv4 RX-Header Declaration of the IP-Core

```
type ipv4_rx_type is record
  hdr              : ipv4_rx_header_type; -- received header
  data             : axi_in_type;         -- RX AXI4-Stream
end record;

type ipv4_rx_header_type is record
  is_valid         : std_logic;
  protocol         : std_logic_vector ( 7 downto 0);
  data_length      : std_logic_vector (15 downto 0); -- user data size, bytes
  src_ip_addr      : std_logic_vector (31 downto 0);
  num_frame_errors : std_logic_vector ( 7 downto 0);
  last_error_code  : std_logic_vector ( 3 downto 0);
  is_broadcast     : std_logic; -- set if the msg received is a broadcast
end record;
```

Listing A.4 UDP RX-Header Declaration of the UDP-Core

```
type udp_rx_type is record
  hdr              : udp_rx_header_type; -- received header
  data             : axi_in_type;        -- RX AXI4-Stream
end record;

type udp_rx_header_type is record
  is_valid         : std_logic;
  src_ip_addr      : std_logic_vector (31 downto 0);
  src_port         : std_logic_vector (15 downto 0);
  dst_port         : std_logic_vector (15 downto 0);
  data_length      : std_logic_vector (15 downto 0); -- user data size, bytes
end record;
```

Listing A.5 UDP TX-Header Declaration of the UDP-Core

```
type udp_tx_type is record
  hdr              : udp_tx_header_type; -- header to tx
  data             : axi_out_type;       -- tx axi bus
end record;

type udp_tx_header_type is record
  dst_ip_addr      : std_logic_vector (31 downto 0);
  dst_port         : std_logic_vector (15 downto 0);
  src_port         : std_logic_vector (15 downto 0);
  data_length      : std_logic_vector (15 downto 0); -- user data size, bytes
  checksum         : std_logic_vector (15 downto 0);
end record;
```

Listing A.6 IPv4 TX-Header Declaration of the IP-Core

```
1  type ipv4_tx_type is record
2    hdr               : ipv4_tx_header_type; -- header to tx
3    data              : axi_out_type;        -- tx axi bus
4  end record;
5
6  type ipv4_tx_header_type is record
7    protocol          : std_logic_vector ( 7 downto 0);
8    data_length       : std_logic_vector (15 downto 0); -- user data size, bytes
9    dst_ip_addr       : std_logic_vector (31 downto 0);
10 end record;
```

Listing A.7 Test pattern generator for non exhaustive test, (more in Listing A.8)

```
#!/usr/bin/env python3
import glob
import random
import os
from PIL import Image
from optparse import OptionParser

class TestImage:
    def __init__(self, w, h, test_pattern, min_pattern, max_pattern, min_noise, max_noise):
        self.__w = w
        self.__h = h
        self.__tp = test_pattern
        self.__min_p = min_pattern
        self.__max_p = max_pattern
        self.__min_n = min_noise
        self.__max_n = max_noise
        random.seed()

    def getImg(self):
        img = Image.new('1', (self.__w, self.__h), "white")
        #min/max_noise is a faktor of the image value
        noise_cnt = random.uniform(self.__min_n, self.__max_n)
        noise_cnt = int(round(float(noise_cnt) *(self.__w*self.__h)))
        pattern_cnt = random.randint(self.__min_p, self.__max_p)

        for i in range(0, pattern_cnt):
            #with overlapping patterns
            #select a randome pattern
            sel_pattern = random.randint(0, len(self.__tp)-1)
            pattern = self.__tp[sel_pattern]
            pattern_s = pattern.size
            #generate position for chosen pattern
            x_coord = random.randint(0, self.__w - pattern_s[0]-1)
            y_coord = random.randint(0, self.__h - pattern_s[1]-1)
            img.paste(pattern, (x_coord, y_coord))

        for i in range(0, noise_cnt): #add the noise
            x_coord = random.randint(0, self.__w-1)
            y_coord = random.randint(0, self.__h-1)
            img.putpixel((x_coord, y_coord), 0)
        return img

class TestPattern:
    def __init__(self, pattern_dir):
        self.__pattern_dir = pattern_dir

        self.pattern = []
        self.__pattern_read(self.__pattern_dir)


    def __pattern_read(self, pdir):
        for files in glob.glob(pdir + "/pattern*"):
            self.pattern.append(Image.open(files))
```

