Design and Development of an FPGA Based Intelligent Stereo Vision System

Bastian Nordmeyer and Olaf Lüke

MASTER'S THESIS

assisted by:
Dipl.-Ing. Frank Schmidtmeier
M.Eng. Mohamed Shafik

submitted to:
Prof. Dr.-Ing. Bärbel Mertsching
Prof. Dr. Franz Josef Rammig
University Paderborn

Paderborn, March 31, 2010

Declaration

We hereby warrant that we wrote this Master's Thesis independently, no resources other than the cited sources were used.

Paderborn, March 31, 2010

Bastian Nordmeyer

Olaf Lüke

Abstract

In this thesis an FPGA based stereo vision system for rapid prototyping development of stereo vision algorithms in Impulse C is proposed. It allows a fast development of algorithms without deep knowledge of *Hardware Description Languages* (HDLs).

A self-engineered hardware platform, consisting of cameras connected to an FPGA development board, is used to capture stereo images.

The captured images are read out by our *Vision Framework*, which is written in VHDL. This framework also allows the access on a DDR2 memory and an gigabit Ethernet interface over Impulse C streams.

Stereo vision algorithm and necessary preprocessing steps, i. e. image filtering, as well as postprocessing are done in the *Vision Core*. The Vision Core is written in Impulse C and communicates over the Vision Framework with hardware components on the FPGA. As an example the so called *Scan-Line based Segmentation* stereo vision algorithm and several preprocessing filters are implemented.

Therefore the proposed system is a complete stereo vision platform capable of capturing images, calculating disparity maps and communicating with a host PC. It incorporates a generic concept that allows software engineers to develop, advance and test algorithms without the need of low-level hardware knowledge.

Results consisting of resource requirements, performance, power dissipation and example disparity maps are presented.

Responsibility Assignment

This thesis was written by Bastian Nordmeyer and Olaf Lüke. In the following table the content of this thesis is assigned.

		Bastian Nordmeyer	Olaf Lüke
1	Introduction	✓	✓
2	Related Work	✓	✓
3.1	Fundamentals - FPGA Design	✓	
3.2	Fundamentals - Impulse C		✓
3.3	Fundamentals - Camera System	✓	
3.4	Fundamentals - Scan-Line based Segmentation		✓
3.5	Fundamentals - Image Filter		✓
4	Requirements Analysis	✓	
5	Self-made Hardware	✓	
6	Vision Framework	✓	
7	Image Preprocessing Comparision		✓
8	Vision Core		✓
9.1	Results - Vision Framework	✓	
9.2	Results - Vision Core		✓
9.3	Results - Stereo Vision System[1]		✓
10	Summary	✓	✓
11	Outlook	✓	✓

Not specified chapters or sections are not assignable since shared aspects are discussed and the respective content was produced together.

[1] except for MT9 Camera color problem analysis

Contents

1 Introduction **1**
1.1 Objectives . . . 3
1.2 Taxonomy . . . 4

2 Related Work **5**

3 Fundamentals **9**
3.1 FPGA Design . . . 9
3.1.1 FPGA Design Flow . . . 10
3.2 Impulse C . . . 13
3.2.1 Programming Model . . . 13
3.2.2 Processes and Process Communication . . . 15
3.2.3 Constraints for Hardware Processes . . . 18
3.2.4 Hardware Optimizations . . . 20
3.3 Camera System . . . 24
3.3.1 Optical Lens . . . 24
3.3.2 Image Sensor . . . 26
3.4 Scan-Line based Segmentation . . . 30
3.4.1 Constraints . . . 31
3.4.2 Line Segmentation . . . 33
3.4.3 Correspondence . . . 34
3.5 Image Filter . . . 37
3.5.1 Gaussian Filter . . . 37
3.5.2 Median Filter . . . 38
3.5.3 Hybrid Median Filter . . . 41
3.5.4 Bilateral Filter . . . 42

4 Requirements Analysis **47**
4.1 System Requirements . . . 47
4.2 Development Board Selection . . . 49
4.2.1 General . . . 50

4.2.2 Altera . . . 50
4.2.3 Xilinx . . . 52
4.2.4 Decision . . . 54
4.3 Camera Selection . . . 55
4.3.1 Requirements . . . 56
4.3.2 Camera Overview . . . 57
4.3.3 Decision . . . 59

5 Self-made Hardware 61
5.1 Requirements . . . 61
5.2 Design . . . 63
5.3 Adapter Boards . . . 64
5.4 Cameras . . . 66

6 Vision Framework 69
6.1 Design . . . 69
6.1.1 Requirements . . . 70
6.1.2 Architecture . . . 71
6.2 Clock Manager . . . 73
6.2.1 FPGA Clocking . . . 73
6.2.2 Clock Generation . . . 74
6.3 Reset Handler . . . 75
6.3.1 Reset Strategies . . . 76
6.3.2 Decision . . . 77
6.4 Vision Core . . . 78
6.5 Block RAM Double Buffer . . . 79
6.5.1 Architecture . . . 80
6.5.2 Applications . . . 81
6.6 Stereo Interface . . . 82
6.6.1 Design . . . 82
6.6.2 Architecture . . . 83
6.6.3 Application and Restrictions . . . 84
6.7 Camera Interface . . . 85
6.7.1 Architecture . . . 85
6.7.2 OV7 Interface . . . 87
6.7.3 MT9V Interface . . . 88
6.8 Camera Configurator . . . 90
6.8.1 IP Core Selection . . . 91
6.8.2 Architecture . . . 92
6.9 DDR Interface . . . 93
6.9.1 IP Core Selection . . . 94

6.9.2 Architecture . . . 95
6.10 DDR Port Interface . . . 96
6.10.1 Architecture . . . 97
6.11 Ethernet Interface . . . 99
6.11.1 IP Core Selection . . . 99
6.11.2 Architecture . . . 100
6.12 Design Challenges . . . 102
6.12.1 Debugging and Verification . . . 102
6.12.2 Clocking and Timing Constraints . . . 104

7 Image Preprocessing Comparison 107
7.1 Gaussian Filter . . . 108
7.2 Median Filter . . . 110
7.3 Hybrid Median Filter . . . 111
7.4 Bilateral Filter . . . 112
7.5 Conclusion . . . 114

8 Vision Core 115
8.1 Design . . . 115
8.1.1 VHDL/Impulse C partition . . . 117
8.1.2 Generic Design . . . 118
8.1.3 Complexity . . . 123
8.2 Bilateral Filter and Median Filter . . . 124
8.2.1 Convolution with Histogram . . . 125
8.2.2 Processing Sequence . . . 127
8.2.3 Partitioning . . . 128
8.2.4 Index Generator . . . 130
8.2.5 Gaussian Lookup Table . . . 131
8.3 Scan-Line Based Segmentation . . . 132
8.3.1 Parameter Tuning . . . 134
8.3.2 Segment Extension . . . 139
8.3.3 Parallelization . . . 141
8.3.4 Debugging . . . 143
8.4 Packet Producer and Packet Serializer . . . 144
8.4.1 Operation . . . 144
8.4.2 Packaging . . . 146
8.5 Simulation . . . 148
8.5.1 Simulation of C Code on Host PC . . . 148
8.5.2 Simulation of Generated VHDL with ModelSim . . . 150
8.6 Host PC Viewer . . . 151
8.6.1 Architecture . . . 151

8.6.2 Features 152
8.6.3 Disparity Map Representation 153

9 Results **157**
9.1 Vision Framework 157
9.1.1 Performance 158
9.1.2 Requirements 159
9.2 Vision Core 161
9.2.1 Methodology of Measurements 162
9.2.2 Analysis 163
9.2.3 Resource Requirements 164
9.2.4 Performance 167
9.2.5 Power Dissipation 170
9.2.6 Conclusion 172
9.3 Stereo Vision System 174

10 Summary **179**

11 Outlook **183**

List of Figures

1.1 Taxonomy of the Stereo Vision System 4

3.1 FPGA block structure 10
3.2 FPGA design flow 11
3.3 Register Transfer Level (RTL) structure 12
3.4 Impulse C programming model 14
3.5 Focal length f and focal point F 25
3.6 C-Mount lens with C to CS-Mount adapter ring 25
3.7 CCD image readout 27
3.8 Bayer Pattern 28
3.9 Bayer Pattern cut-out 29
3.10 Epipolar geometry 32
3.11 Rectified epipolar geometry 32
3.12 Line segmentation 35
3.13 Correspondence 36
3.14 Hybrid Median operation 41
3.15 Hybrid Median sizes 41
3.16 Teddy example image 44
3.17 Bilateral filter convolution 44

4.1 Altera Stratix development boards 51
a Stratix IV E 51
b Stratix IV GX 51
4.2 Altera Cyclone development boards 52
a Cyclone II 52
b Cyclone III 52
4.3 Xilinx ML development boards 53
a ML405 53
b ML605 53
4.4 Xilinx SP development boards 54
a SP601 54

b SP605 . . . 54
4.5 DALSA Falcon 4M60 . . . 59

5.1 Illustration of a hardware camera interface . . . 62
5.2 FMC-LPC connector on SP601 development board . . . 62
5.3 Hardware System . . . 64
5.4 Adapter boards architecture . . . 65
5.5 SP601 Adapter . . . 65
5.6 Camera Adapter . . . 66
5.7 OV7 Camera . . . 67
5.8 MT9 Camera . . . 67
5.9 Illustration of the Camera Mount . . . 68

6.1 Illustration of elements . . . 70
6.2 System architecture . . . 72
6.3 Reset Handler: Reset sequence . . . 76
6.4 Vision Core interfaces . . . 78
6.5 Block RAM Double Buffer architecture . . . 80
6.6 Stereo Interface architecture . . . 83
6.7 Camera Interface architecture . . . 86
6.8 OV7720 timing diagram (RGB555, 320×240) . . . 88
6.9 MT9V032 timing diagram (Bayer Pattern, 320×240) . . . 89
6.10 Bayer Pattern 2×2 cut-out . . . 90
6.11 Wishbone Bus . . . 92
6.12 Camera Configurator architecture . . . 93
6.13 DDR Interface architecture . . . 96
6.14 DDR Port Interface architecture . . . 98
6.15 Ethernet Interface architecture . . . 101
6.16 Simple RTL circuit . . . 105
6.17 RTL circuit with pipelining . . . 106

7.1 Unfiltered left and right Tsukuba image with corresponding disparity map . . . 108
a Left image . . . 108
b Disparity map . . . 108
c Right image . . . 108
7.2 Gaussian filtered images and corresponding disparity maps . . . 109
a $\sigma = 0.15$ . . . 109
b $\sigma = 0.50$ . . . 109
c $\sigma = 0.75$ . . . 109
d $\sigma = 1.00$ . . . 109

7.3 Median filtered images and corresponding disparity maps . . . 110
a $ks = 5 \times 5$ . . . 110
b $ks = 7 \times 7$ . . . 110
c $ks = 9 \times 9$ . . . 110
d $ks = 11 \times 11$ . . . 110
7.4 Hybrid Median filtered images and corresponding disparty maps 112
a $ks = 5 \times 5$ . . . 112
b $ks = 7 \times 7$ . . . 112
7.5 Bilateral filtered images and corresponding disparity maps . . 113
a $\sigma = 0.00$ . . . 113
b $\sigma = 0.05$ . . . 113
c $\sigma = 0.15$ . . . 113
d $\sigma = 1.00$ . . . 113

8.1 Impulse C process flow diagram of the Vision Core . . . 116
8.2 Generic image preprocessing subsystem . . . 120
8.3 Generic postprocessing subsystem . . . 122
8.4 Bilateral filter window sliding sequence . . . 127
8.5 Histogram update procedure . . . 128
a Vertical . . . 128
b Horizontal . . . 128
8.6 Bilateral Filter partitioning . . . 128
8.7 Index generation sequence . . . 130
8.8 Maximum distance between identical objects . . . 134
8.9 Teddy image set with different maximum matching search distance . . . 135
a Left image . . . 135
b Right image . . . 135
c $d = 35$ . . . 135
d $d = 50$ . . . 135
e $d = 65$ . . . 135
8.10 Tsukuba image set with different maximum matching search distance . . . 136
a Left image . . . 136
b Right image . . . 136
c $d = 10$ . . . 136
d $d = 15$ . . . 136
e $d = 20$ . . . 136
8.11 Disparity maps with different input image bit depth and color tolerance . . . 137
a 5 bit, $\tau = 6$ . . . 137

b 8 bit, $\tau = 48$ . . . 137
8.12 Teddy image set with different color tolerances . . . 138
a $\tau = 3$ . . . 138
b $\tau = 6$ . . . 138
c $\tau = 9$ . . . 138
d $\tau = 12$ . . . 138
8.13 Tsukuba image set with different color tolerances . . . 138
a $\tau = 3$ . . . 138
b $\tau = 6$ . . . 138
c $\tau = 9$ . . . 138
d $\tau = 12$ . . . 138
8.14 Segment matching illustration . . . 139
a Without segment extension . . . 139
b With segment extension . . . 139
8.15 Matching and resulting disparity map . . . 140
a Without segment extension . . . 140
b With segment extension . . . 140
8.16 Parallelization of the Scan-Line based Segmentation . . . 142
8.17 Parallelization example . . . 143
8.18 Segmentation and matching debug output with corresponding disparity map . . . 144
a Segmentation . . . 144
b Matching . . . 144
c Disparity map . . . 144
8.19 Ethernet Frame . . . 146
8.20 Host PC simulation . . . 149
8.21 Host PC Viewer . . . 152
8.22 Disparity map representations in Host PC Viewer . . . 154
a Flat . . . 154
b Points . . . 154
c Vertices . . . 154

9.1 Images from MT9 Camera . . . 174
a Left image . . . 174
b Disparity map . . . 174
c Right image . . . 174
9.2 Helmet images from OV7 Camera . . . 175
a Left image . . . 175
b Right image . . . 175
c Filtered MF . . . 175
d Filtered BF . . . 175

e Disparity WF 175
f Disparity MF 175
g Disparity BF 175
9.3 Box images from OV7 176
a Left image 176
b Disparity map 176
c Right image 176
9.4 House images from OV7 177
a Left image 177
b Disparity map 177
c Right image 177

List of Tables

4.1 Development board comparison 55

6.1 ClockManager: Generated clocks 74

8.1 Ethernet types as used in the Vision Core 147

9.1 Ethernet Interface frequency comprehension 158
9.2 DDR Interface frequency comparison 159
9.3 Resource requirements of Vision Framework components . . . 160
9.4 Resource requirements of Bilateral Filter, Median Filter, Scan-Line based Segmentation and Parallel Scan-Line based Segmentation. 164
9.5 Resource requirements of complete Vision Core with Bilateral Filter and Median Filter. 167
9.6 Performance of Bilateral Filter, Median Filter, Scan-Line based Segmentation and Parallel Scan-Line based Segmentation. . . 168
9.7 Performance of complete Vision Core with Bilateral Filter and Median Filter. 170
9.8 Power dissipation of Bilateral Filter, Median Filter, Scan-Line based Segmentation and Parallel Scan-Line based Segmentation. 171
9.9 Power dissipation of complete Vision Core with Bilateral Filter and Median Filter. 172

Chapter 1

Introduction

A human perceives two slightly different projections of his environment. Binocular disparity, i. e. the difference in the two perceived images, allows the human brain to calculate depth information for objects in the surrounding [SSG00]. The process of depth recognition is called *stereopsis*.

Stereo vision is a process that tries to emulate stereopsis. 3D information are extracted from digital images captured by two cameras. Disparity maps obtained by *stereo vision algorithms* are important in many different fields, such as aid for the visually impared [Mol+98], automated systems [SS96], robotics or space exploration.

An example for the latter is the *Solar Terrestrial Relations Observatory* (STEREO) project from NASA. It consists of two identical space crafts which allow a stereoscopic view of the sun. They are arranged in space for capturing images that yield surface properties of the sun [Soc+00].

The application range in robotics is vast. Stereo vision gives information about the relative distance to objects in the environment, which allows object recognition [Ber+96], tracking [SAGS07] and navigation [ML00].

Image processing applications such as stereo vision can be implemented on several different platforms, e. g. computer systems, FPGAs and ASICs are possibilities. Computer systems are flexible and easy to develop on, but they can also be expensive and power consuming. Where ASICs are cost and power efficient, though only viable in high quantities. They also do not allow easy advancements of projects.

FPGAs on the other hand are programmable hardware systems and have a flexibility comparable to computer systems with lower cost and lower power

consumption. Furthermore, the technical progress of semiconductors, in particularly in the development of FPGAs, allows to buy more logic for the same amount of money. Notably, there are now affordable FPGAs on the market with enough logic to be attractive for image processing applications. FPGAs also offer a high potential for parallelization. This is a big advantage compared to DSPs.

They are however programmed with hardware description languages (HDLs), which are relatively low-level languages that require low-level hardware knowledge. The average software developer does not possess deep low-level hardware knowledge. This leads to a division in hardware development and software development. Typically an algorithm is firstly implemented on a higher level of abstraction in software and the parameterization is specified.

Hardware development is a longer and more inflexible process, it does not allow to respond to changes as fast as software development. Thus a higher level of abstraction is also desired in hardware development.

One possibility of abstraction can be found in the SystemGenerator[1] from Xilinx. It allows to simulate blocks of logic in Simulink[2] with subsequent synthesis for FPGAs.

Other approaches try to raise the level of abstraction with the use of programming languages that are known from software development for synthesis. These include so called C-to-VHDL compilers, such as Impulse C from Impulse Accelerated Technologies[3]. These compilers allow the developer to use the C programming language with some constraints for FPGA development. With Impulse C it is also possible to utilize a design flow that starts with a well tested C application, which can than be altered to compile to VHDL. This allows rapid prototyping and simpler testing of changes.

A hardware platform consisting of cameras, FPGA board and interfaces to the host PC is another important part of a stereo vision system. Properties such as resolution of the cameras or available interfaces on the FPGA board can change the possible area of application. Also a prototyping platform that allows to replace individual parts of the hardware platform is crucial in the development phase of a stereo vision system.

[1] http://www.xilinx.com/tools/sysgen.htm
[2] http://www.mathworks.de/products/simulink/
[3] http://www.impulseaccelerated.com/

1.1 Objectives

The goal of this thesis is to develop a rapid prototyping stereo vision system that allows the development of vision algorithms on an FPGA without deep knowledge of HDLs. The system should capture two stereoscopic images, process them on an FPGA and transmit the results to a host PC. To do this, two primary tasks have to be accomplished.

Firstly, an applicable hardware platform has to be selected. This platform includes an FPGA board, appropriate cameras for image capturing and an high speed interface which allows to transmit the results to a host PC. The FPGA board requires an hardware interface that allows to connect and read out the chosen cameras and an external memory that permits the storage of computation results.

Secondly, a vision system has to be developed on the chosen hardware platform. A part of this task is the development of an IP Core that utilizes a certain stereo vision algorithm, in this case the *Scan-Line based Segmentation* [SM09]. Optimization possibilities for this algorithm have to be probed and it has to be tested for performance, power dissipation and resource requirements. Integration of the stereo vision algorithm on the FPGA is done with the Impulse C programming language.

To be able to use Impulse C generated IP Cores with the utilized hardware platform, external HDL interfaces have to be developed. These interfaces abstract the hardware of the platform and allow a hardware independent implementation of an vision algorithm.

Camera images have to be captured by the system and provided over an interface to the IP Core. To be able to transmit results to a host PC, output interfaces for suitable hardware peripherals are necessary. Additionally, if required, an interface to an external memory has to be implemented.

All of the above has to be achieved in a generic manner, that on the one hand allows a later modification of the stereo vision system, e. g. other vision algorithms and on the other hand permits to reuse components in other projects.

As an secondary objective the authors of this thesis decided to develop a cohesive stereo vision system that is deployable in a real world application. This means that low-priced hardware has to be used and if necessary self-engineered. Furthermore, it was already at the very beginning of the work evident that besides the stereo vision algorithm different preprocessing filters

have to be tested. The best filter also has to be implemented in Impulse C. In addition a possibility to evaluate the stereo vision system on a host PC has to be created.

1.2 Taxonomy

The taxonomy of the Stereo Vision System is illustrated in figure 1.1.

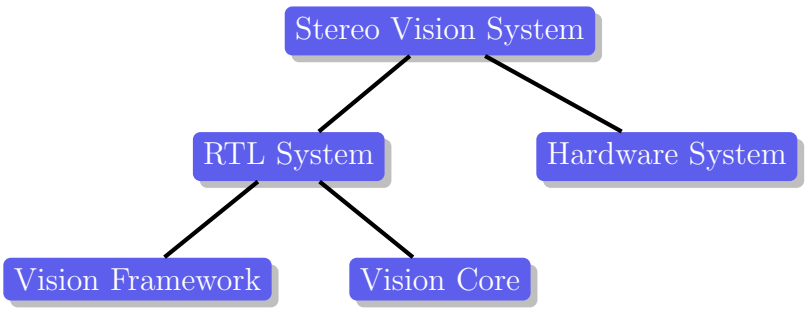


Figure 1.1: Taxonomy of the Stereo Vision System

A devision of the Stereo Vision System into *RTL*[4] *System*, which is the configurable part of the system and could be parameterized or partly replaced by other algorithms and *Hardware System* is performed. Whereas the RTL System is further divided into *Vision Core* and *Vision Framework*.

The latter is written in VHDL and establishes interfaces to the hardware platform. These interfaces are utilized by the Vision Core, which implements the computation intensive vision algorithms in Impulse C. Together the Vision Core and the Vision Framework represent the RTL System that interacts with the Hardware System. The term RTL System refers to the fact that this system processes digital data which is transformed and passed from one register to another.

Whereas the term Hardware System represents the composition of cameras, FPGA development board and other required hardware components.

[4]Register Transfer Level

Chapter 2

Related Work

In the following, some papers are shortly introduced to give an overview over scientific work done on Impulse C, preprocessing, stereo vision and image processing architectures.

In [TS07] an implementation of a Bilateral Filter with Impulse C on a Xilinx ML410 development board is presented. A filter window of size 3×3 for the Bilateral Filter is used.

Three different implementations are analyzed. The first implementation features parallel multi-processes, i. e. multiple processes ere used, each processing data in parallel. This approach led to $3,592$ occupied slices with a frame rate of 12.11 frames per second.

The second implementation uses pipelined multi-processes, i. e. each of the deployed processes uses a pipeline for the calculations. $1,570$ slices are used and a speed of 10.10 frames per seconds is accomplished.

One single process with a large pipeline is used in the third implementation. This approach consumed 930 slices and produces 65.31 frames per second.

It is concluded that it is a little bit faster to unroll a loop and execute it in parallel, but it also is a lot more expensive in terms of slice consumption. Furthermore, one big pipeline is more efficient than several small pipelines, doing the same work.

The processes are clocked with 27 MHz, though the resolution of the input images is not given.

A hardware/software co-design for line detection algorithms written in Impulse C is introduced in [KS09]. Three different edge detection filters (Robert,

Sobel and Prewitt) and a Hough transform are implemented on a Xilinx Spartan-3E development board.

The system has a 24 bit bitmap file as input, which is translated to a grey scale image with a software process. In a first step one of the edge detection filters is executed on the FPGA, whereas the sine and cosine functions are calculated again in software.

In a second step the edge enhanced images are Hough transformed, again on the FPGA with the trigonometric functions executed in software. Another software process writes the results in a file.

Partitioning of the hardware and software parts is done via Impulse C. This fine grained approach to hardware/software co-design yields a speed of about one processed image per second.

64 % of the slices available on the development board are utilized by the VHDL code, which is generated by the Impulse C C-to-VHDL compiler.

A real-time stereo vision system on an FPGA is proposed in [MM03]. A complete system is described, which consists of two primary components.

The first component implements camera calibration. A perspective projection between 3D points in the environment and in the camera images is calculated. This component simplifies the rest of the system.

As the second component, a sum of absolute difference (SAD) algorithm is implemented. It is implemented in different versions with kernel sizes ranging from 5×5 to 15×15. With an image size of 640×480, speeds of 8.7 to 26.0 frames per second are achieved. The performance is limited by the calibration.

In [AHK07] a hardware architecture for stereo vision sensors is proposed. The architecture consists of two DSPs and one FPGA and has three stages.

Preprocessing is done in the first stage. It consists of the rectification of input images to fulfill the epipolar geometry. One of the DSPs is used for the rectification algorithm since it needs to be recalibrated for every stereo vision sensor, which would be unpractical for FPGAs.

The second stage is the stereo vision algorithm implemented on an FPGA. It is an SAD algorithm for an 8 bit grayscale input image with a resolution of 320×240. The SAD kernel size is 3×3. This setup can calculate one disparity map per 2.3 ms, which equals about 435 frames per second.

Postprocessing is again implemented on a DSP, it refines the output of the stereo vision algorithm. Deviations in the vertical alignment can be detected and new calibration values for the preprocessing stage are calculated.

An FPGA based vision system for low-complexity tasks, such as obstacle detection, is presented in [Arr05]. As stated by the author it is a materialization of the hardware section of his Ph.D.

The presented system is optimized for real-time computations (up to 25 images per second) and using a self-made FPGA board. This board is equipped with an Altera FPGA, three memories (input, rom and output memory) and USB as well as Firewire interfaces.

For image input a monochrome camera with a resolution of 640×480 pixel is used. Furthermore, it is possible to connect IEEE-1394 cameras for image input.

The architecture of the vision system consists of a chain of blocks which perform image readout and storage, computations and camera configuration. An algorithm operates on the stored images in the input memories and places the results in an output memory. The input and output memories of the FPGA boards are accessible by connectors such that another board can be connected to it.

In a chain, adjacent modules can therefore use the output memory as input to perform operations. The idea behind it is to create a pipeline consisting of hardware modules which perform operations on a traversing data stream.

Chapter 3

Fundamentals

The Stereo Vision System that is proposed in this thesis encloses a wide variety of different topics, such as FPGAs, Impulse C, camera systems, stereo vision algorithms and image filters.

In the following the fundamentals for all of these topics are presented. Every important basic principle that is needed to comprehend the design and implementation of the Stereo Vision System is depicted.

3.1 FPGA Design

An *Field Programmable Gate Array* (FPGA) is a configurable integrated circuit. It consists of an array of logic elements called Configurable Logic Blocks (CLBs), interconnection signals and I/O blocks (IOB). Mostly integrated memory (block RAM, BRAM), Digital Clock Managers (DCM) to provide different clocks are integrated as well[1].

These blocks are connected via configurable interconnections. Figure 3.1 shows the block structure of an FPGA.

Typically each logic block consists of a lookup table (LUT), selection circuits (e. g. multiplexer) and flip-flops. The lookup tables are configurable to represent a given logic function, whereas multiplexer and flip-flops are necessary to control the data flow.

[1] http://www.xilinx.com/company/gettingstarted/index.htm

[2] http://www.xilinx.com/images/circuitblock.jpg

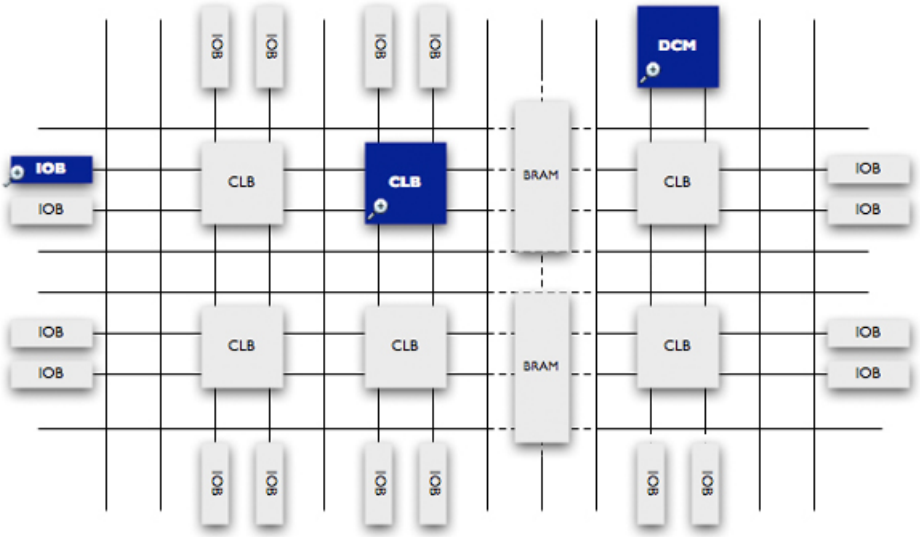


Figure 3.1: FPGA block structure[2]: I/O Block (IOB), Configurable Logic Block (CLB), block RAM (BRAM), Digital Clock Manager (DCM)

Since a great number of CLBs are available on modern FPGAs, very complex circuits can be generated. The user has to create a configuration encoded in a *bitstream* that describes which elements on the FPGA are occupied and how they are interconnected.

In the next section a design flow for FPGA development is introduced.

3.1.1 FPGA Design Flow

The development of an FPGA Design is a complex task with many failure possibilities. To handle this task a developer has to follow a design flow.

In the following a possible design flow is introduced and illustrated in figure 3.2. It is based upon the design flow presented at FPGA Central[3].

Firstly, the system has to be designed based on requirements or specifications. This includes the development of the later system architecture, i. e. the components with their behaviour and interfaces.

Secondly this architecture can be implemented in a *Hardware Description Language* (HDL) which describes the components and their behavior. Com-

[3] http://www.fpgacentral.com/docs/fpga-tutorial/fpga-design-flow-overview

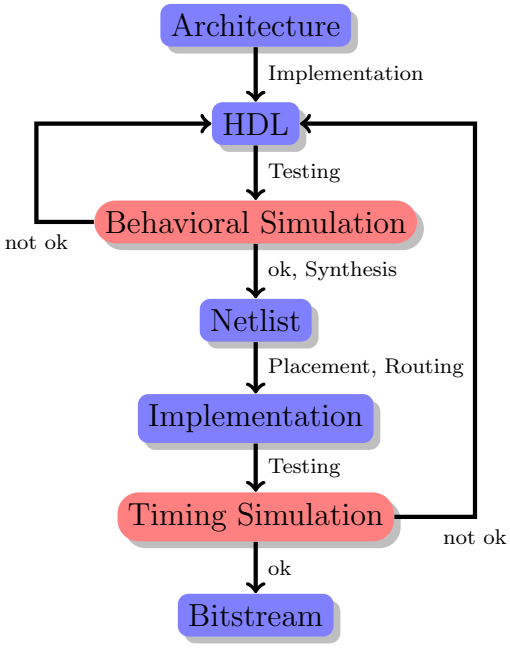


Figure 3.2: FPGA design flow

monly used HDL's are the *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) [Ash01] and *Verilog*. We utilize VHDL to implement our Vision Framework components (see chapter 6).

After the implementation, the developed components can be simulated and tested in a behavioral simulation. Thus, a test environment must be written which simulates the environment of the components. A test framework usually provides the required clock signals and simulates external elements which interact with the component. During the simulation of a component erroneous behavior can be revealed and fixed. The development of a test framework can be a very complex task which might be more complex than the development of the component itself.

If the behavioral test is accomplished a synthesis tool generates a digital circuit description, so called *netlist*, that describes the computed internal wiring of the FPGA on a *Register Transfer Level* (RTL). The term Register Transfer Level refers to the fact that data is transfered and modified between registers. Figure 3.3 shows the basic structure of a RTL circuit.

After a netlist is generated a routing tool creates an implementation for the FPGA. This implementation is composed of the occupied logic elements, memory and other internal blocks as well as the used interconnection signals.

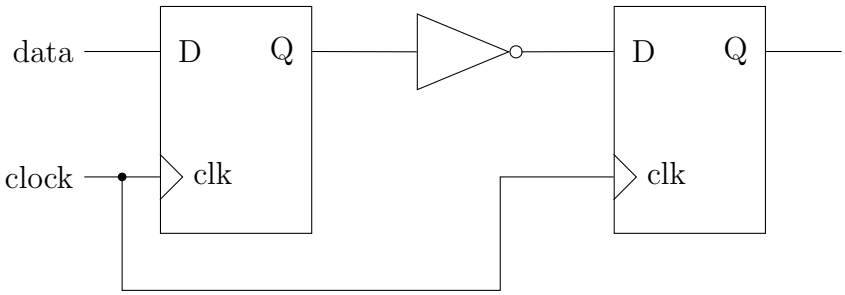


Figure 3.3: Register Transfer Level (RTL) structure: The register reads the input data clock synchronous and transmits it to combinatorial logic elements (shown as inverter). The results are transmitted to another register which passes them in the next clock cycle.

Based on this implementation a timing simulation can be performed since the circuits on the FPGA are established (wiring lengths and occupied elements are known). The developer can specify timing constraints on signals, e. g. maximum latency and maximum clock, which can be analyzed. This analysis determines if the design satisfies the requirements.

If all requirements are satisfied the FPGA can be configured with this design by transmitting the configuration bitstream.

As known from libraries in software development, special proprietary and non-proprietary components can be used by the developer. These are so called *soft cores*. Typical soft cores are microprocessors. Xilinx offers for example two different types of soft processors the 8 bit *PicoBlaze* and the 32 bit *MicroBlaze*. These are defined on a register transfer level and can be synthesized on the FPGA. Other examples for soft cores are communication controllers which simplify the communication over an electrical bus.

Since soft cores are limited to the utilization of logic elements and integrated memory, the performance is sometimes not satisfying. Therefore, so called *hard cores* are integrated in modern FPGAs which are special integrated hardware components. For example integrated gigabit transceivers for high performance communication are integrated in some FPGA devices. Hard cores offer a higher performance than comparable soft cores but are limited in their quantity. Soft cores can be synthesized multiple times until the number of logic elements is depleted.

The development with HDLs can be exhausting since development takes place on a hardware level with little abstraction and many failure possibilities, e. g.

clock failures. Thus, alternatives are welcome which simplify the FPGA development process.

One alternative is the generation of VHDL code by the Impulse C C-to-VHDL compiler presented in the next section.

3.2 Impulse C

Impulse C from Impulse Accelerated Technologies[4] is a C-based development tool for programmable hardware systems consisting of FPGAs and general purpose processors. The language is a subset of ANSI C extended with additional Impulse C libraries.

The Impulse C programming model starts with a C application that is partitioned into *processes* and *streams* or other communication objects. Whereas the processes are segmented into hardware and software processes. All hardware processes run concurrently on the FPGA and can communicate over streams with each other and software processes on the processor.

Some limitations are present in the Impulse C subset of the C language, such as a lack of pointer arithmetic and limited struct and function support.

Optimizations are done automatically by the compiler and it is also possible for the user to give the compiler hints for hardware optimization possibilities such as pipelining and loop unrolling [Imp08].

One big advantage of Impulse C is, it's support of software-oriented design methods. It can be used by software engineers to produce HDL code without deep knowledge of hardware description languages. Therefore, it is easy to use by algorithm developers that have a software engineering background. Some low-level hardware knowledge is needed, otherwise an user of Impulse C would not be able to optimize the C code in a meaningful manner.

3.2.1 Programming Model

The Impulse C programming model is illustrated in figure 3.4 and is described in detail in [PT05]. It starts with a C language program that has to be restructured into processes which are interconnected with streams. In section 3.2.2 the process and streaming model is described more precise. The

[4] http://www.impulseaccelerated.com

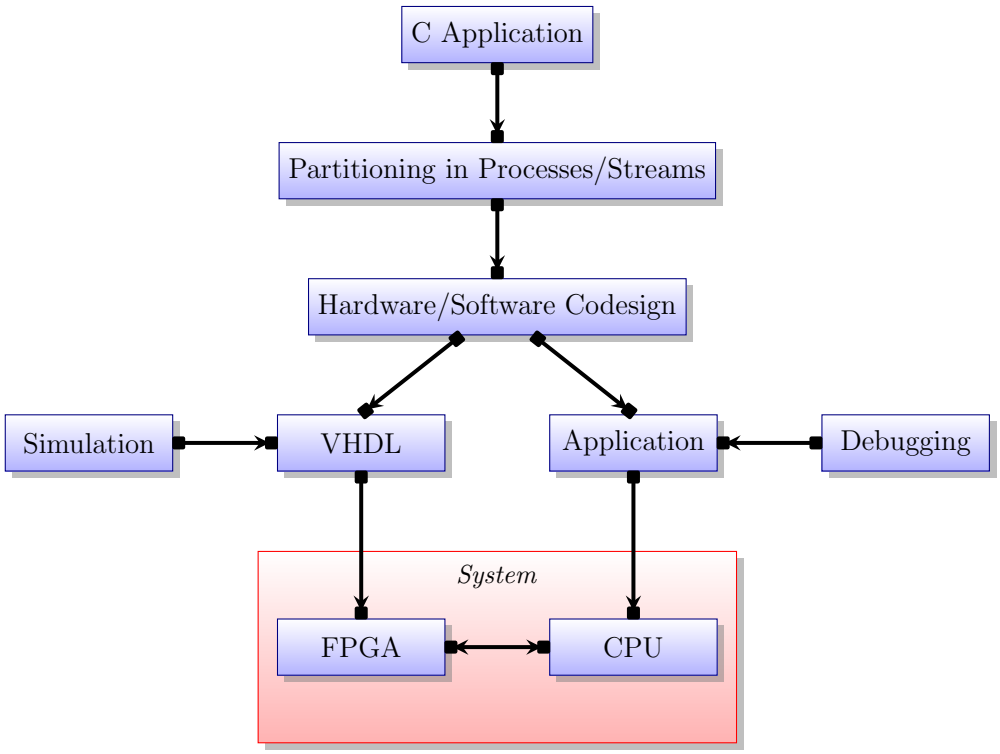


Figure 3.4: Impulse C programming model

software processes are build with a standard C compiler either for debugging on a PC or for the utilized processor. This part of the code can use ANSI C and libraries that are available on the target platform as well as all C extensions that are available in the used C compiler.

After the partitioning in processes and streams, the hardware/software codesign takes place. This means, that the processes are partitioned in hardware processes (running on the FPGA) and software processes (running on the processor). Impulse C allows communication between an FPGA and a general purpose processor. The processor might be attached to the FPGA in actual hardware or might be synthesized on the FPGA. Mostly used are hard cores such as PowerPC processors and soft cores such as MicroBlaze or PicoBlaze (see 3.1.1).

In the next step, the hardware/software partitioned processes and streams can be compiled with a standard C compiler and tested as well as debugged on the PC with standard programming tools such as profilers and debuggers. This part of the development process is only usable for a coarse test of the

flow of the streams and the general operation of the implemented algorithms. A hardware process might run fine in the software simulation and not even compile with the C-to-VHDL compiler or indeed compile to VHDL though not synthesize with the used VHDL synthesis tools.

Afterwards the hardware part is translated into VHDL by the C-to-VHDL compiler. This only compiles the Impulse C subset of C, which lacks some C features such as recursion, function calls in general, pointer arithmetic, pointers to structures, structures as fields in structures, union and floating point arithmetic. A detailed description about the constraints of Impulse C is depicted in section 3.2.3.

When the general operation is demonstrated in the software simulation the generated VHDL code, generated by Impulse C, can be simulated in a simulator such as ModelSim or downloaded to the actual FPGA and tested on the system. If an error occurs in this test or the VHDL code does not synthesize in the first place, the defective VHDL code has to be traced back to the actual Impulse C implementation where the erroneous code can be fixed.

The most important part in the Impulse C programming model is the divsion of a problem in processes and process communication objects, which are described in detail in the following.

3.2.2 Processes and Process Communication

In general, the idea behind Impulse C is to use concurrent processes, which execute algorithms and interconnect them with communication objects. These objects can enable the exchange of data as well as synchronization between processes.

Processes

Each process is a piece of the C code that has a number of input and output communication objects (see section 3.2.2). With these objects the processes can communicate with each other.

A process is either a hardware or a software process. The hardware processes will run concurrently on the FPGA after the code is translated to VHDL and synthesized. Hardware processes are restricted to the Impulse C subset of the C programming language and have to be optimized to run fast on an FPGA, as desribed in section 3.2.4.

Software processes on the other hand are not restricted. It is possible to use the full functionality of the utilized processor system. The processor can be in hardware (e. g. PowerPC) or it can be a synthesized processor on the FPGA (e. g. MicroBlaze or PicoBlaze). It is possible to utilize every C construct that is available on the C compiler that is used to compile to the appropriate machine code. This includes linking against libraries that are ported to the platform.

Each Impulse C process is executed concurrently and has access to local memory that is only available to one process. These processes are in this respect comparable to standard threads that are known from operating systems. Therefore it is possible to transform an existing multithreaded application into an Impulse C application with multiple processes. However, there are differences between threads and Impulse C processes that have to be considered.

It is not possible to use shared global memory as known from threads. This has to be done with communication objects (see section 3.2.2) or via dual-port RAM, that can enable sharing of memory between maximal two processes if the target platform supports it.

Also Impulse C processes cannot be created at run-time as is possible with threads, they rather have to be defined at compile-time. Where threads share one or more processors, each Impulse C process runs in its own environment, can have different execution frequencies and is synchronized only via communication objects with other processes.

Another possibility to use an Impulse C process is to write it in VHDL, for this one has to program the communication with other streams over the communication objects by hand.

Beyond the streams, Impulse C offers several means of communicating between processes.

Communication Objects

Several communication objects which interconnect different processes are utilized in Impulse C. *Signals* and *semaphores* are one-to-one and one-to-many synchronization objects respectively. Whereas *streams* and *registers* can be used for buffered and unbuffered data exchange. *Shared memory* is a possibility to exchange data between a software and a hardware process.

Streams Streams are the most used means of communication between processes. A stream object consists of a signed/unsigned integer, char, float or double message with a bit depth of up to 64 bit and a buffer with a predefined size. The float and double message types are however only usable when communicating between software processes.

Processes that want to exchange data can *read* and *write* on streams. The reading process blocks until at least one element is in the buffer. If more than one element is in the buffer, the buffer acts as a FIFO and the first element is received by the reading process. The writing process on the other hand will block until the buffer is not full and append the data to the buffer otherwise.

In software processes it is also possible to use burst operations. With burst operations an whole array can be written and correspondingly filled on the reading side.

Signals A signal is a one-to-one synchronization object. It is also possible to communicate status information from one process to another with signals. For this purpose the signal can transmit signed and unsigned data with a bit depth of up to 64.

When signals are used one process *posts* a signal with a value and another process *waits* for this signal. The waiting process blocks until the other process posts this signal. Whereas the posting process does not wait after posting a value. Existing values that are posted before and not yet received by a waiting process will be overwritten.

Semaphores With semaphores it is possible to establish a one-to-many synchronization between processes. Unlike signals it is however not possible to send status information from one process to another with semaphores.

A counter is used to form the semaphore. A process that *waits* on the semaphore will decrease the counter and the process can proceed if the counter is greater than zero. The counter is increased by *releasing* processes. With this approach it is possible to employ a multi-process synchronization.

Registers A register is a piece of memory that can be written by one process and read by multiple processes, so called *fan-out register*. Unlike streams, registers are not buffered and unlike signals they are unsynchronized.

Hence, it is possible to e. g. implement a counter with registers that is counted down by one process using the *put* method. This value can then asynchronously be read from multiple other processes with the *get* method and acted upon.

Shared memory Shared memory can be used to communicate between a software and a hardware process over DDR memory without intermediate Impulse C communication objects. To use shared memory a location of the memory has to be declared, whereupon data with arbitrary length can be written to and read from relative to the location specified. This is only possible for development boards that have an Impulse C platform support package.

When shared memory is used, it has to be ensured that two processes are not writing to the same part in memory at the same time. It is possible to safeguard a part of memory with semaphores or similar Impulse C communication objects.

In the following constraints for the described communication objects and other parts of Impulse C are depicted.

3.2.3 Constraints for Hardware Processes

Some constraints have to be considered when Impulse C is used. Firstly, Impulse C does not allow the use of all features present in C. Some of the C constructs are adjusted to ease the translation into VHDL. Secondly floating point arithmetic cannot be used on an FPGA, hence only limited fixed point arithmetic is usable in Impulse C.

C Restrictions

The Impulse C subset of the C programming language has several restrictions. C function calls are only supported in a very limited way. It is not possible for a process to call another process. Processes can only communicate via communication objects (see section 3.2.2), though it is possible to use *hardware primitive functions* or *external HDL hardware functions*. Furthermore, pointer arithmetic and structs are limited.

Hardware Primitive Function A hardware primitive function can have parameters that consist of integers with bit depth of 1 to 64, pointers

to integers and arrays of integers. Pointers to integers are however only used as input/output parameters, it is not possible to use pointer arithmetic.

The address of the pointer must be resolvable at compile-time. Structs and pointers to structs are not supported as parameters in hardware primitive functions. A hardware primitive function will be translated into a distinct VHDL entity that is executed from the corresponding process.

External HDL Hardware Function Another possibility to create functions is to use external HDL hardware functions. These are implemented in VHDL as entities and can be invoked with the *co implementation* pragma.

Pointer Arithmetic The use of pointers in processes is limited. It is not possible to use pointer arithmetic and addresses to variables have to be resolvable at compile-time, as with hardware primitive function parameters. It also it not possible to reuse one pointer to point to different variables, e. g. in

```
p = &array_a[2];
// ...
p = &array_b[3];
```

the second assignment is not allowed. This restriction renders pointers in Impulse C nearly useless and limits the possibility to port preexisting C programs to the Impulse C language subset of C.

Structs In C a struct can be used to represent objects that are comprised of different data types. The Impulse C subset of the C programming language however is limited.

It is not possible to use structs, pointers, arrays or bit fields as data types in structs. Furthermore, pointers to structures are not supported and structs cannot be used as parameters to hardware primitive functions or as global variables. Most of these limitations are not obvious, since it clearly would be possible to interpret each variable in a struct as a local variable of the process. Instead Impulse C translates structs to entities that have a memory interface, hence the access to structs is unnecessary slow.

Besides the restrictions on C language constructs, Impulse C has limitations regarding to floating point arithmetic and math operations. These restrictions are described in the following.

Integer/Floating-Point Math Operations

Impulse C supports the operators +, ++, -, - - and * on arbitrary integer variables with a single-cycle operation. The division operator / is however implemented with a multi-cycle method that needs n cycles for a division that yields a result value of bit depth n. This implementation is unusually slow, when a division of two big integers is used, it might be preferable to use an external HDL hardware function (see section 3.2.3) that uses an $\mathcal{O}(1)$ implementation for division as described in [Vla04]. The modulo operator % uses the division operator implementation and has therefore the same runtime.

Floating-point operations are in general not available in Impulse C. It is however possible to use a fixed-point arithmetic extension to integer variables. In this mode normal integer values are used and the programmer has to keep track of the fixed-point format. Impulse C offers macros to convert between fixed-point variables and floating-point variables in software processes, so that an floating-point variable from the processor can be used in the FPGA. To perform additions, subtractions, multiplications and divisions on the fixed-point variables the macros *FXADD*, *FXMUL* and *FXDIV* can be used. Each of these macros is usable with variables of the same bit depth and takes the two variables as well as the position of the fix-point as a parameter. One has to have knowledge about the employed variables and they must have predictable sizes, such that a adequate position for the fix-point can be determined before compile time.

One of the reasons Impulse C provides only limited C constructs and floating point arithmetic is the need to optimize the code for programmable hardware.

3.2.4 Hardware Optimizations

Impulse C incorporates several optimizations for hardware generation. In a first step the Impulse C code is optimized independent from the hardware generation where possible. Afterwards the generated HDL code is effected by so called *pragma* directives. These directives have to be manually set by the developer to give the Impulse C compiler hints where HDL optimizations are possible.

Impulse C Optimizations

The C-to-VHDL compiler from Impulse C can optimize basic reducible math constructs, i. e. strength-reduction on integer operations such as multiplication, division and modulo. When multiplying or dividing with powers of two values, the operators can be replaced with the more efficient shift operator. This is especially meaningful in the case of divisions, since divisions are implemented in a time consuming manner (see section 3.2.3). In the case of modulo by powers of two, the modulo-operator is replaced by logical *and-operations*.

Impulse C has two tools for code optimizations *impulse_porky* and *impulse_sm*. The former is used for general C language optimizations, such as the power of two optimizations described above. Whereas the *Stage Master* (impulse_sm) optimizes individual blocks of Impulse C code. A block can be a process body, a loop body or a control statements (if-then-else). What kind of optimization is used in these blocks must be specified by the developer with pragmas.

If a block of Impulse C code has a particularly bad performance, it is possible to hand-optimize it using the CO IMPLEMENTATION support. This allows to connect VHDL entities to Impulse C processes, if the inputs and outputs to the entities conform to a specified interface. In Impulse C these entities can then be used in the same way as hardware primitive functions.

Furthermore, Impulse C scalarizes arrays when it is possible. Different elements of an array cannot be accessed at the same time, since Impulse C generates arrays with an memory interface. If the array is big, the generated hardware might use block RAM to store the values. However, if the elements of the array can fit into a distributed RAM Impulse C might translate an array with n entries into HDL that uses n different variables that are stored in distributed RAM.

This does not use a memory interface, therefore it can be accessed in parallel, e. g. when the CO UNROLL pragma is used. CO UNROLL and other user defined pragmas, which are utilized for optimization purposes, are described in the following.

Pragmas

A pragma is a directive that communicates additional pragmatic implementation information. It is processed before compilation. In Impulse C pragma

directives are used to provide information on how a specific block should be handled by the Impulse C C-to-VHDL compiler. In the following the most important pragmas are briefly described.

CO PRIMITIVE The Impulse C subset of the C language does not allow arbitrary function calls. It is however possible to declare a C function and use the *CO PRIMITIVE* pragma. This C function can only have integers or arrays as input and integers as output, therefore it is very limited (see section 3.2.3). Functions with the CO PRIMITIVE pragma can be optimized by also adding the CO FLATTEN pragma described below.

CO FLATTEN The logic of a complex if-then-else or switch block can be explicitly flattened by the *CO FLATTEN* pragma. For example a switch statement with four simple cases could require four stages if unflattened but only one stage if flattened. A flattened block might however require more slices on the FPGA. When the CO PIPELINE pragma is used the compiler will implicitly flatten all control logic, since pipelining would not be possible otherwise.

Another use of the CO FLATTEN pragma can be found when using hardware primitive functions. When used at the top of a hardware primitive function, the whole function will be flattened if possible. This means, that a hardware primitive function that contains e. g. only one switch statement will be translated into a single state and only requires one cycle to be executed.

CO PIPELINE Declaring a CO PIPELINE pragma is possible within the body of a loop. The Impulse C C-to-VHDL compiler will try to pipeline every statement following this pragma. To understand pipelining the following code might be considered:

```
for(i = 0; i < 42; i++) {
    x = x + i*5;
}
```

This example code will result in one adder and one multiplier that are executed 42 times, resulting in an execution time of $42 \cdot (d_{\text{add}} + d_{\text{mul}})$, where d_{add} and d_{mul} are the delays for adding respectively multiplying on the FPGA. If the CO PIPELINE pragma is used

```
for(i = 0; i < 42; i++) {
#pragma CO PIPELINE
```

```
        x = x + i*5;
    }
```

the compiler can generate two stages which execute the adder and multiplier concurrently. Therefore, the execution time is only $42 \cdot \max(d_{\text{add}}, d_{\text{mul}})$ in this case.

It is also possible to define a stage delay when pipelining is used. The pragma *CO SET StageDelay X*, if placed below the CO PIPELINE pragma, sets the maximum stage delay inside the pipeline. This is useful if the stage delay for a particular body is known but Impulse C is not able to calculate a good value.

Furthermore, the stage delays can be controlled with the *co_par_break* function. This function can be used at arbitrary locations inside a loop body that will be pipelined. The statement specifies a *clock cycle boundary*. With every clock cycle boundary a new cycle will be started inside a pipeline. It is therefore possible to further fine-tune the parallelism in the pipeline with this function.

CO NONRECURSIVE A given array is *recursive* if elements of the array are written and then read immediately afterwards. The *CO NONRECURSIVE* pragma tells the Impulse C compiler that an array has the non-recursive property. This property is very important for pipelining. If a loop that is to be pipelined contains access to arrays, the Stage Master cannot optimize the pipeline properly since it is unknown at what point in time the array variables are completely written. When the CO NONRECURSIVE pragma is set, the compiler assumes that all array variables are always in a meaningful state inside the pipeline body.

CO UNROLL Unrolling is possible with the *CO UNROLL* pragma. If a loop is unrolled, the resulting HDL code will duplicate the body of the loop as many times as the loop has iterations. This is therefore only usable with relatively small loops, that have either iterations or a small body.

For this pragma to function the body of the loop cannot include any interdependent calculations or assignments, since this could not be parallelized. Loops that contain arrays can only be unrolled if it is possible for the Impulse C C-to-VHDL compiler to scalarize the array, since otherwise elements could only be accessed one at a time. More details can be found in the user guide of Impulse C.

One application for Impulse C is the implementation of vision algorithms on an FPGA that is connected to cameras. Camera systems that might be used in such a situation are described below.

3.3 Camera System

Another important aspect of the whole system besides the implementation language is the camera system, which is responsible for image capturing. It is a complex system consisting of an optical lens and an image sensor. Different lens parameter and the utilized image sensor affect the shape and the quality of the captured images.

The most common lens parameter and image sensor technologies as well as their properties will be explained in the following section.

3.3.1 Optical Lens

An optical lens is a device which refracts incoming light and transmits it to the backside [Gre04]. Two important parameters of a lens are the *focal length* and the *F-number*. In the following the relevant terms are explained.

The focal length describes a property of a lens, i. e. how strongly the incident light is focused or defocused. For a focusing lens the focal length is positive and the light is focussed to a single spot after the lens. For a defocusing lens this length is negative and the light is diverging after the lens (illustrated in figure 3.5). Therefore, the focussing length is an important parameter of a lens which determines the angle of the view of the camera.

In an optical system the *aperture* is a structure in the light path which controls the amount of light traveling through. Typically an aperture is integrated in a camera lens to control the light amount which reaches the image sensor. Common lenses are marked with a so-called F-number. This number describes the ratio between focal length and aperture diameter. Depending on the camera environment another aperture must be chosen to adapt the camera to the light intensity in the environment.

To attach a lens in front of an image sensor different mounts exist. In case of industrial machine vision commonly *C-Mount* or *CS-Mount* lens mounts are used. A wide range of high quality lenses with different characteristics is available, which is an aspect in favor of this type of lens mount. Both C-

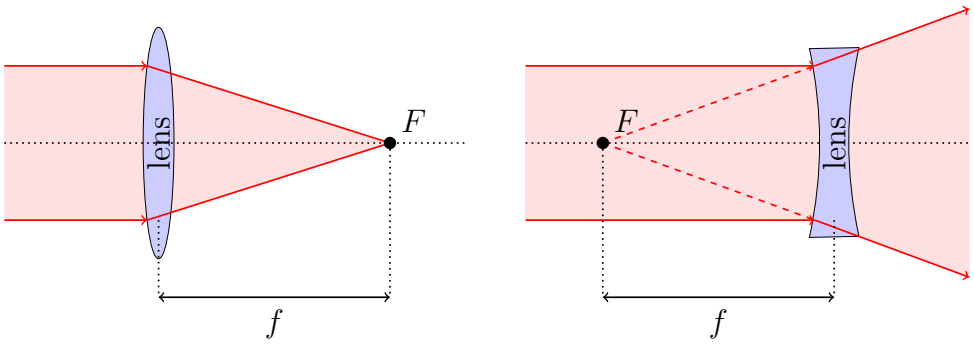


Figure 3.5: Focal length f and focal point F

and CS-Mount have the same diameter of 25.4 mm and only differ in the distance from the lens mount to the image sensor plane (called *flange focal distance*).

A C-Mount lens has a smaller flange focal distance than a CS-Mount. Therefore, a C-Mount lens can be attached to a CS-Mount with an adapter ring. In figure 3.6 a C- to CS-Mount adapter ring and a C-Mount lens is shown.



Figure 3.6: C-Mount lens with C to CS-Mount adapter ring

3.3.2 Image Sensor

An image sensor is a device that converts an optical image into a digital signal. Classic applications for these sensors are video or digital cameras.

Image resolution, frame rate, output format (e. g. color or gray images, RGB or YUV color representation), light sensitivity, shutter type and automatic image control functions such as *Automatic Exposure Control* are typical properties of an image sensor.

Modern sensor technologies are *Charge Coupled Devices* (CCD) and *Complementary Metal Oxide Semiconductors* (CMOS) [Lit01].

Charge Coupled Devices (CCD) sensors consists of a matrix of photodiodes. These photoactive regions accumulate electric charge during light incidence. The electric charge is proportional to the light intensity. After an exposure time the photodiode matrix can be read out. Therefore, in each line of the matrix each photodiode relays its charge to its predecessor. The last photodiode in each line has no predecessor and relays its charge in a capacitor array.

After each line has relayed one charge in this array the first charge can be amplified, measured and converted into a digital value. This value then describes the light intensity of the corresponding photodiode. Afterwards each charge in this array is shifted again such that the next can be converted.

When the whole array is converted, the first line of the image is read out. Then again, each row relays the next charge in the array and the next line can be read out. In this manner a whole image is read out. Afterwards a new exposure is used to charge the photodiodes again and the whole procedure is repeated. Figure 3.7 visualizes this process.

The idea of *Complementary Metal Oxide Semiconductor* (CMOS) image sensors is to create a readout circuit for each pixel. A photodiode or photogate is used to convert the incident light of the pixel into a voltage which is amplified within the pixel circuit. An on-chip analog to digital converter converts the voltage for each pixel into a digital signal.

Thus, in contrast to CCD image sensors, each pixel can be read out independently. This is advantageous for camera functions which operate on specific pixel or subsets of pixel. Those functions are:

Windowing A smaller region than the whole image can be read out. Typically these regions can be defined by the user. The sensor can then

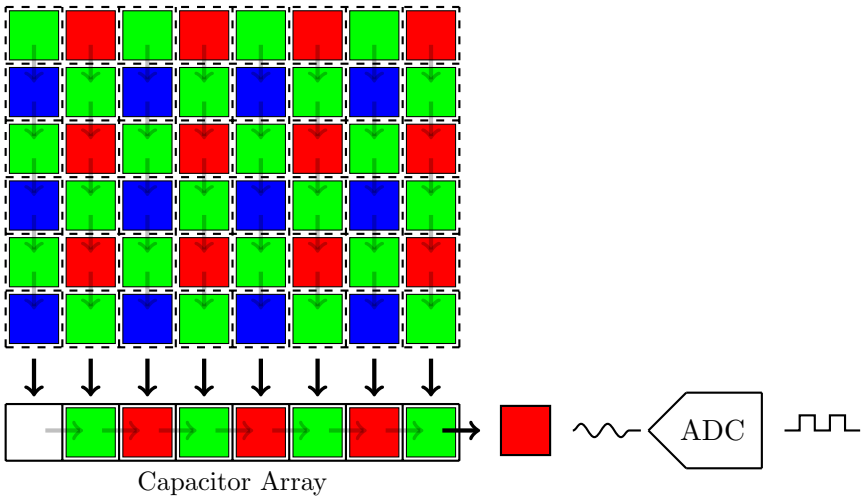


Figure 3.7: CCD image readout procedure: Arrows show the motion of charges and the ADC symbol represents an Analog to digital converter.

deliver this image subset faster. This feature is advantageous for object tracking algorithms which needs a small region of the image with high framerates.

Binning CMOS image sensors can merge several pixels to a single pixel. This leads to a smaller image with lower noise. Typically 2×2 or 4×4 pixels are merged together.

Readout Methods Some CMOS image sensors provide different methods for image readout. For example *Progressive Scan* reads the image line-by-line in a sequential order. *Interlaced Scan* reads first the odd lines then the even ones.

Mirroring Often functions like column or row flip are supported by the sensors.

These functions can be easier implemented by manufacturers of CMOS image sensors than by manufacturers of CCD image sensors.

The presented sensor types have in common that they naturally only can detect light intensity differences. A method to capture color information are *Color Filter Arrays*.

Color Filter Array

To obtain color information from a photodiode array an *color filter array* can be used. It consists of a matrix of small color filters, where each filter is placed over one photodiode. Thus, each diode only records the light passed through the overlying color filter.

One common approach is to use a *Bayer Filter Array [Bay76]*. This Filter is named after Bryce E. Bayer of Eastman Kodak. In case of a bayer filter array 25 % of the filters in the array are red and blue, the other 50 % are green filters. Twice as much green pixel then red or blue are placed since the human eye is more sensitive to green light than to red or blue light.

The structure of a Bayer Pattern is illustrated in figure 3.8.

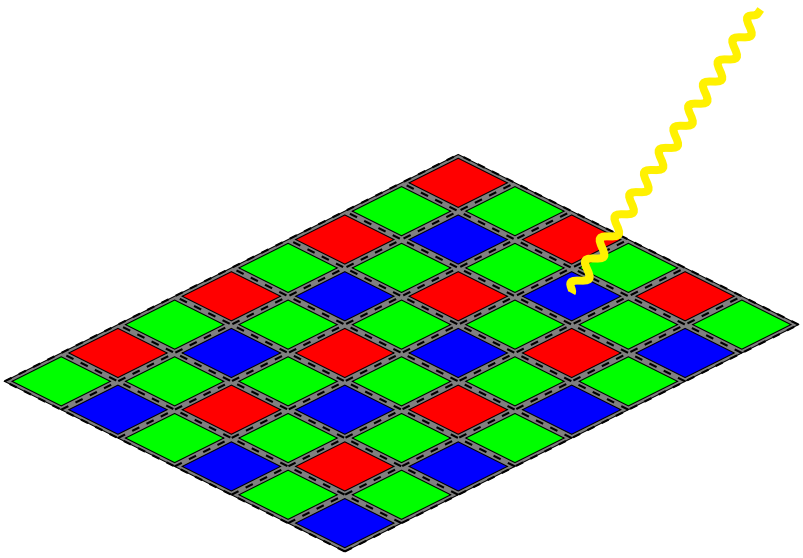


Figure 3.8: Bayer Pattern

Since each pixel can only detect the light intensity difference for one color (red, green or blue) a technique to calculate the RGB-color information is required. The technique is called *Debayering* or *Demosaicing*. This makes it possible to use every photodiode as a pixel, hence the resolution is not reduced. Debayering methods compute the missing color information of one pixel by the color information of its neighbors. One method is *Bilinear Interpolation*.

Bilinear Interpolation is a demosaicing method which uses 3x3 pixels to calculate the missing RGB color values for a pixel [Ram+02]. The bilinear inter-

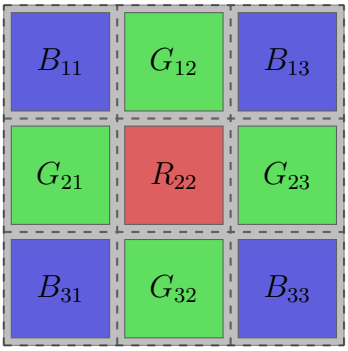


Figure 3.9: Bayer Pattern cut-out

polation formula to compute the RGB color value of pixel position 22 reads (Figure 3.9)

$$\begin{aligned} R'_{22} &= R_{22} \\ G'_{22} &= 1/4 \cdot [G_{12} + G_{21} + G_{23} + G_{32}] \\ B'_{22} &= 1/4 \cdot [B_{11} + B_{13} + B_{31} + B_{33}]. \end{aligned} \tag{3.1}$$

For other pixels this computation can be done analogous. There exist superior and more complex demosaicing methods. In [LGZ08] different modern approaches are compared, which incorporate a bigger pixel neighborhood in the calculations. These approaches lead to better results but have a higher complexity. Hence, they lead to a higher resource usage on a FPGA system.

Shutter

As described in 3.3.2 the photoactive regions have to be exposed for a determined time. This is done by a shutter [Eas03]. A mechanical shutter blinds the photoactive regions to prevent exposure. Another way to prevent light incidence is to use an electronic shutter. Here a circuit deactivates the photodiodes which should not be exposed.

One possibility to implement an electronic shutter for CCD sensors is to reset the photodiodes before a new image. Reset means that all charges are removed from the photodiodes. During the next exposure time new charges are accumulated. After this all charges are simultaneously moved into light

shielded areas on the chip, such that light cannot affect the charges. Then the CCD typical readout procedure is started. At the end all photodiodes are reseted and a new image capture cycle can begin.

Too long exposure time or too powerful light intensity can lead to an effect of CCD sensors, the so called *blooming*. In this case the capacity is exceeded and charge carriers move to adjacent photodiodes. This compromises the photodiodes and the result is an incorrect image with an oversized light spot. Modern consumer CCDs feature *antiblooming*, which means that they use a special design which absorbs excessive charge carriers.

For CMOS image sensors two common electronic shutter techniques exist, so called *Rolling Shutter* and *Global Shutter*. These shutters benefit from the CMOS ability of resetting and reading pixels independently. A Rolling Shutter exposes the lines of the image sensor non-uniformly. Typically the first row of the sensor is reseted, exposed and read out. Then the procedure is repeated for the next line. This can lead to distorted images when fast motions are captured. A Rolling Shutter can be implemented with a small number of transistors and therefore leads to a favorable price. A Global Shutter resets and exposes all pixel of the image sensor uniformly. After exposure the charges are moved into a light-shielded area and then read out sequentially. Therefore, image distortions caused by fast motions or vibrations do not appear. A slobal Shutter however is more complex in implementation and thus more expensive.

Often two cameras are used to capture stereo image, which can be transformed to depth maps with an stereo vision algorithm. In the following one of these algorithms is described.

3.4 Scan-Line based Segmentation

The Scan-Line based Segmentation [SM09] is a stereo vision algorithm. In stereo vision two cameras are placed in different positions to capture the same scene from different angles, similar to human binocular vision. The goal of stereo vision is to obtain relative depth information by comparing the two images.

Disparities are defined as the difference between column and row coordinates between corresponding pixels in the left and right image. Whereby pixels *correspond* to each other, if they are from two different views of the same point in a 3D scene. The task of finding these correspondences is called

correspondence problem. A *disparity map* consists of disparity values for each of the pixels of two images.

Stereo vision algorithms obtain disparity maps, which are inversely proportional to the distances to objects in the scene and can hence be used to calculate the desired depth information.

In general stereo vision algorithms can be classified with the taxonomy described in [SS01a]. They are divided in the two categories *global methods* and *local methods*.

Global methods include *dynamic programming* (DP) [LSY06], *belief propagation* (BP) [KSK06] and *graph cuts* (GC) [KZ01]. These methods generally will yield good results, but they are computationally intensive.

Local methods such as *census transformation* (CT) [ZW94], *sum of absolute differences* (SAD) and *sum of squared differences* (SSD) [Tom+08] on the other hand, can often be executed fast and are parallelizable. The three named algorithms operate on a window of size $n \times n$ such that they can operate on stereo images with a maximum disparity of n.

The Scan-Line based Segmentation is a local method. It uses rows of images, so called scan-lines, as its local domain. This allows bigger disparities than a window based algorithm. To be able to yield good results with this kind of local domain some constraints are imposed. These constraints are described in the following.

3.4.1 Constraints

In order to perform the estimation of the disparity map many times per second, some assumption upon the input data have to be made. The Scan-Line based Segmentation algorithm has two geometric constraints that are assumed to be adhered by the input stereo images and the image scene. Former is assumed to be rectified and the scene has a smoothness constraint.

Rectified Images

The first assumption that the Scan-Line based Segmentation uses are rectified stereo images as input. This means, that the input images must have a common image plane, in which the *epipolar lines* align horizontally [Ora01].

Let O_l and O_r be the focal points of the left and the right camera. Each focal point can be seen on the image plane of the other camera, these are the *epipolar points* e_r and e_l. Let furthermore x_l and x_r be points in the left and right image plane that are corresponding to point X in the real scene (seen by both cameras). The emerging lines $e_\mathrm{r} - x_\mathrm{r}$ and $e_\mathrm{l} - x_\mathrm{l}$ are the epipolar lines [HZ04]. A graphical representation of this can be seen in figure 3.10. This view of the scene is called epipolar geometry.

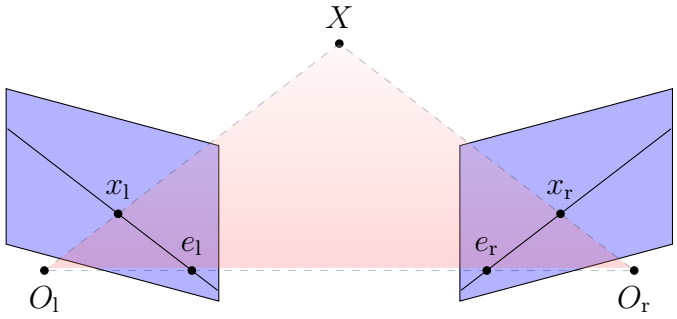


Figure 3.10: Epipolar geometry: Two cameras (blue rectangle), focal points (O_l, O_r), point in real scene (X), points on left and rigt image plane (x_l, x_r), epipolar points (e_l, e_r) and epipolar lines (between e_r, x_r and e_r, x_r).

Thus, the rectified image assumption can be expressed with the equation

$$e_\mathrm{r} - x_\mathrm{r} = e_\mathrm{l} - x_\mathrm{l}. \tag{3.2}$$

Which means that both cameras have to face perfectly parallel at the scene, when the images are captured. The epipolar lines on the image planes are at the same height in the image. Projected objects from the scene have a horizontal disparity (figure 3.11). This requirement cannot be completely satisfied with the hardware that is used in our system (see section 5.4 for detail).



Figure 3.11: Rectified epipolar geometry: Epipolar lines on height y in both image planes with horizontally moved objects from the scene.

Rectified stereo images are needed, since the Scan-Line based Segmentation tries to match belonging epipolar lines of both images. If the stereo images were not rectified, the algorithm would have to search in two dimensions instead of the one dimensional scan-line. This leads to a too high run-time complexity.

Besides the need for rectified images the Scan-Line based Segmentation has a smoothness condition described in the following.

Smooth Scenes

The second assumption is based upon the scene for which the disparity map should be estimated. It is assumed that the disparity map of the two stereo images is smooth for each object in the scene and that there are differences at the border of objects.

This is not always the case, particularly because the Scan-Line based Segmentation uses color information to differentiate between objects. An object and a background that have a similar color may be classified as one single segment in the segmentation phase of the algorithm. The subsequent disparity map for this part of the image will be flat and therefore not show the object.

Furthermore, an object might have patterns, such as a checkerboard pattern. This can lead to a wrong matching, since one segment has multiple possible matching candidates in the pattern. An example of this is the periodic table in the background of figure 3.16. The problem can be circumvented by using a smoothing filter to smooth out the irregularities in objects. This filter must however not blur the borders of objects. A filter that can ensure this property is discussed in section 3.5.4.

The smoothness assumption is important in a line segmentation, since otherwise objects with different depth in the scene might be segmented into the same segment. The operation of the line segmentation that is utilized in the Scan-Line based Segmentation is described below.

3.4.2 Line Segmentation

In the first phase the Scan-Line based Segmentation performs a line by line segmentation of the right input image $\mathfrak{I}_\mathrm{r}$.

The strategy for this is to start at the first pixel of the epipolar line epl_y^{r} from row y of $\mathfrak{I}_{\mathrm{r}}$ and measure the color difference to each of the neighbor pixels. If the color difference is below a threshold value τ, the pixel is added to the segment. On the other hand, if the color difference exceeds τ, a new segment is created.

To calculate the color difference between two neighboring pixels $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2) \in \mathfrak{R}$, with $\mathfrak{R}$ being the RGB color space, the function

$$\begin{aligned} C(x_1, y_1, x_2, y_2) \;=\; & |I_{\mathrm{R}}(x_1, y_1) - I_{\mathrm{R}}(x_2, y_2)| + \\ & |I_{\mathrm{G}}(x_1, y_1) - I_{\mathrm{G}}(x_2, y_2)| + \\ & |I_{\mathrm{B}}(x_1, y_1) - I_{\mathrm{B}}(x_2, y_2)| \end{aligned} \tag{3.3}$$

is used. Whereby $I_{\mathrm{R}}(x, y)$, $I_{\mathrm{G}}(x, y)$ and $I_{\mathrm{B}}(x, y)$ specify the intensity of a pixel (x, y) in the red, green and blue color channel of $\mathfrak{R}$.

For the segmentation a label space Γ is defined in which different label sets $l_i(epl_y^{\mathrm{r}}) \in \Gamma$ for each segment in the epipolar line epl_y^{r} can be specified. For $(x, y) \in l_i(epl_y^{\mathrm{r}})$ this means, that the pixel (x, y) is in segment i of the epipolar line epl_y^{r}.

With this the segmentation can be defined as

$$\begin{aligned} (x_{\mathrm{s}} + k, y) \in l_i(epl_y^{\mathrm{r}}) \quad \forall k, l \in \mathfrak{L}_{x_{\mathrm{s}}} \quad & \text{with} \quad C(x_{\mathrm{s}}, y, x_{\mathrm{s}} + k, y) < \tau \\ & \text{and} \quad k < l \;\forall\; C(x_{\mathrm{s}}, y, x_{\mathrm{s}} + l, y) \geq \tau, \end{aligned} \tag{3.4}$$

where x_{s} is the start or seed pixel of the segment and $\mathfrak{L}_{x_{\mathrm{s}}} = \{1, \ldots, w - x_{\mathrm{s}}\}$ is the set of the remaining pixels in the processed epipolar line epl_y^{r} with width w. Figure 3.12 illustrates this procedure.

Segments in the label sets $l_i(epl_y^{\mathrm{r}})$ are used in the correspondence phase to find a match for each segment in the corresponding epipolar line of the left image $\mathfrak{I}_{\mathrm{l}}$.

3.4.3 Correspondence

The second step in the Scan-Line based Segmentation is to find a good matching between the segment $l_i(epl_y^{\mathrm{r}})$ of the right image $\mathfrak{I}_{\mathrm{r}}$ and the corresponding epipilar line epl_y^{l} in the left image $\mathfrak{I}_{\mathrm{l}}$.

The segment $l_i(epl_y^{\mathrm{r}})$ is slided by one pixel over the corresponding line epl_y^{l}. In each sliding step the summed up color difference is calculated and the sliding position with the lowest color difference is identified as the best matching.

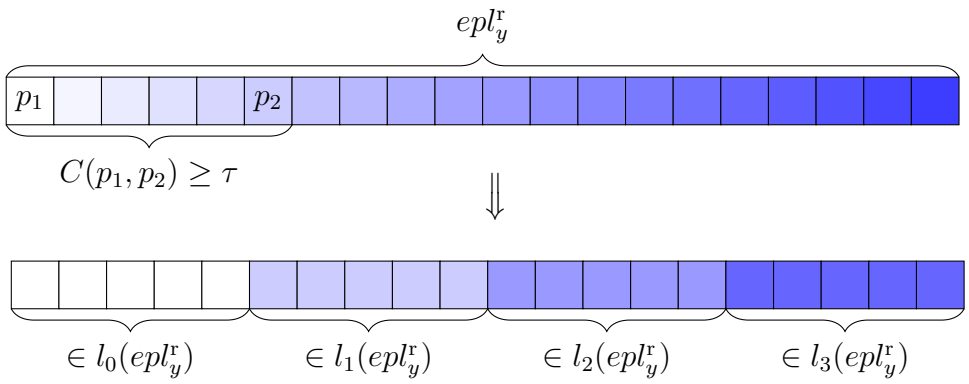


Figure 3.12: Line segmentation: Epipolar line epl_y^r is divided into four segments $l_0(epl_y^{\mathrm{r}})$ to $l_3(epl_y^{\mathrm{r}})$, color distance threshold τ is exceeded for the color difference of pixels $p_1 = (x_1, y)$ and $p_2 = (x_2, y)$.

The horizontal pixel distance between the original segment position in epl_y^{r} and the best matching position in epl_y^{l} is recorded as the disparity value for this segment in the disparity map D. An illustration of this approach is shown in figure 3.13.

Since it is not practical, i. e. it is to slow, to slide each segment over the whole corresponding epipolar line, a maximum search distance d is used. This value should be set to the maximum horizontal pixel distance between the same object in the left and the right image. Hence, the value of d depends upon properties of the used cameras and the observed scene.

If the cameras, placed in a horizontally parallel position to each other, have a big distance between them, the maximum distance between the same object in the left and in the right image will be big. On the other hand, if the camera distance is small, the object position difference in the images will be small.

Likewise if an object in the scene is far away from the camera, the pixel distance between the left and right image will be small. Whereas the distance will be big, if an object is near to the camera.

The level of detail in the disparity map depends upon the average distance between the same objects in the left and right image. Therefore, a trade-off between the position of the camera in the scene, the distance between the cameras and the emerging complexity in the Scan-Line based Segmentation

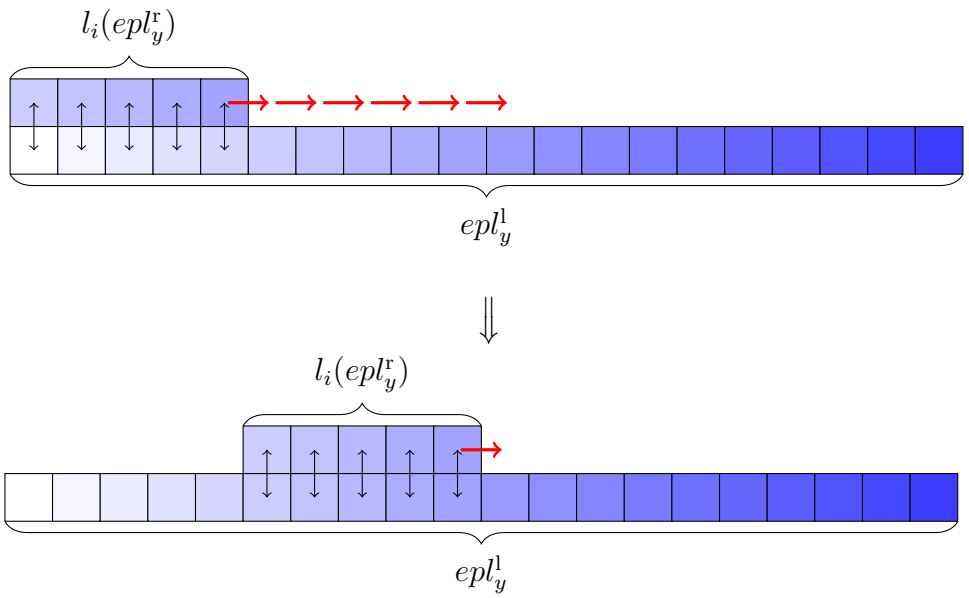


Figure 3.13: Correspondence: A segment from the right image ($l_i(epl_y^{\mathrm{r}})$) is matched on the corresponding epipolar line from the left image (epl_y^{l}), the black arrows show for which pixels the color distance is determined and the red arrows show the sliding path of the segment.

has to be made. Details on the setup that is used in our system are described in section 5.

When a good value for the maximum search distance d is determined, the disparity map D can be defined for each segment $l_i(epl_y^{\mathrm{r}})$ as

$$D(l_i(epl_y^{\mathrm{r}})) = \underset{0<i\leq d}{\arg\min} \left(\sum_{(x_1,y_1)\in l_i(epl_y^{\mathrm{r}})} C(x_1, y_1, x_2 + i, y_2) \right) \text{ with } (x_2, y_2) \in epl_y^{\mathrm{l}} \text{ corresponding to } (x_1, y_1), \quad (3.5)$$

whereby

$$D(l_i(epl_y^{\mathrm{r}})) = X \Leftrightarrow \forall (x, y) \in l_i(epl_y^{\mathrm{r}}) : D(x, y) = X. \quad (3.6)$$

Function C is the color difference function as defined in formula 3.3 in section 3.4.2. In the following four image filter that are able to enhance the input images such that they are more suitable the Scan-Line based Segmentation are discussed.

3.5 Image Filter

Image filter are a set of algorithms that allow to modify images in a specific manner. This modifications can be edge enhancements, warps, blurs, etc. Image filter can operate on intensities or on spatial information or on both at the same time. In [GW92] a wide variety of different intensity transformation filters and spatial filters is presented.

The Scan-Line based Segmentation requires a disparity map that is smooth almost everywhere except at the borders of objects (see section 3.4.1 for details). This means, that input images with dotted or striped parts or blurred object borders will produce flawed disparity maps.

Therefore, an image filter that can blur minor details but does preserve sharp borders is needed. Four different filters that might have a usable quality for the Scan-Line based Segmentation are presented. These also must have an acceptable time complexity to run in real-time on a robot and have an acceptable space complexity to fit on an FPGA.

The tested filters are *Gaussian*, *Median*, *Hybrid Median* and *Bilateral Filter*. The first two are standard blurring and noise reduction filters that do not preserve object borders. The Hybrid Median Filter is a modified Median Filter that does not completely blur transitions from one color to another. A Bilateral Filter on the other hand has a blurring and an edge enhancing part. It can be fine tuned to compute images that are of good quality to be used in the *Scan-Line based Segmentation*. It also has a high run-time complexity in its standard form.

One image filter that comes to mind in regard to the demands of the Scan-Line based Segmentation, but is not described, is the *Anisotropic Diffusion*. It is a noise reducing filter that does not remove significant parts from the image [Bla+97]. The Anisotropic Diffusion uses lots of floating point operations in one filter run and it needs a great deal of iterations to yield good results. Hence, it could a priori be deemed unfitting for an FPGA.

In the following the different filters are described and there run-time and space complexity is discussed.

3.5.1 Gaussian Filter

The Gaussian Filter for image processing or Gaussian Blur is a widely used image filter to reduce noise and detail. Gaussian filtered images are blurred

as if the images were views through an out-of-focus lens. The Gaussian Filter can be described as

$$G_\sigma(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{3.7}$$

where x is the horizontal distance from the origin pixel, y is the vertical distance from the origin pixel and σ is the standard deviation of the Gaussian distribution.

With the values that are calculated by $G_\sigma(x, y)$, the neighbor pixels of the origin pixel are multiplied and summed up to replace the origin pixel. In theory one would have to use every pixel of the image, but since the values of $G_\sigma(x, y)$ are sufficient small for $|x| > 3\sigma$ or $|y| > 3\sigma$ to have no effect, a filter kernel of size 6σ is big enough to provide good results [NA08].

For an implementation on the FPGA the Gaussian Blur possess two problems: The number of entries in the filter kernel might be to big to be stored in block RAM and it uses floating point numbers that can't directly be used on the FPGA.

The former can be counteracted. Since the Gaussian Blur is linearly separable, it is possible to apply the filter in two one-dimensional phases. In the first phase the image is only blurred in the horizontal or vertical direction and in the second phase the image is blurred in the other direction. In both passes only a one-dimensional filter kernel is needed. This reduces both, run-time complexity and space needed to store the filter kernel in block RAM.

Besides the Gaussian Filter it is possible to also use a Median Filter to reduce noise. The operation of the Median Filter and differences to the Gaussian Filter are discussed below.

3.5.2 Median Filter

The Median Filter is a noise reducing image filter. It is particularly good at removing *salt and pepper noise*, i. e. randomly occurring white or black pixels [Lim90].

The idea of the Median Filter is to use a filter window that slides over the image. For each step the intensities of the pixels in the window are sorted and the median of the sorted window is picked to replace the center pixel of the window.

It can easily be seen that there are no floating point operations, or arithmetic operations at all for that matter, needed to implement a Median Filter. Merely a sorting function and space to store the elements of the filter window is necessary. However, it will not blur out a checkerboard pattern or stripes and is therefore not optimal for our purpose.

Since we need a big filter window to get good filter results for the Scan-Line based Segmentation, a big number of intensity values have to be stored and sorted. Neither is possible on the FPGA, since there is not enough block RAM to save that many values, nor is the complexity of $\mathcal{O}(\log(r)r^2)$ to sort the $4r^2$ values in a filter window with radius r acceptable.

Since the Median Filter is a filter that is used in several image processing fields, lots of research is done on optimization possibilities, e.g. the linear optimization described in the following.

Linear Optimization

In [Hua81] an approach to Median Filtering that lowers the time complexity from $\mathcal{O}(\log(r)r^2)$ to $\mathcal{O}(r)$ for a filter window with radius r is discussed. The original run-time stems from the time that is needed to sort the intensity values in the filter window.

By using a histogram to store the content of the filter window a significant improvement can be made. For the first pixel, the histogram is filled with the intensity values of the filter window. In the subsequent steps the histogram has not to be newly filled, it can be updated by subtracting the intensity values that are not longer in the filter window and adding the new ones.

This takes only $2r + 1$ subtractions and additions per pixel. The median value can be extracted from the histogram by summing up the number of appearance of each intensity in the histogram starting by intensity 0. If the counting number exceeds $\frac{4r^2}{2}$, which is half of the number of pixels in the filter window, the median value is found.

Let i be the counter for the histogram entries and $H(i)$ be the number of pixels with intensity i in the histogram, then i is the median intensity if

$$\sum_{j=0}^{i-1} H(j) < \frac{4r^2}{2} \quad \text{and} \quad \sum_{j=0}^{i} H(j) \geq \frac{4r^2}{2}. \tag{3.8}$$

Identifying the median value takes at most 2^d steps, where d is the color depth of the used images. This number of steps is independent from the used window size.

The histogram idea can be used in a hierarchical manner to achieve a better than linear run-time for the Median Filter.

Constant Optimization

An implementation of the Median Filter with a time complexity of $\mathcal{O}(1)$ is introduced in [PH07]. It uses the fact, that histograms are distributive, i. e.

$$H(A \cup B) = H(A) + H(B). \tag{3.9}$$

for disjoint image regions A and B. Since the number of elements in the histograms are independent from the filter window size that is used in the Median Filter, this operation has a time complexity of $\mathcal{O}(1)$.

The proposed algorithm has a histogram for each column. Updating of the histograms is done via a filter window, as in the $\mathcal{O}(r)$ implementation, but only for the effected histograms. Together with the distributivity property of the histograms a complexity of $\mathcal{O}(1)$ can be achieved.

A time complexity of $\mathcal{O}(1)$ for the Median Filter is very unexpected, since in [GW93] it is proven that the lower bound for the Median Filter is $\mathcal{O}(\log r)$, with r being the radius of the filter window. The proof uses the fact that the time complexity of an *comparison sort algorithm* is not lower then $\mathcal{O}(n \log n)$. However, the scheme as proposed in [PH07] does not use comparisons to sort the values and there is no such lower bound for non comparison sort algorithms. Thus, the proof in [GW93] has an erroneous assumption and the lower bound for Median Filtering can be revised to $\mathcal{O}(1)$.

With this method, the Median Filter is faster than the $\mathcal{O}(\log r)$ implementation starting from a filter window radius of 60. Even though we need a big filter window, this might be too big. This is discussed in detail in section 7.

There are Median Filter modifications which use shapes different from the square filter window and are thus not compatible with the histogram optimization ideas. One of these filters is described in the next section.

3.5.3 Hybrid Median Filter

The Hybrid Median Filter – like the Median Filter (see section 3.5.2) – removes noise from the input image. It does however have better edge preserving properties than the standard Median Filter [MVE05].

There are three steps to calculate the hybrid median value. Each step involves a filter window with a different shape. Instead of the square filter window, one filter with a +-shape and one filter with a x-shape is used (see figure 3.14).

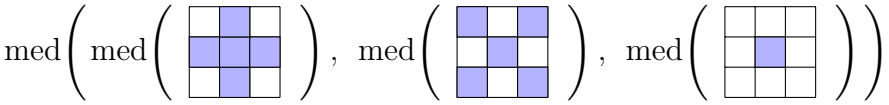


Figure 3.14: Hybrid Median operation: At first the median of the blue pixels in the +-shaped filter window is determined, than the median of the x-shaped filter window is determined and in a last step the median of the two former medians and the center pixel is calculated to generate the Hybrid Median value.

With these varying shaped filter windows, the median value is calculated as before in the standard Median Filter by sorting the values in the filter window and picking the median of the sorted values.

After this is done, the median of the +-shape median, the x-shape median and the center pixel is determined. This median of medians is the result of the Hybrid Median Filter [HN87]. As shown in figure 3.15 the Hybrid Median Filter can be constructed with arbitrary odd filter window sizes.

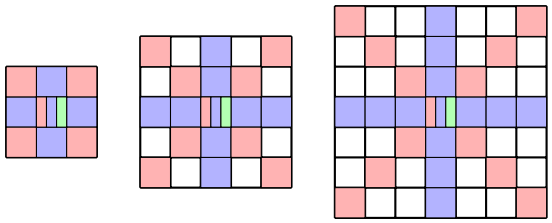


Figure 3.15: Hybrid Median sizes: The Hybrid Median Filter can be used with arbitrary odd filter sizes. Each color represents a different filter window. The Hybrid Median value is the median of the medians of each of the filter windows.

At a first glance the Hybrid Median Filter seems like a good candidate for the preprocessing step of the Scan-Line based Segmentation. It reduces noise and preserves borders. These properties are both needed to get good results in the segmentation, as discussed in section 3.4. Also there is no need for floating point operations or the like.

The time complexity of the obvious implementation is with $\mathcal{O}(r \log r)$, for sorting the $\mathcal{O}(r)$ values (two times), relatively high. The histogram idea in the $\mathcal{O}(r)$ or $\mathcal{O}(1)$ optimization of the standard Median Filter, as described in sections 3.5.2 and 3.5.2, is not applicable here since the histograms cannot easily be updated when the filter window moves. Not only the outer most pixels are changed, but also most of the inner pixels of the +-shaped and x-shaped filter windows are altered when the window is moved.

Because of this, a maximum filter size of 7×7 is possible if the real-time constraint has to be adhered. Filter windows with these sizes might however be to small to provide filtering as it is needed by the Scan-Line based Segmentation. Comparisons of this and the other filters can be seen in chapter 7.

A filter that has even better edge preserving properties than the Hybrid Median Filter but is optimizable in the same fashion as the Median filter is described in the following.

3.5.4 Bilateral Filter

Bilateral filtering can smooth images while preserving edges, it was first described by Tomasi et al. in 1998 [TM98]. This is exactly what is needed in the Scan-Line based Segmentation, which requires a disparity map that is smooth almost everywhere except at borders.

The filter has two functions in order to be able to perform smoothing and edge enhancing in one run. While the smoothing is done with a geometric distance weighting function g, the edge enhancing part uses an intensity distance weighting function f.

For the functions f and g arbitrary weighting functions can be used, e. g. a Gaussian function or a linear function. This is crucial for our system, since this allows us to fine-tune the filter for our needs. It is for instance possible to use a Gaussian function with a small σ for the smoothing function g and a Gaussian function with a big σ for the edge enhancing function f, to achieve a small amount of blurring with sharp edges. This together with a relatively large filter kernel would be a good setup for our system.

Let (x, y) be a coordinate of a point in the image, $I(x, y)$ the intensity of (x, y) and $I'(x, y)$ the resulting intensity after filtering. The Bilateral Filter can then be described as

$$I'(x,y) = n^{-1}(x,y) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(\xi,\psi) f(I(\xi,\psi) - I(x,y)) g(\delta((\xi,\psi),(x,y))) \,\mathrm{d}\xi \,\mathrm{d}\psi, \tag{3.10}$$

whereas n is the normalization

$$n(x,y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(I(\xi,\psi) - I(x,y)) g(\delta((\xi,\psi),(x,y))) \,\mathrm{d}\xi \,\mathrm{d}\psi \tag{3.11}$$

and δ is the geometric distance

$$\delta((\xi,\psi),(x,y)) = \sqrt{(\xi-\psi)^2 + (x-y)^2}. \tag{3.12}$$

The (ξ, ψ) depict nearby points of (x, y) that are used to determine values corresponding to the devised weighting functions f and g.

Operation

From formula (3.10) it can be derived that a filtered value is dependent upon the difference of intensity as well as the geometric distance between the looked at pixels. This means that an actual discretized implementation of the Bilateral Filter is a convolution that combines two different weighting functions.

In figure 3.17 three cut outs from the example image in figure 3.16 with their corresponding filter kernel are shown.

The first cut out is from the background and shows a uniformly grey image, hence the corresponding filter kernel is a simple Gaussian filter kernel (with white depicting high values). The Gaussian look comes from the geometric distance weighting function, whereas the color distance function does not have any influence in this cut out since the color is nearly equal everywhere.

In the second cut out one can see a sharp edge with a big color difference. The filter kernel is therefore a Gaussian Filter kernel with an edge at the same location. This edge is there because the color distance weighting function weighted the pixels with a low value since there is a high color distance between the two sides of the edge. Therefore, the resulting bilateral filtered



Figure 3.16: Example image: Convolution examples in figure 3.17 are taken from the black spots in the teddy example image from the 2003 Middlebury dataset [SS03].

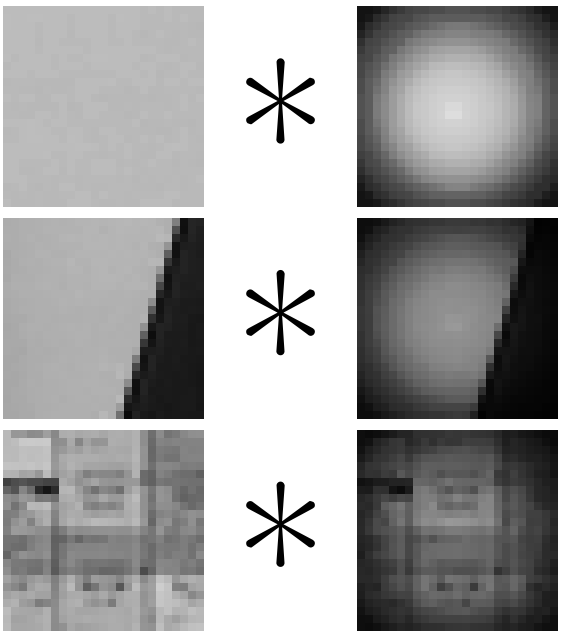


Figure 3.17: Bilateral filter convolution: Left column shows cut outs from figure 3.16, right column shows corresponding convolutions for smoothing and edge enhancing part of Bilateral Filter

pixel will only contain very little color information of the pixels right from the edge, hence the edges persist in the filtered image.

The third image is a complex image with lots of different color intensities, the Bilateral Filter will most likely completely blur all the small edges and the inscription, which is a good approach for the Scan-Line based Segmentation.

The discretization of the Bilateral Filter looks like

$$I'(x,y) = \frac{\sum\limits_{(\xi,\psi)\in\Omega} I(\xi,\psi) f(I(\xi,\psi) - I(x,y)) g(\delta((\xi,\psi),(x,y)))}{\sum\limits_{(\xi,\psi)\in\Omega} f(I(\xi,\psi) - I(x,y)) g(\delta((\xi,\psi),(x,y)))}, \tag{3.13}$$

where Ω is the set of pixels that are in the filter kernel radius r of (x,y).

An straight forward implementation of this formula has a run-time complexity of $\mathcal{O}(r^2)$, which might be to slow to run in an adequate speed on the FPGA.

Optimization

In [Wei06] an optimization idea for the Bilateral Filter is introduced. It is based upon the histogram optimization idea for the Median Filter (see section 3.5.2). At first the median optimization from [Hua81] is improved from $\mathcal{O}(r)$ to $\mathcal{O}(\log r)$. A hierarchy of histograms together with the distributivity property is used to achieve this run-time complexity. This result was later improved in [PH07] to $\mathcal{O}(1)$, as already described in section 3.5.2.

The idea that makes it possible to carry the histogram idea from the Median Filter over to the Bilateral Filter, is to use the special case of a spatial box filter for the distance weighting function. When a spatial box filter is used, the distance becomes constant for a given intensity. Because of the build-up of the discrete Bilateral Filter formula (3.13), the complete distance weighting term can be canceled out.

Since the resulting formula

$$I'(x,y) = \frac{\sum\limits_{(\xi,\psi)\in\Omega} I(\xi,\psi) f(I(\xi,\psi) - I(x,y))}{\sum\limits_{(\xi,\psi)\in\Omega} f(I(\xi,\psi) - I(x,y))} \tag{3.14}$$

has only terms that can be read out of an histogram from the filter kernel, all optimizations that have been done for the Median Filter can now be applied to the Bilateral Filter as well. The exact transformation of this formula to a histogram based formula is described in detail in section 8.2.

With the use of a spatial box filter instead of a Gaussian distance weighting function a part of the blurring potential of the Bilateral Filter is lost. However, the spatial box filter does approximate a Gaussian after multiple consecutive executions [Jah02]. This might therefore be sufficient for our purposes, since a totally blurred out image is not needed in the Scan-Line based Segmentation.

The Stereo Vision System in which one of the four described filters is utilized, has several requirements on general properties of the system and the used hardware. In the following these requirements are described. Some of the requirements, such as the desired speed of the Stereo Vision System are applicable on the used image filter.

Chapter 4

Requirements Analysis and Hardware Selection

The Stereo Vision System has to cope with different requirements such as flexibility, cost effectiveness and speed, which affect the required hardware as well as the system design. Since the development of an FPGA board is not a task of this thesis, an available FPGA development board has to be chosen which satisfies our requirements. As well, cameras for image capturing has to be chosen.

The following sections explain the requirements and give an overview over existing development boards and cameras. At the end of every section a decision is made which development board or camera should be used.

4.1 System Requirements

The Stereo Vision System should be *flexibly usable* in various applications. Designed RTL System components should be generic, i. e. parts should be usable for other projects or in another context. Often a vision system programmer wants to implement vision algorithms but is preoccupied with system environment tasks such as creating interfaces to data storage, data input or data output. These tasks are time consuming and can lead to a failing project. Therefore, one of our goals is to develop a system which gives the programmer the ability to implement the desired algorithms while using our interfaces to access data storage, getting image data and sending the results to a PC.

As mentioned, creating own FPGA hardware is not a task of this thesis. Thus we have to choose an existing development board. The offered boards differ in capabilities and prices. We want to choose a platform which is capable of computing complex algorithms and is *cost effective*. Another cost-intensive domain in designing hardware can be the use of IP Cores. There exist many companies which create IP Cores for commercial purposes. However, most cores that might be interesting for our system are expensive. A price of 1,000 $ or above for an complex IP Core is not an exception. Therefore, only free and open source IP Cores can come into consideration.

In many cases one reason to implement algorithms on an FPGA instead on a processor system is *speed*. FPGAs normally have a slower clock rate than modern processor systems but can process data highly parallel. Thus the used algorithms have to be optimized to get a high efficiency on an FPGA.

Due to flexibility of the system no speed characteristic of the used vision algorithm can be considered, i. e. it is not known how fast the Vision Core processes the camera images. Therefore, it cannot be assured that the implemented algorithms are fast enough to process the captured images without caching. Likewise it is not guaranteed that the Vision Core only needs image data in a sequential order.

Thus, it is necessary to store complete images and provide the Vision Core with access to the data such that each pixel can be accessed independently. Stereo vision algorithms need access on data of two simultaneous captured images. In order to assure data *consistency* a good option is to access and store both camera images together.

Modern FPGAs feature internal block RAMs but these are to small to store two complete camera images. Hence, a requirement for the FPGA development board is that the board has an *external memory* which is large enough to cache the incoming images.

An *high speed interface* is needed to transmit camera images and computation results to a PC. Two interfaces are widespread: USB and Ethernet. Since we don't want that communication is a bottleneck we have to choose an interface with a high data transfer rate. The transmission of camera images and vision algorithm results in parallel would be a desirable feature. We can estimate the needed data transfer rate by the transmission of two camera images with a resolution of 640×480, 24 bit color and 25 fps. In the case of a stereo algorithm a resolution of 640×480 pixel, 8 bit disparity information per pixel and 25 results per second. This estimation includes enough reserves for other applications.

This estimation implies:

$$\begin{aligned} D_{\mathrm{r}} &= 2 \cdot D_{\mathrm{r}_{\mathrm{image}}} + D_{\mathrm{r}_{\mathrm{disparity}}} \\ &= 2 \cdot 640 \cdot 480 \cdot 24\mathrm{bit} \cdot 25\tfrac{\mathrm{frames}}{\mathrm{s}} + 640 \cdot 480 \cdot 8\mathrm{bit} \cdot 25\tfrac{\mathrm{frames}}{\mathrm{s}} \\ &\approx 410\,\mathrm{Mbit/s}, \end{aligned} \tag{4.1}$$

where D_{r} is the overall data rate measured in megabit per second and $D_{\mathrm{r}_{\mathrm{image}}}$ and $D_{\mathrm{r}_{\mathrm{disparity}}}$ are the data rates for the two input images and the disparity map respectively.

Thus, a Full-Speed USB 2.0 device interface with 12 Mbit/s is not sufficient. The minimum requirement for USB is a High-Speed USB 2.0 device interface with 480 Mbit/s to transmit the data. A gigabit Ethernet interface is more tolerant and is therefore preferred.

FPGAs, like other ICs, get replaced by newer versions over time. The hardware which will be chosen should not be discontinued in the near future and should be *available* for a long time.

The *cameras* for image input that come into consideration have to be accessible by the development board and usable in a robotic environment. Hence, the development board needs a *hardware interface* that is able to interact with the cameras. Refer to section 4.3 for details on the used cameras and their interface.

For the most stereo vision algorithms a *parallel mount* of the used cameras with a fixed distance between the two cameras lenses is necessary. Therefore, the mounting of the cameras has to assure that the cameras stay in parallel and in the defined distance. Ideally this mount would allow to adjust the distance between the two cameras.

4.2 Development Board Selection

This section gives an overview over different FPGAs and their characteristics and features. Afterwards different development boards are presented and compared. At the end a development board for our Stereo Vision System is chosen.

4.2.1 General

Before a development board can be selected, an FPGA that fits our requirements (see section 4.1) has to be chosen. FPGAs differ in price, number of slices/logic elements[1], size of integrated block RAM, hardware modules for I/O, number of I/Os, DSPs and integrated processors.

Primarily, there exist two main manufacturers of FPGAs, Altera and the current market leader Xilinx. In the following sections an overview over the FPGAs and development boards is given.

4.2.2 Altera

Altera offers three different FPGA device families[2].

The Stratix family is a high-end FPGA family with high I/O performance, large number of logic elements, integrated memory and other features. The newest device is the Stratix IV which was introduced in 2008. It is a 40 nm device with high density (up to 820 k logic elements) and low power consumption. Some devices have integrated transceivers for high speed communication applications.

The Arria family is a midrange FPGA family. The newest series called Arria II was introduced in 2009, uses a 40 nm technology and has up to 256 k logic elements. It is designed for performance intensive systems which are cost- and power-intensive as well. For high speed communications such as gigabit Ethernet the Arria II GX series has integrated transceivers.

For low-cost systems, Altera offers the Cyclone family. The newest member of this family is the Cyclone IV. It is a 60 nm low-power device introduced in 2009 and offers up to 150 k logic elements. A version with integrated transceivers exist as well, called Cyclone IV GX.

Development Boards

A wide range of development boards are offered by Altera or other companies[3]. These boards are based on Altera FPGAs and assembled with different

[1] manufacturer dependent term, Xilinx: slices, Altera: logic elements

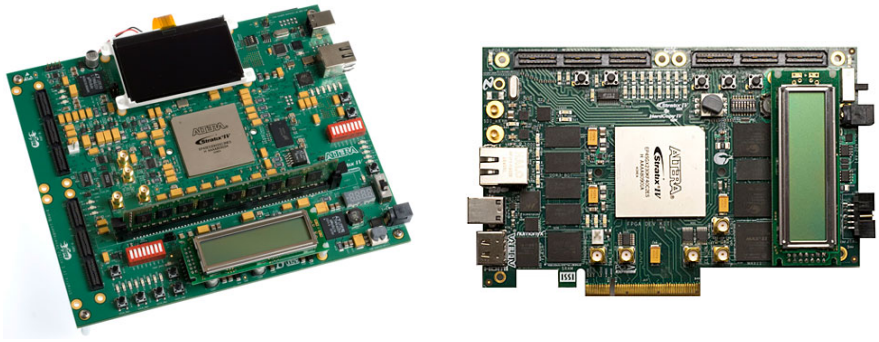
[2] http://www.altera.com/products/devices/

[3] http://www.altera.com/products/devkits/

secondary hardware like USB or Ethernet interfaces, general purpose I/Os and memory. A selection of these boards is presented.

Stratix development boards are equipped with a wide range of interfaces, displays and memory. The cheapest price for these boards is around 2,500 $ for a Stratix III development kit. This kit include some software and a board equipped with gigabit Ethernet, USB 2.0, two displays, 128 MB DDR2 SDRAM and a FPGA with 142 k logic elements. This board seems not to be obtainable.

The successors are the Stratix IV GX with a 124 k FPGA for round about 4,500 $ and the Stratix IV E development kit with 531 k logic elements for round about 10,000 $. Both included boards have a gigabit Ethernet interface, LCD displays and a several different memory types. Figure 4.1 shows these boards.



(a) Stratix IV E (b) Stratix IV GX

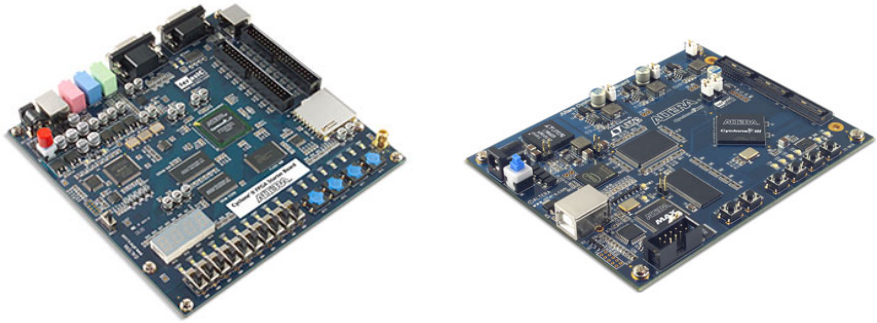
Figure 4.1: Altera Stratix development boards[4] [5]

The Arria II GX development kit contains a board with a 124 k logic elements Arria II GX FPGA. It is a PCI-Express board and costs round about 1,500 $. It is equipped with 128 MB DDR3 RAM, 1 GB DDR2 SO-DIMM, PCI Express interface, LCD display and a gigabit Ethernet port.

Low-priced development kits are equipped with the Cyclone family. For the new Cyclone IV currently no kit is offered. Older Cyclone kits have a low logic element count and slow communication interfaces. One example is the 200 $ Cyclone II Starter Development Kit. This kit is equipped with a

[4] http://www.altera.com/products/devkits/altera/images/stratix-ive-devki.jpg
[5] http://www.altera.com/products/devkits/altera/images/stx4gx-fig1-devboard.jpg

20 k logic element FPGA has a 8 Mbit SDRAM and no high speed communication interface. Another example is the 200$ Cyclone III FPGA Starter Kit equipped with a 24 k FPGA, 256 Mbit DDR SDRAM, and a USB interface. These boards are shown in figure 4.2.



(a) Cyclone II (b) Cyclone III

Figure 4.2: Altera Cyclone development boards[6] [7]

4.2.3 Xilinx

Currently Xilinx offers three different FPGA families[8]. The Virtex family is a high performance, up to 760 k slices, low-power series with integrated features such as DSP, PowerPC-Processor and high speed serial transceivers. This family consists of different FPGA versions which differ in their configuration and features. The newest version is the Virtex-6 introduced in 2009.

The second family is called EasyPath, it is a solution for high volume products. A customer can create a Virtex-6 project, can ship it to Xilinx and receives a non-reprogrammable device with lower costs than the Virtex-6 FPGA. Therefore, this product family is irrelevant for this thesis.

The Spartan family is the third FPGA family from Xilinx. It is a solution balanced between low-cost, low-power and high performance. The newest version is the Spartan-6 introduced in 2009. The devices of this series differ in their configuration but can include gigabit transceivers, integrated memory controllers, block RAM, DSP slices, PCI express endpoints and differential

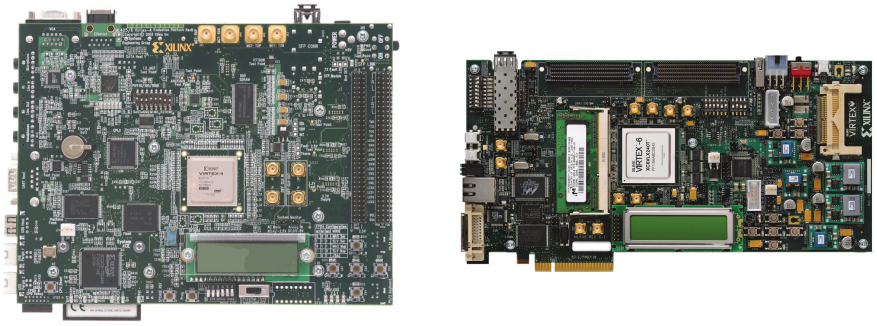
[6] http://www.altera.com/products/devkits/altera/images/fig1-cyclone2-staterkit.jpg
[7] http://www.altera.com/products/devkits/altera/images/cyc3starter-angled.jpg
[8] http://www.xilinx.com/products/devices.htm

pair I/O. The integrated memory controllers are on chip and achieve higher speeds than synthesized IP Core memory controller.

Development Boards

A variety of different development boards are offered by Xilinx and other companies[9]. For example high-end boards such as the older Virtex-4 FX *ML405* Evaluation Platform for 800 $, which is used in the GET Lab. It is equipped with 19 k slices and a Virtex-4 FPGA containong a PowerPC core. The board is also equipped with gigabit Ethernet, LCD display, 256 MB DDR2 SDRAM, USB Full Speed host and general purpose I/Os.

The new 2,500 $ Virtex-6 FPGA Embedded Kit *ML605* utilizes a 241 k slices containing Virtex-6, which includes in contrast to the Virtex-4 FPGA no PowerPC core. The board is designed as a PCI Express card and is equipped with 128 MB DDR3 SDRAM, USB Full Speed host, general purpose I/Os and gigabit Ethernet components. Figure 4.3 shows these boards.



(a) ML405 (b) ML605

Figure 4.3: Xilinx ML development boards[10] [11]

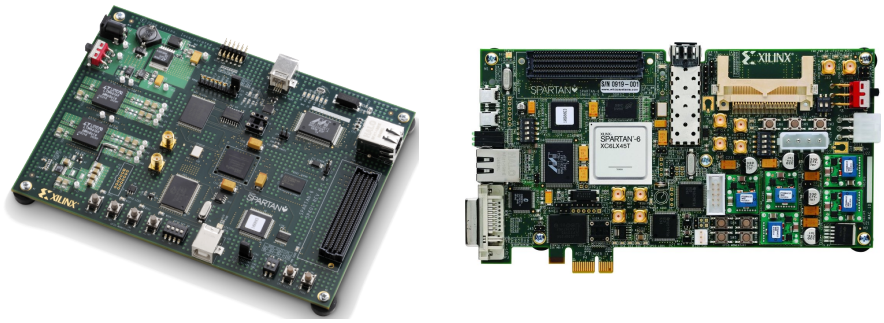
The Spartan-6 FPGA SP601 Evaluation Kit is a brand-new kit. The board is equipped with a 14 k slices FPGA, gigabit Ethernet, 128 MB DDR2 SDRAM and different general purpose I/Os. It can be purchased for round about 300 $.

[9] http://www.xilinx.com/products/boards_kits/index.htm

[10] http://www.xilinx.com/products/devkits/HW-V4-ML405-UNI-G-image.htm

[11] From ML605 Product Brief:http://www.xilinx.com/publications/prod_mktg/ml605_product_brief.pdf

A feature rich development board is the Spartan-6 FPGA Embedded Kit SP605. A 43 k slices element FPGA is placed on a board with PCI Express card form factor. The board provides features such as 128 MB DDR3 memory, gigabit Ethernet and a support for *MicroBlaze* (see section 3.1.1) soft processor cores. The board has an expected price of 500 $ with a promotional discount, unfortunately it is not available yet. Both boards are pictured in figure 4.4.



(a) SP601 (b) SP605

Figure 4.4: Xilinx SP development boards[12] [13]

4.2.4 Decision

Table 4.1 shows a comparison of the selected development boards.

After some investigation Altera was found to offer no affordable development board with the required hardware configuration. Boards with a gigabit Ethernet interface are too expensive and the low-priced boards neither have gigabit Ethernet nor a USB 2.0 High Speed interface. Thus, the option of a system with an Altera FPGA was discarded.

Xilinx's high-end Virtex-6 on the ML605 development board provides a huge amount of slices, but it is to high-priced for our application.

The ML405 is used as a reference because it is utilized by students and employees in the GET Lab. It is an old board with an outdated FPGA, the integrated PowerPC processor could be used to implement algorithms in C or other programming languages but that would lead to a PC like system. Our

[12] http://www.xilinx.com/products/devkits/EK-S6-SP601-G-image.htm
[13] http://www.xilinx.com/products/devkits/EK-S6-SP605-G-image.htm

	Development Board	Number of Slices/LE[14]	Price	USB 2.0 High Speed	Gigabit Ethernet
Altera	Stratix IV GX	124 k	4,500 $	no	yes
	Stratix IV E	531 k	10,000 $	no	yes
	Arria II GX	124 k	1,500 $	no	yes
	Cyclone II	20 k	200 $	no	no
	Cyclone III	24 k	200 $	no	no
Xilinx	ML405	19 k	800 $	no	yes
	ML605	241 k	2,500 $	no	yes
	SP601	14 k	300 $	no	yes
	SP605	43 k	500 $	no	yes

Table 4.1: Development board comparison

goal is to implement the whole system in hardware without any processor assistance, thus low-priced FPGAs can be used.

The low-priced development boards SP601 and SP605 combine an acceptable amount of slices, a gigabit Ethernet interface and a DDR2 SDRAM which is big enough to store multiple images. For our application both boards differ only in number of slices.

SP601 and SP605 were brand-new at the begin of our work. We intended to use the SP605 but is was not obtainable at the time. Therefore, we decided to purchase the SP601.

4.3 Camera Selection

In this section firstly the requirements for the cameras are discussed. Afterwards different camera types sorted by their interfaces are presented and a decision is made which camera type should be utilized in this thesis.

[14]Logic Elements

4.3.1 Requirements

Our Stereo Vision System should be usable in a robotic environment. Changing illuminations and vibrations caused by motion influences the captured pictures. Thus, special requirements for cameras that can be used in our system exist.

The *image sensor* should capture color images. It should have high sensitivity for low-light operation and should handle changing lighting. In [HKT07] a comparison of state of the art CCD and CMOS cameras is made. It is shown that current CMOS image sensors deliver high quality images but the quality of current CCD image sensors is still better. A decision for one technology has to be made. See section 3.3.2 for an explanation of CCD and CMOS technologies.

Some algorithms need a high *camera resolution* but can not be executed fast enough to take advantage of a high *framerate*. In contrast when fast changing elements shall be detected or tracked, a high framerate is required and a smaller resolution can be accepted. Therefore, a configurable resolution and a high framerate are desired. We accept cameras with a framerate of 30 fps or above and a minimum resolution of 640×480 pixel.

In robotic environments vibrations can cause rolling shutter cameras to capture distorted images. Thus, a *global shutter* is desired. See 3.3.2 for information about different shutter types.

Distortion-free camera lenses are preferable. Since vision algorithms often compute characteristics in association with the position in the image, good camera lenses are required. Especially in the border area lense distortions are significant and can compromise algorithms. For different applications other lenses are applicable. Parameters like *focal length* and *F-number* have to match the requirements of the application (see section 3.3.1 for explanations of focal length and F-number).

In regard to costs the same requirements that must be met by the Stereo Vision System have to be met by the cameras. We want to choose cameras with *reasonable results* for a *low-price*.

Finally, the interface of the cameras must be accessible by the chosen development board.

4.3.2 Camera Overview

There exist many different cameras on the market. In this section an overview over different camera types, ordered by there interfaces, is given.

Cameras with USB or Firewire Interface

USB cameras are primarily known because of USB webcams. These cameras are wide-spread but most of the offered cameras do not give proper results for image algorithms. One reason for the insufficient results are the used lenses which are not user replaceable.

Some manufacturer of industrial cameras offer high quality CMOS or CCD products with USB interface and user selectable and replaceable lenses. These cameras are applicable for image algorithms but depend on a PC system for image readout. Without such a system the provided drivers are not usable and readout is challenging because a USB host must be emulated.

Firewire cameras typically are not used by end-users. They are used for industrial machine vision or research in high performance applications. Without PC support these cameras are complicated to handle, too.

One manufacturer for USB and Firewire products is *The Imaging Source*[15]. High quality USB cameras begin at a price of round about 250 €. Some Firewire cameras are offered for a similar price. High end cameras for industrial purposes are more expensive and can cost over 1,000 €.

Cameras with Ethernet Interface

Ethernet cameras use the IEEE standardized Ethernet network for image transmission. For industrial purposes a standard called *GigE Vision*[16] is established.

The standard is based on a UDP/IP protocol which allows camera configuration and readout. It features fast transfer rates up to 1 Gbit/s, high data transfer lengths and easy usage with other Ethernet devices and computers.

For this standard many companies have developed GigE Vision conform cameras. One manufacturer is *DALSA Coorporation*[17]. These cameras are of-

[15] http://www.theimagingsource.com/

[16] http://www.machinevisiononline.org/vision-standards-details.cfm?id=36\&type=5

[17] http://www.dalsa.com/

fered with a price of round about 500 €[18] for the cheapest and well beyond 1000 € for high end products.

Cameras with CameraLink

CameraLink[19] is a special interface for cameras in high performance applications. It uses *Low Voltage Differential Signaling*(LVDS) for data transmission. LVDS is an interface standard which is efficient (uses low voltage signals) and can transmit data with high transmission rates in areas with interferences without data corruption.

The CameraLink standard defines a so called *Channel*. It consists of three LVDS pairs to transmit 8 bit each and one LVDS pair to transmit synchronization signals such as *Line Valid* and *Frame Valid*. Another LVDS pair is used to transmit the clock.

Additionally four LVDS pairs are reserved for camera control signals defined by the camera manufacturer and two LVDS pairs for camera configuration by serial communication.

CameraLink is available in three versions: Base Configuration, Medium Configuration and Full Configuration with one, two and three channels respectively to transmit 24, 48 and 64 bit data per clock.

A few CameraLink cameras are available for a price below 1,000 €. Most of the cameras cost well over 1,000 €.

Cameras with Direct Interface

Many manufactures of image sensors offer development boards for their products with a direct interface to the sensor. The signals from the sensor are directly joined to a connector. These boards are primarily for hardware developers which want to develop their own camera and need a test board.

A wide range of camera modules with this interface and miscellaneous features are offered for a low-price.

[18] http://www.theimagingsource.com/en_US/products/cameras/gige-ccd-color/
[19] http://www.vision1.com/pdf/CameraLink5.pdf

4.3.3 Decision

The first idea was to purchase a camera that can also be used by GET Lab employees in future projects. After a discussion of the different requirements a common denominator was found in the DALSA Falcon 4M60 (see figure 4.5).



Figure 4.5: DALSA Falcon 4M60[20]

This color CMOS camera has a resolution of up to 2352×1728 pixel with a maximum framerate of 30 fps and a CameraLink Medium Interface. A request for quote unveiled a price tag of about $4,000$ €, which is way over the aspired price limit. Other cameras with CameraLink are comparable high priced, therefore only cameras with Ethernet, USB, Firewire or a direct interface came into consideration for our Hardware System.

Since Ethernet cameras are offered for comparable prices like the CameraLink cameras and the Ethernet interface of the SP601 is occupied for the host PC connection, Ethernet cameras are not suitable for our application.

Directly interfacing USB cameras with an FPGA without a computer system is not reasonable, since the drivers functionality would have to be emulated by the FPGA. The driver sources are mostly closed source which leads into an unsolvable problem. In addition only high priced development boards feature an USB Host interface.

Interfacing Firewire cameras over an FPGA is possible and performed, for example in [Arr05]. We decide against this possibility because it is not documentated how to perfom this and the risk of a failure was to high.

[20] http://www.dalsa.com/images/mv/4m60_200w.jpg

Therefore, we decide to use cameras with a direct interface. The advantage of this approach is that we are able to build a hardware interface to the SP601 development board which allow the connection of a wide range of these cameras (see chapter 5). Compared to cameras with other interfaces the huge amount of direct interfaced cameras is offered. Thus, an adequate camera for our application can be selected. For the first evaluation we decided to use the OV7720 CMOS image sensor from OmniVision Technologies[21].

This color image sensor captures images with a maximum resolution of 640×480 pixel and a framerate of 60 fps. The featured high sensitivity for low light operation and the high response to changing illumination are ideal in robotic applications. OmniVision sponsored us two of these color CMOS image sensors.

Furthermore we build an own camera with an MT9V032 image sensor from Aptina Imaging[22] to test if satisfying results can be achieved without spending much money. See chapter 5.4 for more information.

In the next chapter the the self-made hardware to interface the selected cameras is introduced in detail. Likewise the details of the used cameras are presented.

[21] http://www.ovt.com
[22] http://www.aptina.com

Chapter 5

Self-made Hardware

In this chapter we present the developed, self-made hardware for the connection of the camera to the FPGA board. The SP601 development board has no interface for a direct connection of a camera. Hence, additional hardware has to be engineered for interfacing the chosen cameras.

In the next sections the requirements for the self-made hardware are presented followed by a detailed description of the used hardware in this thesis.

5.1 Requirements

Cameras with a direct interface were selected in chapter 4. The interface of these cameras consist of an input clock to drive the internal camera functions and the image readout process. Over parallel signal outputs (typically 10) the image data and control signals are transmitted. These signals are synchronous to a provided output clock. Figure 5.1 illustrates the interface of the cameras.

The only available connector on the SP601 with enough in- and outputs to connect two cameras is the FMC-LPC connector. It implements an industry standard called *VITA 57.1 FMC* which provides 68 single-ended signals with a logic voltage of 2.5 V(see figure 5.2 for picture of the connector). Thus a board has to be engineered which connects the camera signals with the signals of this connector.

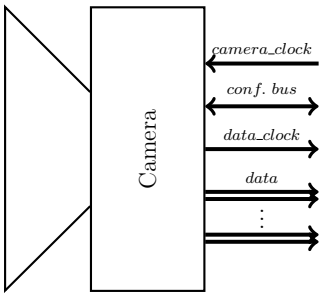


Figure 5.1: Illustration of a hardware camera interface (top down): clock input for camera internal functions (camera_clock), conf. bus for camera control and configuration, data_clock and data as outputs for delivered data.



Figure 5.2: FMC-LPC connector on SP601 development board

Typically cameras with a direct interface are driven by a voltage of 3.3 V. Therefore a bidirectional voltage translation is required for all signals between the camera and the development board.

Since the cameras have to be flexibly mounted a cable connection to the development board is necessary. A direct transmission of the camera's parallel output signals to the development board is inadvisable because transmission errors caused by interferences has to be expected. Therefore a technique to maintain signal integrity is necessary.

During the usage of the Stereo Vision System misfortunes and defective components can lead to heavy hardware damages. Thus preventive measures have to be made against electrical short circuits and user errors.

5.2 Design

After some investigation we decided to use the *FIN212AC*[1] IC of Fairchild Semiconductor International Inc. for data transmission between the camera and the SP601 development board.

This IC can be configured as serializer or deserializer and is capable of transmitting a clock and 12 clock synchronous signals in one direction and another clock in the opposite direction. Data is transferred over a proprietary low-power interface called *Current Transistor Logic* (CTL™). In contrast to the well known *Low Voltage Differential Signaling* (LVDS) bits are not encoded as differential voltages, they are encoded as different current directions.

The I/O voltage and the supply voltage of an FIN212AC are electrically insulated. Thus a voltage translation can easily be achieved by supplying a different I/O voltage to the serializer than to the deserializer.

We use two of these ICs to interface each camera. Each camera is mounted on a *Camera Adapter* circuit board. These boards feature a FIN212AC which serializes the parallel data output and transmits it over a *Flexible Flat Cable* (FFC) to the *SP601 Adapter*. This board is connected to the FMC-LPC connector on the SP601 development board. It is equipped with two FIN212AC ICs configured as deserializer.

OmniVision uses a *Serial Camera Control Bus* (SCCB) to configure their image sensors [Tec03]. The utilized protocol is a variant of the *Inter-Integrated Circuit* (I^2C) bus invented by Philips Semiconductors [Sem07]. The main difference between SCCB and I^2C bus is that the SCCB specification defines special transmission cycles which are not regulated in the I^2C specification. Other image sensors can be controlled directly over I^2C, therefore voltage translators for I^2C can be used. We use the bidirectional *PCA9306*[2] voltage translator of NXP Semiconductors for the 2.5 V to 3.3 V translation.

Figure 5.3 shows our engineered system consisting of the camera mount, SP601 development board with attached SP601 adapter, FFC Cables and Camera Adapter with attached MT9 Cameras. In the following all selfmade components are explained in detail.

[1] http://www.fairchildsemi.com/pf/FI/FIN212AC.html
[2] http://www.nxp.com/pip/PCA9306_4.html

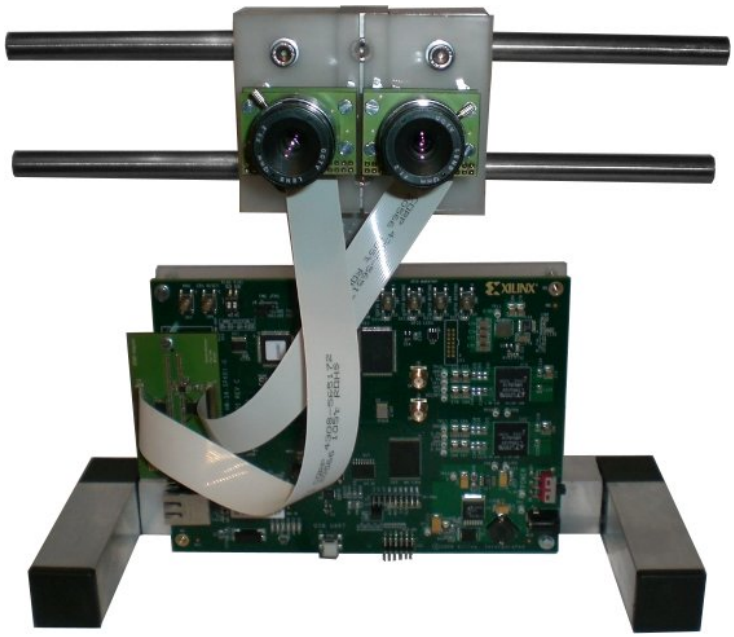


Figure 5.3: Hardware System (top down):
Camera Mount with two MT9 Cameras and Camera Adapter boards connected via FFC Cables with the SP601 Adapter on the SP601 development board.

5.3 Adapter Boards

To interface the chosen cameras two boards were engineered, the *SP601 Adapter* which is attached to the FMC-LPC connector on the development board and the *Camera Adapter*, whereas the camera is mounted on this board.

The used serializer/deserializer for data transmission FIN212AC are configured by several control pins provided by the IC. The selected configuration does not support higher output clocks of the cameras than their input clock. This has to be considered for camera configuration since the serializer blocks data transmission if the output clock is faster than the input clock. A schematic representation of the adapter boards is illustrated in figure 5.4.

The *SP601 Adapter* is a small printed circuit board engineered to fit on the development board. It is equipped with two FFC connectors, one for each camera. Per camera a camera clock signal is transmitted which drives the camera and a clock with belonging synchronous data is received. Data and

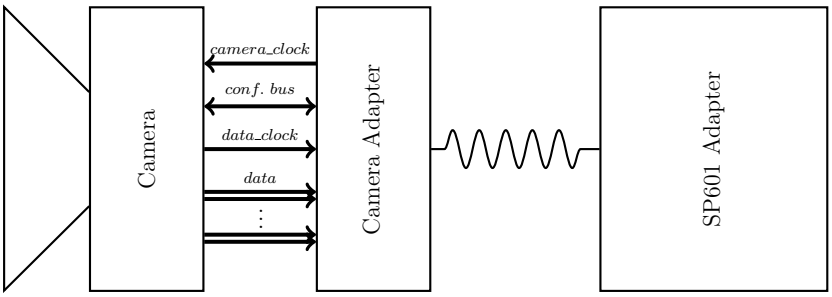


Figure 5.4: Adapter boards architecture

clock transmission is performed by one FIN212AC IC per camera. Since this IC is connected to the SP601 its I/O voltage is driven by 2.5 V. The outgoing voltages are fixed to 3.3 V. As mentioned above a PCA9306 translates the voltage of the camera configuration bus. To avoid hardware damage the 2.5 V and 3.3 V supplies, generated by the development board, are protected by fuses.

As described in chapter 4 we intended to use the superior equipped SP605 development board but it was not obtainable at the time. Since both boards, SP601 and SP605, share the same FMC-LPC connector our engineered hardware is also usable on the SP605 development board.

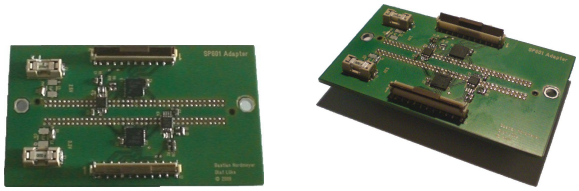


Figure 5.5: SP601 Adapter

Figure 5.5 depicts the SP601 Adapter board with the described components.

A printed circuit board called *Camera Adapter* is designed to connect cameras with a direct interface with the SP601 Adapter. It is engineered to transmit 10 data signals and a clock to the development board, whereas on clock can be received. Likewise this is performed by a further FIN212AC IC.

The connection with the SP601 Adapter is established by another FFC connector. A camera can be connected to this board over a 20 pin socket terminal strip, whereas a robust mounting of the camera can be obtained by screwing

it with the board. Since the pinout of the cameras can slightly change, the pin assignment can be configured for the most common camera pinouts.

The adapter is illustrated in figure 5.6.

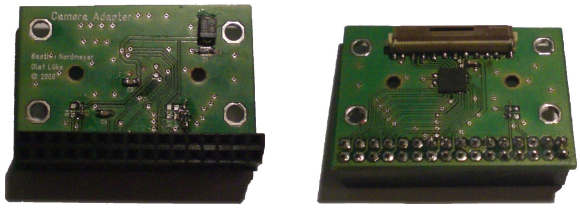


Figure 5.6: Camera Adapter

5.4 Cameras

Two different camera models where used during this work. The *OV7 Camera* and the *MT9V Camera* both equipped with different image sensors. The cameras are configured over an I^2C compatible bus by setting register values. For example the image size, the readout mode and control functions like automatic exposure control and automatic gain control can be parameterized. Both cameras possess about 170 and 200 registers respectively, which have to be partially adapted to work as desired.

In the following sections both camera types are presented.

The OV7 Camera is an evaluation board directly manufactured by OmniVision. It is equipped with an OV7720[3] color VGA image sensor with a maximum resolution of 640×480 and a maximum frame rate of 60 fps. It features automatic image control functions like Automatic Exposure Control and Automatic Gain Control which allow us to focus on image readout without concerning of exposure and gain determination. The camera is configured over OmniVisions SCCB bus and uses a lens with a diameter of 8 mmand a unknown focal length.

Figure 5.7 shows two different views of the OV7 Camera. The camera is covered with black tape to inhibit light leakage into the PCB which caused artefacts in the images.

[3] http://www.ovt.com/products/detail.php?id=62

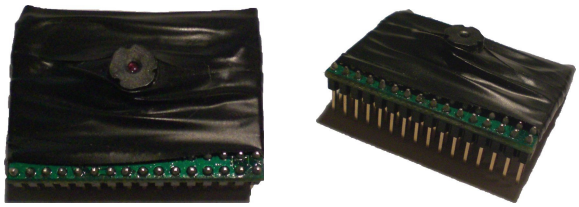


Figure 5.7: OV7 Camera covered in black tape to inhibit light leakage in to the PCB

As mentioned in chapter 4 we decided to develop our own camera, which has to fulfill the defined requirements. Briefly a minimum resolution of 640×480 pixel with a framerate of 30 fps and a global shutter were required. A wide range of image sensors are available on the camera market which fulfills these requirements.

We choose the MT9V032 CMOS image sensor from Aptina Imaging[4]. It is a WVGA image sensor with a maximum resolution of 752×480 pixel, global shutter, extremely low noise levels, low-light sensitivity and an image quality which is usually only provided by CCD image sensors. Especially the near-infrared sensitivity is extremely high on this sensor model.

Since one possible application of system is robotics, and there image capturing has to be performed under extremely different lightening, a camera with this sensitivity is very useful to gain usable image information.

To permit the usage of different lenses, we equipped the board with a C-Mount lens mount such that a wide range of available lenses can be utilized. For our evaluations a low-priced lens without IR-filter, with a focal length of 12 mm and a F-value of 1/3" is used (see section 3.3.1 for lens details).

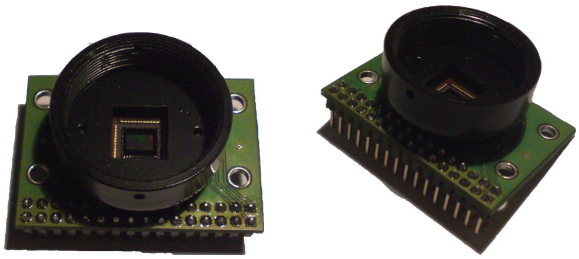


Figure 5.8: MT9 Camera

[4] http://www.aptina.com/products/image_sensors/mt9v032c12stc/

Figure 5.8 shows our camera called *MT9 Camera* with the equipped CS-Mount and the socket board connector fitting to our Camera adapter. Compared to the cameras mentioned in chapter 4.3 the costs for the parts of the camera are well below 40 €, and the camera could be customized in size and the interfaces for our requirements.

Another important factor besides the cameras is the mounting. Vision algorithms depend on defined geometrically relations between the cameras. In case of stereo vision algorithms often rectified images are necessary (see section 3.4.1 for further details). Aside from lens distortions, which are not directly covered in this work, the mounting of the cameras is essential for rectified images. Hence a solid mounting for the cameras had to be constructed which ensures a parallel orientation and a adjustable distance between the cameras.

We have designed the mounting in the 3D CAD software Solid Edge[5]. It is constructed of aluminium with steel shafts. Figure 5.9 depicts the construction.

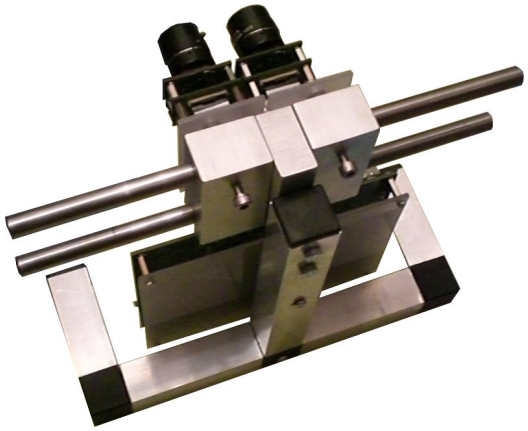


Figure 5.9: Illustration of the Camera Mount

[5] http://www.solidedge.com/

Chapter 6

Vision Framework

In this chapter the framework, called *Vision Framework*, for the utilized vision algorithms is presented. Since tasks like clock generation, access on the external in- and outputs of the development board and communication over Ethernet are not directly possible by an Impulse C generated VHDL code, a framework has to be developed which handles these tasks. Thus the Vision Framework provides the access on camera images, DDR memory and a Ethernet connection over Impulse C streams to the Vision Core.

6.1 Design

The design of the Vision Framework has to satisfy several requirements, which are already discussed in chapter 4. In the following sections the relevant requirements for the Vision Framework are discussed and our developed architecture is presented.

In this chapter, the main components of the Vision Framework are illustrated in a simplified manner. For example clock signals, reset signals, signals for synchronisation and handling signals (e. g. for FIFOs) are not depicted. Likewise only the main processes of each component are shown.

Figure 6.1 specifies the graphical elements which are used in component illustrations.

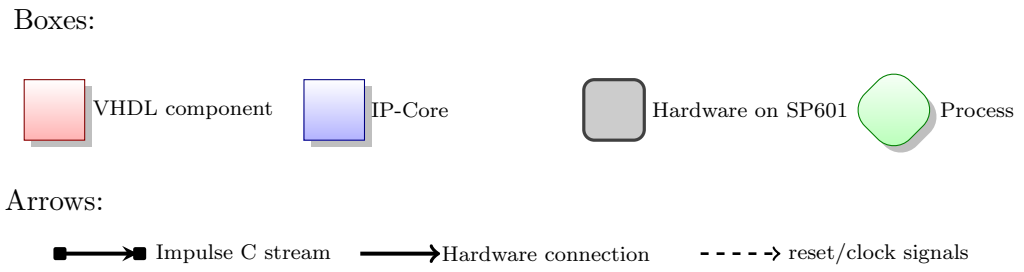


Figure 6.1: Illustration of elements

6.1.1 Requirements

In chapter 4.1 the requirements for the Stereo Vision System are discussed and a hardware platform is chosen. Based on this decision the requirements for the system design are selected.

Firstly, the system structure should be modular, such that individual components can be *flexibly* replaced by others for different applications. To be able to test other vision algorithms than the implemented ones without the need of an adaption of the Vision Framework, interfaces to the Vision Core have to be *generic*.

Cost effectiveness is another requirement and an essential element in this work, i. e. we depend on a system without expensive commercial IP Cores. Also the system should be optimized for *speed*. In cases where an optimization interferes with flexibility a trade-off must be found.

An accessible *data storage and data consistency* are further requirements. The SP601 features DDR memory, this memory should be used to cache two simultaneous captured images, such that the Vision Core can process these images without any time constraints and all image pixels in an arbitrary order.

Also a gigabit Ethernet transceiver is featured by the SP601. This *high speed interface* must be accessible by the Vision Core over an Impulse C stream. To achieve highest flexibility the interface to this transceiver should only pass the stream data to the transceiver, i. e. the Ethernet interface should not do data framing or similar tasks.

In the next section an adequate architecture for the Vision Framework is presented, which fulfills the previous requirements.

6.1.2 Architecture

The Vision Core is embedded in the Vision Framework which handles image capturing and buffering as well as data output. This framework allows vision algorithm developers to implement their algorithms with Impulse C without the need to be concerned with difficult low-level HDL implementations.

Images are simultaneously captured by the framework and buffered. The Vision Core can access each pixel of these images individually. The data that is generated by the Vision Core is transmitted over gigabit Ethernet to a host PC. To achieve this, the developer has to frame the data and send it over a stream to the component that handles the Ethernet communication.

Great importance was attached to a generic interface for each component such that it is flexibly usable in other applications or that it can easily be replaced by a component with other behavior. For example the camera data input can be substituted by a component which reads out another type of camera without the need to change any of the other components. Vision algorithms developed on our platform can be used on other platforms by adapting the framework without the need to reimplement the whole algorithm.

An important property of the Stereo Vision System is the flexibility, it is ranked higher than the speed of the system. In RTL development a system optimized for high performance is a highly specialised system. Therefore, some concessions in speed have to be made to preserve generic interfaces.

The whole RTL System except for some used IP Cores is written in VHDL. Figure 6.2 illustrates the basic architecture and the data flow in the Stereo Vision System.

Different clocks are supplied to the system by the *Clock Manager*. It generates the required clocks for all components from an external oscillator placed on the development board.

Reset signals are handled by the *Reset Handler*. It controls the reset signals for all components, i. e. it is responsible for the initialisation sequence of the system. Some components, such as the DDR memory, need some time for initialisation until they are usable. This handler releases the reset of specific components successively until all components are running.

Two functions are provided by the *Stereo Interface* component. Firstly, it configures the connected cameras to behave as user defined. For example it can set the resolution or the output format of the cameras. Secondly, if both cameras are configured, it reads out the camera images and takes care of

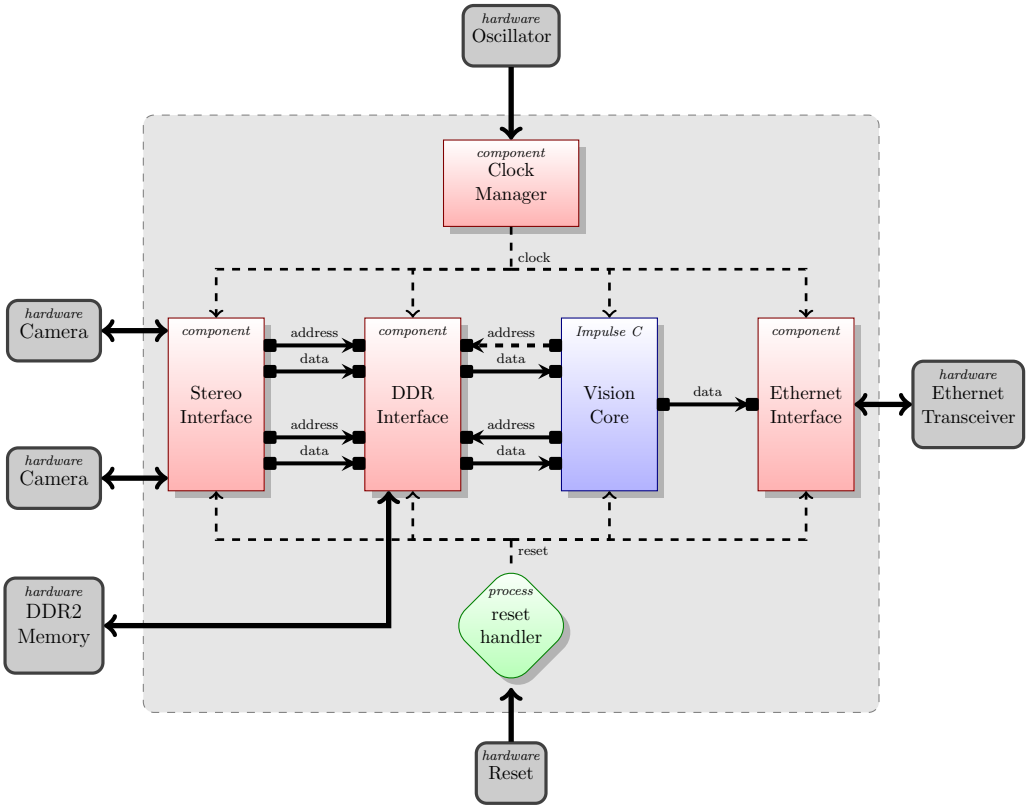


Figure 6.2: System architecture: Simplified data flow between hardware components (grey), HDL components (red) and the Vision Core (blue) written in HDL languages and generated by Impulse C C-to-VHDL compiler respectively.

stereo image consistency, i. e. it assures, that only images which are captured at the same time, are transmitted. The captured images are stored in two different buffers in the DDR memory by setting the corresponding address and data. Each data in the memory is a 32 bit value, whereas 30 bit contain a pixel in both images encoded in RGB555 color format (5 bit for red, green and blue).

The *DDR Interface* provides an interface to the external DDR memory on the SP601 and adds some double buffering features, i. e. a simultaneous read and write on two individual memory areas, so called buffers, can be performed. After reader and writer have finished their operations these buffers are swapped. Hence, the reader can read the buffer which was written last by the writer. And the writer can overwrite the buffer which was lastly read

by the reader. Two double buffers with a write and a read port each are provided. With both write ports image data is written into memory.

An independent readout of the stored image data is performed by the *Vision Core*. Each DDR Double Buffer features an *address stream* and a *data stream*. The component can request data by transmitting an address over the address stream and receives as response the corresponding data over the data stream. In our implementation one address stream is not driven by the Vision Core. It is driven by a separate VHDL process which transmits pixel after pixel, such that the data stream acts as a plain image stream.

Data that is generated by the Vision Core is transmitted over an 32 bit Impulse C stream to the *Ethernet Interface*. The Vision Core sends 1018 bytes successively and the Ethernet Interface then sends them to the hardware Ethernet transceiver.

6.2 Clock Manager

In digital circuits described by RTL (refer to section 3.1.1), data is propagated between combinational logic by clocking. The maximum clock speed of the generated circuit mostly depends on the maximum delay in the combinational logic paths of the sub-circuits.

A stable operation of the whole circuit can be assured by clocking it with the minimum of the maximal clock of all paths. It is also possible to drive particular sub-circuits with higher clocks when their maximum clock frequency is not exceeded.

To achieve high performance it is necessary to run some sub-circuits with different clock speeds, such that the clock speed of these sub-circuits is exhausted. Therefore, different clocks in the RTL System are required.

6.2.1 FPGA Clocking

Modern FPGAs have an integrated infrastructure to generate clocks. Typically this infrastructure includes dedicated clock inputs, buffers, routing lines and clock generation components.

In case of Spartan-6 so called *Clock Management Tiles* (CMTs) [Xil10a] are special hardware components inside FPGAs, which are used to generate

different clocks from an input clock. These components include *Digital Clock Manager* (DCM) and *Phase Lock Loops* (PLL) [Xil09c].

These components can be coupled together over dedicated clock routes and can be used to eliminate clock skew, shifting the phase of a clock or multiply and accordingly divide the frequency of an incoming clock to generate a new clock. The Spartan-6 FPGA on the SP601 development board features two CMTs composed oftwo DCMs with one PLL each.

6.2.2 Clock Generation

Since the quantity of DCMs and PLLs is limited and clock generation is error-prone, it is not recommendable that each system component generates its own clock. Therefore, in our system a central component called *Clock Manager* generates the required clocks for the whole system.

Long clock paths can lead to a huge clock skew which on the other hand leads to a smaller maximum frequency for this path. To achieve high clock frequencies it is necessary to minimize the length of the clock paths, i. e. a long clock path driving a huge clock domain should be avoided. Preferably several clock domains driven by their own clock are used.

The Clock Manager uses two PLLs to generate the required clock for each clock domain. An external 200 MHz signal generated by an oscillator on the development board is used to drive the PLLs.

Table 6.1 shows the different clocks.

Signal	Speed	Purpose
$CLK600$	600 MHz	Doubled clock for DDR memory
$CLK300$	300 MHz	Clock for DDR memory
$CLK200$	200 MHz	Clock for DDR Interface operations
$CLK125$	125 MHz	Clock for Ethernet transceiver
$CLK100$	100 MHz	Clock for Stereo Interface
$CLK66$	66 MHz	Clock for Vision Core
$CLK25$	25 MHz	Clock to drive external cameras

Table 6.1: ClockManager: Generated clocks

The DDR Interface utilizes a memory controller IP Core which uses 600 MHz and 300 MHz clocks to drive the internal functions of the core and the external DDR memory. A 200 MHz clock is utilized to handle the communication with the memory controller IP Core (see section 6.9).

For 1 Gbit/s Ethernet connections a 125 MHz clock is required by the hardware Ethernet transceiver. The Ethernet Interface which controls this transceiver also communicates with the hardware driven by the 125 MHz clock (refer to section 6.11).

The Stereo Interface requires a clock to handle image data buffering and transmission to the DDR Interface. This clock has to be at least as fast as the clock which drives the Vision Core to avoid performance loss. To create an own clock domain an unused clock with a frequency above the Vision Core clock has to be used. Therefore, this component is driven by 100 MHz clock.

Impulse C generated VHDL code is clocked over the connected stream clocks. Each Impulse C stream has its own clock and the C-to-VHDL compiler takes care of the clock distribution within the generated code. Therefore, the maximum clock to drive the different streams of the Vision Core depends on the used C code. To ensure a stable operation, the Vision Core is driven by a frequency of 66 MHz regardless of the current implemented algorithms.

Image sensors need a clock to drive sensor internal functions and a data clock for image transmission. The required speed of this clock differs between image sensor models. Our cameras are both driven by a 25 MHz clock.

6.3 Reset Handler

After the hardware is powered up the Hardware System components such as the Ethernet transceiver and connected cameras are initialised. Each component requires a different initialisation time until it can be used. FPGAs include so called *hard cores*(see section 3.1.1), these have a specific initialisation phase, e. g. CMTs take some time until they produce a stable clock. Therefore, an immediate system run is not possible and a well-regulated power up sequence is required.

Additionally an external triggerable reset of the whole system is desired. This is needed to allow the user to e. g. reconfigure the cameras if the camera connection was lost. Otherwise a restart of the whole system would be needed.

Thus, it has to be possible for the system to always reach a well-defined state when the reset signal is used.

These tasks are handled by the *Reset Handler*, it controls the power up sequence and is used whenever the system is turned on or an external reset is triggered. This is done by holding the system in a reset state for a fixed time and then successively releasing the reset signals for different components until the whole system is in a running state. Figure 6.3 shows the reset release sequence.

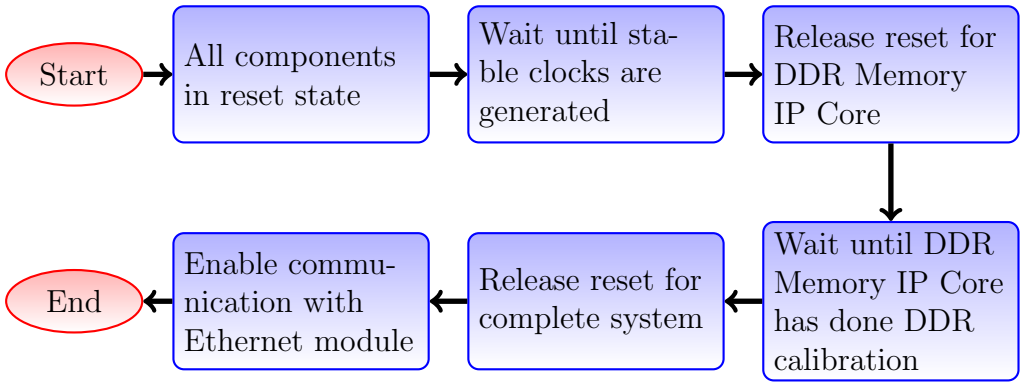


Figure 6.3: Reset Handler: Reset sequence of the system.

6.3.1 Reset Strategies

Two different reset strategies exist for RTL circuits. A reset is either performed clock synchronous or it is performed clock asynchronous. The following VHDL example illustrates the different implementation.

```
entity sync is
   port (
     clk : in std_logic;
     reset : in std_logic
   );
end sync;

architecture rtl of sync is
begin
  process(clk, reset)
  begin
     if rising_edge(clk) then
        if reset = '1' then
           -- perform reset
        else
           ...
        end if;
     end if;
  end process;
end rtl;
```

```
entity async is
   port (
     clk : in std_logic;
     reset : in std_logic
   );
end async;

architecture rtl of async is
begin
  process(clk, reset)
  begin
     if reset = '1' then
        -- perform reset
     elsif rising_edge(clk) then
        ...
     end if;
  end process;
end rtl;
```

In this example two different VHDL entities called *sync* for the synchronous component and *async* for the asynchronous component are defined. A VHDL entity is a component which can alter signal states. It consists of a interface definition and a behavioral definition.

The interface definition in the first lines defines that the corresponding entity has *clk* and *reset* as inputs.

The behavioral definition of an entity begins with the *architecture* keyword. In both entities the architecture only consists of a process which has the clock and reset signal in its sensitivity list. A VHDL process is evaluated when a signal of the sensitivity list is changed.

In the synchronous component a reset is performed when a rising edge of the clock was detected and the reset signal is high. Whereas in the asynchronous component the state of the clock signal is irrelevant. To perform a reset only a high reset signal is necessary.

6.3.2 Decision

Both asynchronous and synchronous reset strategies can be used in a VHDL project to handle a reset. In [CMG03] techniques related to asynchronous and synchronous reset designs are discussed, it is concluded that an optimal decision for the utilized strategy has to be application dependent. The pro-

grammer is free to choose a reset strategy but has to retain a strategy in the whole project to avoid faulty behavior.

Since the VHDL code that is generated by Impulse C uses an asynchronous reset strategy, the Vision Framework also performs resets in an asynchronous manner to avoid a mixed reset design.

6.4 Vision Core

Vision algorithms are encapsulated in the Vision Core. This core utilizes the Vision Framework such that the developer does not have to pay attention to tasks such as image capturing, data storage, data output and clock generation. For various algorithms different Vision Cores can be implemented which are simply *plugged* in the system, such that the whole RTL System does not have to be adapted to run another algorithm.

The Vision Core is intended to be generated by the C-to-VHDL Impulse C compiler. All incoming and outgoing interfaces are Impulse C streams. But it is also possible to implement a Vision Core in a HDL description language like VHDL or Verilog. In this case the Impulse C streams have to be driven from components written in a HDL language.

Figure 6.4 shows the interfaces of the Vision Core.

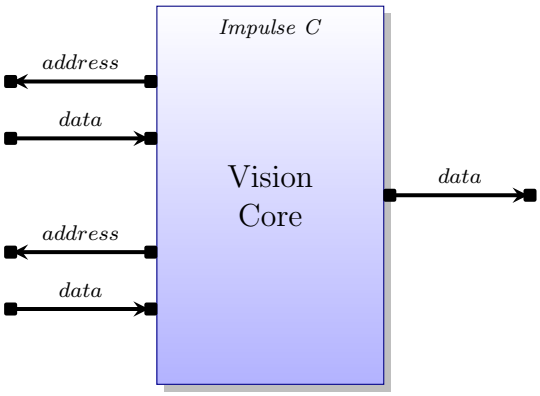


Figure 6.4: Interfaces of the Vision Core: Impulse C streams for camera data input and computation data output.

Bit widths of the streams can be adapted to the current application, for example if a higher image resolution is used a larger bit width is necessary to address the image. In our work we decided to use an 30 bit encoding of

the two camera images that consists of six 5 bit color channels (two times RGB555). The reason is, that firstly the OV7 Camera transmits images in this color format and secondly a color format with more bits than the 5 bit color encoding of RGB555 would lead to a higher complexity of the histograms used in the Vision Core, as described in section 8.2.

Therefore, a 32 bit data stream for the pixel input, an up to 25 bit address stream which defines the pixel position in the image and a 32 bit stream for the outgoing data are provided for the Vision Core (currently a 19 bit address stream is used).

The implemented Vision Core does not depend on using all provided interfaces. In our case one of the address streams is not driven from the Vision Core. A VHDL process drives the address stream such that camera images are streamed successively over the first data stream to the Vision Core. A Vision Core only utilizing the first address and data streams is possible as well.

A full description of the Vision Core used in this thesis is discussed in chapter 8.

6.5 Block RAM Double Buffer

The *Block RAM Double Buffer* is a component which is used by the Ethernet Interface and the different camera interfaces. It features double buffer functionality, i. e. independent memory areas, so called buffers, can be independently written or read. After both reader and writer finished their operations these buffers are swapped.

This concept is advantageous in cases where a process wants to transmit data to another process among different clock domains. The usage enables the writer as well as the reader to write/read without waiting on the other process. Only in the case when a process has finished its operation it has to wait for the other process to swap the buffer. Therefore, double buffering can increase the performance of the system.

6.5.1 Architecture

Spartan-6 FPGAs feature internal block RAM memory [Xil10b].

These memory blocks can store up to 18 Kbit data and can be addressed through two independent ports. The data width and memory depth of each block can be configured, e. g. 512 words with a data width of 32 bit or 16 k words with 1 bit data width. Memory access is fast, write and read operations requires only one clock cycle of the connected memory clock. The XC6SLX16 FPGA on the SP601 development board has 32 of these blocks which leads to a combined memory of 576 Kbit.

Figure 6.5 illustrates the architecture of the Block RAM Double Buffer.

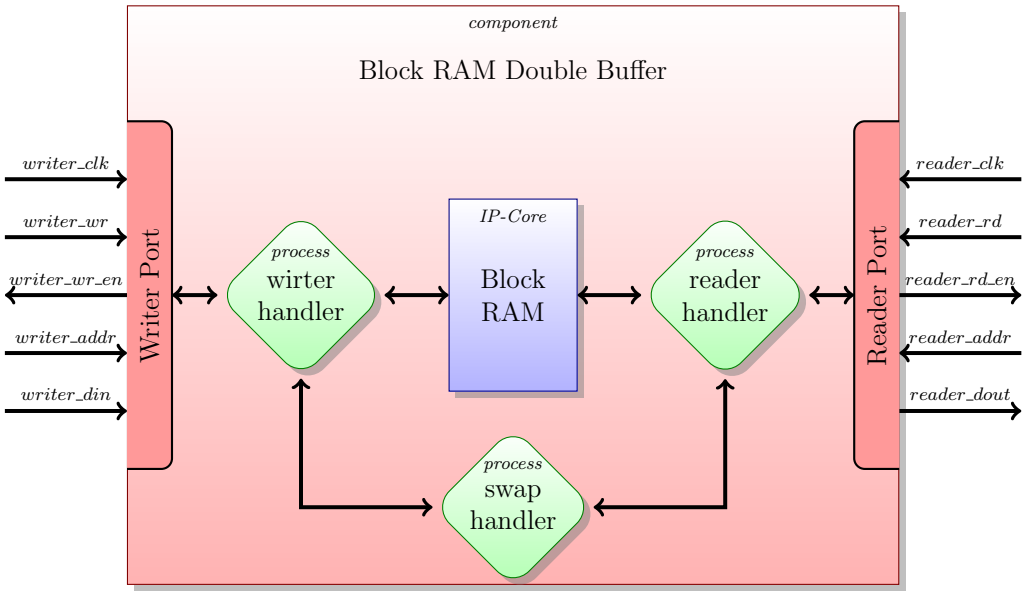


Figure 6.5: Block RAM Double Buffer Architecture: Data flow between writer handler, reader handler, swap handler and block RAM. To simplify the illustration signals are merged such that only the direction of data flow is shown.

With the *Core Generator* tool provided by Xilinx a block RAM IP Core can be generated. Such an IP Core is used by the Block RAM Double Buffer to access a memory block. The memory space provided by the block RAM is partitioned into two equal sized chunks, called *buffers*, which are each assigned to writer or reader.

Writer and a reader can access their buffers over two interfaces called *Writer Port* and *Reader Port*. All signals of a port are synchronous to an assigned clock signal (writer_clk or reader_clk).

The *Writer Port* is handled by a process called *writer handler* which controls the access to the memory. The signal writer_wr is used to inform the Double Buffer that the writer wants to write data to the buffer (called *buffer request*). If the current writer buffer is writeable this is confirmed by the component by setting writer_wr_en. Now in every writer_clk clock cycle data can be written to the buffer by setting writer_addr and writer_din. If the writer has finished its write operations it releases writer_wr. As response writer_wr_en is released by the component. Now the *Writer Port* is locked and another setting of writer_wr is not confirmed until the reader has finished its operations and the buffers are swapped.

For the reader the port handling is analogous. A process called *reader handler* controls the access of the reader to the memory. With reader_rd the reader can signal that it wants to read from the buffer. The Block RAM Double Buffer acknowledges read operations by setting reader_rd_en. Now in every reader_clk clock cycle the corresponding data value of reader_addr is with one clock latency present at reader_dout. If the reader has finished its read operations it releases reader_rd. Then reader_rd_en is released by the Block RAM Double Buffer and the *Read Port* is locked.

If both *Read* and *Write Ports* are locked a process called *swap handler* swaps the buffers and unlocks the ports. In the next read cycle the reader can read the data written by the writer in the previous cycle.

Depending on the application a bit width of 32 or 16 bit can be selected for writer_din and reader_dout, whereas the address inputs are fixed to a width of 8 bit for the 32 bit version and 9 bit for the 16 bit version.

6.5.2 Applications

The Block RAM Double Buffer is used twice in the RTL System.

The Ethernet Interface component utilizes a Block RAM Double Buffer instance to buffer Ethernet packets. This is advantageous because the used Ethernet IP Core has an interface which uses FIFOs for data transmission. To send an Ethernet packet, in every clock cycle data can be transmitted to the FIFO. Since it can not be assured that the Vision Core provides data in every clock cycle, a whole Ethernet packet is buffered to eliminate latency

during Ethernet packet transmission. A high latency can cause packet loss and other problems.

The second application of the Block RAM Double Buffer are the Camera Interfaces. Cameras with a direct interface transmit image data in every clock cycle. This data has to be processed or it is lost, which would result in corrupt images. Therefore, a Block RAM Double Buffer is used to buffer a line of the incoming image. The reader of the image data then does not have to process data in every clock cycle. It only has to be fast enough to process every image line. This enables the reader to process the data line by line with intervals without data processing.

6.6 Stereo Interface

The *Stereo Interface* component provides stereo images to the system and transmits these over two stream pairs. It utilizes two camera specific Camera Interfaces to read out the connected cameras.

6.6.1 Design

During the development process some decisions on the integration of the capturing process in the system are made. Due to the required flexibility it is necessary that other cameras, than the one we use, can be utilized with our system. These cameras can differ in the readout process or in the interface, e. g. CameraLink instead of a direct interface.

Our hardware modules consist of an *SP601 Adapter* (refer to section 5.3) and an *Camera Adapter* (see section 5.3) which allow to transmit 12 clock synchronous signals from the connected camera to the Vision Framework. These signals can be control signals or data signals, the synthesized RTL circuits decide how to handle these.

In principle all cameras with direct interface, which use 12 or less signals for communication, can be utilized with the Camera Adapter and the SP601 Adapter. Other Interfaces like CameraLink cannot be used directly, since it uses an LVDS interface that is incompatible to the CTL interface on the SP601 Adapter. See chapter 5 for more information.

To avoid that the whole RTL System has to be re-engineered if other cameras ought to be used, the whole handling of a camera is encapsulated in a

single component. This component has to be implemented for every camera type, corresponding to the required readout process. We introduce an *Camera Interface*, which describes the interface and the internal design of these components. (refer to section 6.7 for more information).

An Camera Interface configures a connected camera and captures images by handling the 12 input signals. For each of our two cameras an implementation of the Camera Interface exists. This concept is comparable to *drivers* known from operating systems. The Stereo Interface uses two Camera Interfaces to receive pixel data of two connected cameras simultaneously. The pixel data is assured to stay in synchronicity and is transmitted to two outgoing streams pairs together with the position in the image.

One advantage of this design is that the Stereo Interface can be removed and the outgoing image data streams can be utilized by other HDL components or an Impulse C process to provide images.

6.6.2 Architecture

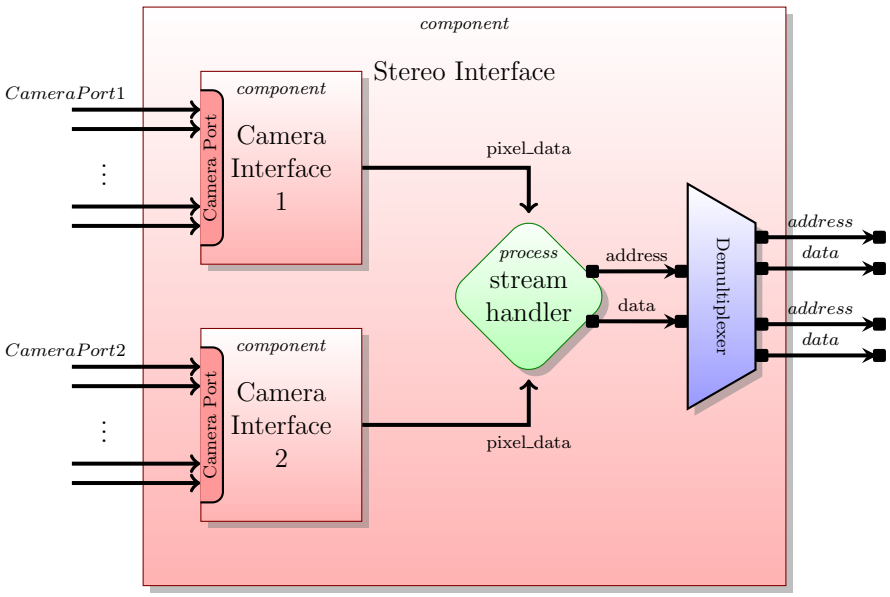


Figure 6.6: Stereo Interface architecture: Simplified data flow from the connected cameras to the outgoing Impulse C stream pairs.

As shown in figure 6.6 the Stereo Interface consists of two Camera Interfaces and a process which handles data output. Each Camera Interface serves as

a driver for a specific camera type and writes successively each line of the captured image to a Block RAM Double Buffer (refer to section 6.5 and to section 6.7).

At the beginning of a new image the *stream handler* process tries to lock the reader ports of both Camera Interface buffers. If both ports can be read, from each buffer a pixel value is read out successively and a new value is generated by concatenation.

This new value is transmitted together with an address to the demultiplexer, where the address is the position of the pixel data in both images. Since the stream handler takes care of data synchronicity, the pixel position is identical in both images. After this, the multiplexer forwards the composed pixel data and the address to one of two stream pairs. The demultiplexer distributes the incoming images alternating to the two stream pairs.

Images are read as RGB555 for each pixel from the Camera Interfaces, thus the outgoing data stream has a bit width of 30 bit. If the camera resolution changes, the address bit width can be adapted accordingly, unless the maximum buffer size of the camera interfaces is not exceeded (see section 6.7). The Vision Framework used in this thesis transmits images with a resolution of 320×240 pixel.

6.6.3 Application and Restrictions

Since only one line of the image can be buffered by the Camera Interfaces, the reader has to read out the buffered image line, before the Camera Interfaces have to write pixel data of the next line.

If the readout process is too slow the Block RAM Double Buffer is not swapped fast enough and image data is lost. Since it cannot be assured that the Vision Core can process the data fast enough a complete image has to be buffered by the Vision Framework.

Another restriction of this component is that both connected cameras must have the same timing characteristic. If one Camera Interface has finished the write operation on its buffer, it is required that the read operation on the other buffer finishes in the near future such that the buffers can be swapped. Since the pixel values are synchronously read from both Camera Interfaces one Camera Interface can block the read operation of the other. In this case this can lead to a locked Block RAM Double Buffer and therefore to lost

image data. Thus, it is necessary to connect two cameras of the same type to the Stereo Vision System to achieve equal timing characteristics.

An image is always transmitted over one stream pair at once. A concurrent transmission over both stream pairs would be desirable, but cannot be implemented since it cannot be guaranteed that both stream pairs work identically. If the stream timings are unequal the stream handler has to wait on the pending stream. This is possible for a short period, if this period takes too long, image data is lost.

6.7 Camera Interface

The *Camera Interface* is the interface and the internal design of a component which configures and reads out a specific camera type (driver concept). Since VHDL does not allow inheritance like other languages, this concept is only a guideline how Camera Interfaces can be implemented.

Each camera type has its own Camera Interface implementation. In our thesis two implementations of the Camera Interface exists: The *OV7 Interface* for the OV7720 based camera and the *MT9V Interface* for our MT9V based camera.

6.7.1 Architecture

In Figure 6.7 the basic architecture of an Camera Interface is illustrated.

Each Camera Interface provides a *Camera Port* which offers a clock signal (camera_clock) to the attached camera, a bus for camera configuration and 12 data inputs which can transmit pixel data and control signals clock synchronous to the received data_clock.

After a system reset or system powering, the run signal is set high to order the *Camera Configurator* (see section 6.8) to configure the attached camera. If the camera is configured an init_done signal informs the *data handler* process that the initialisation process is completed and the image readout can begin. Image readout is performed by the data handler by interpreting the delivered 12 data signals in a synchronous manner to the received data_clock. The image readout is a camera type specific detail and must be implemented according to the used camera. Typically a subset of the signals is used to

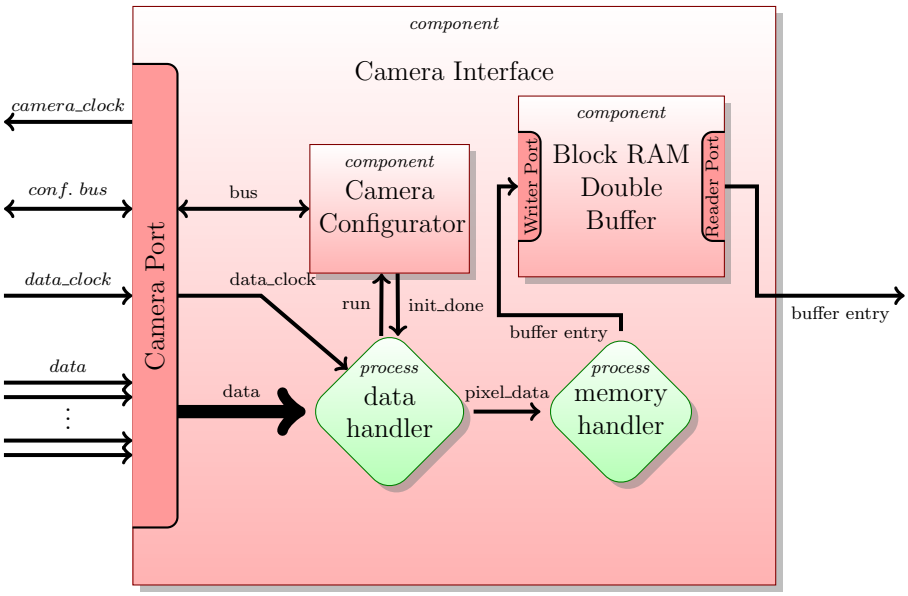


Figure 6.7: Camera Interface architecture: The Camera Interface reads out the cameras and stores the captured image line by line in an internal Block RAM Double Buffer.

transmit the image data and the other part is used to notify when a new image begins and when correct image data is delivered.

The data handler transmits the pixel_data to another process called *memory handler*, which interprets the data. For example a camera can transmit RGB565 in two bytes or Bayer Pattern raw data in one byte. This data has to be converted to a RGB555 pixel format by the memory handler. The processed data is written into a 16 bit value in the Block RAM Double Buffer. After each line the buffer is released and a new buffer is requested (line by line image transmission).

To simplify the implementation of a new Camera Interface, data handler and memory handler are two separated processes.

The Reader Port of the Block RAM Double Buffer can be accessed by an external component to read out the image data in RGB555 pixel format. This component has to be fast enough, such that the memory handler can write a buffer at any time pixel data has to be written. Since typically in image transmission a long interval of not valid image data is between two transmitted image rows, this time can be utilized by the reader to read out the remaining image data and to swap the buffers.

6.7.2 OV7 Interface

OV7 Interface is the Camera Interface implementation for the OV7720 camera boards sponsored by OmniVision Technologies[1]. This component performs camera clocking (25 MHz) and configuration as well as image readout.

The equipped image sensor features a maximum image resolution of 640×480 pixels, transmitted with a maximum frame rate of 60 fps. An internal DSP can be configured to generate the output formats RGB555, RGB565 or YUV. Image sub-sampling, i. e. reduction of the image resolution can be performed by the DSP as well.

Since RGB555 as output format is directly supported by the camera, no color format conversion has to be done. For camera configuration the Camera Configurator (refer to section 6.8) resets the camera and configures the output format as well as the image size.

Since the camera features an internal DSP with clocking infrastructure, as default a higher clock is returned by the camera than the incoming clock. The used serializer/deserializer for data transmission (see section 5.3) does not support higher output clocks than the input clock, it blocks data transmission if the output clock is faster than the input clock. Therefore, an output clock with a speed identically to the input clock is configured.

Image Capturing

The 12 data inputs of the Camera Port are used by two control signals (VSYNC and HREF) and 8 data signals for the image data. The remaining two data inputs are unused.

Figure 6.8 illustrates the image readout process for the OV7720 image sensor, configured to transmit images with a resolution of 320×240 and a RGB555 output format.

All data inputs are clock synchronous to the returned Pixel Clock. A high period of VSYNC for several clock cycles marks the beginning of a new image. If the HSYNC signal is high, valid line data is available and in every clock cycle a byte of the data is transmitted. In case of a RGB555 output format, for each pixel two bytes are transmitted. These bytes include three 5 bit values for red, green and blue.

[1] http://www.ovt.com

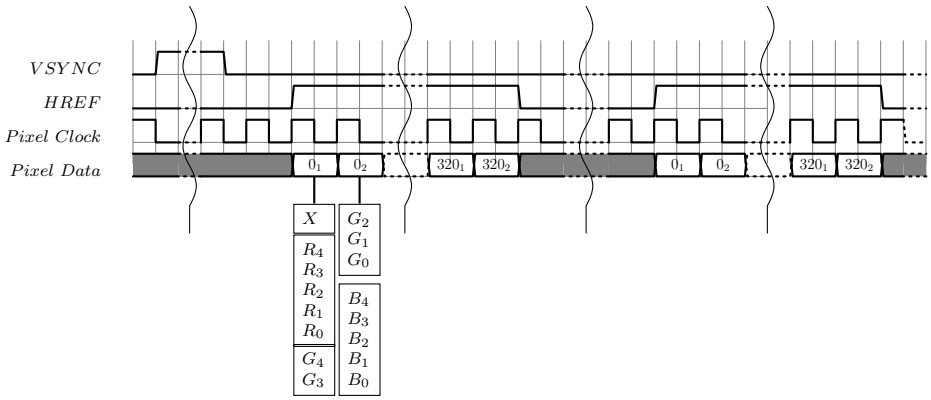


Figure 6.8: OV7720 timing diagram (RGB555, 320 × 240)

The Camera Interface and the attached image data reader can use the period between two valid lines to read the remaining image data and to swap the buffers.

6.7.3 MT9V Interface

The *MT9V Interface* is a Camera Interface implementation for our own MT9V Camera Board (see section 5.4). This component clocks the camera board with a 25 MHz clock generated by the Clock Manager, configures the camera and reads the captured images.

The camera board is equipped with a MT9V032 image sensor from Aptina Imaging Incorporation[2]. The sensor has a maximum resolution of 752 × 480 pixel and a maximum frame rate of 60 fps. Image windowing and binning are directly supported by the image sensor (see section 3.3.2 for an explanation of the terms).

Since the camera does not support RGB or YUV output formats the MT9V Interface has to convert the transmitted Bayer pattern raw data into the RGB555 color format.

Blankings and the active sensor area are configured by the Camera Configurator (see section 6.8). The component firstly resets the camera by setting bits in the corresponding camera registers and afterwards sets the desired camera register values. A pixel clock adjustment as discussed in section 5.3

[2] http://www.aptina.com

is not necessary since the camera output is directly clocked by the input clock.

Image Capturing

In case of the MT9V Camera Module the 12 data signals consist of 10 pixel data plus two control signals (Frame Valid, Line Valid). All signals are clock synchronous to the data clock, the so called Pixel Clock.

Figure 6.9 shows the readout process. Frame Valid signals that a new image begins and is high until image transmission is completed. Line Valid is high during an image line cycle. On every rising edge of the Pixel Clock during this cycle, a new pixel of the current image line can be read by the first 8 bit of Pixel Data. If Line Valid is low, the current Pixel Data is not valid. As for the OV7 Interface, the period between two valid lines can be used by the Camera Interface and the attached image data reader to read out the remaining image data and to swap the buffers respectively.

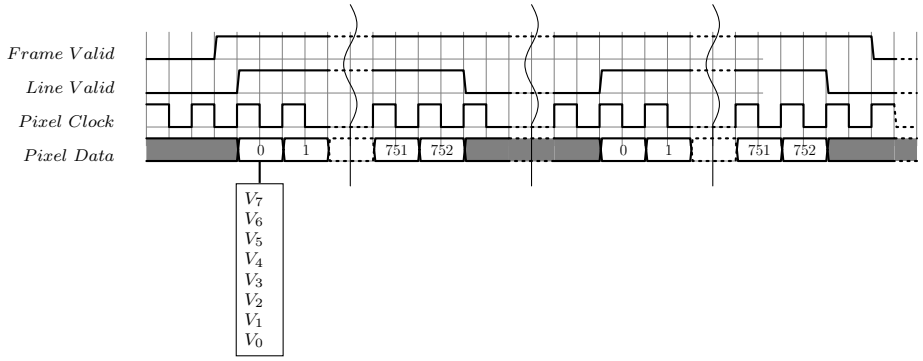


Figure 6.9: MT9V032 timing diagram (Bayer Pattern, 320×240)

Debayering

Since only a resolution of 320×240 is required a resolution preserving debayering like *Bilinear Interpolation* (see section 3.3.2) is not necessary. Therefore, the received 752×480 raw data images are downsized to a resolution of 640×480 by windowing. An image with a resolution of 320×240 pixel is afterwards computed by summing up a 2×2 window of the raw data image to one pixel of the new image.

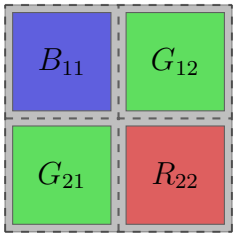


Figure 6.10: Bayer Pattern 2×2 cut-out

A 2×2 window in a Bayer pattern contains the color information of a red pixel, two green pixels and a blue pixel (see figure 6.10). Therefore, the RGB555 color of a pixel p in the downsized image can be defined as

$$\begin{aligned} p_{\mathrm{R}} &= bit_5(R_{22}) \\ p_{\mathrm{G}} &= bit_5(1/2[G_{12} + G_{21}]) \\ p_{\mathrm{B}} &= bit_5(B_{11}), \end{aligned} \tag{6.1}$$

where $bit_5(x)$ returns the 5 upper bits of x.

6.8 Camera Configurator

Each Camera Interface needs a component which can configure the attached camera. This is done by the *Camera Configurator*.

For each camera the developer has to define a configuration vector which contains the necessary write instructions for camera configuration. The developer can also define read instructions to verify register settings.

Each instruction consists of the camera address, the register address and the value which should be written to the register or is expected to be read from the register. If the read values match the expected ones and the camera has acknowledged each instruction, an outgoing signal indicates the completion of the configuration process.

As depicted in chapter 5 the used cameras are configureable over an I^2C bus. Therefore an IP Core has to be selected which handles the transmission of the configuration to the cameras.

6.8.1 IP Core Selection

Several applicable IP Cores exists, but these are mostly commercially offered and price information must be requested. An example is the PHILLIPS I²C-BUS COMPATIBLE BUS CONTROLLER from Evatronix[3]. Since one requirement for the Stereo Vision System is cost effectiveness, we intended to use a free IP Core.

OpenCores[4], a community which publishes core designs under free licenses, offers a wide range of free IP Cores. We decided to use the *I²C controller core*[5], an I²C compatible IP-core under LGPL license with a Wishbone Slave Interface.

The GNU Lesser General Public License (LGPL) is a free software license which enables everyone to use the source code with some restrictions[6]. The software can be modified, distributed and used for all purposes. Modified software which is commercial distributed has to be distributed with its source code and under LGPL licence. If the source code is used without modifications by another software, this software can be licensed under any other licence. Therefore, the LGPL license does not restrict the usage of the I²C IP Core for this work and the usage of this free I²C-Core produces no costs.

The *Wishbone Bus* is an open source microcomputer multi master bus intended to define a standard interface for IP Cores[7]. Many Opencore designs offer this bus as an interface, which is typically used by a soft core micro controller. Since it only defines how information is exchanged it can also be utilized directly from a RTL component.

Figure 6.11 illustrates the architecture of a master slave Wishbone Bus.

Typically data transfer over a Wishbone Bus is done by reading or writing a register of the slave. These registers are mapped to an address space. For example for the I²C-Core with Wishbone interface a register is used to configure the transmission rate, to control data transmission, to read data and to write data.

[3] http://evatronix-ip.com/
[4] http://www.opencores.org
[5] http://www.opencores.org/project,i2c
[6] http://www.gnu.org/copyleft/lesser.html
[7] http://opencores.org/downloads/wbspec_b3.pdf
[8] http://opencores.org/downloads/wbspec_b3.pdf

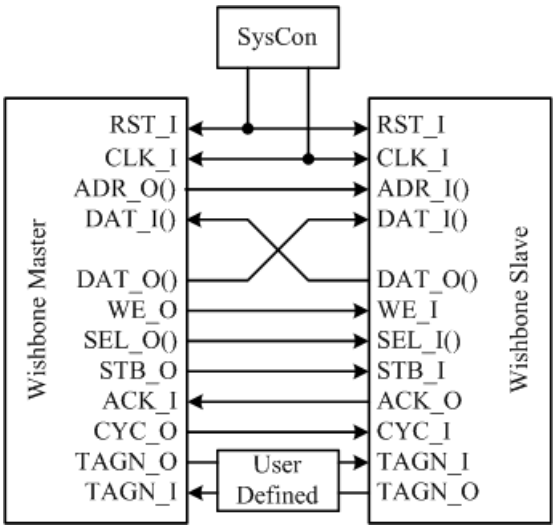


Figure 6.11: Wishbone Bus[8] between one master and one slave (Illustration from Wishbone specification).
Reset signal (RST) and clock signal (CLK) are generated by an external component (SysCon). Data input (DAT_I) and data output (DAT_O) transfer the data between master and slave. The address input (ADR) of the slave defines the address which should be read or write. Write Enable (WE) indicates whether the current bus cycle is a read or write cycle. Select (SEL) selects which bits in the Data input or output are valid, whereas strobe (STB) indicates valid data. With acknowledge (ACK) the slave informs the master that a transfer cycle is completed, whereas the master indicates that a valid bus cycle for the slave is in progress with cycle (CYC). TAGN are user defined signals.

6.8.2 Architecture

In Figure 6.12 the architecture of the Camera Configurator is illustrated.

The *camera configuration vector* contains read and write instructions. Each write instruction consists of the camera address, the register which should be written as well as the value. An automatic verification of each write instruction by a following read instruction is unfortunately not possible, since some camera registers can only be written and are always read as zero. The user can define specific read instructions consisting of the camera address, the register which should be read and the expected value.

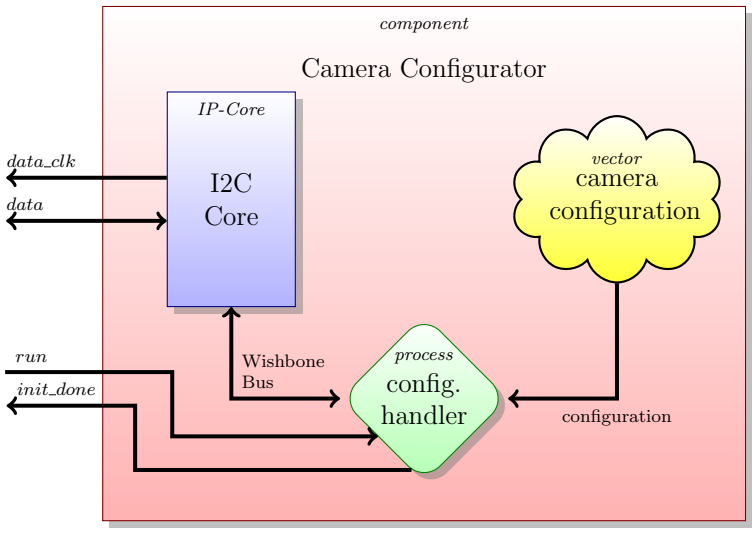


Figure 6.12: Camera Configurator architecture: Basic data flow from an encoded configuration to the attached camera on the I^2C data bus.

After the run signal goes high, the configuration vector is processed by the *configuration handler* process. It controls the I^2C Core over the Wishbone Bus, delivers the read and write instructions to the core and waits for an acknowledgement of the camera. If the last instruction was a read instruction, the expected value and the received value from the camera are compared.

If all instructions are acknowledged by the camera and all read instructions match with the expected values, the init_done signal is set high to inform that the configuration process is successfully completed. The init_done signal remains low if any read value does not match its expected value or one instruction was not confirmed by the camera.

6.9 DDR Interface

The Vision Core can contain different algorithms with different behaviours. For some algorithms it may be sufficient to get the image data pixel by pixel in a sequential order. Other algorithms may need pixel in an arbitrarily order.

Since the image readout process of the cameras cannot be slowed down and the timing behaviour of the pixel access can be arbitrarily a buffering of

the captured images is required. This buffering is performed by the *DDR Interface*.

To read data from the memory and write data to it, we decided to use an Impulse C stream address, data pair as interface for the reader and writer. This is the simplest interface for an Impulse C generated Vision Core and can also be used by other HDL components.

To achieve a high performance it is necessary that the Vision Core must not wait for an image. Therefore, a double buffering is necessary such that the Vision Core can read out one buffer while the other buffer is filled with a new image. Once the Vision Core has finished processing of the image the buffers can be swapped and the next image can be processed without much idle time.

The internal block RAM memory of the assembled FPGA on the the SP601 development board is not sufficient to store two complete images. The only sufficient memory is the DDR2 memory on this board which is very complex to handle because DDR memory needs refresh cycles to preserve the stored data. An IP Core which handles this memory would simplify the usage of this type of memory.

Therefore, an IP Core must be found to handle the DDR2 memory and ease the reading and writing of the memory.

6.9.1 IP Core Selection

Spartan-6 FPGAs offer *Memory Controller Blocks* (MCB) [Xil10c] which can perform the initialisation and refresh of an attached DDR memory. Also it is possible to access the memory over these hard cores which feature support for DDR, DDR2 and DDR3 memory standards. Transfer rates up to 800 Mbit/s are offered when operating in 400 MHz double data rate mode by these high performance blocks (800 MHz clocked memory).

One of two MCBs of the assembled Spartan-6 FPGA are connected to the assembled DDR2 memory on the SP601 development board. To interface this MCB Xilinx offers a tool called *Memory Interface Generator* (MIG) which can be used to generate an IP Core which provides an Interface to this MCB [Xil10d].

An external component can access the wrapped memory over ports. The IP Core can be configured to provide 1 to 6 ports for accessing the memory. These configurations differ in their bit width (32, 64 and 128 bit are

possible) and the access, for example a 6 port configuration with two 32 bit bi-directional ports and four uni-directional ports (read or write). Since only one port can access the physical memory at a particular time the memory access of each port is regulated by the IP-core. The used regulation method can be free defined.

Since multiple ports can operate in *parallel* on the memory and a high data transfer rate should be achieved the read and write operations are implemented with FIFOs.

Each port offers a write FIFO in which the data which should be stored in the memory before a write operation has to be placed. Another FIFO contains the read data from the memory after a read operation. These operations are controlled over a command FIFO. In this FIFO commands can be placed, which describes the direction of the command (read or write) and on which address it is performed. The developer has to guarantee that in case of a write operation no FIFO underrun occurs, i. e. if a write operation is executed data has to be in the write FIFO. And in case of a read operation a overrun must not occur, i. e. when a read operation is executed the read FIFO must have space for a new data. These requirements has to be guaranteed during the whole operation although all operations are buffered in a FIFO.

The MIG generates a whole project, ready for testing on the SP601, which performs different read and write operation patterns on the memory. The needed IP Core is highly integrated in the generated project, therefore the generated code had to be adapted such that the IP Core can be used separately in our system.

6.9.2 Architecture

Since initialisation and refresh cycles are performed by the MCB, only write and read operations has to be controlled by the DDR Interface. We decided to use two 32 bit ports to offer two independent double buffers to the Vision Core.

Figure 6.13 illustrates a simplified architecture of the DDR Interface.

The *DDR Wrapper* has a interface to the hardware memory assembled on the development board. Each port of the DDR Wrapper is controlled by a *DDR Port Interface* instance which handles the *write*, *read* and *command* FIFO of this port.

Each DDR Port Interface instance operates on a separated memory area to read and write data. These data are read and written via two address and data stream pairs, each for one data direction. The reader and writer buffers are swapped when both has finished their operations which is signaled over the corresponding address stream by transmitting a particular address. Refer to section 6.10 for more information.

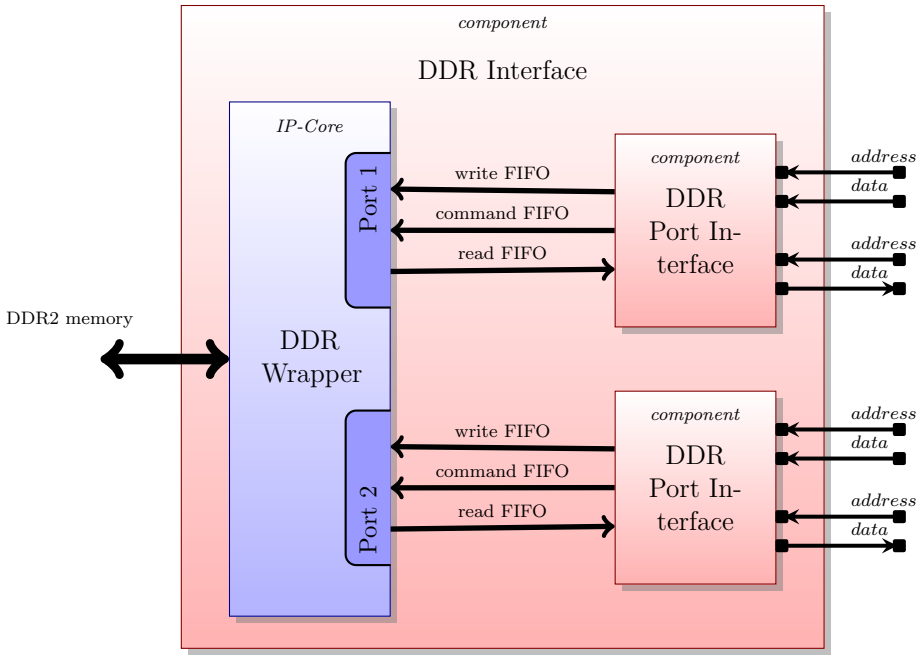


Figure 6.13: DDR Interface architecture.

6.10 DDR Port Interface

Each port of the *DDR Wrapper* is handled by an *DDR Port Interface* instance (refer to section 6.9 for more information).

This component allows a reader and a writer to read/write data in the DDR memory without collisions. To achieve a high performance, reader and writer operate on two different memory regions, so called *buffers*.

Each buffer contains two images, where each buffer entry is the concatenation of one pixel from each image (interleaved image storage). Both pixel have the same position in their image. The buffer sizes can be configured

corresponding to the size of the image which is stored in the buffer. If both, reader and writer, have finished their operations on the buffers, they signal this by transmitting an address over the address stream which is outside of the address range.

The reader can request data from its buffer by an 25 bit address stream and gets the requested data over a 32 bit data stream back. To write data, the writer sets the position in its buffer over the address stream and the corresponding data is send over its data stream.

This component can be used to store any 32 bit value. The double buffer functionality can be deactivated such that both reader and writer operate on the same buffer. Therefore, this component could be used by the Vision Core to cache computation results if both reader and writer stream pairs are connected to the core.

6.10.1 Architecture

The DDR Port Interface consists of three processes.

Incoming address and data stream pairs driven by an external writer are handled by the *write handler* process. In each transmission an address and data is submitted simultaneously. If an address/data pair is received by the write handler it tries to transfer the data part to the write FIFO. If this FIFO is not full, the data can be written to it and the next step is to transfer the address to the *command handler*. This process handles the command FIFO which controls the read and write operations to the memory. To write data to the memory the command handler has to write the address and a write operation encoding to the command FIFO. If the write FIFO and the command FIFO are written in the next cycles of the DDR Wrapper, this data is written to the specified address. Thereafter the write handler acknowledges the transmission over both streams, such that the writer can transmit the next address/data pair.

The readers address and data stream pair is handled by the *read handler* process. Its behaviour is analogous to the behaviour of the write handler. If the reader transmits an address, the read handler process forwards this address to the command handler. This process writes the address and the encoding for a read operation to the command FIFO if the FiFo is not full. After the read command is written to the FIFO, during the next clock cycles the data is written and placed in the read FIFO. If data is in the read FIFO this is detected by the read handler which reads out the data and transmits

it over the data stream to the reader. To minimize idle time, address and data streams are intentionally not synchronous such that multiple data can be requested over the address stream before the first data is returned on the data stream.

If the DDR Port Interface is configured to use double buffering, reader and writer operate on different memory regions called *buffer*. The buffer size can be configured, such that it can store images of different sizes. When the writer has finished operation on its buffer, it signals this to the write handler by transmitting an address outside the buffer address range. The write handler informs the *swap handler* process to swap the buffers by a swap signal. The procedure is analogous for the reader.

If both reader and writer are ready to swap the buffers, the swap handler swaps the address offset of the reader and writer buffer such that each is now working on the other address range. Then each handler is informed by a swap_ready signal that the swap is performed and both can resume work.

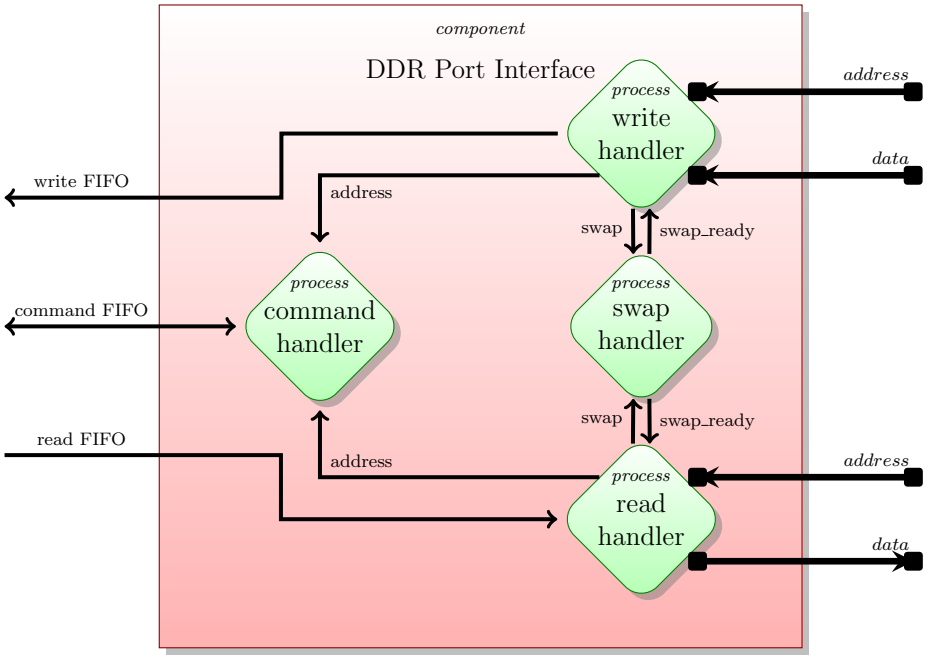


Figure 6.14: DDR Port Interface architecture: Data flow between the read, write and command FIFO interface of the DDR Wrapper IP Core and the stream handling processes.

6.11 Ethernet Interface

In chapter 4 gigabit Ethernet was selected for data transmission to the host PC. This interface should be used to receive data over an Impulse C stream from the Vision Core and deliver the received data to the PC.

The SP601 development board is equipped with a 10/100/1000 Mbit/s Ethernet hardware transceiver which has to be configured to work in gigabit Ethernet mode. To use this transceiver an IP Core has to be selected, which controls the configuration and the data flow to the transceiver.

6.11.1 IP Core Selection

Xilinx offers an *Soft Tri-Mode Ethernet MAC* (TEMAC)[9] which is ideally suited to be used, since it is well documented and easy to interface. It has to be licensed with a commercial licence which costs several hundred dollars. The exact price has to be requested but a licence renewal price could be figured out. The renewal licence cost 1,600 $[10]. A usage of this IP Core does not come into consideration since we want to develop a cost effective system.

Xilinx offers also free Ethernet MAC IP Cores but these feature only transmission rates up to 100 Mbit/s. Therefore, the cores are unsuitable, since the required transmission rates (see section 4.1) cannot be achieved.

Several other gigabit Ethernet MAC IP Cores can be found, but these are all under commercial licences, with expected costs comparable to these of the TEMAC.

OpenCores offers a free 10/100/1000 Mbit/s Ethernet MAC IP Core[11] written in Verilog. This core is licensed under a free LGPL license (see section 6.8.1) and supports 10, 100 and 1000 Mbit/s transfer rates. The core is ideally suited for our purpose but several users report problems about this core.

Most problems were caused by the fact that the core is not written for a usage on an FPGA, it is written only for simulation. Without changes it cannot be synthesized for an FPGA. Therefore, several modifications in the code of the core were necessary to get it working in a real system.

The 10/100/1000 Ethernet MAC Core [Gao06] does not feature a Wishbone Bus interface like the used I^2C Core (see section 6.8.1). Core configuration is

[9] http://www.xilinx.com/products/ipcenter/TEMAC.htm

[10] license: *EFR-DI-TEMAC-SITE* on http://www.avnet.com/

[11] http://www.opencores.org/project,ethernet_tri_mode

performed by a host interface which allows it to read and write configuration register.

Ethernet packets are transfered by the IP Core via FIFOs. Received Ethernet packets can be read from the receive FIFO, whereas Ethernet packets must be placed into the transmit FIFO for sending. Via other control signals *start of package* and *end of package* are marked. Since the core uses a 32 bit interface for data exchange, single bytes can be written by masking the 32 bit values.

Raw IEEE 802.3 Ethernet packets can be transfered with the IP Core. The CRC checksum of the Ethernet packets is calculated and appended by the IP Core. It is not possible to transmit UDP/IP or TCP/IP packets, since the Ethernet packets that are send to the IP Core must have already been framed. UDP/IP or TCP/IP packets would have to be generated by a protocol stack which is very complex to implement. Commercial IP Cores containing these stacks exist. Therefore, an extension of our system to send UDP/IP or TCP/IP packets is possible.

To access the external hardware transceiver, a so called *PHY*, the Ethernet Core provides a *Gigabit Media Indepentent Interface* (GMII). This interface is defined in the IEEE 802.3 standard and transfers data with a 125 MHz clock. The equipped 88E1111 PHY of Marvell Semiconductor[12] features a GMII interface and can be directly connected to the Ethernet IP Core.

Other gigabit Ethernet PHYs feature this interface as well, therefore the Ethernet Interface can be utilized on other development boards with an gigabit Ethernet transceiver without modifications.

6.11.2 Architecture

The architecture of the Ethernet Interface is illustrated in figure 6.15.

An incoming Impulse C stream with a bit width of 32 bit transfers data framed with an Ethernet header (see section 8.4.2). Each 1018 byte a new Ethernet packet begins.

Each of this packets is written by the *stream handler* process to the *Block RAM Double Buffer*. After 1018 bytes are written the stream handler releases the buffer such that it can be swapped, i. e. entire Ethernet packages are buffered.

[12] http://www.marvell.com/products/tranceivers/alaska_gigabit_ethernet_transceivers/Alaska_88E1111-002.pdf

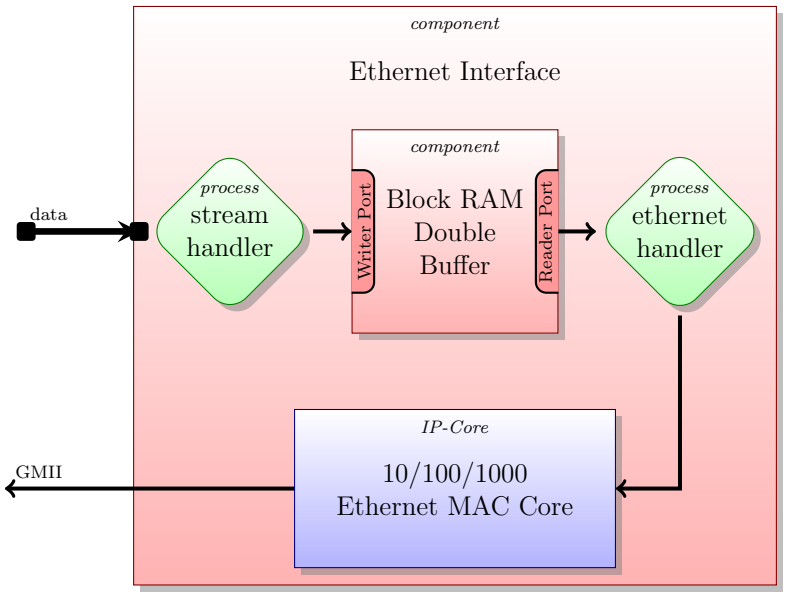


Figure 6.15: Ethernet Interface architecture

The *ethernet handler* process reads the Ethernet packet stored in the buffer. It signals the *Ethernet MAC Core* that a new package begins and transmits the first 32 bit. After the whole package is transmitted it signals the Ethernet MAC Core that the package is complete. After that, the ethernet handler releases its buffer such that the next buffer swap can be performed and the next Ethernet packet can be transmitted.

The *Ethernet MAC Core* transmits received Ethernet packet over the GMII interface to the Ethernet PHY on the development board.

6.12 Design Challenges

During the development of the Vision Framework we have to cope with different challenges, for example not working code or failing timing constraints. In the following an overview over our possibilities to debug not working code is given. Afterwards timing problems and solutions are discussed.

6.12.1 Debugging and Verification

HDL components can become very complex, a VHDL entity definition can have 1,000 lines of code and more, in spite of a good programming style. This entity again can interact with other entities such that the resulting structure gets more complex. Therefore, good debugging mechanisms are necessary to locate an error. In software development this is typically done by using special debugging tools.

In circuit development by HDLs a simple way to debug the generated hardware does not exist. Two different methods to locate errors are applicable [Xil09b].

Testbench Simulation

Testbench Simulation is a method to test a developed RTL component in a testbench specifically written for this component. Typically this method is used to verify a specific behaviour of a component in respect to a given specification. To gain meaningful results the testbench has to behave like the real environment of the component.

The advantage of this method is, that in a HDL simulator each step of the circuit can easily be comprehended. HDL Simulators feature methods known from software development simulators like *Single Step Mode*, *Step Over* or *Breakpoints*.

A disadvantage of this method is that additionally to the component a simulation framework has to be developed. The development effort to create a testbench can be high, since all the necessary connections to the environment must be simulated with their behavior. For example when a component works as a I^2C bus master, the connected I^2C slaves and their behavior have to be simulated. Since the testbench, as well as the component which is tested, can contain errors an erroneous behavior can be caused by both.

Another problem is that, if the component works correctly in the testbench, it is not a guarantee that it will work in the later framework. Therefore, subsequent testing is necessary when the component is embedded in the real system.

Since all signals of a RTL circuit have to be simulated, the resulting amount of data is often underestimated. Several Gigabyte of data can be generated for a simulation of only some milliseconds, depending on the complexity of the RTL circuit.

Chipscope Debugging

Another possibility to locate the cause of erroneous behavior is to use a tool like *Chipscope*. Chipscope is a program that generates a RTL circuit, which is placed on the FPGA and records previously defined signals of the system. The number of signals and the duration of the record can be defined, but is limited since the record has to be stored on the FPGA which consumes block RAM memory.

The advantage of this method is that no extensive testbench has to be written, only the signal names have to be added to the Chipscope block and the signals can be observed in the running system.

A synthesis of our whole RTL System often takes more than 30 minutes, depending on the used optimisation strategies. Therefore, it takes a while until debugging can begin. Frequently it happens that an important signal is not added to the Chipscope block, whereafter the it is not possible to locate the error. Then the missing signal must be added and it must be waited until the new synthesis is complete.

Another problem is that Chipscope uses block RAM, which is then not available for the RTL System. If the RTL System needs a huge amount of block RAM it can lead to an increase of the utilized slices since block RAM memory has to be emulated by distributed RAM assembled on slices. Since the monitoring of the signals can affect the timing behavior of the system, errors can be altered when Chipscope is utilized. In this case debugging is very difficult.

Debugging Approach

During the development process of the Vision Framework different methods were used for debugging.

A testbench for the whole RTL system such that it can be run completely in the simulation was not possible for various reasons. For example the DDR memory which is equipped on the SP601 could not be simulated, the manufacturer does not provide the needed simulation files.

Instead we create testbenches for specific components to test their behavior. For example, for the Camera Configurator a testbench was build such that a correct behavior could be tested without a complete synthesis of the system.

For the Vision Core we developed a testbench which simulates the Vision Framework. This testbench generates the necessary clocks and handles all streams of the Vision Core. It reads out two specified image files and stores the data into buffers. The Vision Core can read out these images over the address, data stream pairs in a unaltered manner. The data which is generated by the Vision Core is received over the outgoing Impulse C stream and written to a file, from where it can be read for further usage.

As mentioned, only single components could be tested via a simulation. Since the complete Vision Framework could not be simulated, testing with Chipscope was the only way to fix some errors.

6.12.2 Clocking and Timing Constraints

HDL languages are used to synthesize Register Transfer Level circuits. These circuits consist of two different elements: Registers and combinational logic. The registers are used to synchronize the data flow between the combinational logic blocks to an attached clock, whereas the combinational logic blocks compute the desired logic functions. Figure 6.16 shows a simplified illustration of a RTL system with a combinational logic block depicted as two inverters.

In each clock step of this circuit data is transfered from an input register to a logic block, which computes a logic function clock asynchronously. The computation of a logic function can take a different length of time. When the clock of the outgoing register is too fast, wrong results are transmitted since the internal logic block does not produce correct results.

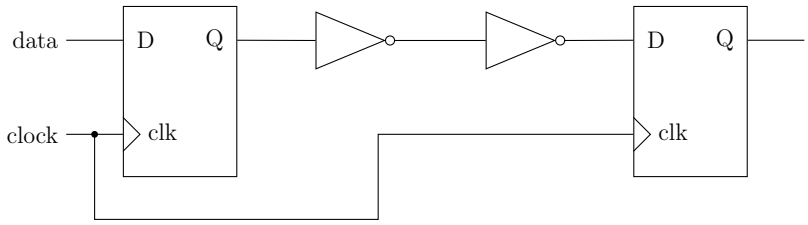


Figure 6.16: Simple RTL circuit without Pipelining

Therefore, for a high performance circuit it is necessary that the register clock and the computation time of the internal combinational logic are coordinated.

In HDL development it is often necessary to create code which can be synthesized to a RTL circuit that is clocked with a desired clock. For example the stream handling processes of the DDR Port Interface have to be fast enough to process the incoming data and deliver commands to the command handler. The delay of the resulting combinational logic blocks can be so long that it is not possible to compute them in one clock cycle.

The obvious solution of such a clock problem is to slow down the driving clock, which leads to an decreasing performance of the system. Another possibility is the utilization of a method called *Pipelining* [Xil09a].

Pipelining

The idea of pipelining is to divide a big combinational logic block into several smaller blocks each separated by a register. The resulting Pipeline needs multiple clock cycles to compute the result of the logic blocks (higher latency) but each clock cycle can be calculated with a higher frequency.

Figure 6.17 shows the result of the pipelining method applied on the previous mentioned RTL circuit.

If pipelining is used to increase the clock frequencies, the timing behavior of signals changes. If former data exchange takes one clock cycle to deliver data to another component, with pipelining multiple clock cycles are necessary to transmit data. This has to be considered during the development process and can prohibit the applicability of pipelining, e. g. when a clock synchronous interface to an IP Core is required.

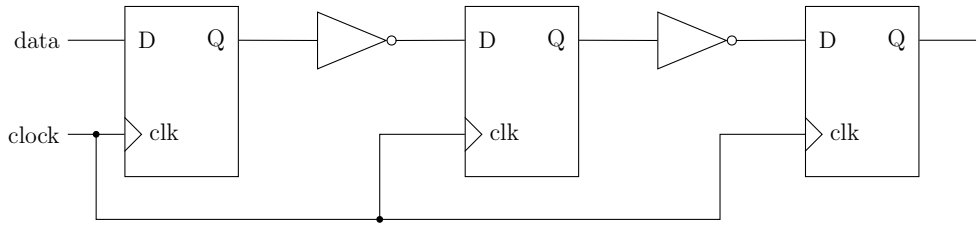


Figure 6.17: RTL circuit with pipelining: A long combinational logic block is separated into two blocks by an additional register.

In the Vision Framework in many components pipelining is used to cope with the timing constraints. For example the DDR Port Interface uses multiple processes only to separate long computation paths.

Chapter 7

Image Preprocessing Comparison

In section 3.5 the fundamentals of the Gaussian, Median, Median Hybrid and Bilateral Filter are discussed. We implemented all four filters in C to be able to compare the quality of the filters for the Stereo Vision System.

The comparison of the image filters is challenging. Because of the histogram optimization possibilities in the Bilateral and Median Filter arbitrary kernel sizes are possible, whereas the Gaussian and Hybrid Median Filter can only be implemented with a maximal kernel size of 5×5 to 7×7. Furthermore, the filters have different parameters to tinker with and different color tolerances and maximum matching search distances are proper for each of the filters. Therefore, this chapter can only cover a small fraction of the input images and parameters that were tested.

Figure 7.1 shows the unfiltered Tsukuba stereo image set from the 2001 Middlebury dataset [SS01b], with the corresponding disparity map originated from the Scan-Line based Segmantation.

The color tolerance and maximum matching search distance that are used in the illustration are 5 and 13 respectively. These values were found to result in the best disparity map for the Tsukuba stereo image set without preprocessing.

In the following sections results for the four image filter with different filter parameters are depicted. The parameters for the Scan-Line based Segmentation are chosen as specified above. This is needed to allow a meaningful comparison, since otherwise there would be so many parameterization pos-



(a) Left image (b) Disparity map (c) Right image

Figure 7.1: Unfiltered left and right Tsukuba image with corresponding disparity map

sibilities that every tested filter could be tweaked to yield very good results for a given input. Additionally implementation ideas and problems for each of the filters are discussed.

7.1 Gaussian Filter

The Gaussian Filter is one of the best known and easiest to understand image filters. The Gaussian formula contains several floating point operations and an exponential function. Since floating point operations are not available on an FPGA and there are to many consecutive floating point operations in the Gaussian formula to convert them into integer operations, the only viable option is a lookup table.

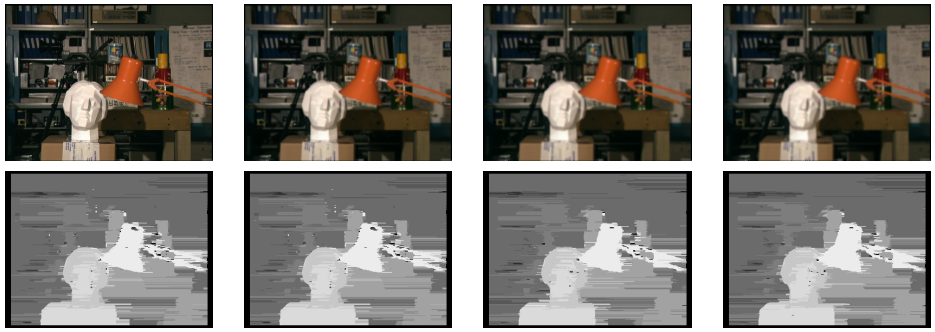
Since the Gaussian Filter has to be applied on each color channel independently three Impulse C processes are required to compute the resulting image. Each of these processes cannot access the same lookup table, since Impulse C does now allow access to global variables. Therefore, the lookup table would have to be stored in block RAM three times to compute the filtered image of one camera.

In addition only a small kernel size ks of maximal 5×5 or 7×7 can be used. A Gaussian filter implementation with a kernel size of 7×7 would need to buffer 7 lines of the input image. Since the filter only operates on one of the three color channels and the stereo vision system has two input images, there would be the need to store $6 \cdot 7$ lines in block RAM. This would pretty much fill the whole available block RAM and force the rest of the system to use distributed RAM. Therefore, 7×7 or perhaps even 5×5 is the maximal kernel size with this approach.

As already described in the fundamentals chapter in section 3.5.1, the Gaussian Filter is linearly separable and thus consecutive one dimensional vertical and horizontal sweeps approximate a two dimensional Gaussian Filter. This would however double the amount of Impulse C processes and thus also double the number of LUT slices that are needed for the implementation on an FPGA.

An index generator that queries the DDR Interface for the needed image data in the correct order is another viable option. Since an access to the DDR memory with aid of the Vision Framework requires more cycles than an access to block RAM, it might however slow down the whole system.

Figure 7.2 shows Gaussian filtered images of the Tsukuba stereo image set with a kernel size of 7×7 and standard deviations of $\sigma = 0.15$, $\sigma = 0.50$, $\sigma = 0.75$ and $\sigma = 1.00$ as well as the corresponding disparity maps.



(a) $\sigma = 0.15$ (b) $\sigma = 0.50$ (c) $\sigma = 0.75$ (d) $\sigma = 1.00$

Figure 7.2: Gaussian filtered images with $ks = 7 \times 7$, different standard deviation σ (top row) and corresponding disparity maps (bottom row)

The first thing that stands out when examining the results is that already disparity map (a) with an standard deviation of only $\sigma = 0.15$ contains far less noise than the disparity map from the unfiltered input images. As expected there is less noise with bigger σ values.

A kernel size of 7×7 is not big enough to blur out some of the smaller features of the input image, such as the text on the board in the background. The reflections on the lamp are getting less apparent with growing σ, they do however not disappear completely.

When only the filtered images are viewed, they do not look as if they were fitter for the Scan-Line based Segmentation than the unfiltered images. The disparity maps are however of unexpected good quality. In disparity map (d) nearly no noise is present. Only the holder of the lamp appears to be a problematic area, a big white block instead of the thin holders can be seen.

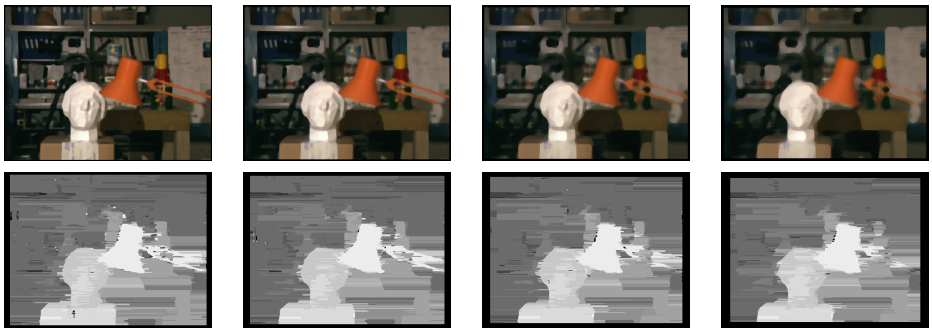
7.2 Median Filter

An implementation of the Median Filter on an FPGA is comparably simple. There are no floating point operations needed, a sorting of the filter kernel yields the result. This obvious implementation does have a rather bad runtime, again only a filter kernel size of 5×5 to 7×7 would be possible.

The Median Filter, contrary to the Gaussian Filter, can however be optimized with the histogram idea presented in section 3.5.2. With an index generator to produce indices that are needed for updates of the histogram, the median can be determined independently from the filter kernel size.

Thus, an Impulse C implementation of the Median Filter is comparably fast and does not need to store any data in block RAM.

Figure 7.3 illustrates the Median Filter on the bassis of median filtered Tsukuba test images with kernel sizes of $ks = 5 \times 5$, $ks = 7 \times 7$, $ks = 9 \times 9$ and $ks = 11 \times 11$.



(a) $ks = 5 \times 5$ (b) $ks = 7 \times 7$ (c) $ks = 9 \times 9$ (d) $ks = 11 \times 11$

Figure 7.3: Median filtered images with different kernel sizes (top row) and corresponding disparity maps (bottom row)

A decrease of noise in the disparity map can be seen with increased kernel size, however also the sharp edges disappear. For example the lamp in disparity map (a) with $ks = 5 \times 5$ has a relatively sharp outline, whereas the lamp in disparity map (d) with $ks = 11 \times 11$ appears to be more blurry.

The median filtered images exhibit less detail with increased kernel size, e. g. in filtered image (d) the lamp does not possess reflections anymore. It looks rather cartoon like, which is good for the Scan-Line based Segmentation. Though it does not retain sharp edges, hence objects in the disparity map become blurry.

7.3 Hybrid Median Filter

The Hybrid Median Filter, in contrast to the Median Filter, can not use the histogram optimization approach, as explained in section 3.5.2. It also is not linearly separable like the Gaussian Filter, i. e. it cannot operate in two separate horizontal and vertical sweeps.

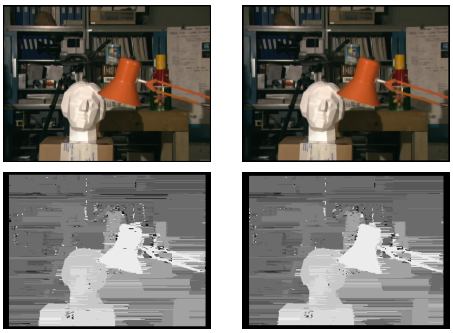
Thus, an implementation with a maximal kernel size of only 7×7 could fit on the FPGA that is utilized in our Stereo Vision System.

Figure 7.4 shows Tsukuba test images that are hybrid median filtered with kernel sizes of $ks = 5 \times 5$ and $ks = 7 \times 7$. No other kernel sizes are meaningful, since a kernel size smaller than 5×5 is to small to have any effect on the Scan-Line based Segmentation and a kernel size greater than 7×7 is not implementable on the utilized development board.

The disparity maps that result from the hybrid median filtered images feature a little less noise compared to the disparity maps resulting from unfiltered images. Compared to the results of median filtered or Gaussian filtered images they are however of lesser quality.

Only small differences between hybrid median filtered input images (a) and (b) can be observed. Objects in both retain sharp edges compared to the median filtered images with the same kernel sizes, however small features like the black writing in the background are not blurred out.

Thus, it can be concluded that a kernel size of 7×7 is to small for the Hybrid Median Filter to have a meaningful effect on the Scan-Line based Segmentation. This is probably the case since only a few of the pixels in the filter window are considered in each run.



(a) $ks = 5 \times 5$ (b) $ks = 7 \times 7$

Figure 7.4: Hybrid Median filtered images with different kernel sizes (top row) and corresponding disparity maps (bottom row)

7.4 Bilateral Filter

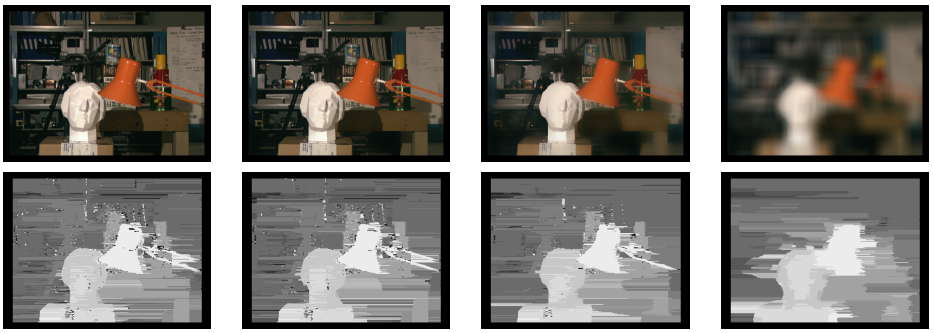
The Bilateral Filter is the most complex of the considered filters. It is however usable on the FPGA if a spatial box filter for the geometric distance weighting function and the histogram optimization are used (see section 3.5.4).

If a Gaussian function for the intensity distance weighting function is used, a lookup table can be stored for fast access. Compared to the Gaussian Filter this Gaussian function is only one dimensional and therefore only a small lookup table has to be stored.

Implementing the Bilateral Filter might be more challenging than implementing the other filters, it will however have a faster run-time than the Gaussian Filter or Median Hybrid Filter.

Figure 7.5 illustrates bilateral filtered Tsukuba test images with a kernel size of 25×25 and different standard deviation values for the Gaussian intensity distance weighting function as well as the corresponding disparity maps.

One can see, that small artifacts like the black writing in the background get blurrier with increased σ. Filtered image (c) shows a very nice mix of sharp edges and blurred out details. For example the small black dots that can be seen in images (a) and (b) are bleached out completely in the bilateral filtered image with $\sigma = 0.15$.



(a) $\sigma = 0.00$ (b) $\sigma = 0.05$ (c) $\sigma = 0.15$ (d) $\sigma = 1.00$

Figure 7.5: Bilateral filtered images with $ks = 25 \times 25$ and different standard deviation σ (top row) and corresponding disparity maps (bottom row)

Although the filtered image does feature nearly exactly the properties that are needed by the Scan-Line based Segmentation, the resulting disparity map shows some noise and most notably errors at the left and the right from the bottom of the bust. This happens because the value of 5 that was chosen for the color tolerance at the beginning of this chapter is to high for the bilateral filtered images. Since the edges are quite sharp, smaller color tolerance will lead to better results.

It is however noteworthy that disparity map (c) is the only disparity map of all of the disparity maps that are shown in this chapter, which does not have a big white block at the place of the holder of the lamp. Only a small part of the upper holder is missing, otherwise the disparity data for this part of the disparity map is very good.

Bilateral filtered image (d) shows the result of a comparably high σ value of 1.00. It is blurred out as one would expect it from a Gaussian Filter. This shows that, since a big filter window is possible because of the histogram optimization, the Bilateral filter can approximate a Gaussian Filter if kernel size and standard deviation are chosen accordingly. A smaller kernel size with a high standard deviation will produce results comparable to the results in figure 7.2 of section 7.1.

In the following the presented image filter results are compared and it is specified which image filter is best fitting as the preprocessing component for the Scan-Line based Segmentation.

7.5 Conclusion

There is no clear winner in the image preprocessing comparison when only the resulting disparity maps are considered. Because of the properties of the Bilateral Filter, one would have assumed that it performs best as a preprocessing filter for the Scan-Line based Segmentation.

Several reasons for the bad performance of the Bilateral Filter can be given, firstly in the whole comparison the same color tolerance and maximum matching search distance were used. While a good maximum matching search distance only depends on the maximum distance between an object in the left and the right image, a good value for the color tolerance can change if an image filter changes color transitions from one object to another. Exactly this happens when the Bilateral Filter is used because of the edge enhancing properties.

Secondly the Tsukuba stereo image set does not exhibit any noise and the two images compose a perfect rectified epipolar geometry. Our own Hardware System will not be able to produce images of this quality. The bilateral filter will show better results compared to the other tested image filters, since it is the only tested filter that can reliably remove small features from the input images (e. g. writing in the background and black dots at the front of the table). Results for bilateral filtered images that are taken with our cameras can be seen in section 9.2.

We chose to implement the Bilateral Filter in the Vision Core, since it can approximate the Gaussian Filter with the right parameterization. Also the calculation part of the Bilateral Filter can easily be substituted with a simple access to the median value to yield an implementation for the Median Filter.

Therefore, it will be possible to switch between a Gaussian Filter approximation, a Median Filter and a Bilateral Filter in the Vision Core. This allows to adjust the image preprocessing subsystem according to the respective environment. Thus, we can assure a good image filtering quality for the Scan-Line based Segmentation, although a clear winner in the comparison could not be determined.

In the following the implementations of the preprocessing part of the Vision Core as well as the vision algorithm and the postprocessing part are discussed and described in detail.

Chapter 8

Vision Core

The Vision Core is the heart of the Stereo Vision System. It contains the utilized stereo vision algorithm, conducts the necessary preprocessing and packages output data such that it can be transmitted to a host PC.

These different parts of the Vision Core are integrated in a generic design, which allows exchange of different parts and an easy advancement of the project. In the following the design concept that allows this genericness is described.

8.1 Design

The Vision Core is divided into several processes, figure 8.1 gives an overview of all used processes and their interconnection with streams. It is contained in two VHDL components, which are the *DDR Interface* and the *Ethernet Interface* provided by the Vision Framework (refer to chapter 6 for more information). Impulse C nodes communicate only over these two components with the rest of the system.

Each image that is taken with the cameras traverses the following path in the Vision Core: The *Index Generator* (IG) generates indices that are in the correct order to be used in the *Bilateral Filter* (BF), i. e. indices to the pixels in the filter kernel that are added or removed when the filter kernel window is moved.

With the information from the Index Generator the VHDL DDR Interface can produce the correct pixels and send them to the *Bilateral Distributor*

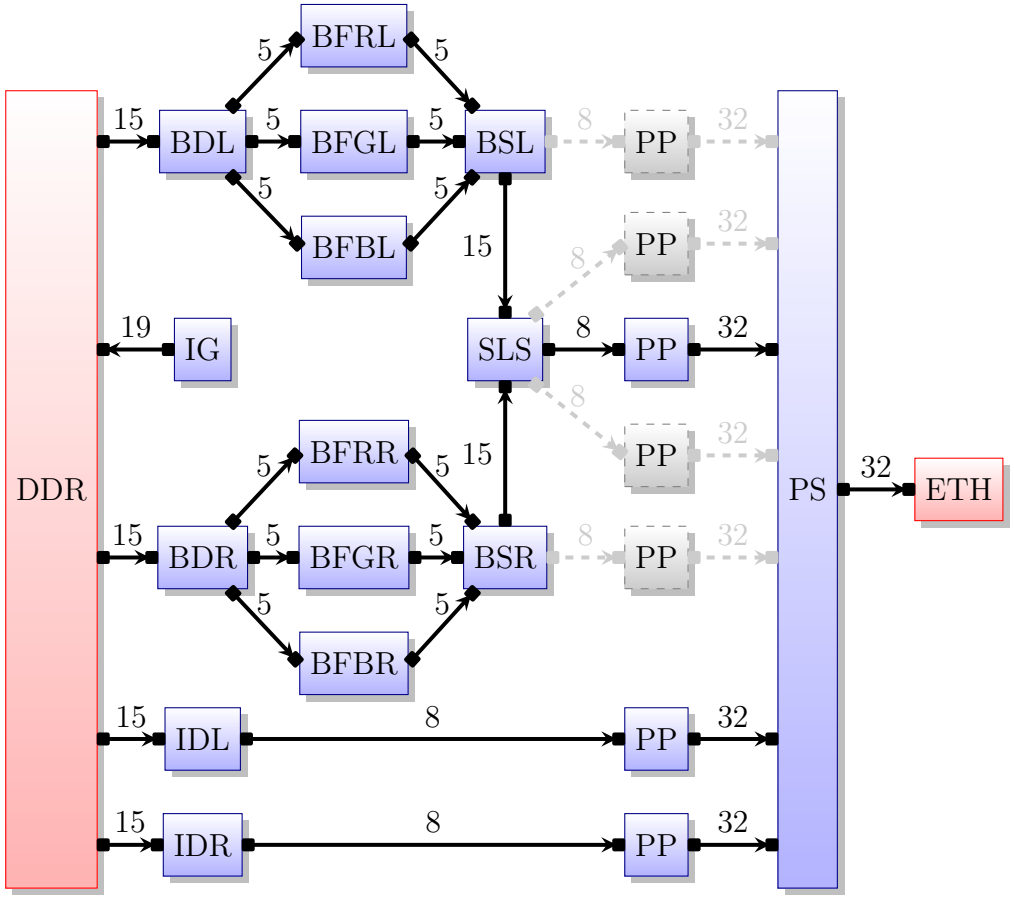


Figure 8.1: Impulse C process flow diagram of the Vision Core. Impulse C processes consist of Index Generator (IG), Bilateral Distributor for left and right image (BD(R|L)), Bilateral Filter for the left and right image as well as for the red, green, and blue color channels (BF(R|G|B)(L|R)), Bilateral Serializer for left and right image (BS(L|R)), the Scan-Line based Segmentation (SLS), the Packet Producer (PP) and the Packet Serializer (PS). (DDR) and (ETH) are the DDR Interface and the Ethernet PHY Interface. Red processes are written in VHDL and blue processes are written in Impulse C. The dashed streams and processes are optional (for debugging purposes) and the numbers are bit depth of corresponding streams.

(BD) processes which subsequently segment the image data in six different color channels to be processed by 6 parallel Bilateral Filter processes (for the red, green and blue color channels of two input images). Thereafter the filtered image data is merged again with the *Bilateral Serializer* (BS) processes.

Computing the actual disparity information is done with the Scan-Line based Segmentation process. In the basic design this is done in one single process (see section 8.3.3 for possible extensions).

After the computational part of the system is done, the computed data is combined into IEEE 802.3 conform Ethernet packets in the *Packet Producer* (PP). The Ethernet packets are merged into one continuous stream via the *Packet Serializer* (PS) to be handled by the VHDL Ethernet Interface component (ETH).

In parallel to the processes described above, the plain image output of the cameras is also packaged and subsequently serialized and send to the host PC via the Ethernet Interface.

In the following the reasons behind the partitioning in VHDL/Impulse C as it is done, the methodology of the process and stream structure and the complexity of the overall system are described in detail.

8.1.1 VHDL/Impulse C partition

The cameras in our system have to be read out in a HDL, since they are connected to general purpose pins on the SP601 which are not accessible by Impulse C generated code (see section 5 for detail).

Furthermore, the selected I^2C IP Core, which is utilized to configure the cameras uses an Wishbone interface that cannot be directly accessed with Impulse C (see section 6.8).

The utilized Ethernet IP Core, responsible for results transmission, has an FIFO interface which requires the handling of several signals (see section 6.11). Since Impulse C does not feature an access on single signals, a HDL component as interface is necessary as well.

Since different vision algorithms exhibit a different timing behavior and timing constraints during image readout must be met, buffering of the images is required. The Memory Control Block, which interfaces the DDR memory on SP601, has also an interface which is not directly accessible by Impulse C processes (refer to section 6.9 for detail information).

Therefore, the RTL System cannot completely be written in Impulse C. A partition in Impulse C and VHDL parts as well as a communication interface between these parts is needed. There are two possible ways to communicate between Impulse C and VHDL. The first is to use external HDL hardware

functions as described in section 3.2.3 and the second is to utilize the standard Impulse C stream communication object. Since the stream communication objects have internal buffers, which are needed anyway to communicate between VHDL and Impulse C in a fast manner, we chose to implement the Impulse C stream protocol in VHDL.

Since one of the goals of this thesis is to investigate the capabilities of Impulse C and to study power dissipation, resource requirements and performance of Impulse C processes, the part of the system that is written in Impulse C is as big as possible. Two VHDL components are used as input and output streams to communicate with the Vision System.

One VHDL component generates image data in the correct order for the corresponding Impulse C processes. The image is read from a double buffer in the DDR memory, where it is stored after reading from the cameras. In the simulation, this process is replaced by a process that reads an image from a file (see section 8.5).

The second VHDL component reads the generated packages from the Packet Serializer and sends them to the Ethernet PHY of the SP601. In the simulation, this process is replaced by a process that writes the packages to a file and a small program that opens a socket and sends the data like the real hardware would do.

With this design it is possible to write the whole computational part of the Stereo Vision System in Impulse C. Another advantage is the possibility to simulate a large part of the system on the host PC, since Impulse C allows compilation with a standard C compiler, as described in section 3.2.

Additionally the utilized partitioning allows flexibility and genericness. In the following the ideas behind the proposed generic design are discussed.

8.1.2 Generic Design

One of the objectives of this thesis is to make a generic system that is flexibly usable and extensible by other people in further projects (see section 4). The methodology that was used in the design of the Vision Core endorses this.

It is possible to coarsely divide the Vision Core the three subsystems *stereo algorithm*, *preprocessing* and *postprocessing*, comparable to the proposed architecture in [AHK07]. All subsystems have Impulse C streams as input and output and can therefore be used independently in different Impulse C projects.

Stereo Algorithm

The main computational task in the Vision Core is the stereo algorithm. In the standard version of the Vision Core, as mentioned above, it consists of only one process that has two input streams (left and right image), one disparity output stream and two optional debugging output streams.

Since we assume images with a bit depth of 5 per color channel as input, the two Impulse C input streams have each a bit width of 15 bit for the red, green and blue color channels. The disparity output is directly connected to a Packet Producer process and needs therefore a bit depth of 8. Calculated disparity values are scaled between 0 and 255 to use the full bit width. Hence, it is possible to utilize the disparity map as a grey color image without further conversions.

It is possible to activate the two optional debugging streams. They transmit the segmentation of the right image and the matching positions of the segments on the left image. The 5 bit red, green and blue color values are each scaled up to 8 bit and send to corresponding Packet Producers.

Another stereo vision algorithm can be tested in our system without the need to change any of the preprocessing or postprocessing Subsystems. Only the two 15 bit input and the 8 bit output stream have to be used. It is therefore easily possible to test other algorithms while utilizing the image filtering and the output over Ethernet.

A chain of processes with two 15 bit input streams at the beginning and one 8 bit output stream at the end is also possible as a replacement for the stereo algorithm. An example of this is the parallelized Scan-Line based Segmentation as described in section 8.3.3.

The generic preprocessing subsystem, which produces the input data for the stereo algorithm is described in the following.

Preprocessing

A generic concept composed of Distributor, Filter, Serializer and Index Generator has been conceived for the preprocessing subsystem. The idea is, that an Index Generator generates indices of pixels that are then split into their color channels to be filtered and afterwords merged again with the serializer. This cycle is depicted in figure 8.2.

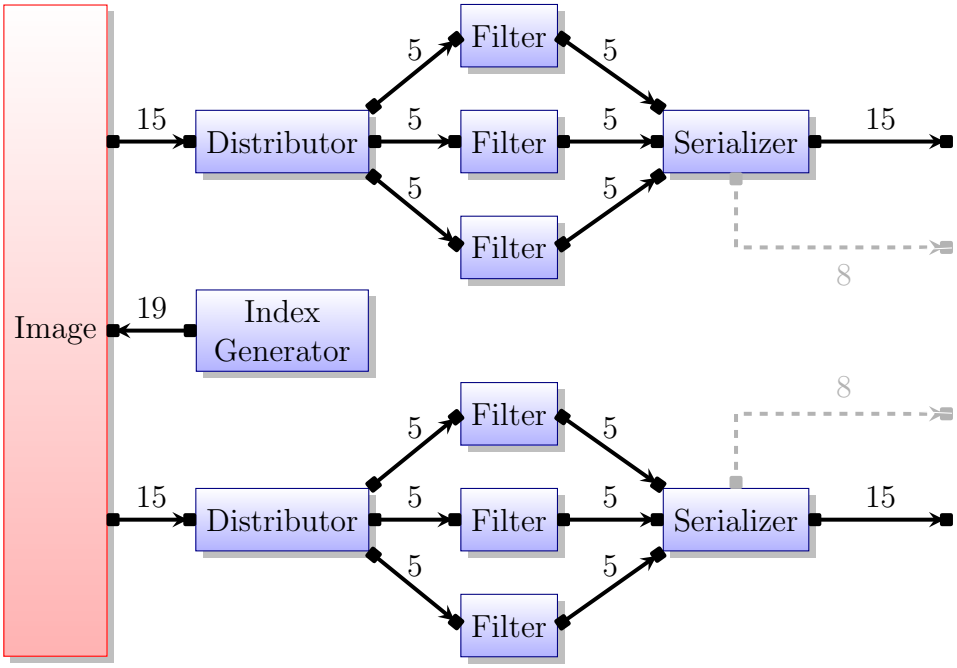


Figure 8.2: Generic image preprocessing subsystem with red VHDL processes and blue Impulse C processes. Numbers depict stream bit width and greyed out dotted streams are optional (for debugging purposes).

The Index Generator can be adapted for a variety of filters. In the standard Vision Core it produces indices that belong to the pixels that are added or removed when the kernel window is moved, so that the histogram in the Bilateral Filter (see section 8.2) can be updated without the need to buffer any image data. The Median Filter with the histogram optimization is usable with the same Index Generator. An Image filter that acts upon another concept, e. g. a Hybrid Median Filter, needs an adapted Index Generator which generates the indices in the correct processing order of the pixels.

Indices are handled by a VHDL component that sends the corresponding pixel from both input images via Impulse C streams to two Impulse C distributor processes. The streams have a bit width of 15 and they expect an RGB-triple with a bit depth of 5 for each color channel. In the Vision Framework a VHDL component accesses the DDR-SDRAM on the SP601 that double buffers the input image data. This process can however be replaced with another process that supplies the image data by other means, e. g. by directly excessing the camera or via an external input source over Ethernet.

Distribution and serialization of the pixel data rarely needs to be altered. Most image filter algorithm operate on one color channel only. With the distributor and serializer processes six filters can be executed in parallel, based on the left and right input images. The distributor simply splits the 15 bit input pixels into red, green and blue color channels, which are fed into the three filter via Impulse C streams of bit width 5.

After the filtering process the belonging color channels are streamed into serializer processes. These processes merge the three color channels into an RGB-triple again. Besides the normal output stream that is connected to the stereo algorithm, an optional debug output stream can be used. It sends data directly into a Packet Producer process and therefore allows the examination of filtered input images on the host PC.

Filtering itself is done in a single process in the Vision Core. All the tested filters (Gaussian, Median, Median Hybrid and Bilateral) are not easily parallelized or pipelined into several Impulse C processes. But, comparable to the stereo algorithm, it is possible to use a chain of processes that has a 5 bit input and a 5 bit output stream, which are connected to the distributor and serializer respectively.

Besides the preprocessing subsystem the Vision Core needs a postprocessing step that prepares the output of the stereo algorithm for the transmission to the host PC.

Postprocessing

After the stereo algorithm is done, the computed data has to be packaged in packets that can be send via Ethernet. A Packet Producer for every Impulse C output stream and a Packet Serializer that serializes all produced packets for an VHDL Ethernet process are used. Figure 8.3 illustrates this concept.

The whole postprocessing subsystem is very generic. It is completely independent from the rest of the Stereo Vision System or even vision algorithms in general. Every Impulse C process that has an outgoing Impulse C stream object with a bit width of 8 can be connected to the postprocessing subsystem to transfer data over Ethernet to the host PC.

Each Packet Producer generates 1018 byte long packets that are conform to the IEEE 802.3 Ethernet standard from the 8 bit input values (see section 8.4.2). This is done by attaching an Ethernet header at the front of each packet.

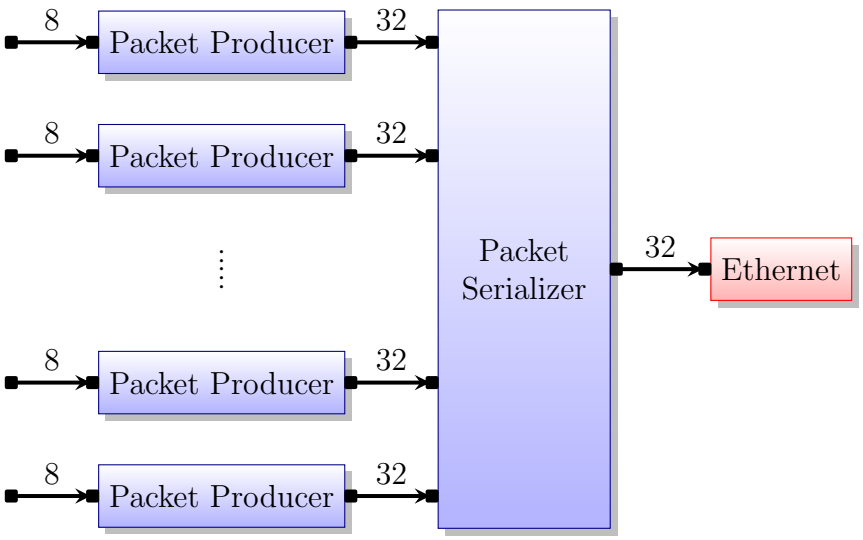


Figure 8.3: Generic postprocessing subsystem with red VHDL processes and blue Impulse C processes. The numbers depict stream bit width of the corresponding streams.

Completed packets are send to the Packet Serializer, which buffers a maximum of one complete Ethernet packet per Packet Producer. These packets are then successively transmitted via an Impulse C streaming object to the Ethernet Interface component in 32 bit parts.

The Ethernet Interface component is a VHDL entity that double buffers the 32 bit input values and controls the Ethernet IP Core. If our system ought to be ported to another FPGA board which features an gigabit Ethernet PHY the Ethernet Interface can be utilized since the Ethernet IP Core has a standardized GMII interface as connection to the PHY.

One possible purpose for the preprocessing subsystem, alongside the output of image and disparity data, is debugging. Sometimes it is not easy or even not possible to debug a complex Impulse C system with a VHDL simulator such as ModelSim or a hardware debugger such as ChipScope. The VHDL code, that is generated from the Impulse C subset of the C programming language, is not easily matchable to the original C code.

Therefore, some of the more abstract parts of an algorithm, such as the value of an iterator in a for-loop at a particular location, cannot be debugged with these standard VHDL debugging tools. With the introduced postprocessing subsystem it is however possible to create a new Packet Producer that receives

the iterator value as an Impulse C stream input. The values of interest can then be read out with a simple program on the host PC (a detailed description on how this works can be found in section 8.6).

In our system this is used to output the filtered images as well as debugging data from the Scan-Line based Segmentation such as the segments of the segmentation or the matching of the segments in the corresponding image.

The genericness and flexibility of the proposed Vision Core design also yields a high complexity.

8.1.3 Complexity

The presented Vision Core uses an uncommonly complex Impulse C configuration. Most publications on vision systems that utilize Impulse C, such as [TS07] or [KS09], use only one process that computes the actual image filter or image algorithm. The remaining part of the system is done on a coprocessor that processes the data. With this approach e. g. tasks such as the controlling of the Ethernet PHY can be done by the Linux kernel.

Our unusual approach however has several advantages but also some disadvantages.

Advantages

One obvious advantage is that our system does not need any coprocessor, such as an external PowerPC as present on the ML405. Hence, it was possible to use a comparably cheap FPGA board.

Another benefit of our design is that after this framework of preprocessing, stereo algorithm and postprocessing is implemented and tested extensively, the system can be easily used by software engineers without experience in VHDL or low-level hardware programming. Someone that only knows the C programming language can e. g. test a new image filter in the Vision Core by just substituting the C code of the Bilateral Filter with the new filter code.

It also is possible to implement a new image algorithm that runs in parallel to the existing system and uses its own Packet Producers. Most of the complexity that normally has to be considered when such an image algorithm is implemented on an FPGA is already contained in our Stereo Vision System.

Disadvantages

The complex Impulse C framework also introduces some disadvantages. One is, that in general an Impulse C process uses more slices and is slower than a comparable hand crafted VHDL entity. Since the computational system is written in Impulse C, it uses a relatively large amount of slices and parts like the Packet Serializer that are used by every process on the device might slow down the whole system. Whereas, if only a small part of the system would be written in Impulse C, neither the increased slice use nor the slowness would be dramatic.

Another disadvantage and probably the biggest problem is the impossibility to comprehend the generated VHDL code. If only one small Impulse C process is used, the generated VHDL code can easily be understood and therefore debugged when not working. As described in section 8.1.2 is it however possible to use the postprocessing subsystem to generate debugging data that can be read and interpreted on the host PC. This works very well, but only as long as the system does not come to a standstill. This can happen if e. g. an accidental deadlock occurs or some of the Impulse C peculiarities are not considered (see section 3.2.3).

Whenever something like this happens the bug can only be found by trial and error, i. e. removing parts of the system until it functions and then reinserting the functionality in small parts to find the part that is erroneous. This approach leads to another problem. Since the System is complex and uses lots of slices, the synthesis of the whole system can take up to several hours. Which makes the debugging process very cumbersome and might outweigh the rapid development that is otherwise possible with Impulse C.

Also the complexity of our design and Impulse C possesses some disadvantages, the implementation of the Bilateral Filter and the Median Filter, as described in the following, would not be possible without either of the two.

8.2 Bilateral Filter and Median Filter

As described in section 7.5 we decided to implement the Bilateral Filter and the Median Filter. The latter can easily be created from an implemented Bilateral Filter be replacing the convolution part of the Bilateral Filter with a selection of the median value from the Histogram.

Thus only the implementation of the Bilateral Filter is depicted. The convolution part of the Bilateral Filter, which has to be replaced for the Median Filter, is described in the following.

8.2.1 Convolution with Histogram

Since the theory and the optimization possibilities of the Bilateral Filter are already described in detail in section 3.5.4, the derivation from the theoretical formula to the discretized and optimized version of the Bilateral Filter will not be discussed here. The optimized Bilateral Filter formula (3.14) reads

$$I'(x,y) = \frac{\sum\limits_{(\xi,\psi)\in\Omega} I(\xi,\psi) f(I(\xi,\psi) - I(x,y))}{\sum\limits_{(\xi,\psi)\in\Omega} f(I(\xi,\psi) - I(x,y))}$$

as described in [Wei06], where (x,y) is a pixel coordinate, $I(x,y)$ is the intensity of pixel (x,y), f is an arbitrary weighting function for the edge enhancing part of the Bilateral Filter, Ω is is the set of pixels that are in the filter kernel radius r of (x,y) and $I'(x,y)$ is the intensity value of the filtered image at location (x,y).

If histogram H for filter kernel Ω of center pixel (x,y) is given, it is possible to make the formula independent from the size of the filter kernel. Since the number of occurrences of an intensity i is converted into the histogram, it is possible to describe the sum of all

$$I(\xi,\psi) f(I(\xi,\psi) - I(x,y)) \tag{8.1}$$

with $I(\xi,\psi) = i$ in the single term

$$i \cdot f(i - I(x,y)) \cdot H(i). \tag{8.2}$$

This has to be done for all i from 0 to the biggest possible intensity to get the same value as for the sum in the numerator of formula (3.14). For a color channel bit depth d, which is 5 in our system, the following holds:

$$\sum_{(\xi,\psi)\in\Omega} I(\xi,\psi) f(I(\xi,\psi) - I(x,y)) = \sum_{i=0}^{2^d-1} i \cdot f(i - I(x,y)) \cdot H(i). \tag{8.3}$$

As the weighting function f a Gaussian function G_σ is used. A Gaussian function can relatively easy be implemented with a lookup table (see section 8.2.5) and is capable of being parameterised with the standard deviation σ to be adjusted for our needs.

Therefore, the actual convolution of the Bilateral Filter as it is implemented can be described as

$$I'(x,y) = \frac{\sum_{i=0}^{2^d-1} i \cdot G_\sigma(i - I(x,y)) \cdot H(i)}{\sum_{i=0}^{2^d-1} G_\sigma(i - I(x,y)) \cdot H(i)}. \tag{8.4}$$

That the convolution is independent from the filter kernel size can easily be deduced from this formula. The complexity depends only on the color bit depth d.

Convolution is implemented with a hardware primitive function (see section 3.2.3 for operation and constraint details on hardware primitive functions). Input values for the function are the current center pixel and a pointer to the histogram array.

The former is an unsigned 5 bit value and the histogram array has a length of 32, which equates to 2^d for the used color bit depth of $d = 5$. Elements of the array can hold 10 bit values. 10 bit values are used since a filter kernel radius of $r = 25$ is utilized in our system. With a filter kernel radius of 25, the maximum value of an element in the histogram is 25^2. Since $2^9 < 25^2 < 2^{10}$, a bit width of 10 is the optimal value for the histogram array elements. If the filter kernel radius is changed, the bit width for the histogram array has to be adapted accordingly.

An exact complexity for the convolution can be given. It uses $3 \cdot 2^d$ multiplications, $2 \cdot 2^d$ additions, 2^d subtractions, 2^d lookups in the Gaussian lookup table, 2^d lookups in the histogram and 1 division. It is obvious that the complexity only depends on the color depth d and grows exponentially with increased color depth. A color depth of $d = 8$, which is used by most non low-cost cameras, needs an histogram array of size $2^8 = 256$. This might already make the implementation as it is used in our system to slow. For a color depth of $d = 16$, as it is used in high-end cameras, the histogram approach to optimize the Bilateral Filter is not practical on an FPGA anymore.

The convolution part of the Bilateral Filter has to be applied on every pixel. In the following the processing sequence on the input image that is needed to update the histogram accordingly is described.

8.2.2 Processing Sequence

For the Bilateral Filter with the histogram optimization it is not possible to do the processing in simply line by line. After every convolution the histogram has to be updated according to the next center pixel that is worked upon. This is only possible by sliding the filter window by a maximum of one x or y value.

Therefore, the filter kernel window has to slide in a continuous manner over the image without visiting a center pixel twice. This is possible by filtering the pixels line by line and changing the direction of the sliding in every line. The filter window sliding sequence is illustrated in figure 8.4.

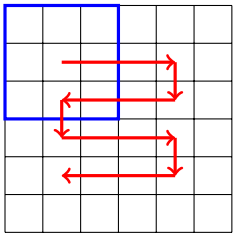


Figure 8.4: Bilateral filter window sliding sequence

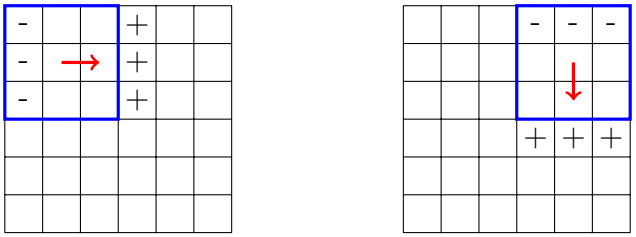
Updating the histogram after each sliding step in a fast manner is possible because of the distributivity property of histograms. When the filter kernel window is moved, a histogram for the removed pixels H_{removed} and one for the new pixels H_{new} can be constructed. The updated histogram H' can then be calculated by simply subtracting the former and adding the latter histogram

$$H' = H - H_{\text{removed}} + H_{\text{new}}. \tag{8.5}$$

An illustration of the updating procedure for vertical and horizontal updates can be seen in figure 8.5.

The update process can be done directly by adding new and subtracting old pixel intensities to the histogram h. Let $i_0, i_1, \ldots, i_{r-1}$ be the pixels that are new in the filter window with kernel radius r and $j_0, j_1, \ldots, j_{r-1}$ be the removed pixels. The histogram can be updated as follows:

$$\begin{array}{llll} H(j_0)--; & H(j_1)--; & \ldots & H(j_{r-1})--; \\ H(i_0)++; & H(i_1)++; & \ldots & H(i_{r-1})++; \end{array}$$



(a) Vertical (b) Horizontal

Figure 8.5: Vertical and horizontal histogram update procedure

It is evident that the updating procedure unlike the convolution is dependent upon the filter kernel radius r. Therefore, the overall complexity of the Bilateral Filter algorithm is $\mathcal{O}(r)$.

The processing sequence as described above has to be applied on every color channel of the two RGB input images of the Stereo Vision System. How the partitioning of these color channels works is described below.

8.2.3 Partitioning

Since the Bilateral Filter operates on only one color channel, it is necessary to split the input images. The obvious approach for this is to split the image in its red, blue and green color channels and execute the Bilateral Filter algorithm on each of the color channels in parallel. After this, the color channels have to be merged again. Figure 8.6 illustrates this procedure.

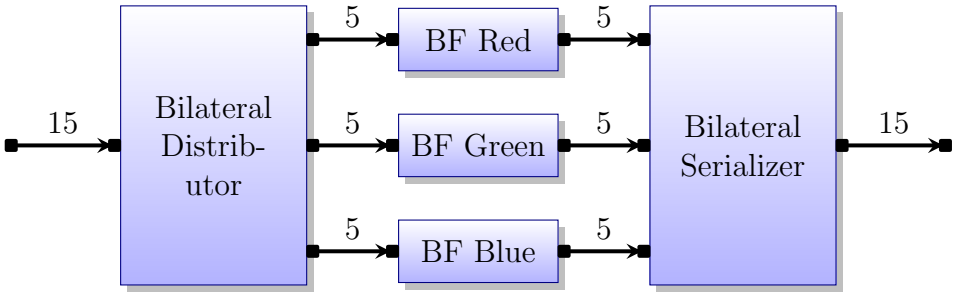


Figure 8.6: Partitioning in red, green and blue color channels, followed by parallel Bilateral Filtering, with subsequent serialization.

The generic concept of the preprocessing phase is already discussed in section 8.1.2 and the operation of the index generator is described in detail in section 8.2.4, therefore only the workings of the Serializer Process and Distributor Process will be depicted here.

Distribution of the input image is implemented in a straight forward manner. The input image is read from an Impulse C stream object with a bit width of 15, containing one pixel of the image at a time. From this pixel the red, green and blue color channels are successively extracted and send via three Impulse C stream to the corresponding Bilateral Filter processes.

After the Bilateral Filtering however, the Impulse C streams of bit width 5 that are transmitting the filtered pixels to the Bilateral Serializer are not in the right order to be processed in the Scan-Line based Segmentation.

The reason behind this is the filter window sliding sequence of the Bilateral Filter as illustrated in figure 8.4 in section 8.2.2. It can be seen, that the alignment of the filtered image as computed in the Bilateral Filter changes in every line. This means that every second line is aligned from right to left, whereas the other lines are aligned from left to right (as in the input images).

Therefore, the Bilateral Serializer process not only merges the 5 bit color channels to full 15 bit pixels, it also buffers a whole line of the filtered image in block RAM. Every second line of the buffered pixels is then send from right to left to the Scan-Line based Segmentation where it can be utilized without further preprocessing.

Although the buffer in the serializer is used out of necessity, it has the unforeseen property to speed up the preprocessing subsystem. Since the Scan-Line based Segmentation only reads a new line of pixels when one complete line is processed, the Bilateral Filter would have waiting times whenever the Scan-Line based Segmentation is operating. The buffer in the serializer process essentially acts like a double buffer. Whenever the Scan-Line based Segmentation is running, the Bilateral Filter processes can fill the buffer in the serializer instead of having to wait.

In the following the Index Generator is described. It has not to be partitioned as the Bilateral Filter, since the indices that are needed to update the histogram are valid for all input images and color channels at the same time.

8.2.4 Index Generator

The first approach to implement the Bilateral Filter with histogram optimization that comes to mind is to use a buffer that can store kernel radius r many lines. This buffer could be updated after every histogram update and the processing sequence as described in section 8.2.2 could be used. The FPGA on the SP601 does not have enough block RAM to buffer the aspired kernel radius of $r = 25$.

As a workaround for this problem an Index Generator was implemented. The Index Generator has a defined sequence of indices that is the access sequence of the Bilateral Filter on the histogram. In figure 8.7 this access sequence is illustrated.

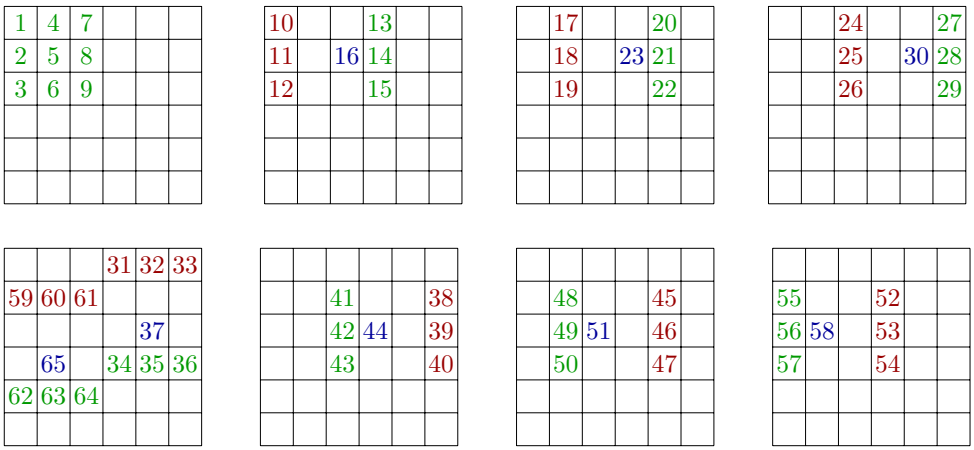


Figure 8.7: Index generation sequence: 3×3 filter kernel; Numbers are sequence of index generation; Green, red and blue for new, removed and center pixel; Windows on left side are initialization and vertical updates, other windows are horizontal updates

At first the Index Generator has to produce the initialization indices. These consist of the complete first filter window, since the histogram has to be filled one time before the distributivity can be used.

The next steps can be divided in horizontal and vertical updates. Vertical updates are used for left and right movement, whereas horizontal updates are used for up and down movements. Such an update consists of indices of

pixels that are removed when the filter window is moved, indices of pixels that are new, as well as the index of the new center pixel.

These indices are sent over an Impulse C stream with bit width 19 to the VHDL DDR Interface, that reads the corresponding pixel of the image from the DDR memory (see figure 8.1 or figure 8.2). A bit width of 19 for the streaming interface comes about because a maximum image resolution of 640×480 was specified and indices up to $640 \cdot 480 = 307200$ fit optimal in a variable of bit width 19.

Thus, the only storage capacity needed in the Bilateral Filter is for the histogram array itself, since the transmitted intensities can directly be applied to the histogram. The histogram array has a length of $32 = 2^d$ for the color depth $d = 5$ that is used in our system. This relatively small array of 32 elements (compared to the 25^2 values that are needed to store one filter kernel window) can be stored in the block RAM.

Besides the Index Generator to cope with the memory consumotion, the Bilateral Filter needs a Gaussian lookup table to cope with the floating point multiplications in the convolution part of the filter.

8.2.5 Gaussian Lookup Table

The Gaussian function

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}, \tag{8.6}$$

in the Bilateral Filter convolution (8.4) has two problems for the implementation on an FPGA. Firstly, it has several multiplications and one square root and is therefore costly to calculate. The second problem is the usage of floating point numbers, since it is not possible to use floating point numbers in Impulse C.

A lookup table can address the first problem. To avoid floating point arithmetic the values in the lookup table can be multiplied by a large fixed constant and rounded to be usable as normal integer variables.

This means, that in a first step for every possible intensity i the Gaussian value $G_\sigma(i)$ is determined. With these values the Gaussian lookup table G_σ^L can be constructed:

$$G_\sigma^L(i) = \left\lfloor G_\sigma(i) \cdot 2^{16} \right\rfloor \qquad \forall\, 0 \leq i < 2^d. \tag{8.7}$$

With this one can update formula (8.4) to incorporate the Gaussian lookup table

$$I'(x,y) = \frac{\sum_{i=0}^{2^d-1} i \cdot G_\sigma^L(i - I(x,y)) \cdot H(i)}{\sum_{i=0}^{2^d-1} G_\sigma^L(i - I(x,y)) \cdot H(i)}. \tag{8.8}$$

It can easily be seen, that the constant of 2^{16} that is multiplied to the Gaussian function does not change the result of the convolution since the factor is present in both, the numerator and the denominator. 2^{16} was chosen, because a resolution of 16 bit for the Gaussian function was found to be enough for our system.

Since the division is a very costly function if done in Impulse C (see section 3.2.3), numerator and denominator should not be huge numbers. The result of the sum in the numerator has a maximum bit width of $5+16+10+5 = 36$ for i, the Gaussian lookup, $H(i)$ and the sum. A division of a variable with a bit width of 36 can use up to 36 cycles when implemented in Impulse C. This is about as much time as is needed for the combined rest of the Bilateral Filter convolution. Fortunately it was found that with real world data a bit width of more than 12 to 16 was rarely occupied.

The Scan-Line based Segmentation, which is described below, does not require any floating point arithmetic or integer division. Thus the Gaussian lookup table is the only lookup table utilized in the Stereo Vision System.

8.3 Scan-Line Based Segmentation

The theory behind and the general idea of the Scan-Line based Segmentation is already described in detail in section 3.4. Therefore, only the implementation specific details will be discussed here.

Since the Scan-Line based Segmentation works on a line by line basis and in the matching phase one has to read a non-predefined number of pixels of the line, it is mandatory to buffer a minimum of one entire line of pixels from the left and for the right image. This lines are saved in three arrays for the red, green and blue color channels with a bit width of 5 bit for each array.

In C it would be a good approach to use an array of a struct such as

```
typedef struct {
    co_uint5 red;
    co_uint5 green;
    co_uint5 blue;
} RGB;
```

to buffer the image data. This has the advantage, that an element of the array can be accessed as a whole pixel (with bit width 15), as well as a particular color channel of a pixel of the array (with bit width 5) without need of any calculations to split the color channels. In Impulse C it is not possible to directly write in a struct from an Impulse C stream object. Therefore, the pixels would have to be split before they could be inserted into a struct. Another problem, which renders structs in Impulse C pretty much useless, is the VHDL code that is generated for structs. For every struct a memory interface is generated over which an access is executed (see section 3.2.3 for details), hence Impulse C processes that use structs are slowed down significantly.

After one complete line of both images is read into the three color channel arrays, the size s_i of a segment with start index i of the right image has to be identified. To do this, the color distance between pixel i and the following pixels is calculated until it exceeds the color tolerance τ. The distance between this pixel and the first pixel that exceeds the color tolerance is size s_i of the segment.

In the next phase, the segment size is given and the best match for a segment of this size between both image lines has to be calculated. For that, the minimum accumulated color difference over the segment size s_i, starting with index i in a predefined maximum matching search distance d is determined. This is done by comparing all d accumulated color differences and taking the index i' of the matched segment with the smallest color difference.

Subsequently, the disparity for a total of s_i pixels can be deduced. Since a segment of the right image was matched to a segment of the same size in the left image, the disparity between these segments is $i' - i$ and the disparity for s_i many pixels can be calculated by scaling the disparity according to the maximum matching search distance.

8.3.1 Parameter Tuning

It is evident from the section above that the Scan-Line based Segmentation has two parameters which can be fine-tuned according to the environment: The maximum matching search distance d and the color tolerance τ.

Maximum Matching Search Distance

The maximum matching search distance d is the maximum distance that is used to search for the best matching candidate of a segment from the right image in the left image.

In theory this value should be set to the maximum distance between identical objects d_{opt} in the left and right image, as illustrated in figure 8.8.

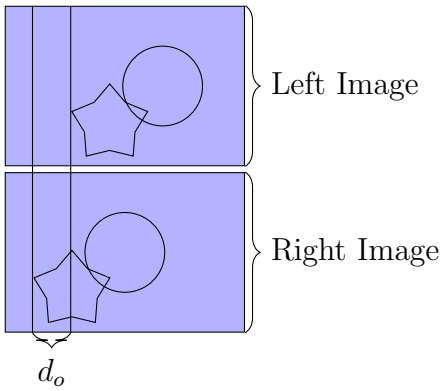


Figure 8.8: Maximum distance d_{opt} between identical objects in left and right image is best possible value for maximum matching search distance d

This optimal value for d can however obviously not be calculated a priori for arbitrary environments, since d_{opt} is dependent on several properties of the used camera system and environment.

An object that is arbitrarily close to the camera will also have an arbitrary large d_{opt}, independent of the used camera alignment. Thus, the expected average distance from camera to object of interest has to be considered. This average distance can be used to deduce a good estimation for d_{opt} and therefore for d.

To make this estimation, the distance between the cameras has to be known. If the two cameras are moved apart from each other, the distance between

identical objects will grow. Likewise the distance between identical objects will shrink, if the cameras are moved nearer to each other.

Therefore, it is possible to get good results in the matching phase by either moving the camera system in a good position, aligning the cameras suitably or adapting the maximum matching search distance according to environment and camera alignment.

In figure 8.9 and figure 8.10 two different example stereo images with two different optimal maximum matching search distances are shown.



(a) Left image (b) Right image

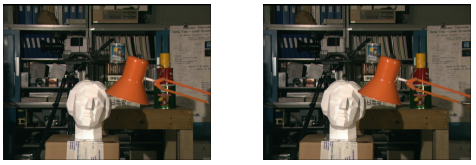


(c) $d = 35$ (d) $d = 50$ (e) $d = 65$

Figure 8.9: Teddy stereo image set from [SS03] (top row) and corresponding disparity maps with different maximum matching search distance (bottom row).

The teddy stereo image set has a relatively large optimal maximum matching search distance of about $d = 50$. The illustration also shows disparity maps for $d = 35$ and $d = 65$. It can be seen, that in the former case some of the leaves in the bottom right of the images could not be matched, since the pixels are further than 35 pixels apart. With $d = 65$ the conversion between disparity map and greyscale image produces a very dark image, since the largest disparities are never reached. This might be fixed by adjusting the image intensity afterwards, but there also is an increased border where matching is not possible.

Figure 8.10 shows disparity maps for three different maximum matching search distances for the Tsukuba stereo image set.



(a) Left image (b) Right image



(c) $d = 10$ (d) $d = 15$ (e) $d = 20$

Figure 8.10: Tsukuba stereo image set from [SS01b] (top row) and corresponding disparity maps with different maximum matching search distance (bottom row).

The optimal distance in this image set is reached at about $d = 15$. With a smaller distance problems at the bottom of the sculpture and at the top of the lamp occur. With increased d the disparity of the camera in the background of the images begins to blur with the background.

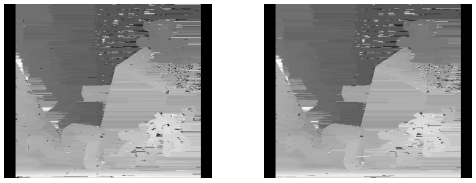
Color Tolerance

The second parameter that has to be considered in the Scan-Line based Segmentation is the color tolerance τ. It is used in the segmentation phase of the algorithm and indicates the maximal possible color distance between the first and the rest of the pixels in each segment.

Obviously the biggest factor that affects the optimal value of the color tolerance is the bit depth of the pixel. For the same color in a 5 bit pixel c_5 and an 8 bit pixel c_8 the relation is approximately $c_8 \approx 2^3 \cdot c_5$.

Therefore, a color tolerance that was identified to be good in an particular environment for an 8 bit camera should be usable with a 5 bit camera by

dividing with 2^3 without huge differences in quality. We found this to be true in all of the test stereo image sets we tried with 5 bit and 8 bit pixels. Disparity maps of the teddy stereo image set with 5 bit input data and a color tolerance of $\tau = 6$ as well as 8 bit input data and a color tolerance of $\tau = 6 \cdot 2^3 = 48$ are illustrated in figure 8.11.



(a) 5 bit, $\tau = 6$ (b) 8 bit, $\tau = 48$

Figure 8.11: Disparity maps of teddy stereo image set (see figure 8.9) with different input image bit depth and color tolerance τ.

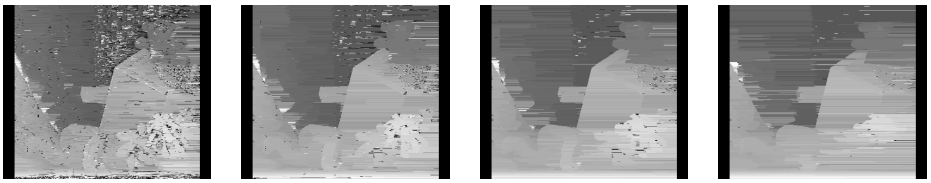
There are lots of small changes between the disparity maps of the 5 bit and 8 bit input images that occur because of a slightly different segmentation. These changes do not degrade the overall quality of the result.

Additionally there are however more errors in the background of the image (where the periodic table is in the input images), as well as at the right side of the house and right of the teddy identifiable. Hence, we can deduce that cameras with an 8 bit depth per pixel would indeed produce slightly better results, without fundamental changes in the used algorithms.

In the following different color tolerances on the teddy stereo image set and the Tsukuba stereo image set are tested. Figure 8.12 shows the result of the Scan-Line based Segmentation on the teddy stereo image set with color tolerances of 3, 6, 9 and 12. The pixel bit depth of the input images is 5, as in our own camera system.

One can see a continuous decrease of noise with an increased color tolerance. The sharpness of the edges however deceases with increasing color tolerance. Therefore, a tradeoff between fringed edges and noise has to be made.

Most of the noise stems from small segments that are produced because of details in the input images. This noise however can be eliminated with good preprocessing, such as the Bilateral Filter as described in section 8.2.



(a) $\tau = 3$ (b) $\tau = 6$ (c) $\tau = 9$ (d) $\tau = 12$

Figure 8.12: Disparity maps of teddy stereo image set (see figure 8.9) with different color tolerances.

Thus, the best color tolerance for the teddy stereo image set is at about $\tau = 6$. With this value, most of the edges are preserved while a bulk of the noise that is present for $\tau = 3$ is absent.

Results for the Tsukuba image set with a bit depth of 5 for the input images and color tolerances of 3, 6, 9 and 12 are shown in figure 8.13.



(a) $\tau = 3$ (b) $\tau = 6$ (c) $\tau = 9$ (d) $\tau = 12$

Figure 8.13: Disparity maps of Tsukuba stereo image set (see figure 8.10) with different color tolerances.

Compared to the teddy stereo image set it shows blurred edges already with a color tolerance of $\tau = 6$, whereas the noise decreases with increased color tolerance as expected. With $\tau = 6$ there are failures left from the sculpture and at the lamp holder that are not present with a color tolerance of $\tau = 3$.

The results for color tolerances of $\tau = 9$ and $\tau = 12$ do have very reduced noise. They are however of a poor quality, since the whole space below the table is presented with roughly the same disparity as the table. This error might be to big in many applications of the Scan-Line based Segmentation.

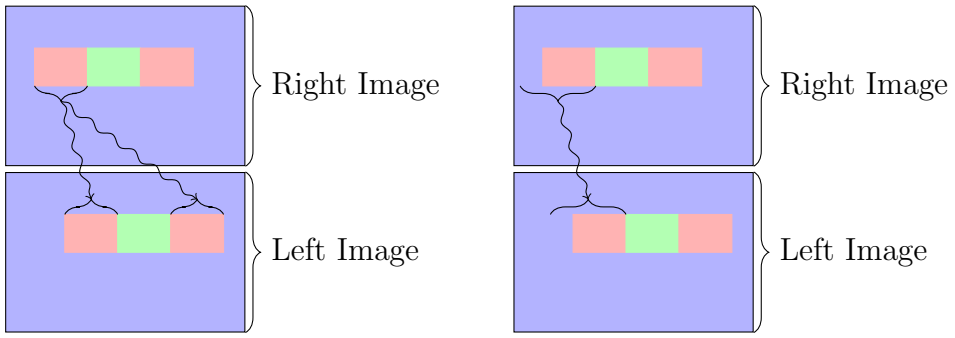
Therefore, a color tolerance of $\tau = 3$ is preferable for the Tsukuba stereo image set, if a noise reducing preprocessing step is taken before the Scan-Line based Segmentation is executed.

This result compared to the result for the teddy stereo image set shows that an different environment can have a different optimal color tolerance. An environment with high color differences between objects of different depth delivers a better result for high color tolerances, while an environment with a low color difference between objects will yield better results if a lower color tolerance is used.

Instead of the usage of high color tolerance to reduce noise in the result of the Scan-Line based Segmentation, it is also possible to extend the segments after the segmentation phase. This approach is described in the following.

8.3.2 Segment Extension

One problem with the Scan-Line based Segmentation occurs when a segment from the right image is matched in an area of the left image that has the same color but is not the corresponding area. This can happen in any scene that has a checkerboard pattern. This problem is illustrated in figure 8.14.



(a) Without segment extension (b) With segment extension

Figure 8.14: Segment matching ambiguous without segment extension (left) and unambiguous with segment extension (right).

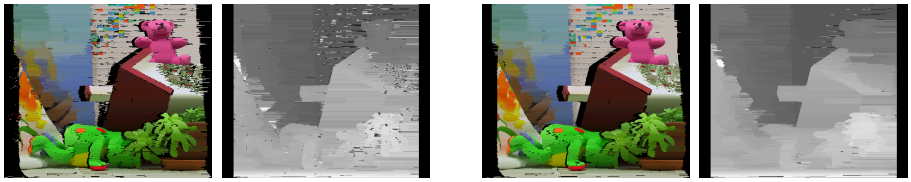
The algorithm as described in [SM09] is flawed in these checkerboard pattern situations. A possible improvement on this was found by extending the segments in the matching phase of the algorithm. The calculation of the color

distance is started some pixels before the beginning of the original segment. Thereby some color values from the former segment are also evaluated.

In the example in figure 8.14 the input images have consecutive rectangles with the colors blue, red, green, red and blue. In the segmentation phase, the Scan-Line based Segmentation will generate five segments, one for each color.

When the matching phase is started for the red segment, the matching algorithm has two possible matching position in the left image. The Scan-Line based Segmentation by chance might match the left red rectangle of the left image to the right red rectangle of the right image because of different noise in the left and the right camera. If the segment is extended to the left, some of the blue color values will be included in the accumulated color distance and the matching becomes unambiguous.

Figure 8.15 illustrates the effect of this improvement on the teddy stereo image set.



(a) Without segment extension (b) With segment extension

Figure 8.15: Results with and without segment extension, shown are segments of left input image matched onto right input image (left) and resulting disparity maps (right).

In the first pair of images it can be seen, that the algorithm has problems matching the white spaces in the periodic table to the correct segments in the other image. This problem does not occur in the second pair of images, which is produced with the segment extension improvement. Also problematic is the very detailed front of the house. This however is also fixable with a proper preprocessing filter, that blurs the details.

The proposed improvement is unfortunately not without disadvantages. In the special case, when left of an object in the left image is the background of the scene and left of the same object in the right image is another object or another part of the background with a different color, because of the different

perspective, the segment extension might yield in an incorrect matching. This constellation was found to rarely happen in real world examples.

One side effect of the segment extension is an increase of the run-time in the matching phase of the Scan-Line based Segmentation. A possible way to increase the run-time lays in parallelization.

8.3.3 Parallelization

As presented so far, the Scan-Line based Segmentation only consists of one process and is not parallelized. It is not possible to execute the Scan-Line based Segmentation in concurrent manner as it is done in the Bilateral Filter, i. e. one process per color channel. The Scan-Line based Segmentation requires all three color channels of one pixel at once in two different locations of the algorithm.

Firstly, in the segmentation phase the color difference between the first pixel of the segment and the currently considered pixel is determined. Only if for each color channel the color tolerance is not exceeded, the pixel is added to the segment. Therefore, if the color channels were separated into several processes, at this stage a merging process that can access information from all three color channel processes would be required.

Secondly in the matching phase an accumulated color distance between a segment and a possible matching position is calculated. This procedure needs access to all color channels of all pixels of the considered segments in the left and right image. Again a merging process would be needed to execute the color distance calculation.

Since the color difference and color distance calculations are the most time-consuming parts of the Scan-Line based Segmentation, a concurrent execution based on the color channels does not come into further considerations.

For the Scan-Line based Segmentation it is however the case that each line of the input images is processed independently from each other. Hence, it is possible to distribute different lines of the stereo images into several parallel running copies of the complete Scan-Line based Segmentation process.

Figure 8.16 illustrates this approach in an Impulse C process flow diagram.

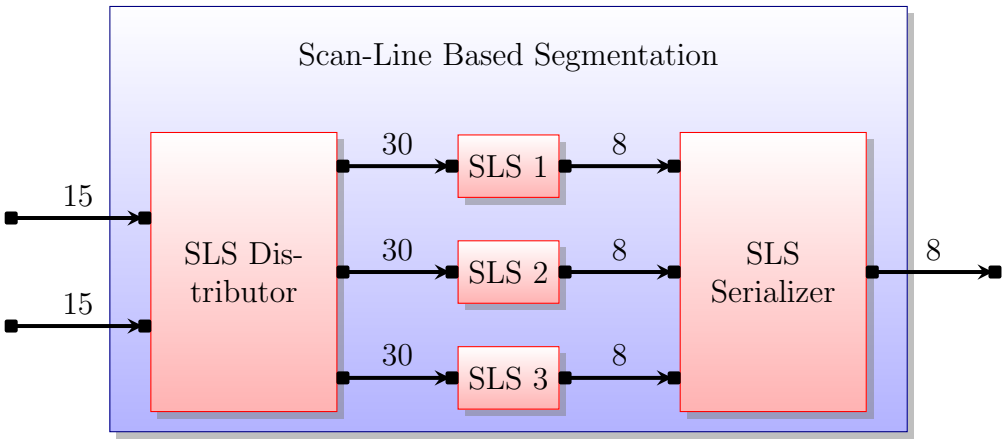


Figure 8.16: Parallelization of the Scan-Line based Segmentation, processes shown as red boxes, streams as arrows with bit depth, content of blue box can replace SLS process in figure 8.1.

Although the illustration looks like the distribution, processing and serialization scheme used in the preprocessing subsystem, distribution and serialization do work differently here.

The two 15 bit input streams are the outgoing streams of Bilateral Serializer left and right (figure 8.1). The *SLS distributor* splits the lines of the input images into n different images composed of every n-th line. Let l_i^r and l_i^l be line i of the right and left image, m be the number of lines per image and n be the number of independent Scan-Line based Segmentation processes s_j. Then the input images are segmented as follows

$$\forall 0 \leq j < m: \qquad l_i^r \triangleright s_j \text{ and } l_i^l \triangleright s_j \quad \Leftrightarrow \quad i \equiv j \mod n, \tag{8.9}$$

where $l \triangleright s$ means that line l is processed in Scan-Line based Segmentation process s.

This procedure is illustrated in figure 8.17.

Each of the Scan-Line based Segmentation streams produces an 8 bit output stream, consisting of disparity values. These are serialized in the *SLS Serializer*, where they are assembled in the correct order and send to a Packet Producer in 8 bit parts.

This approach accomplishes a speedup of n times, if n Scan-Line based Segmentation processes are used (see section 9.2.2 for details). Therefore, the run-time of an input stereo image with a resolution of 320×240 could in theory be reduced to a fraction of the run-time of the non-parallel approach.

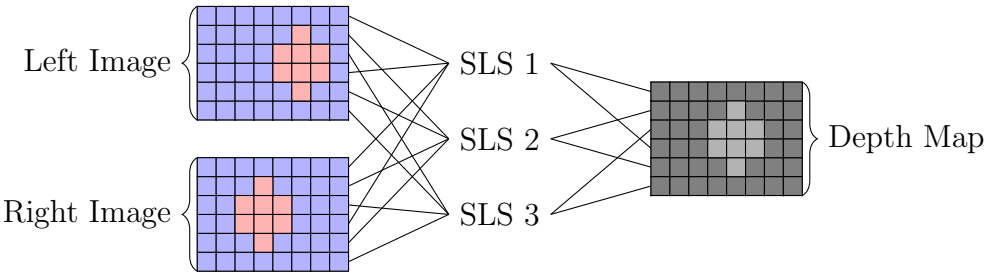


Figure 8.17: Parallelization example with two input images consisting of six lines each and three Scan-Line based Segmentation processes.

Each of the parallel processes needs as much slices as a complete Scan-Line based Segmentation process. Thus, the parallelization can only be utilized on the SP601 if no preprocessing is used. The maximum amount of parallel SLS processes that fit on the SP601 is three (see section 9.2.2).

Verification of the functionality of the parallelized Scan-Line based Segmentation can be done with the debugging possibilities described in the following.

8.3.4 Debugging

As can be seen in figure 8.1, the Scan-Line based Segmentation process has two optional debugging output streams connected to two packet producers.

These streams can be activated if required, to get information about the segmentation and the matching phase of the algorithm. The debug output is helpful in finding good values for the parameterization of the algorithm itself and the Hardware System.

Both streams produce RGB images. The segmentation debug image consists of the segmented right input image, where each segment originated in the segmentation phase of the Scan-Line based Segmentation. Each segment has the color of the first pixel in the segment. The average color of each segment is not calculated. It would consume needless run-time and slices, since only the position and length of the segments are relevant in the algorithm.

The matching debug image consists of the same segments as the segmentation debug image, but this time they are placed at the corresponding matching position, as determined by the matching phase of the Scan-Line based Segmentation. Therefore, if two neighboring segments possess a disparity of n, the matching debug image will show n black pixels between the two segments.

Segmentation and matching debug images with the corresponding disparity map for the teddy stereo image set can be seen in figure 8.18.



(a) Segmentation (b) Matching (c) Disparity map

Figure 8.18: Segmentation and matching debug output with corresponding disparity map

Debug output from the Scan-Line based Segmentation is packaged in the postprocessing subsystem consisting of packet producer and packet serializer as every other output of the Stereo Vision System. In the following the Packet Producer and Packet Serializer processes are described.

8.4 Packet Producer and Packet Serializer

The Vision Core has several Impulse C streams that need to be transfered to the host PC. These steams are the plain left and right image, the disparity map, the Bilateral Filter left and right debug images and the segmentation and matching debug images from the Scan-Line based Segmentation.

Since the Ethernet Interface that is provided by the Vision Framework is a component that takes a 32 bit input value, double buffers it and sends it to the Ethernet PHY of the SP601, the output data has to be serialized and packaged in a way that allows the host PC to distinguish between different images as well as starting and end positions of images.

8.4.1 Operation

The Ethernet frames (as described in section 8.4.2) are assembled in the Packet Producer. Each output stream is connected to an associated Packet Producer that knows properties such as width, height or bit depth per pixel

for the image that is send over the corresponding output stream. Hence, there exist one Packet Producer for each output stream, all working in parallel.

Each Packet Producer fragments the data into packets and attaches the corresponding header. This is done by building small blocks with a size of 32 bit each. These are send one after another to the Packet Serializer over the connected Impulse C stream with a corresponding bit depth of 32. Thus, the Packet Producers operate by only working on small amounts of data at the same time and do not need any block RAM, distributed RAM or DDR RAM besides the utilized 32 bit variable.

Data from all Packet Producers is send to one Packet Serializer process. Since the Ethernet PHY expects Ethernet packet in one piece, the Packet Serializer has to buffer one whole 1018 byte Ethernet packet per Packet Producer in block RAM. Otherwise the Packet Serializer would have to rotate between the Packet Producers and always read out one Ethernet Packet at a time. This would reduce the performance severely.

Instead only one 32 bit block at a time is read from each Packet Producer with a non-blocking read function provided by Impulse C. This non-blocking function does not wait for data if the buffer of the accessed stream is empty, instead the next stream is queried.

Sequential polling of all Packet Producer streams is done until one of the Ethernet packet buffers is filled completely. Then this buffer contains an IEEE 802.3 Ethernet frame without the respective CRC checksum. The Ethernet frame is send to the Ethernet Interface in the Vision Framework, which sends it to the utilized Ethernet IP Core where the CRC checksum is added and from where it is transmitted to the host PC via the Ethernet PHY on the SP601.

This approach allows the serialization of an arbitrary amount of Impulse C streams and is independent from the rest of the Vision Core or the used development board. The generic idea of the postprocessing subsystem is described in detail in section 8.1.2.

Although the Impulse C implementation of the Packet Serializer processes functions well, it demonstrates a weakness of Impulse C. Since global variables can only be accessed by a maximum of two processes, the Packet Serializer cannot be parallelized into multiple processes. Therefore, the filling of the Ethernet frame buffers has to be done sequentially. If many different payload types are used, this can slow down the whole system. In VHDL the different buffers could be filled with multiple parallel processes without any problems.

What the actual Ethernet packages look like and how they are produced is described in the following.

8.4.2 Packaging

An optimal solution for the packaging would be to use TCP/IP or UDP/IP as a means of data transfer. These protocols are supported by any modern operating system and it would be possible to plug our system into a switch to stream the output data over a local area network or even the internet. An implementation of a TCP/IP or UDP/IP stack on the FPGA is far beyond the scope of this thesis. It is also questionable if the FPGA on the SP601 has enough slices to store one of the stacks.

Thus, we decided to use raw IEEE 802.3 Ethernet frames. Such a frame is on the link layer in the ISO/OSI reference layer model [Zim80] and thus the lowest possible protocol level that can be used with the Ethernet PHY on the SP601.

An IEEE 802.3 Ethernet frame consists of destination MAC address, source MAC address, Ethernet type and CRC checksum [Eth]. The Ethernet type describes the format of the payload (e. g. UDP or TCP). A frame, as it is generated in the Packet Producer, is illustrated in figure 8.19.

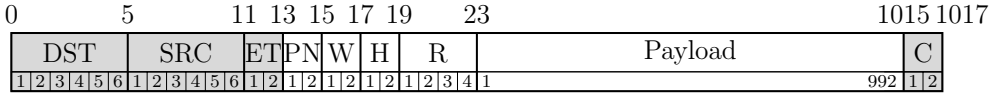


Figure 8.19: Ethernet frame consisting of destination (DST), source (SRC), Ethernet type (ET), packet number (PN), width (W), height (H), reserved part (R), payload and checksum (C). The gray parts are defined by the IEEE 802.3 standard.

In the following the usage for each of the fields in the Ethernet frame in our system is described.

Source and Destination MAC Address The source MAC address is fixed, it is assigned to the Ethernet PHY on the SP601. Whereas the destination MAC address has to be adjusted for the corresponding host PC. Hence, our system is not flexible in this regard. If the host PC is

changed, the destination MAC address in the Impulse C code has to be reassigned and the generated VHDL code resynthesized.

Ethernet Type It is possible to use the Ethernet type field to specify the size of the payload. If the field is set to 1500 or lower it is interpreted as payload size, where it is interpreted as the type of data in the payload for values over 1535. In our system this plays a vital role. The Ethernet type is used to identify the different output streams (see table 8.1).

	Ethernet type	Output stream
mand.	0x55A0	Left plain image
	0x55A1	Right plain image
	0x55A2	Disparity map
optional	0x55A3	Segmentation debug image
	0x55A4	Matching debug image
	0x55A5	Left Bilateral Filter debug image
	0x55A6	Right Bilateral Filter debug image

Table 8.1: Ethernet types as used in the Vision Core

This approach allows a readout of different streams by different independent programs on the host PC. Why this is important and how this works is described in section 8.6 in detail.

Packet Number Since it is not guaranteed that each packet that is send by the Ethernet PHY will arrive at the host PC when raw Ethernet frames are used, a packet number has to be attached to each packet. The packet number is a consecutive number, that is increased in each packet per image type until the image is completely send. It declares where the data in the payload belongs in the image. A packet number of n means, that this packet is the n-th packet for one particular image. Therefore, the position of the payload of this packet in the image can simply be calculated by multiplying n with the payload size 992. This means that one lost packet will not imply the loss of an whole image but only of a small part in the image.

Width and Height The W and H fields in the Ethernet frame specify the width and height of the image. These fields are needed by the host PC to display the image in the right format but also to deduce when the last packet of an image is send. This can be done since it is possible

to calculate the number of packets per image with knowledge of width and height of the image, therefore the last packet number is known and the last packet of an image can be identified.

Reserved To allow future improvements a four byte long field is reserved for future use. It is also possible to use this field for debugging output per packet. This is very helpful whenever timing problems occur, since additional image sequence information can be transmitted.

Payload Storage of the actual image data is done in the payload field. The size of 992 is specified due to technical details in the VHDL implementation of the double buffer in the Ethernet Interface.

CRC Checksum Data corruption is avoided with the use of a CRC checksum at the end of each raw Ethernet packet. This checksum is part of the IEEE 802.3 Ethernet standard and it is calculated in the Ethernet IP Core that is utilized in the Vision Framework.

When the Stereo Vision System is simulated with the simulation described in the next section, the CRC checksum has to be attached by other means.

8.5 Simulation

One important part of the proposed Vision Core is the possibility to simulate parts of it or the whole core. Since the Vision Core is relatively complex, the generated VHDL code is large and complex too. Thus a solution to simulate the Impulse C code on the host PC without the need for synthesis was developed.

Simulation of the host PC only allows to debug the functionality of the C code, it does not ensure that the generated VHDL code functions. Hence, it must also be possible to simulate the generated VHDL code.

8.5.1 Simulation of C Code on Host PC

Since Impulse C is a subset of the C programming language, it is possible to compile the Impulse C processes and streams with a standard C compiler. Thereby the basic functionality of the implemented algorithms can be tested (see section 3.2 for details).

Parts of the Stereo Vision System are written in VHDL, these parts have to be replaced by C code. The software simulation code can however use all of the ANSI C language features (such as reading from a file and writing to a file), since it is only for simulation on the host PC.

C simulation software has to be written for the actual cameras, the DDR Interface, the Ethernet Interface and the Ethernet IP Core. Figure 8.20 illustrates the host PC simulation system.

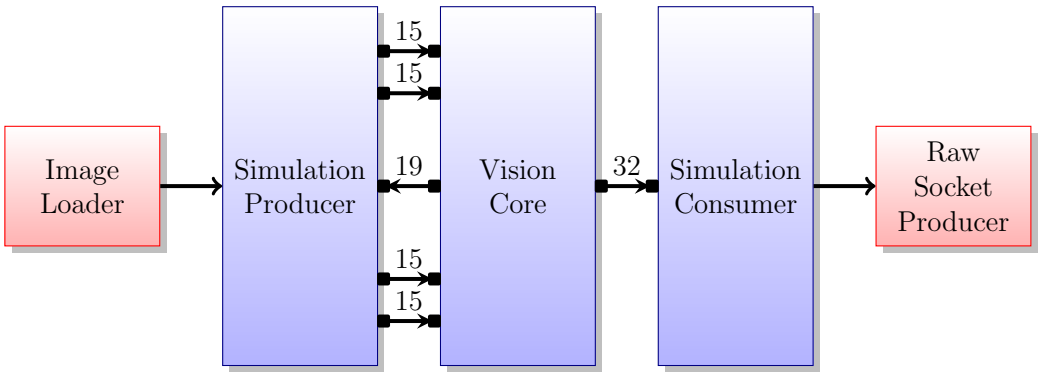


Figure 8.20: Host PC simulation: With Impulse C processes (blue) and ANSI C simulation code (red).

Firstly, the cameras are replaced by an *Image Loader*, which can load bitmap images from the host PC that can be used as input images for the Vision Core. The loaded images are transmitted to a *Simulation Producer* Impulse C process, which converts the image data into a fitting format.

That is, it scales the pixel bit depth to 5 bit per color channel, fills the Impulse C streams for the plain input images and transmits data to the bilateral filter corresponding to the input from the Index Generator.

For the simulation of the output part of the Vision Framework, a *Simulation Consumer* Impulse C process is written. This process receives the Ethernet packets that are produced by the Packet Producer and Packet Serializer processes. It buffers them and sends them as byte buffers consisting of complete Ethernet packets to the *Raw Socket Producer*.

The Raw Socket Producer simulates the Ethernet IP Core. It utilizes the Ethernet stack of the Linux kernel. A raw IEEE 802.3 Ethernet socket is opened with the same configurations that are used by the Ethernet IP Core. Ethernet packets from the Simulation Consumer are send over this socket without

further modifications. The CRC checksum, which is normally attached by the Ethernet IP Core, is now calculated and attached by the Linux kernel.

Therefore, the packets are received by the host PC in the same manner as they are when the actual Stereo Vision System is used. Thus, the software simulation is transparent for the Host PC Viewer, it can not differentiate between the genuine hardware and the software simulation. This allows authentic debugging when the software simulation is used.

8.5.2 Simulation of Generated VHDL with ModelSim

Simulation via the C code on the host PC, as described in section 8.5.1, is unfortunately not enough. Processes that run fine on the host PC might not run at all after they are translated into VHDL (see section 3.2 for details).

Hence, also a VHDL simulation is required. We used ModelSim to simulate the VHDL code that is generated from the processes and streams by the Impulse C C-to-VHDL compiler.

In principal the illustration in figure 8.20 consisting of Image Loader, Simulation Producer, Vision Core, Simulation Consumer and Raw Socket Producer can also be used to describe the VHDL simulation. However, since all of these components except the Vision Core are written in ANSI C, they need to be modified to be simulateable with a VHDL simulator.

Firstly, the Image Loader is changed into a standalone program that can open a bitmap file and convert the image data in the same format as it will be requested by the Vision Core.

The Simulation Producer is rewritten as a VHDL process, which opens the file that is saved by the Image Loader. It can use the read image data byte by byte and send it to the Vision Core without further calculations, as it is already saved in the right format.

Data from the Packet Serializer is received by the Simulation Consumer, which again is rewritten as a VHDL process. It opens a new file and converts the data as it arrives into this file (see section 6.12.1).

At last the Raw Socket Producer is also modified to run as a standalone program. Functionality to read in the file saved by the Simulation Consumer and to revive the Ethernet packets from the unformatted data is added.

With this approach the system consisting of Simulation Producer, Vision Core and Simulation Consumer is self-contained with image input from a file

and data output to another file. Since Simulation Producer and Simulation Consumer are written in VHDL and the Vision Core is completely written in Impulse C and can therefore be translated to VHDL by the Impulse C C-to-VHDL compiler, the whole system can be simulated with ModelSim.

Furthermore, since Image Loader and Raw Socket Producer are written in C and function as standalone programs, they can use C libraries for the opening of image files and the Linux kernel Ethernet stack as described in the section above. Thus, the same data as with the host PC simulation can be used and the Host PC Viewer can also be utilized to view the results.

8.6 Host PC Viewer

Alongside the Vision Core to execute vision algorithms, the Vision Framework to interface the Vision Core with a host PC and the Hardware System to make the actual images, an Host PC Viewer was developed. The Host PC Viewer is a GUI that enables visualizations of all of the data which is send by the Stereo Vision System to the host PC.

Such a viewer is important for debugging purposes. It allows to tinker with the options of the algorithms while getting a feedback in real-time. It also is important for the setup of the Hardware System. Small changes in the alignment of the cameras can change the epipolar lines significantly and thereby spoil the required rectified epipolar geometry constraint (see section 3.4.1).

8.6.1 Architecture

The GUI is written in Python, it uses the cross platform GUI development framework Qt[1] and utilizes OpenGL to display the debug images and disparity maps. Python and Qt are chosen because they allow a very rapid development of GUIs [Sum07], albeit Python is slow on executing calculations.

Thus, the actual code to calculate the representation of the debug images and the 3D version of the disparity map are done in C. This is necessary since data that is received from the Stereo Vision System over Ethernet is lost if it is not read out immediately. It was found that Python is not fast

[1] http://qt.nokia.com/

enough to reliably read out the data that is transmitted by the Stereo Vision System.

One thread for each of the utilized Ethernet types (see section 8.4.2) is used to read out all of the debug data (left/right image, disparity map, bilateral filtered left/right image, segmentation and matching of Scan-Line based Segmentation) at the same time. Each of these threads has access to a double buffer that can store two images. Every time one of the buffers is filled, they are swapped and another thread is started that calculates an *OpenGL display list* for the newly filled buffer.

The OpenGL display list is read 60 times per seconds by the Python GUI to allow a maximum of 60 fps. This is done independently by one Python thread per Ethernet type to make different frame rates for the features described below possible.

8.6.2 Features

A screenshot of the Host PC Viewer is shown in figure 8.21.

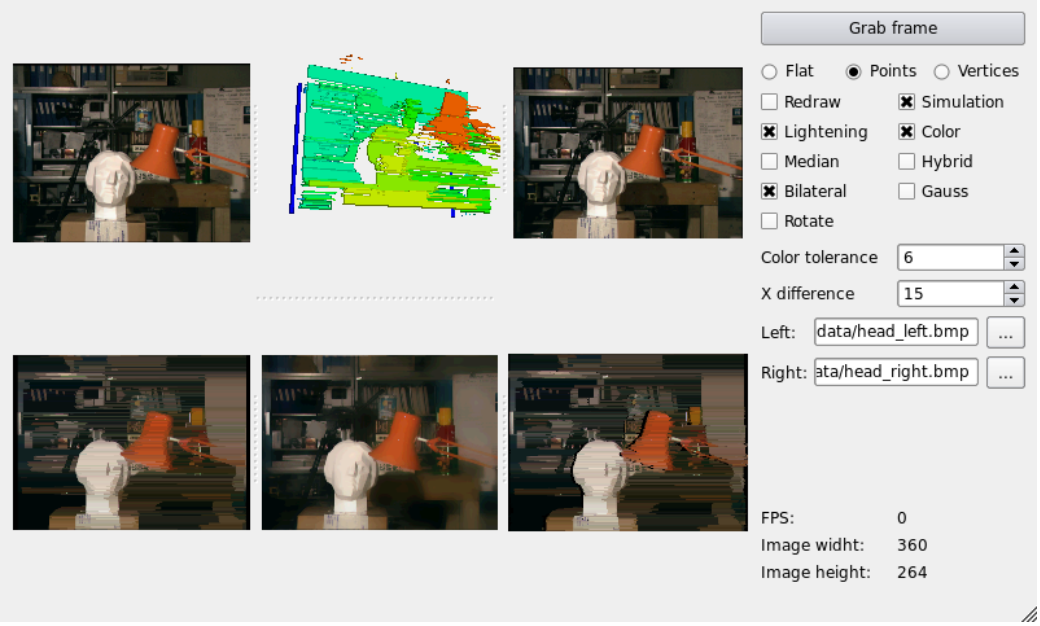


Figure 8.21: Host PC Viewer with left image, 3D view of disparity map, right image in top row and segmentation, bilateral filtered image and matching in bottom row as well as configuration panels on the right

Besides the real-time displaying of the debug images and the disparity map in 2D or 3D, which is described in detail in section 8.6.3, the Host PC Viewer can also be used to set a series of properties if the simulation mode is utilized.

Some adjustments can be made with the configuration panel (depicted on the right hand in figure 8.21), they are described in the following.

The *simulation checkbox* enables the simulation mode of the Host PC Viewer, if checked. All of the configuration possibilities described below are only applicable in simulation mode. After the desired configuration is selected, the simulation can be run by clicking the *grab frame* button. The simulation is then executed on the selected left and right images. If the redraw checkbox is checked, the simulation is run successively until it is unchecked. This allows to test the performance of the simulation and the Host PC Viewer and it would in theory allow the simulation of input videos instead of images, which is however not yet implemented.

Selectable by checkboxes are four *preprocessing filter options*: Gaussian Filter, Median Filter, Median Hybrid Filter and Bilateral Filter. They can be selected individually and in simulation mode it is also possible to use different filters consecutively.

Two *Scan-Line based Segmentation properties* are important for results with good quality: The color tolerance and the maximum matching search distance (see section 8.3.1). Both can be set to individual integer values.

Width and height of the received images, as well as the number of frames currently drawn per second are displayed in the bottom of the configuration panel.

For the *illustration of the disparity maps* it is possible to choose between the illustration types *flat*, *points* and *vertices*. Flat represents the disparity map in 2D, whereas points and vertices are different 3D representations.

It is also possible to choose between colored and grey disparity maps and to enable OpenGL lightning for the 3D representations. The different disparity map representations are described and illustrated below.

8.6.3 Disparity Map Representation

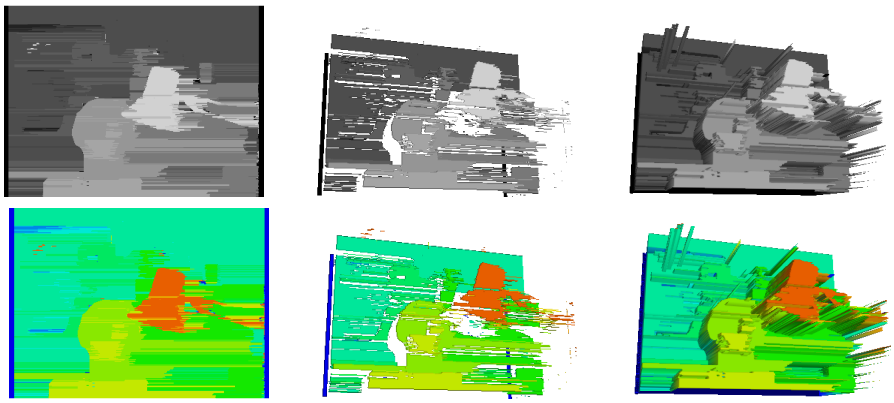
The Host PC Viewer allows different representations of the received disparity map. These representations can be drawn in color or in a gray scale.

A good way to color disparity information is presented in [Loz03]. If the height h of pixel $p = (r, g, b)_p$ is assumed to be in the interval $[0, 4]$, the color of p can be calculated with

$$(r, g, b)_p = \begin{cases} (0, h, 1) & \text{if } 0 \leq h < 1, \\ (0, 1, 2 - h) & \text{if } 1 \leq h < 2, \\ (h - 2, 1, 0) & \text{if } 2 \leq h < 3, \\ (1, 4 - h, 0) & \text{if } 3 \leq h < 4, \end{cases} \tag{8.10}$$

for RGB-values in the interval $[0, 1]$.

Figure 8.22 shows the three different disparity map representations with and without coloration as described above.



(a) Flat (b) Points (c) Vertices

Figure 8.22: Disparity map representations in Host PC Viewer

In the following, the different disparity map representation are described and advantages and disadvantages are given.

Flat The flat representation is a normal 2D representation of the disparity map. Near pixels are white and pixels that are far away are black when gray scale mode is used. In color mode the colors are scaled between blue, green and red from far to near, as described in formula (8.10). Coloration in 2D representations is however not intuitive and should therefore not be used [BI07].

Points In the points representation a point cloud is drawn, where each pixel is arranged in an 3D space according to its disparity. This method

does not require more calculations than the flat representation and can therefore also be drawn in real-time. In this view the grey scale representation also seems to be more intuitive. The 3D representation of the disparity map can be rotated arbitrarily on the x and y axis by the user, which allows an examination of the disparity map from different angles.

Vertices When the vertices representation type is chosen, vertices between each of the pixels and there neighbour pixels are calculated. These allow a solid 3D representation of the disparity map. With a solid 3D representation errors in the disparity map can easily be spotted. Additionally the normal for each vertex is calculated, which allows the use of the OpenGL lightening facilities. Thereby an artificial light source is introduced and the 3D representation features darker spots where the light does not impact. This gives a very good visibility of the 3D structure. Since the disparity is perceived with these means, the coloration adds additional recognizability of small disparity changes and thus small errors.

Calculation of the vertices and normals is complex and can not be done with full 60 fps, even though the calculations are made in C. It is however possible to rotate the 3D disparity map along the x and y axis as with the points representation.

The described representations allow a detailed analysis of the disparity map. Results of this analysis and other aspects of the Stereo Vision System are discussed in the following.

Chapter 9

Results

The Stereo Vision System is a generic development platform for vision algorithms. It is not constructed to achieve a high amount of performance or space-saving.

Nevertheless, information about execution times of the different parts of the Stereo Vision System as well slice usage and reachable frames per second are important to identify the usefulness of the proposed system.

Another important factor is the quality of the images that are captured by our camera system and the quality of the disparity map, which is calculated by the Vision Core.

9.1 Vision Framework

In the following sections an analysis of the Vision Framework is performed. As mentioned in section 6.12.1 a simulation of the whole system was not realizable. Therefore all stated details are measured in the running system.

At first, the performance of the Ethernet Interface and the DDR Interface components are measured regarding different stream clock speeds. Afterwards the resource requirements of each main component in the Vision Framework is analyzed.

Since no simulation of the whole system could be performed a power dissipation analysis was not possible.

9.1.1 Performance

In the following the performance of the Ethernet Interface and the DDR Interface is analyzed. A high throughput of these components is necessary such that the Vision Core can access the memory or transmit results to the host PC without slowing down, i. e. these components should not represent a bottleneck in the system. Other components in the Vision Framework do not have this characteristic, and are therefore not analyzed.

Frequency	Transfer rate
10 MHz	154 Mbit/s
25 MHz	386 Mbit/s
50 MHz	772 Mbit/s
100 MHz	954 Mbit/s
200 MHz	954 Mbit/s

Table 9.1: Ethernet Interface frequency comprehension: Transfer rate in relation to the clocking frequency of the data stream input of the component.

Table 9.1 illustrates the transfer rates of the *Ethernet Interface* to the host PC in relation to the frequency of the clock which drives the incoming data stream.

The illustrated transfer rates are mean values determined by interfacing the Ethernet Interface component with an Ethernet packet generator written in VHDL. This generator produces packages and transmit these clocked by the respective clock to the component. Therefore these results represent the maximum transfer rates which are possible by using the corresponding clock.

As depicted, the maximum frequency of gigabit Ethernet is reached at a frequency of 100 MHz. On lower frequencies the relationship between transfer rate and frequency is nearly linear (interfered by buffer swaps of the internally used Block RAM Double Buffer). We measure an average of 954 Mbit/s for the maximum transfer rate on a PC connected over a 0.5 m CAT 5e cable.

The performance of the *DDR Interface* is measured by interfacing it with external components. A stream reader and writer, each written in VHDL, read/write data over streams from/to the DDR Interface. Both are designed to transmit the maximum amount of data in each clock cycle. In each test case the reader and writer streams are clocked with the denoted clock.

The measuring was performed by using ChipScope, an logic analyzer tool from Xilinx and counting the clock cycles between each data transfer cycle. In each data transfer cycle 32 bit data are transfered, therefore the maximum read and write performance can be calculated.

Table 9.2 shows the resulting values.

Frequency	read clock cycles	read transfer rate	write clock cycles	write transfer rate
10 MHz	4	80 Mbit/s	6	53 Mbit/s
25 MHz	4	200 Mbit/s	6	133 Mbit/s
50 MHz	4	400 Mbit/s	6	267 Mbit/s
100 MHz	7	457 Mbit/s	7	457 Mbit/s
200 MHz	12.5	533 Mbit/s	12.5	533 Mbit/s

Table 9.2: DDR Interface frequency comparison

For low frequencies the limiting factor is the handling of the streams and the filling of the FIFOs, since this is all handled with the low frequencies. If higher stream frequencies are used, the speed of the memory core (running at 300 MHz) is also relevant for the performance since multiple clocks are necessary to handle data in a FIFO.

The proposed performance of 800 Mbit/s in [Xil10c] for a 800 MHz clocked DDR memory cannot be reached, since our DDR memory is clocked with 600 MHz. This leads to the maximum possible performance of 600 Mbit/s[1]. Our maximum achievable transfer rate is 533 Mbit/s when clocking the streams with 200 MHz.

In the next section the resource requirements of the Vision Framework components are discussed.

9.1.2 Requirements

The occupied hardware of a generated system is not constant, because random decision are made during the synthesis. Therefore the hardware requirements cannot be specified in general, minimal variances are possible for repeated synthesises.

[1] A higher clocking of the memory was not possible since the timing constraints for the clock paths could not be meet

During the development code optimizations where applied to reduce the required hardware. Optimization parameters as in Impulse C do not exist in VHDL. The only way to manipulate the resource requirements besides code modifications are the optimization parameters of the synthesis, router and placer tool. Basically an optimization for space (occupied resources) and timing performance is possible. For the stated values a timing performance optimization was applied, but a space optimization leads to similar results.

In table 9.3 the hardware requirements are illustrated in respective to *Slices*, *Slice Register*, *Slice LUTs* and *block RAM blocks* (The hardware requirements of instantiated components are added to the requirements of the main component).

Component	Slices	Slice Register	Slice LUTs	BRAM blocks
Available on SP601	2278	18224	9112	32[2]
Clock Manager	5	25	1	0
Block RAM Double Buffer	16	52	14	1
Stereo Interface (OV7)	382	675	889	2
OV7 Interface	181	295	407	0
Stereo Interface (MT9)	374	675	909	4
MT9 Interface	173	295	417	2
Camera Configurator	120	183	330	0
DDR Interface	336	634	668	0
DDR Port Interface	66	215	105	0
Ethernet Interface	536	1259	1040	4

Table 9.3: Resource requirements of Vision Framework components

The *Clock Manager* does not occupy many slices since it is basically a wrapper for the clock managing hardware (PLLs), which are not depicted.

In case of the *Block RAM Double Buffer* the swap handling procedures are the only relevant elements for the hardware requirements since it is also basically a wrapper for a block RAM block.

The *Stereo Interface* is depicted twice. The reason is that this component can be configured to handle OV7 or MT9 cameras. In case of a OV7 camera a *OV7 Interface* is instantiated, the hardware requirements of the OV7

[2] $RAMB16BWER$ blocks

Interface are added to those of the Stereo Interface. As depicted, the hardware requirements of a OV7 or MT9 configured Stereo Interface does not vary much. The only appreciable difference is the usage of two additional block RAM blocks by the MT9 Interface. Each MT9 Interface needs one additional block RAM block, compared to a OV7 Interface, to store the last image line such that a debayering can be performed (refer to section 6.7.3 for more information).

Both Camera Interfaces instantiate a *Camera Configurator* for camera configuration. The hardware requirements of the internal I^2C IP Core are included in the values of the configurator. Since a huge state-machine is required to handle the data transfer to the IP Core a noticeable amount of slices is required.

The *DDR Interface* uses the memory controller IP core and two *DDR Port Interfaces* which handle the data flow between streams and the ports of the IP Core. Each DDR Port Interface consists of processes which handle the data transfer and are implemented as state-machines. Therefore, a larger quantity of slices is necessary.

The most resources are occupied by the *Ethernet Interface*, whereas the largest amount is used by the Ethernet IP Core. To reduce the slice requirements only the transmit part of this core is operated such that the receive part could be removed. The block RAM amount consists of one block RAM block for the instantiated Block RAM Double Buffer and three blocks occupied by the Ethernet IP Core.

9.2 Vision Core

In the following sections the Vision Core is analyzed in three different measurement categories. It was not possible to analyze the Vision Framework in the same way, since some of the utilized IP Cores such as the DDR IP Core and the Ethernet IP Core could not be simulated.

At first the methodology of the measurements is shown and the simulation workflow is depicted.

9.2.1 Methodology of Measurements

To be able to compare the Vision Core with other stereo vision algorithms, several measurements are carried out. We picked three measurement categories: *Resource requirements*, *performance* and *power dissipation*.

Resource requirements consist of the slices and the block RAM that are occupied on the FPGA. The most important slice types are *Slices*, *Slice Register* and *Slice LUTs* (lookup tables).

For synthesis of the generated VHDL code the electronic design automation (EDA) suite Xilinx ISE[3] is utilized. After each synthesis it produces a module level utilization map, which shows the number of slices and block RAM blocks that are utilized by the synthesized VHDL code.

The FPGA on the SP601 features 32 *RAMB16BWER* and 64 *RAMB8BWER* blocks. The former is an 18 Kbit block of block RAM and the latter is a 9 Kbit block of block RAM. Module level utilization maps only show the sum of these two types, thus only the sum is depicted in the analysis.

Performance is measured in frames per second (fps), that is how many complete input images are processed per second. Some of the used algorithms are deterministic and the fps can therefore be calculated easily with only a small amount of processed data. Other of the used algorithms, such as the Scan-Line based Segmentation, have different run-times for different inputs. Thus, a big amount of data is measured and averaged for these types of algorithms.

To be able to determine exact performance values, the simulation of the generated VHDL code (see section 8.5.2 for details on the operation of this simulation) with ModelSim is utilized. The simulator can produce waveforms for selected signals. With these it is possible to identify the precise moments when an algorithm starts with an input and ends the calculation. Thus, a nanosecond accurate measurement can be taken.

Power dissipation is given in milliwatt and only for the Vision Core parts. Other parts such as the Camera Interface, the Ethernet Interface or the cameras and the Ethernet PHY themselves are not included in this analysis.

For calculation of the power consumption Xilinx XPower[4] is utilized. It takes a *Value Change Dump* (VCD) file as input, which describes every switching operation that is taken on a simulation of synthesized VHDL code. For-

[3] http://www.xilinx.com/ise/

[4] http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm

tunately ModelSim can generate VCD files at the same time it generates waveforms. Therefore, the needed VCD files are already available after the performance measurements.

In the following section different configurations of the Vision Core with different optimization primitives are analyzed with the methodology as described above.

9.2.2 Analysis

To analyze the Vision Core, it is firstly split in Median Filter, Bilateral Filter, Scan-Line based Segmentation and Parallel Scan-Line based Segmentation. Each of these parts is firstly synthesized and afterwards analyzed without any Impulse C optimizations. Thereafter different Impulse C optimization pragmas (refer to section 3.2.4 for a list and details of optimization pragmas) are added successively.

With this approach a chain of optimizations is applied and one can see changes in the values of the different measurement categories for each of the optimization possibilities. Thus, a detailed analysis of the measurements is feasible.

Utilized pragmas are the pipeline pragma to build pipelines for bodies of non-nested loops, the non-recursive pragma to declare arrays that are never written and read consecutively and the flatten pragma to flatten if-then-else logic as well as hardware primitive functions.

The unroll pragma, which in theory is able to unroll whole loops and therefore make important parts significantly faster by using more slices in parallel, was not applicable. Whenever an unroll pragma was used, the Impulse C C-to-VHDL compiler crashed. Several different positions for the pragma and even a different compiler version were tried, but this feature was not usable in any of the configurations.

Simulating the algorithms is done on the teddy stereo image set, stripped down to a resolution of 320×240 to be comparable to the resolution of our own cameras. For the Scan-Line based Segmentation the maximum matching search distance is 50 and a color tolerance of 6 is used, which are determined to be good values in section 8.3.1. A standard deviation of 0.15 is used for the Gaussian function in the Bilateral Filter. The Vision Core is clocked with a frequency of 100 MHz

In the following the resource requirements for each of the tested algorithms are analyzed and compared.

9.2.3 Resource Requirements

In table 9.4 slice and block RAM usage for the Median Filter, the Bilateral Filter and the parallel as well as non-parallel Scan-Line based Segmentation are shown.

		Slices	Slice Register	Slice LUTs	BRAM blocks
	Available on SP601	2278	18224	9112	32/64[5]
	Optimization				
Bilateral Filter	without	138	309	339	9
	+ pipel. convolution	35	106	93	9
	+ pipel. histogram update	51	103	132	18
	+ pipel. distributor/serializer	55	106	131	11
	+ non-recursive arrays	54	103	132	9
	+ primitive function flatten	56	106	130	14
Median Filter	without	64	107	145	14
	+ pipel. convolution	65	127	164	14
	+ pipel. histogram update	90	146	222	14
	+ pipel. distributor/serializer	96	146	218	14
	+ non-recursive arrays	94	145	211	14
	+ primitive function flatten	96	140	224	14
SLS	without	320	340	1046	16
	+ pipel. pixel conversion	279	398	874	36
	+ pipel. distance calculation	424	825	1089	36
	+ pipel. image calculation	403	826	1078	36
	+ non-recursive arrays	438	826	1065	36
PSLS	without	1104	1020	2987	33
	+ SLS optimizations	1101	2406	3295	30
	+ pipel. distributor/serializer	1155	2406	3309	30

Table 9.4: Resource requirements of Bilateral Filter, Median Filter, Scan-Line based Segmentation and Parallel Scan-Line based Segmentation.

The Bilateral Filter is configured with a kernel size of 25×25 for the analysis. All six color channels are used, such that the Bilateral Serializer and Bilateral Distributor can also be analyzed. The slice consumption is depicted for only one of the Impulse C Bilateral Filter processes (not the whole distributor/-filter/serializer scheme).

Six different measurements are taken, the first measurement is without any optimizations. The second measurement uses pipelining for the convolution part of the Bilateral Filter, i. e. the sum over the histogram. Histogram updates are pipelined in the third measurement and the Bilateral Distributor as well as the Bilateral Serializer are pipelined in the fourth measurement. After that the non-recursive pragma is used on arrays and the hardware primitive function (which is used for the convolution) is flattened.

When the convolution is pipelined, a drop in the usage of all slice types can be seen. This is because of different optimizations in the synthesis. Besides the listed slice types, the FPGA on the SP601 features a small amount of complex slices, such as DSP slices with up to 48 input signals. Most likely the synthesis tool can use one of those slices when the pipelining for the convolution is activated, hence the consumption of normal slices is decreased.

After the pipelining of the histogram update procedure a small increase in all slice types is identifiable. The rest of the optimizations do not seem to trigger any changes which have an impact on the slice usage. Another puzzling result can be seen in the block RAM usage. The number of used block RAM blocks seems to change randomly between the optimizations. Most likely this is due to the non-deterministic behavior of the synthesis tool. Since only a small amount of the available slices are utilized, the synthesis tool can decide between the use of block RAM and distributed RAM for the storage of the Gaussian lookup tables.

In the Median Filter the convolution part is the determination of the median from the histogram and the kernel size is 11×11 instead of 25×25, otherwise the same code and the same optimizations as in the Bilateral Filter are used. There is no drop in slice usage when the convolution part is pipelined, otherwise the slice usage also behaves similar to the Bilateral Filter. Since no lookup table or other big arrays are used in the Median Filter, the block RAM usage stays stable.

For the Scan-Line based Segmentation it was first tried to pipeline the segmentation phase of the algorithm. The Impulse C C-to-VHDL compiler did

[5] 32 *RAMB16BWER* and 64 *RAMB8BWER* blocks of block RAM are available on the SP601.

generate VHDL code that was synthesizeable, though the pipeline did not generate meaningful data, neither in simulation nor on the SP601. Thus, optimizations for the segmentation phase could not be tested.

The first functioning optimization for the Scan-Line based Segmentation is the pipelining of the pixel conversion, i. e. the part which reads the 15 bit input values from the stream and writes them to three different arrays (for the red, green and blue part of the RGB image). It seems like the synthesis tool can utilize lots of block RAM for this pipeline while saving slices.

Pipelining the distance calculation in the matching phase is done as the second optimization. Since the distance calculation does include lots of arithmetic operations, the emerging pipeline is big and thus an increase in the used slice registers can be seen. Pipelining the grey image calculation (the calculation of grey color pixels from the disparity map) and using the non-recursive pragmas for the arrays does not trigger any relevant slice usage changes in the Scan-Line based Segmentation.

The Parallel Scan-Line based Segmentation is configured to use three Scan-Line based Segmentation processes. It is tested with the full Scan-Line based Segmentation optimizations and with pipelining in the SLS Distributor and SLS Serializer. The former yields a slice usage that is about three times as big as in the normal Scan-Line based Segmentation, which is expected. Optimizing the SLS Serializer and SLS Distributor does not seem to have a big impact in slice utilization.

The resource requirements of the complete Vision Core are illustrated in table 9.5. The first line in the table shows the number of slices and block RAM blocks that are available on the SP601.

The shown slice utilization does include all distributor and serializer processes as well as all Impulse C streaming objects and buffers of the Vision Core. It does not include the slice consumption of VHDL components such as the Ethernet Interface, the Camera Interface or the DDR IP Core.

An average Impulse C stream buffer does occupy about 3 slices, 8 slice registers and 3 slice LUTs. The Bilateral and Median Distributor use 8 slices, 22 slice registers and 2 slice LUTs, whereas the Bilateral and Median Serializer utilize 30 slices, 81 slice registers and 20 slice LUts each.

The Median Filter, Bilateral Filter and Scan-Line based Segmentation themselves occupy up to 3 times as many slices as they do when tested alone. This most likely comes about because of the small amount of special slices that are depleted in the synthesis of the complete Vision Core.

		Slices	Slice Register	Slice LUTs	BRAM blocks
Available on SP601		2278	18224	9112	32/64
Filter	Optimized				
Median	no	1742	3382	5279	35
	yes	1949	4122	5953	35
Bilateral	no	2118	4474	6718	35
	yes	2224	5677	7623	35

Table 9.5: Resource requirements of complete Vision Core with Bilateral Filter and Median Filter.

After the resource requirements are analyzed, the following section will show if the occupied slices for optimizations are worth it. The performance for the same algorithms with the same optimizations is tested.

9.2.4 Performance

Table 9.6 shows the execution time of different measurement units and the frames per second for the Bilateral Filter, the Median Filter, the Scan-Line based Segmentation and the Parallel Scan-Line based Segmentation.

Performance of the Bilateral Filter can be determined by only measuring the execution time for one single pixel. This is possible since the Bilateral Filter does use the exact same amount of calculations for arbitrary input values. Thus, the calculated frames per second are very accurate.

Pipelining the convolution part of the Bilateral Filter brings an improvement of about 17.5 % and pipelining the histogram update procedure speeds the Bilateral Filter up be another 16 %. Using the pipeline pragma on in the Bilateral Distributor and Bilateral Serializer does not result in any improvements, which means that neither the Bilateral Distributor nor the Bilateral Serializer are bottlenecks in the Bilateral Filter.

The non-recursive pragma for the arrays and the flatten in the hardware primitive function did not affect the performance of the Bilateral Filter in any way. This probably means that the Impulse C compiler optimized these parts even without the pragmas.

	Optimization	Time per pixel	Frames per second
Bilateral Filter	without	3480 ns	3.742
	+ pipelining convolution	2870 ns	4.537
	+ pipelining histogram update	2410 ns	5.403
	+ pipelining distributor/serializer	2410 ns	5.403
	+ non-recursive arrays	2410 ns	5.403
	+ primitive function flatten	2410 ns	5.403
		per line	
Median Filter	without	389843 ns	10.69
	+ pipelining convolution	344243 ns	12.10
	+ pipelining histogram update	281800 ns	14.79
	+ pipelining distributor/serializer	281800 ns	14.79
	+ non-recursive arrays	281800 ns	14.79
	+ primitive function flatten	281800 ns	14.79
		per 5 lines	
SLS	without	9342852 ns	2.23
	+ pipelining pixel conversion	8665593 ns	2.40
	+ pipelining distance calculation	3656740 ns	5.70
	+ pipelining image calculation	3657429 ns	5.70
	+ non-recursive arrays	3657081 ns	5.70
		per 5 lines each	
PSLS	without	12116600 ns	5.16
	+ SLS optimizations	5175221 ns	12.08
	+ pipel. distributor/serializer	5175078 ns	12.08

Table 9.6: Performance of Bilateral Filter, Median Filter, Scan-Line based Segmentation and Parallel Scan-Line based Segmentation.

Calculating the execution times for the Median Filter is not possible by only executing it for one pixel. The run-time of the convolution part of the Median Filter depends on the distribution of the values in the histogram, i. e. if the median of a particular kernel window has low intensity value, the Median Filter is will execute fast and vice versa. Thus, the Median Filter is executed on a whole line (320 pixels) and the average running time value is determined.

One would expect, that pipelining the convolution in the Median Filter results in less improvements than in the Bilateral Filter and that pipelining the histogram update procedure brings about the same improvement as the Bilateral Filter. This is due to the fact that the Bilateral Filter has more arithmetic operations in the convolution that can be pipelined and the histogram update part being the same.

Improvements of about 11.7 % for the convolution and about 18.1 % for the histogram update conform to the expectations. Pipelining the Median Distributor and Median Serializer processes does not yield in any improvements, as with the Bilateral Filter. The same holds also true for non-recursive arrays and flattening the hardware primitive function.

The run-time of the Scan-Line based Segmentation is dependent on the input values, as in the Median Filter. Therefore, 5 lines are executed and averaged per tested configuration. These 5 lines are a relatively small example size, though the simulation of 5 lines takes hours and the generated VCD file exceeds sizes of several gigabytes. This means that a larger sample size is not practical.

Using the pipeline pragma in the for the pixel conversion part of the Scan-Line based Segmentation does only improve the runtime by about 0.07 % or by about 0.17 frames per second. The pipeline pragma in the distance calculation of the matching phase however does deliver an improvement of over 50 %. This is by far the best improvement that was achieved by a single pragma. It is interesting to note that the occupied register slices increased by over 50 % for the same optimization.

Pipelining the grey image calculation part of the Scan-Line based Segmentation and using non-recursive arrays does not improve the run-time in a significant manner.

The Parallel Scan-Line based Segmentation was tested with 5 lines of input for each of the utilized Scan-Line based Segmentation processes. Using all optimizations in the underlying Scan-Line based Segmentations yields an improvement of 134 %. The SLS Distributor and SLS Serializer processes are again yielding no improvements and therefore shown to be no bottlenecks.

With all optimizations the parallel version of the Scan-Line based Segmentation is about 2.12 times faster than the non-parallel version. This means that the algorithm lost about 29 % of its efficiency because of the added latency in distributing and serializing.

The performance of the complete Vision Core is illustrated in table 9.7. The Vision is tested with Median Filter and Bilateral Filter and is executed on 5 input lines.

Filter	Optimized	Time per 5 lines	Frames per second
Median	no	10111864 ns	2.06
	yes	4138100 ns	5.04
Bilateral	no	13228400 ns	1.58
	yes	5486792 ns	3.80

Table 9.7: Performance of complete Vision Core with Bilateral Filter and Median Filter.

A complete Vision Core with the Bilateral Filter shows a performance of about 3.8 frames per second. This means that more than 1 frame is lost in the latencies that are added by the serializer and distributor processes as well as Impulse C buffer objects.

When the Vision Core utilizes the Median Filter, the result is better. A performance of about 5 frames per second can be achieved, this is only little behind the theoretical possible value of 5.7 frames per second that was shown for the Scan-Line based Segmentation.

The last measurement that is taken is the power dissipation. It measures the usage of milliwatt for each of the tested algorithms and configurations.

9.2.5 Power Dissipation

In table 9.8 the power consumption of the Bilateral Filter, the Median Filter, the Scan-Line based Segmentation and its parallel counterpart are shown.

Unlike to the measurements of performance and resource requirements, the power consumption turns out to be not at all dependent on any of the tested optimizations.

The only effects that can be seen within the algorithms when using the optimizations show themselves in the pipelining of the serializer and distributor processes. A decrease of about 0.98 %, 3.92 % and 5.45 % in power consumption can be seen in the Bilateral Filter, Median Filter and Scan-Line based Segmentation respectively.

	Optimization	Overall power consumption
Bilateral Filter	without	316.36 mW
	+ pipelining convolution	309.79 mW
	+ pipelining histogram update	309.89 mW
	+ pipelining distributor/serializer	306.85 mW
	+ non-recursive arrays	307.73 mW
	+ primitive function flatten	311.46 mW
Median Filter	without	309.14 mW
	+ pipelining convolution	310.79 mW
	+ pipelining histogram update	312.13 mW
	+ pipelining distributor/serializer	300.75 mW
	+ non-recursive arrays	299.14 mW
	+ primitive function flatten	298.61 mW
SLS	without	322.79 mW
	+ pipelining pixel conversion	321.77 mW
	+ pipelining distance calculation	321.28 mW
	+ pipelining image calculation	321.67 mW
	+ non-recursive arrays	320.43 mW
PSLS	without	379.84 mW
	+ SLS optimizations	379.84 mW
	+ pipel. distributor/serializer	359.15 mW

Table 9.8: Power dissipation of Bilateral Filter, Median Filter, Scan-Line based Segmentation and Parallel Scan-Line based Segmentation.

This is interesting, since the pipelining of distributor and serializer processes neither increased the slice usage nor the frames per second. Thus, it might be a good practice to pipeline processes that only buffer, distribute and merge data without any calculations, since it seems to decrease the power consumption.

A comparison of the power consumption between the tested algorithms does yield expected results. The Bilateral Filter and the Median Filter do use about the some amount of power, where the Scan-Line based Segmentation utilizes more power and the parallel counterpart even more. There are however quite obviously no simple linear correlations, e. g. the Parallel Scan-Line based Segmentation that utilizes three Scan-Line based Segmentation does not use about three times as much power as the non-parallel version.

The power dissipation of the complete Vision Core with Median Filter and Bilateral Filter is illustrated in table 9.7.

Filter	Optimized	Overall power consumption
Median	no	340.13 mW
	yes	343.76 mW
Bilateral	no	357.69 mW
	yes	354.89 mW

Table 9.9: Power dissipation of complete Vision Core with Bilateral Filter and Median Filter.

As expected, there is only small difference in power consumption between optimized and non-optimized versions of the Vision Core. The Vision Core with the Median Filter shows less power usage.

Power dissipation of a complete Vision Core is lower than the power dissipation of one Parallel Scan-Line based Segmentation. Which is expected, since the Parallel Scan-Line based Segmentation runs three big Impulse C processes, while Vision Core is only running one Scan-Line based Segmentation and one preprocessing component.

In the following the results of the measurements of resource requirements, performance and power dissipation are summed up and a conclusion is drawn.

9.2.6 Conclusion

The analysis of the Vision Core revealed some interesting facts about the Impulse C optimization possibilities. For example the flatten pragma for hardware primitive functions and the non-recursive pragma for arrays do not seem to have any effect on resources, performance or power consumption worth mentioning. It can be concluded that either these pragmas do not have any or only very little impact or these optimizations are already executed by the Impulse C C-to-VHDL compiler without the pragmas.

Resource requirements are hard to compare, since the synthesis tool exhibits a great deal of non-deterministic behavior. Especially if only a small system is synthesized, the synthesis tool can chose between different possibilities such as the use of distributed RAM and block RAM or the use of few special slices and the standard slice types.

The performance of the Vision System is shown to be 3.8 frames per second when the Bilateral Filter is used, 5.04 frames per second when the Median Filter is used and 12.08 frames per second with the Parallel Scan-Line based Segmentation, without preprocessing. These numbers might seem low compared to other stereo vision platforms, such as [Mur+07]. They are however respectable if one considers the facts that a rapid development process was used and the focus was on a generic Stereo Vision System, which is modifiable and reusable.

Lastly the power dissipation was determined to be between 300 mW and 380 mW. A difference of 80 mW is probably only relevant for environments in which power efficiency is the number one concern. As a comparison, the Ethernet PHY that is utilized in our Stereo Vision System and needed to transfer data consumes about 750 mW on average. Other components of our system such as the cameras and the DDR memory might even exceed this value. It is evident that the power consumption of the FPGA does not play a big role in the overall power consumption of the Stereo Vision System.

9.3 Stereo Vision System

To measure the quality of images captured by the camera system and the resultant disparity maps from the Scan-Line based Segmentation, different images from our two cameras are shown and analyzed.

The Scan-Line based Segmentation is parameterized with a maximum matching search distance of 50 pixel and a color tolerance of 6 for all of the shown examples. All images are captured with a resolution of 320×240 pixel.

Figure 9.1 illustrates images captured by the MT9 camera with a corresponding disparity map.



(a) Left image (b) Disparity map (c) Right image

Figure 9.1: Images from MT9 Camera with corresponding disparity map from Scan-Line based Segmentation

As one can see, the quality of the images taken by the MT9 Camera are not optimal. It seems that the green color channel is overdriven, because green parts in the image are displayed very bright. The brightness distribution in the images is correct since the light intensities in the images are as expected. Tests were performed which suggest that the readout process and the debayering work correctly. Unfortunately, the reason could not be determined within the scope of this master thesis.

There are several different possible reasons for the erroneous images. Firstly, the camera possesses about 200 registers, that each configure multiple parameters. A simple misconfiguration of the cameras might be be the problem.

Secondly, the camera has high near-infrared sensitivity, such that infrared light may cause this problem. The problem was discovered too late to purchase an infrared filter, since compatible infrared filters are hard to obtain. Furthermore, a programming error in some part that is involved in handling of the camera output data is still possible.

Meaningful data is however shown in the disparity map. The background is fairly uniformly distributed and the book in the foreground is recognizable. Problems in the lower right are most likely due to too low color differences between book and background.

All of the following example images are captured by the OV7 Camera. This camera works as expected. If one considers the low price and the small lens, the produced images are good.

Figure 9.2 shows a detailed illustration of example images consisting of a helmet and a book. The images as well as bilateral and median filtered images with corresponding disparity maps are shown.



(a) Left image



(b) Right image



(c) Filtered MF



(d) Filtered BF



(e) Disparity WF



(f) Disparity MF



(g) Disparity BF

Figure 9.2: Helmet images from OV7 Camera with right image filtered and corresponding disparity maps from Scan-Line based Segmentation without filter (WF), with Median Filter (MF) and with Bilateral Filter (BF)

The Median Filter did blur out most of the inscription of the book and the helmet consists of only a few single-colored segments, which is good. Edges of the book are however also blurred out. Bilateral Filtering yields a completely blurred out description, while preserving sharp edges.

All of the shown disparity maps are of good quality. The Scan-Line based Segmentation produces some artifacts for the unfiltered images, the bilateral filtered disparity map has one artifact in the lower left and the median filtered disparity map does not exhibit any artifacts. Though, the book in the disparity map originated from the bilateral filtered image does have sharp edges, while the edges of the median filtered disparity map are blurred out.

The following two examples are produced by a Scan-Line based Segmentation without any preprocessing. In figure 9.3 a colorful box with corresponding disparity map is shown.



(a) Left image (b) Disparity map (c) Right image

Figure 9.3: Box images from OV7 Camera with corresponding disparity maps from Scan-Line based Segmentation

Several different colors are present on the box, which is hard to process by the Scan-Line based Segmentation. The generated disparity map does have some errors at the top of the box, but it generally is of good quality. Also the disparity map shows a gradient from bright to dark on the surface from the foreground to the background.

In figure 9.4 a house is captured in natural light, in the dawn. It shows lots of details and a high rate of difficult color differences.

The disparity map does feature parts of the roof and the house wall. The quality of the disparity map is however poor. The Scan-Line based Segmentation has problems to construct meaningful segments that can be matched to a good position in the matching phase, since the texture of the wall and the roof is very uniform. This image shows the limitations of the Scan-Line based Segmentation.

Also most of the shown examples do have a acceptable quality, all of the images were captured after multiple tries. It turned out that the rectification requirement of the Scan-Line based Segmentation (see section 3.4.1) is hard to satisfy.



(a) Left image (b) Disparity map (c) Right image

Figure 9.4: House images from OV7 Camera with corresponding disparity maps from Scan-Line based Segmentation

Although we used a steady aluminium mount (see section 5.4), is was very hard to synchronize the angles of both cameras. A vertical difference of only 3 or more pixels will yield in poor results of the Scan-Line based Segmentation.

One of the most important application areas for stereo vision is robotics. For this application special attention has to be paid that the adjustment of the cameras is correct and could not alter.

Chapter 10

Summary

In this thesis a rapid prototyping system for stereo vision algorithms is proposed. The system allows the development of algorithms without deep knowledge of the underlying hardware platform and HDL experience. Two primary tasks were achieved, consisting of the selection of an applicable hardware platform and the development of a vision system.

We selected the SP601 development board from Xilinx as an FPGA development board and implemented the complete vision system on this board. A market analysis was performed to select applicable cameras for image capturing. We decided to utilize cameras with a direct hardware interface to the equipped image sensor. For the tests, cameras with OmniVisions OV7720 image sensors were used. Since cameras with higher performances are to high priced we decided to test the manufacturing of a self-made camera.

To connect the cameras with the SP601 development board printed circuit boards were engineered which interface the cameras and transmit image data to the FPGA over an interference resistant connection. These boards are designed such that other cameras than ours can be interfaced. A necessary voltage translation is also performed by these boards.

During the development it became clear that a precise adjustment of the cameras is essential to enable the possibility of meaningful results from the Scan-Line based Segmentation algorithm. Also, the distance between the cameras has to be adaptable. Therefore we decided to engineer a mounting for the cameras that allows an adjustable distance of the cameras and a stable parallel mounting.

The secondary task was the development of a generic vision system which allows developers without much HDL experience the development of vision algorithms on an FPGA platform in the C language by generating VHDL code with the C-to-VHDL Impulse C compiler.

We followed a new approach since we implemented the system without an integrated processor, which is present in all found publications that utilize Impulse C for vision applications. The advantage of this approach is that costly FPGAs with integrated processors are not necessary and a huge slice consumption for synthesized processors can be prevented.

A direct access of cameras or other hardware components from Impulse C generated VHDL is not possible. Therefore, the vision algorithms were encapsulated in an Impulse C generated IP Core, called Vision Core, which communicates with a hardware abstracting framework, called Vision Framework. This approach offers the possibility of using unaltered vision algorithms on other hardware platforms by only adapting the framework.

The Vision Framework consists of several developed VHDL components which perform various tasks. A Clock Manager generates the required clocks for the system. To read out the connected cameras, so called Camera Interfaces are used. Each interface handles one connected camera of a specific type. It configures the camera over a configuration bus and reads the camera out correspondingly to the required readout sequence. The image data of the camera are provided over a standardized interface. Thus, only a new Camera Interface has to be implemented to read out a new type of camera without adaption of the remaining system.

Another component assures that the captured images stay in synchronicity and transmits these over streams to a DDR memory wrapping component, which performs a double buffering on the images. Memory read and write operations are performed over streams such that access is possible by Impulse C generated VHDL as well as other HDL components.

To communicate the results of the Vision Core to a host PC, a Ethernet Interface was developed which can transmit Ethernet packets with a speed of nearly 1000 Mbit/s.

During the development of the mentioned components we payed attention for generic interfaces such that components of the Vision Framework can be used for other applications.

As an example application we implemented the Vision Core. A generic concept for the Vision Core, consisting of preprocessing subsystem, stereo vision algorithm and postprocessing subsystem was designed.

The preprocessing subsystem consists of an image filter. Before the implementation of the image filter in Impulse C, different filters were tested. We implemented a Gaussian Filter, a Median Filter, a Hybrid Median Filter and a Bilateral Filter in the C language. The filters were tested extensively for there practicability to be implemented in Impulse C and for there suitability to function as input for the Scan-Line based Segmentation.

It was found that all filters can deliver good results if parameterized well. We concluded that the Bilateral Filter and Median Filter can be implemented with reasonable slice usage and good speed in Impulse C by utilizing histograms and lookup tables.

To implement the two filters, a system consisting of distributor, filter and serializer was used. It allows to run six filter processes in parallel, for the red, green and blue color channels of our two input images. Furthermore an Index Generator was implemented to allow large filter kernel sizes without the need to buffer any lines in the filter processes.

As an stereo vision algorithm, the predetermined Scan-Line based Segmentation was converted from an existing C++ source to Impulse C. An improvement was found by extending the determined segments to the left, so that a better matching could be achieved. This implementation uses one single large Impulse C process.

A parallel version of the Scan-Line based Segmentation was designed and implemented to yield higher frame rates. It utilized the fact that the Scan-Line based Segmentation only operates on one line at a time and opened the possibility to run as many Scan-Line based Segmentation processes in parallel as there are lines in the input images.

Furthermore, we tried to find out good values for the two important parameter of the Scan-Line based Segmentation, the maximum matching search distance and the color tolerance. It was concluded that good parameter values are dependent on the environment in which the images are taken and the space between the stereo cameras.

The postprocessing subsystem consists of several Packet Producers and one Packet Serializer. A Packet Producer is created for every Impulse C stream that should transfer data to a host PC. It generates an IEEE 802.3 conform Ethernet packet, that can be received by most PC systems. A generic concept

for the Ethernet packet format was proposed. It allows to send Ethernet packets from different Impulse C streams by utilizing the Ethernet type field. The packets are also constructed in a way, that allows the host PC to show parts of images when packet loss is present. Packet serialization is done in the Packet Producer, it was implemented to function as a buffer, between packets from the Packet Producer and the Ethernet PHY.

Additionally, two different simulations were introduced. Firstly, a simulation on the host PC, which can simulate the basic behavior of the Vision Core, was implemented. An Image Loader and a Raw Socket Producer were implemented to allow the execution on a host PC without the assistance of the Vision Framework.

As a second simulation, a more accurate simulation on the generated VHDL code utilizing ModelSim was implemented. For this, similar to the host PC simulation, Image Loader and Raw Socket Producer were reimplemented as stand-alone programs. This allowed a transparent use of the simulations such that they behave in the same way as they would have, when executed on the FPGA.

To be able to debug the Stereo Vision System, the Host PC Viewer was conceived. It allows the debugging of the output streams by providing a possibility to view the plain camera images, debug output and 3D versions of the disparity map.

In a last step the resource requirements, performance and power dissipation of single components form the Vision Core and the Vision Core as a whole were determined. It was found, that the Stereo Vision System can have a speed of 3-12 frames per second for different configurations of the system. Also example output data of the Stereo Vision System was presented. The results were good, but it was found that it was not possible to achieve these results without several tries, since it was very hard to adjust both cameras to the exact same angle.

Possible solutions for this problem and possible advancements of this project are described in the following.

Chapter 11

Outlook

The Stereo Vision System is a complete system from camera input to output. Nevertheless there are several possible advancements and possible improvements.

Rectified input images are essential to obtain satisfying results, as can be seen in the output image analysis. We address this problem by a mechanical solution, i. e. we try to orientate the cameras correctly. An improvement would take advantage of an FPGA based solution, which rectifies the images in a preprocessing step.

Independently of other preprocessing steps, the disparity map can contain artefacts. If a disparity map with artefacts is used in an robotic environment these artefacts can lead to undesirable behavior of the robot, such as the detection of objects where no objects are present. Therefore a postprocessing step could be used to filter these artefacts, e. g. a 3×3 Median Filter removes one pixel wide artefacts.

A configurable system at runtime would also be a desirable, i. e. that different filter can be activated or configured at runtime from the host PC. This was not possible to implement since the SP601 development board does not feature enough slices to contain our system extended with these features. The utilization of the SP605 development board, which is equipped with a larger FPGA, would allow these extensions. The developed system architecture now allows easy development of other stereo vision algorithms. A variety of different stereo vision algorithms exist, which now could be implemented and compared without concerns of hardware related tasks. Furthermore, an obvious desire is to solve the MT9 Camera image problem.

Bibliography

[AHK07] Kristian Ambrosch, Martin Humenberger, and Wilfried Kubinger. “A Novel Hardware Architecture for an Embedded Stereo Vision Sensor”. In: *Annals of DAAAM*. 2007. See pp. 6, 118.

[Arr05] Pedro Cobos Arribas. “Real time hardware vision system design”. In: *ICS’05: Proceedings of the 9th WSEAS International Conference on Systems*. World Scientific, Engineering Academy, and Society (WSEAS), 2005, pp. 1–6. ISBN: 960-8457-29-7. See pp. 7, 59.

[Ash01] Peter J. Ashenden. *The Designer’s Guide to VHDL*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558606742. See p. 11.

[Bay76] Bryce E. Bayer. “Color imaging array”. In: U.S. Patent No. 3971065 (July 1976). URL: http://www.freepatentsonline.com/3971065.html. See p. 28.

[Ber+96] Massimo Bertozzi et al. “A Stereo Vision System For Real-Time Automotive Obstacle Detection”. In: *IEEE International Conference on Image Processing*. 1996, pp. 681–684. See p. 1.

[BI07] David Borland and Russell M. Taylor II. “Rainbow Color Map (Still) Considered Harmful”. In: *IEEE Computer Graphics and Applications* 27 (2007), pp. 14–17. ISSN: 0272-1716. DOI: http://doi.ieeecomputersociety.org/10.1109/MCG.2007.46. See p. 154.

[Bla+97] M. Black et al. “Robust anisotropic diffusion and sharpening of scalar and vector images”. In: *Image Processing, International Conference on* 1 (1997), p. 263. DOI: http://doi.ieeecomputersociety.org/10.1109/ICIP.1997.647755. See p. 37.

[CMG03] Clifford E. Cummings, Don Mills, and Steve Golson. “Asynchronous & synchronous reset design techniques - part deux”. In: *In SNUG'03*. 2003. See p. 77.

[Eas03] Eastman Kodak Company. *Shutter Operations for CCD and CMOS Image Sensors*. 2003. URL: http://www.isgchips.com/pdf/Shutter_Operations_Kodak_App_Note.pdf. See p. 29.

[Eth] *IEEE Std 802.3 - 2005 Part 3: CSMA/CD (Ethernet) Access Method*. Tech. rep. 2005, p. 3.2.6. See p. 146.

[Gao06] Jon Gao. *10_100_1000 Mbps Tri-mode Ethernet MAC Specification*. Opencores.org. 2006. URL: http://www.opencores.org/project,ethernet_tri_mode. See p. 99.

[Gre04] John E. Greivenkamp. *Field guide to Geometrical Optics*. First. SPIE Society of Photo-Optical Instrumentation Engineering, ISBN: 0819452947, 2004. See p. 24.

[GW92] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1992. ISBN: 0201508036. See p. 37.

[GW93] J. Gil and M. Werman. “Computing 2-D Min, Median, and Max Filters”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 15.5 (1993), pp. 504–507. ISSN: 0162-8828. DOI: http://dx.doi.org/10.1109/34.211471. See p. 40.

[HKT07] Rainer Hain, Christian J. Kaehler, and Cam Tropea. “Comparison of CCD, CMOS and intensified cameras”. In: *Experiments in Fluids* V42.3 (Mar. 2, 2007), pp. 403–411. DOI: 10.1007/s00348-006-0247-1. URL: http://dx.doi.org/10.1007/s00348-006-0247-1. See p. 56.

[HN87] P. Heinonen and Y. Neuvo. “FIR-Median Hybrid Filters”. In: *ASSP* 35 (1987), pp. 832–838. See p. 41.

[Hua81] Thomas S. Huang. *Two-Dimensional Digital Signal Processing II: Transforms and Median Filters*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1981. ISBN: 0387103597. See pp. 39, 45.

[HZ04] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, ISBN: 0521540518, 2004. See p. 32.

[Imp08] Impulse Accelerated Technologies. *Impulse C User Guide*. 2008. See p. 13.

[Jah02] B. Jahne. *Digital Image Processing*. Springer, 2002, pp. 303–306. ISBN: 0387537821. See p. 46.

[KS09] W. Kayankit and W. Suntiamorntut. "Hardware/software co-design for line detection algorithm on FPGA". In: *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2009. ECTI-CON 2009. 6th International Conference on*. Vol. 01. Pattaya, Chonburi 2009, pp. 604–606. DOI: 10.1109/ECTICON.2009.5137079. See pp. 5, 123.

[KSK06] Andreas Klaus, Mario Sormann, and Konrad Karner. "Segment-Based Stereo Matching Using Belief Propagation and a Self-Adapting Dissimilarity Measure". In: *ICPR '06: Proceedings of the 18th International Conference on Pattern Recognition*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 15–18. ISBN: 0-7695-2521-0. DOI: http://dx.doi.org/10.1109/ICPR.2006.1033. See p. 31.

[KZ01] Vladimir Kolmogorov and Ramin Zabih. "Computing Visual Correspondence with Occlusions via Graph Cuts". In: *International Conference on Computer Vision*. 2001, pp. 508–515. See p. 31.

[LGZ08] Xin Li, Bahadir Gunturk, and Lei Zhang. "Image demosaicing: a systematic survey". In: ed. by William A. Pearlman, John W. Woods, and Ligang Lu. Vol. 6822. 1. San Jose, CA, USA: SPIE, 2008, 68221J. DOI: 10.1117/12.766768. URL: http://link.aip.org/link/?PSI/6822/68221J/1. See p. 29.

[Lim90] Jae S. Lim. *Two-dimensional signal and image processing*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990, pp. 469–476. ISBN: 0-13-935322-4. See p. 38.

[Lit01] D. Litwiller. *CCD vs. CMOS: Facts and Fiction*. Dalsa Corporation. 2001. URL: http://www.dalsa.com/public/corp/Photonics_Spectra_CCDvsCMOS_Litwiller.pdf. See p. 26.

[Loz03] Daniel W. Lozier. "NIST Digital Library of Mathematical Functions". In: vol. 38. 1-3. Hingham, MA, USA: Kluwer Academic Publishers, 2003, pp. 105–119. DOI: http://dx.doi.org/10.1023/A:1022915830921. See p. 154.

[LSY06] Cheng Lei, Jason Selzer, and Yee-Hong Yang. "Region-Tree Based Stereo Using Dynamic Programming Optimization". In: *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on* 2 (2006), pp. 2378–2385. ISSN: 1063-6919. DOI: http://doi.ieeecomputersociety.org/10.1109/CVPR.2006.251. See p. 31.

[ML00] Don Murray and James J. Little. "Using Real-Time Stereo Vision for Mobile Robot Navigation". In: *Auton. Robots* 8.2 (2000), pp. 161–171. ISSN: 0929-5593. DOI: http://dx.doi.org/10.1023/A:1008987612352. See p. 1.

[MM03] Yosuke Miyajima and Tsutomu Maruyama. "A Real-Time Stereo Vision System with FPGA." In: *FPL*. Ed. by Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa. Vol. 2778. Lecture Notes in Computer Science. Springer, 2003, pp. 448–457. ISBN: 3-540-40822-3. See p. 6.

[Mol+98] N. Molton et al. "A stereo vision-based aid for the visually impaired". In: *Image and Vision Computing* 16.4 (1998). Vision-Based Aids for the Disabled, pp. 251–263. ISSN: 0262-8856. DOI: 10.1016/S0262-8856(97)00087-5. See p. 1.

[Mur+07] Chris Murphy et al. "Low-Cost Stereo Vision on an FPGA". In: *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on* 0 (2007), pp. 333–334. DOI: http://doi.ieeecomputersociety.org/10.1109/FCCM.2007.44. See p. 173.

[MVE05] O. Uma Maheshwari, G. B. Vanisree, and D. Ebenezer. "Performance of several types of median filters in spectral domain". In: (2005), pp. 135–142. See p. 41.

[NA08] Mark Nixon and Alberto S. Aguado. *Feature Extraction & Image Processing, Second Edition*. 2nd ed. Academic Press, 2008. ISBN: 0123725380. See p. 38.

[Ora01] D. Oram. "Rectification for any epipolar geometry". In: *BMVC01*. 2001, Session 7: Geometry and Structure. See p. 31.

[PH07] Simon Perreault and Patrick Hébert. "Median Filtering in Constant Time". In: *IEEE Transactions on Image Processing* 16.9 (Sept. 2007), pp. 2389–2394. See pp. 40, 45.

[PT05] David Pellerin and Scott Thibault. *Practical fpga programming in c*. Upper Saddle River, NJ, USA: Prentice Hall Press, 2005. ISBN: 0131543180. See p. 13.

[Ram+02] Rajeev Ramanath et al. “Demosaicking methods for Bayer color arrays”. In: *Journal of Electronic Imaging* 11 (2002), pp. 306–315. See p. 29.

[SAGS07] Rafael Muñoz Salinas, Eugenio Aguirre, and Miguel García-Silvente. “People detection and tracking using stereo vision and color”. In: *Image and Vision Computing* 25.6 (June 1, 2007), pp. 995–1007. DOI: 10.1016/j.imavis.2006.07.012. URL: http://dx.doi.org/10.1016/j.imavis.2006.07.012. See p. 1.

[Sem07] NXP Semiconductors. *I2C-bus specification and user manual*. NXP Semiconductors. 2007. URL: http://www.nxp.com/acrobat/usermanuals/UM10204_3.pdf. See p. 63.

[SM09] Mohamed Shafik and Bärbel Mertsching. “Real-Time Scan-Line Segment Based Stereo Vision for the Estimation of Biologically Motivated Classifier Cells”. In: *KI 2009: Advances in Artificial Intelligence*. Vol. 5803. LNAI. 2009, pp. 89–96. ISBN: 978-3-642-04616-2. See pp. 3, 30, 139.

[Soc+00] Dennis G. Socker et al. *The NASA Solar Terrestrial Relations Observatory (STEREO) Mission Heliospheric Imager*. Proc. SPIE Vol. 4139, Instrumentation for UV/EUV Astronomy and Solar Missions, 2000, pp. 284–293. See p. 1.

[SS01a] Daniel Scharstein and Richard Szeliski. “A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms”. In: *International Journal of Computer Vision* 47 (2001), pp. 7–42. See p. 31.

[SS01b] Daniel Scharstein and Richard Szeliski. “A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms”. In: *INTERNATIONAL JOURNAL OF COMPUTER VISION* 47 (2001), pp. 7–42. See pp. 107, 136.

[SS03] Daniel Scharstein and Richard Szeliski. “High-Accuracy Stereo Depth Maps Using Structured Light”. In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 1 (June 2003), pp. 195–202. See pp. 44, 135.

[SS96] Zhongchao Shi and Ryosuke Shibasaki. “An Approach to Automated House Recognition from Digital Stereo Imagery”. In: *MVA*. 1996, pp. 233–236. See p. 1.

[SSG00] Barbara Steinman Scott Steinman and Ralph Garzia. *Foundations of Binocular Vision: A Clinical Perspective*. McGraw-Hill Medical; 1 edition, 2000. ISBN: 978-0838526705. See p. 1.

[Sum07] Mark Summerfield. *Rapid GUI Programming with Python and QT: The Definitive Guide to PyQt Programming*. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780132354189. See p. 151.

[Tec03] Omnivision Technologies. *Omnivision Serial Camera Control Bus (SCCB) - Functional Specification*. Omnivision Technologies. 2003. URL: http://www.ovt.com/products/sccbspec_an_2_1.pdf. See p. 63.

[TM98] C. Tomasi and R. Manduchi. "Bilateral Filtering for Gray and Color Images". In: *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*. Washington, DC, USA: IEEE Computer Society, 1998, p. 839. ISBN: 81-7319-221-9. See p. 42.

[Tom+08] Federico Tombari et al. "Classification and evaluation of cost aggregation methods for stereo correspondence." In: *Computer Vision and Pattern Recognition*. IEEE Computer Society, 2008, pp. 1–8. DOI: http://dx.doi.org/10.1109/CVPR.2008.4587677. See p. 31.

[TS07] Ed Trexel and Dov Stamler. "Accelerating Bilateral Filtering in Xilinx FPGAs". In: *Xcell Journal* 2 (2007), pp. 53–55 (visited on 01/2010/16). See pp. 5, 123.

[Vla04] V. Erokhin Vladimir. *Single Clock Unsigned Division Algorithm*. OpenCores. Apr. 2004. URL: http://www.opencores.org/project,single_clock_divider,overview. See p. 20.

[Wei06] Ben Weiss. "Fast median and bilateral filtering". In: *ACM Trans. Graph.* 25.3 (2006), pp. 519–526. ISSN: 0730-0301. DOI: 10.1145/1141911.1141918. URL: http://dx.doi.org/10.1145/1141911.1141918. See pp. 45, 125.

[Xil09a] Xilinx. *Constraints Guide*. Xilinx Incorporation. 2009. URL: www.xilinx.com/support/documentation/sw_manuals/xilinx11/cgd.pdf. See p. 105.

[Xil09b] Xilinx. *Design Guide Synthesis and and Simulation*. Xilinx Incorporation. 2009. URL: www.xilinx.com/support/documentation/sw_manuals/xilinx11/sim.pdf. See p. 102.

[Xil09c] Xilinx. *Spartan-6 Libraries Guide for HDL Designs*. Xilinx Incorporation. 2009. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/spartan6_hdl.pdf. See p. 74.

[Xil10a] Xilinx. *User Guide: Spartan-6 Clocking Resources*. Xilinx Incorporation. 2010. URL: http://www.xilinx.com/support/documentation/user_guides/ug382.pdf. See p. 73.

[Xil10b] Xilinx. *User Guide Spartan-6 FPGA Block RAM Resources*. Xilinx Incorporation. 2010. URL: http://www.xilinx.com/support/documentation/user_guides/ug383.pdf. See p. 80.

[Xil10c] Xilinx. *User Guide: Spartan-6 Memory Controller*. Xilinx Incorporation. 2010. URL: http://www.xilinx.com/support/documentation/user_guides/ug388.pdf. See pp. 94, 159.

[Xil10d] Xilinx. *User Guide: Spartan-6 Memory Interface Solutions*. Xilinx Incorporation. 2010. URL: http://www.xilinx.com/support/documentation/ip_documentation/ug416.pdf. See p. 94.

[Zim80] H. Zimmermann. "OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection". In: *IEEE Transactions on Communications* 28.4 (1980), pp. 425–432. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1094702. See p. 146.

[ZW94] Ramin Zabih and John Woodfill. "Non-parametric Local Transforms for Computing Visual Correspondence". In: *ECCV '94: Proceedings of the Third European Conference-Volume II on Computer Vision*. London, UK: Springer-Verlag, 1994, pp. 151–158. ISBN: 3-540-57957-5. See p. 31.