Listing A.8 Test pattern generator for non exhaustive test - main part

```
if __name__== "__main__":

    parser = OptionParser()
    parser.add_option("--max-pattern", dest="max_p", default=800, type="int",
            help="The maximum number of paterns to place on the image")
    parser.add_option("--min-pattern", dest="min_p", default=100, type="int",
            help="The minimum number of paterns to place on the image")
    parser.add_option("--min-noise", dest="min_n", default=.001, type="float",
            help="The minimum factor of noise to place on the image")
    parser.add_option("--max-noise", dest="max_n", default=.003, type="float",
            help="The minimum factor of noise to place on the image")
    parser.add_option("-w", "--img-width", dest="width", default=1024, type="int",
            help="Image width")
    parser.add_option("-y", "--img-height", dest="height", default=768, type="int",
            help="Image height")
    parser.add_option("-p", "--pattern-dir", dest="p_dir",
            default="../img/pattern/",
            help="Directory with the image pattern \"pattern*\"")
    parser.add_option("-o", "--out-dir", dest="o_dir",
            help="Directory for the generated files")
    parser.add_option("-n", "--num-images", dest="img_num", default=1, type="int",
            help="Number of test images to generate")
    (op, args) = parser.parse_args()

    if not op.o_dir:
        o_dir = "../img/" + str(op.width) + "_" + str(op.height) +"/"
    else:
        o_dir = op.o_dir + "/"

    p_dir = op.p_dir + "/"

    tp = TestPattern(p_dir)

    if not os.path.exists(o_dir):
        os.makedirs(o_dir)

    for i in range (0, op.img_num):
        ti = TestImage(op.width, op.height, tp.pattern, op.min_p, op.max_p,
                op.min_n, op.max_n)

        ti.getImg().save(o_dir + "test" + str(op.max_p)+ "_"
                +str(op.min_p)+ "_" + str(op.min_n)+ "_" +
                str(op.max_n) + "_" + str(i) + ".png")
```

Bibliography

[axi10] AMBA 4 AXI4-Stream Protocol Specification (ARM IHI 0051), 2010. URL http://infocenter.arm.com/help/topic/com.arm.doc.ihi0051a/index.html. (Cited on page 43)

[axi13] AMBA AXI and ACE Protocol Specification (ARM IHI 0022E), 2013. URL http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022e/index.html. (Cited on page 43)

[Bai11] D. G. Bailey. *Design for embedded image processing on FPGAs*. John Wiley & Sons (Asia), 2011. (Cited on pages 19 and 21)

[Ban06] D. Banerjee. *PLL Performance, Simulation and Design*. Dog Ear Publishing, LLC, 4 edition, 2006. URL http://www.ti.com/tool/pll_book. (Cited on page 47)

[BB05] W. Burger, M. J. Burge. *Digitale Bildverarbeitung*, volume 1. Springer, 2005. (Cited on page 50)

[BK08] G. Bradski, A. Kaehler. *Learning OpenCV: [computer vision with the OpenCV library]*. O'Reilly, Beijing; Köln[u.a.], 1. ed. edition, 2008. URL http://swbplus.bsz-bw.de/bsz28096496xcov.htm. (Cited on page 49)

[CDS74] V. Cerf, Y. Dalal, C. Sunshine. RFC 675: Specification Internet Transmission Control Program. 1974. URL http://www.ietf.org/rfc/rfc675.txt. (Cited on page 54)

[DH98] S. Deering, R. Hinden. RFC 2460: Internet Protocol. 1998. URL http://www.ietf.org/rfc/rfc2460.txt. (Cited on page 54)

[FF13] P. Fall, A. Fiergolski. OpenCores 1G eth UDP / IP Stack, 2013. URL http://opencores.org/project,udp_ip_stack. (Cited on pages 7, 43 and 45)

[FGP10] M. Fujita, I. Ghosh, M. Prasad. *Verification techniques for system-level design*. Morgan Kaufmann, 2010. (Cited on page 13)

[Gra10] J. O. Grady. *System verification: proving the design solution satisfies the requirements*. Academic Press, 2010. (Cited on pages 17 and 33)

[HP12] J. L. Hennessy, D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2012. (Cited on page 39)

[IEE02] IEEE Std 802.3-2002: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. Technical report, 2002. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=988967. (Cited on pages 42 and 52)

[JKS95] R. Jain, R. Kasturi, B. G. Schunck. *Machine vision*, volume 5. McGraw-Hill New York, 1995. (Cited on pages 18 and 19)

[Kan02] S. H. Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002. (Cited on page 13)

[Knu98] D. E. Knuth. *Sorting and searching*. Addison-Wesley, Reading, Mass., 2. ed. edition, 1998. (Cited on page 35)

[KRW+12] M. Klaiber, L. Rockstroh, Z. Wang, Y. Baroud, S. Simon. A memory-efficient parallel single pass architecture for connected component labeling of streamed images. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pp. 159–165. IEEE, 2012. (Cited on pages 14 and 22)

[M+65] G. E. Moore, et al. Cramming more components onto integrated circuits, 1965. (Cited on page 13)

[Nyq32] H. Nyquist. Regeneration theory. *Bell System Technical Journal*, 11(1):126–147, 1932. (Cited on page 47)

[Oli06] T. E. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006. (Cited on page 49)

[ON 12] ON Semiconductor. *LUPA3000: CMOS Image Sensor, High Speed, 3 Megapixel, 485 FPS, 13.3 Gb*, 9 edition, 2012. URL http://www.onsemi.com/pub/Collateral/NOIL1SN3000A-D.PDF. (Cited on page 19)

[PF05] D. Perry, H. Foster. *Applied formal verification*. McGraw-Hill, Inc., 2005. (Cited on page 17)

[Plu82] D. C. Plummer. RFC 826: An Ethernet Address Resolution Protocol. 1982. URL http://tools.ietf.org/html/rfc826. (Cited on page 44)

[Pos80] J. Postel. RFC 768: User datagram protocol. 1980. URL http://www.ietf.org/rfc/rfc768.txt. (Cited on page 54)

[Pos81] J. Postel. RFC 791: Internet protocol. 1981. URL http://www.ietf.org/rfc/rfc791.txt. (Cited on page 53)

[PW09] M. Pilgrim, S. Willison. *Dive Into Python 3*. Springer, 2009. (Cited on page 58)

[RIPE] RIPE. IPv4 Exhaustion. URL https://www.ripe.net/internet-coordination/ipv4-exhaustion. (Cited on page 54)

[RP66] A. Rosenfeld, J. L. Pfaltz. Sequential Operations in Digital Picture Processing. *J. ACM*, 13:471–494, 1966. (Cited on page 18)

[Sum07] M. Summerfield. *Rapid GUI Programming with Python and Qt*. Prentice Hall, 2007. (Cited on page 61)

[Tos09] S. Tosi. *Matplotlib for Python developers: build remarkable publication quality plots the easy way*. Packt Publishing Ltd, 2009. (Cited on page 50)

[Wie08] A. Wiemann. *Standardized Functional Verification*. Springer, 2008. (Cited on pages 13 and 17)

[Xil12a] Xilinx. *LogiCORE IP Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC Wrapper v2.2*, 1.2 edition, 2012. URL http://www.xilinx.com/support/documentation/ip_documentation/v6_emac/v2_2/ug800_v6_emac.pdf. (Cited on pages 7 and 42)

[Xil12b] Xilinx. *Virtex-6 Family Overview*, 2.4 edition, 2012. URL http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf. (Cited on page 28)

[Xil14] Xilinx. *Virtex-6 FPGA Clocking Resources*, 2.5 edition, 2014. URL http://www.xilinx.com/support/documentation/user_guides/ug362.pdf. (Cited on pages 7 and 46)

All links were last followed on March 1, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature