Low Latency Implementation of a Resource Utilization Optimized State Discrimination Algorithm

Semester thesis

Max Glantschnig

June 2021

Supervised by:
Roland Matt
Dr. Abdulkadir Akin
Prof. Dr. Jonathan Home

Trapped Ion Quantum Information
Institute for Quantum Electronics

ETH zürich

Abstract

The aim of this semester thesis is to continue the development of a new hardware platform for low latency parallel readout of trapped ions using images from an EMCCD camera. I adapted a state discrimination algorithm, which was designed to use fewer hardware resources than the current implementation by Nick Schwegler, and implemented it in the programmable logic. I also developed an Ethernet application which runs on the processor integrated in the FPGA chip which allows to send the configuration data for this algorithm from a control PC to the board. Using a module which emulates the behavior of the camera, I verified that the algorithm works correctly. The resource utilization of the algorithm (in terms of LUTs and registers) for reading out 10 ions is 6x less than in the previous version. The new platform can support the parallel readout of up to 100 ions while (in total) only using 16 % of all LUTs and 10.45 % of all registers, while maintaining the low latency of the previous implementation, which makes this platform suitable to be used in feedback based quantum error correction codes.

Contents

1 Introduction

Low latency parallel qubit readout is a key component of most error correcting codes and hence a stringent requirement for building a universal quantum computer [1, 2]. In trapped ions, the quantum information is encoded in electronic states of the atom [3, 4]. To perform readout, we make use of the fact that if we shine a laser of certain frequency onto an ion, it will only scatter photons if it is in a certain state, which we refer to as the bright state. If it does not scatter photons, we say that it is in the dark state. Which electronic states actually serve as bright and dark states depends on both the type of ion used as well as on how the quantum information is encoded (see Chap. 3.1 in [5] for how this is done in $^{40}Ca^{+}$). To decide which state the ion is in, one collects scattered photons while the readout laser is turned on (either using a PMT or a camera) and compares the number of photons with a threshold (Chap. 3.2.2 and 5.2.2 in [5]).

In [6], Nick Schwegler simulated and implemented a readout device based on processing the image of an EMCCD camera in an FPGA. He showed that with his architecture, parallel readout of up to 60 ions in a chain is possible in 225 μs with an upper bound infidelity of 10^{-4}.

This project aims at migrating the image processing algorithm to a new hardware platform, the Mars XU3 + EB1 from Enclustra. Last semester, Adithyan Radhakrishnan started working on this and modified the algorithm such that it uses less resources. He also explored how to enable Ethernet on the new platform and developed a Python script that can detect regions of interest (ROIs) given an image of the ions [7].

In this report, I will outline the work I contributed as part of a semester thesis. In Sec. 2, I introduce the reasons for changing the hardware platform as well as the new platform itself. This section also contains an overview over the different parts of the hardware and software design. Sec. 3 contains an in-depth discussion of the Ethernet application which I designed to interface the FPGA with a control PC and is currently capable of sending configuration data to the FPGA board. In Sec. 4, I explain what modifications where made to the image processing algorithm compared to Schwegler's version and how these were implemented in the programmable logic. This is followed by a discussion of the system level verification and by an analysis of the resource utilization. Lastly, Sec. 5 contains an overview of the steps that still need to be taken such that the new platform can be used to perform readout in quantum information experiments.

2 Hardware platform

2.1 Advantages of the new hardware

In [6], Nick Schwegler simulated and implemented a device, which can perform parallel readout of trapped ions with high fidelity. Schwegler used the Xilinx VC707 evaluation board in combination with the Abaco FMC422 frame grabber, which deserializes the pixel data sent via the Camera Link interface and makes it available to the image processing algorithm in the FPGA. There are a few reasons why the decision was made to move to a new platform, which is the Mars XU3 SoC module in combination with the Mars EB1 base board from Enclustra[1,2]. Adithyan Radhakrishnan, the student who was working on this project before me, elaborated on this in his report [7]. I would like to briefly summarize his points:

- The Mars XU3 + EB1 platform is approximately 8 times cheaper than VC707 + FMC422.[3]
- The FMC422 has entered restricted production, which means that Schwegler's architecture cannot easily be replicated for use in other experiments. The EB1 base board has two mini Camera Link connectors. Together with a custom implementation of the frame grabber in the FPGA fabric (developed by Giacomo Bisson in the Quantum Optics group), the functionality is equal to the FMC422.
- The Mars XU3 features a Zynq Ultrascale+ MPSoC (XCZU3EG-2SBVA484I) chip, which contains a Quad-Core ARM Cortex-A53 processor. So far, this microprocessor is used to run a program which communicates via Ethernet with the PC. In the future, this can also be used to do software based on-board image processing.

One downside of the new platform is that the resources in the programmable logic (PL) are smaller than in the VC707. For this reason, Schwegler's image processing algorithm had to be adapted such that it used less resources. Radhakrishnan proposed such an algorithm in [7] and tested it in a behavioral simulation. In Sec. 4, I will explain how I modified his code to make it synthesizable and how I interfaced it with the processing system (PS) in order to set the configuration data for the algorithm.

[1] `https://www.enclustra.com/en/products/system-on-chip-modules/mars-xu3/`

[2] `https://www.enclustra.com/en/products/base-boards/mars-eb1/`

[3] Mars XU3 + EB1 cost 482 CHF + 288 CHF = 770 CHF. VC707 + Abaco frame grabber cost 6200 CHF. Taken from [7] and Enclustra website.

2.2 High level view of the design

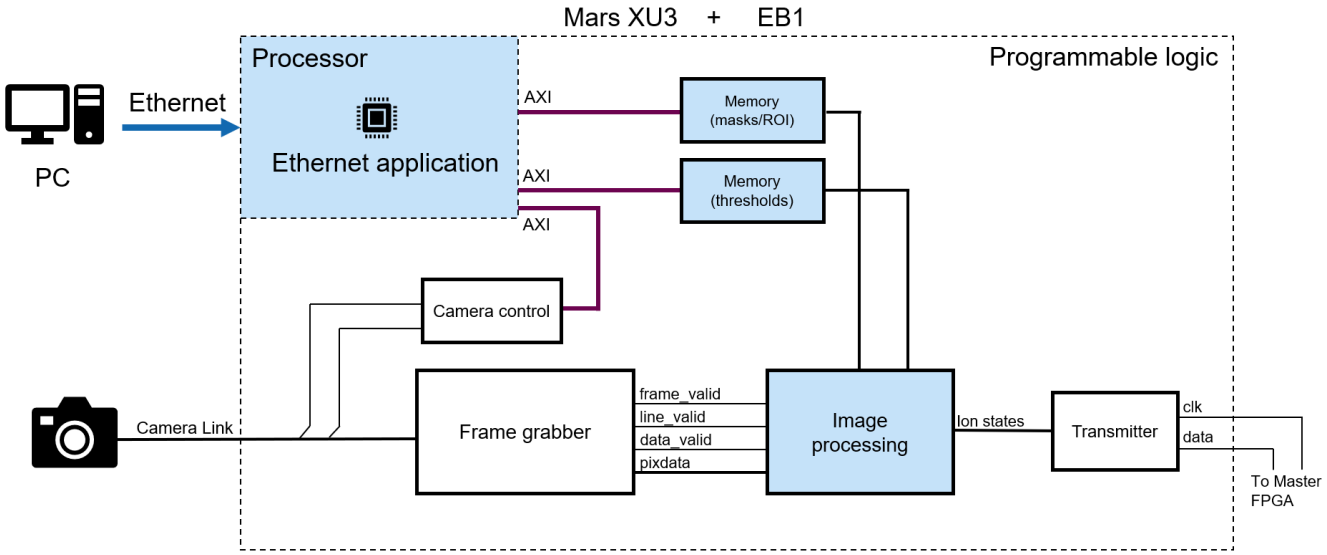


Figure 1: Overview over the different parts of the design in the programmable logic and processor. The large dotted box represents the Xilinx Ultrascale+ chip on the Mars XU3 module. The blue parts were implemented and tested inside the FPGA for this semester thesis.

Fig. 1 shows a high-level block diagram of the design. On the very left of the PL part of the design, we see the Camera Link interface, whose signals are routed to the frame grabber. This module was designed and tested by Giacomo Bisson during a semester thesis in the Quantum Optics group [8]. It provides the pixel intensity as a 16 bit integer (within the Verilog code it is usually called `data` or `pixdata`) and the three video sync signals `frame_valid`, `line_valid` and `data_valid`, which are used to detect the beginning of a new frame or a new line within the pixel stream (see Chap. 1.3.3 in [6] for a detailed discussion of the Camera Link interface). The pixel intensity and the video sync signals are connected to the image processing module which implements the actual state discrimination algorithm. An in depth discussion of this part can be found in Sec. 4.

The top left corner shows the ARM processor. As mentioned above, it is currently used to run a program that receives commands from the PC via Ethernet and communicates with the programmable logic. The PS communicates with the PL via AXI buses. AXI (Advanced eXtensible Interface) is a high-performance, multi-master, multi-slave communication interface that Xilinx uses to connect peripherals to the processor. Most IP cores that Xilinx provides for its Zynq devices come with an AXI interface. In our design

we used the following IP cores:

- Two block memory (BRAM) modules for storing calibration data for the image processing algorithm. BRAM is a convenient way of storing larger amounts of data where only one item needs to be accessed per clock cycle. For a more detailed discussion see Sec. 4.2.
- Two general input output (GPIO) modules, each with a data width of 1 (not shown in Fig 1). They are used to drive a signal called `calibration_valid`, which indicates to the algorithm that valid calibration data has been written to the BRAMs.
- A UART module for communicating with the camera. The Camera Link interface has two dedicated LVDS pairs which serve as RX and TX signals for UART serial communication. The camera can be controlled by sending ASCII characters over this channel and answers from the camera can be received in the same way. This module is depicted in a light gray, because I have not interfaced it with the Ethernet communication yet. I have, however, instantiated the necessary modules in Vivado Block Design already and Bisson tested the hardware by sending data from the PS to the camera [8].

On the right of Fig. 1, the transmitter module acts as an interface between the image processing platform and the experimental control system. It sends the result of the readout, the vector (bright, dark, dark, ...), using a serial communication protocol explained in Chap. 2.1 of [6]. During the course of this project, it was unfortunately not possible to test this module together with the other parts of the platform.

3 Ethernet communication

Controlling lab devices via Ethernet has several advantages over using more basic protocols such as UART. There is no direct wired connection necessary between control PC and device, they merely need to be within the same network. Also, when using the TCP/IP (acronym for Transmission Control Protocol / Internet Protocol) protocol for transmission, it is guaranteed that the data arrives in the exact order it was sent and that no data gets lost or corrupted on the way [9].

In this chapter, I explain how the communication using the TCP/IP protocol works on the software side. First, I introduce the software library I used, then I cover the details of the implementation in the board and finally, I briefly explain how to send data from the PC using a Python script.

As part of his semester thesis, Radhakrishnan set up the hardware of the Mars XU3 + EB1 such that it is capable of running an Ethernet application

[7]. I heavily relied on this work for designing the software.

3.1 LightWeight IP (lwIP)

LightWeight IP (lwIP) is a software stack that has been developed to provide TCP/IP networking functionality for embedded systems [10]. It is designed to be used for a so-called "baremetal" application, meaning it runs directly on the processor and does not require a full-fledged operating system such as Linux. The Ethernet application is entirely written in C and I used Vitis (successor of SDK), an IDE from Xilinx which has an inbuilt version of lwIP[4], to develop and test it. lwIP's working principle is based on events and callbacks. Whenever lwIP detects a certain event (such as "New TCP connection established", "Data received" or "Data sent"), it calls a function that has been specified by the user beforehand. I will give an example further below.

Before being able to send and receive data, a TCP connection has to be established. This involves a handshaking process, where one device acts as a server, listening to connection requests, and the other one as a client, who is actively initiating a connection. After the connection has been established, both devices can send and receive data. In our implementation, the FPGA acts as the server and the PC is the client.

Appendix A.1.1 explains the code required to start up the application and configure it to act as a server.

3.2 Structure of the Ethernet message

Data sent on a TCP/IP connection can be thought of as a continuous stream of bytes. As already mentioned, TCP guarantees that the bytes arrive in the correct order and uncorrupted. It is, however, our job to keep track of the beginning and end of a message and to interpret the bytes in terms of which data they represent (e.g. ASCII characters, 32 bit integers, etc.). We decided that every message should begin with a **header** composed of 11 bytes, followed by the actual data (also referred to as **payload**). Sometimes I refer to header + payload as one "packet" (to not confuse it with Ethernet packets or segments, I will put quotation marks around it). The interpretation of the bytes in the header is as follows (the names in the left column are also the variable names in the C code):

[4] `https://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf`

Bytes	Name	Data type
1	`level`	`s8`
2	`num_packet`	`u8`
3	`num_tot_packet`	`u8`
4-5	`num_bytes`	`u16`
6-7	`num_data`	`u16`
8-11	`address`	`u32`

`num_bytes` are the number of bytes in the payload. Using this information, we know where the message ends and (possibly) a new one begins. Because `num_bytes` is a 16 bit integer, the maximum number of bytes in the payload is $2^{16} - 1 = 65535$. If we want to sent data which is longer than this number, we have to send it as two "packets". `num_tot_packet` is the total number of "packets" one message consists of, `num_packet` denotes which "packet" it is. For this project, all messages were so short that they fitted in one "packet", so these two variables are currently not used. The idea behind `num_data` is to indicate how many individual data items there are. For example, if we send 32 bit integers, then `num_data = num_bytes/4`. The 32 bit `address` can be used to write values to an arbitrary memory location which the processor has access to.

The most important byte of the header is the first one, referred to as `level`. It is used to tell the receiver (i.e. the FPGA board) how to interpret the payload. Currently, the application works with `level` values of 1 (for setting the mask configuration data, see Sec. 4.2.1) and 2 (for setting the threshold configuration data, see Sec. 4.2.2). In the code, I defined an enumerated type `level_t` to refer to these values using `MASK` and `THRESH`, so it is more obvious what they mean. Refer to Appendix A.1.3 for more information on how this is implemented in the code.

3.3 Implementation - Zynq

This section covers the details of the implementation in the board. The source code, which is referred to here and in Appendix A.1, can be found in the TIQI Gitlab. In Appendix A.3 I included a small guide on how to recreate the Vitis project after having cloned the GitLab repository.

3.3.1 Struct for buffering received data

To buffer the data which we receive on the Ethernet connection, I use a C struct called `w_job`. Listing 1 shows its declaration.

```
1  struct w_job {
2    u16 header_pos; //header length is 11 bytes
3    char header[HEADER_LEN];
4
5    //unpacking the header
6    level_t level;
7    u8 num_packet;
8    u8 num_tot_packet;
9    u16 num_bytes;
10   u16 num_data;
11   u32 adress;
12
13   //for intermediate storage of the payload
14   char* data_point;
15   char* write_point;
16   u16 recv_bytes;
17 };
```

Listing 1: Definition of `struct w_job`

The array `header` serves as an intermediate storage for the bytes belonging of the header. `header_pos` keeps track of how many bytes have already been read from the Ethernet interface. Below are the different components of the header as explained in Sec. 3.2. There is a function called `resolve_header` which interprets the raw bytes and assigns the variables at lines 6-11. To store the payload data, we do not use arrays, because the length of the payload is not known at compilation time, we have to allocate memory dynamically (i.e. use malloc). `data_point` points to the first element of the payload data and `write_point` to the position **after** the last payload byte (it is called `write_point` because the next byte we receive will be **written** to that position).

3.3.2 Receive callback function

The receive callback function `tcp_recv_perf_traffic` is executed every time data is successfully received by lwIP. All other user defined functions (e.g. write mask configuration data to BRAM, etc.) are called from within this function. The function's signature is shown in Listing 2:

```
1 static err_t tcp_recv_perf_traffic(void *arg,
2                                    struct tcp_pcb *tpcb,
3                                    struct pbuf *p,
4                                    err_t err)
```

Listing 2: Function signature of `tcp_recv_perf_traffic`

Here, the "protocol control block" `tpcb` contains information about the current TCP connection. `struct pbuf *p` is the most interesting argument: it contains the actual data that was received, stored as a `void`-pointer `payload` to the first element and the number of bytes `len`. In fact, `struct pbuf` also contains a pointer to another object of type `struct pbuf` - it is a linked list. However, when I tested the application, the pbuf chain always had length 1. Nevertheless, the code does handle the general case.

It may well happen that not the whole message we sent arrives with one pbuf. The rest of the message will then be presented to the user the next time when `tcp_recv_perf_traffic` is called (i.e. right after the current execution of `tcp_recv_perf_traffic` finishes). But because the contents of the current pbuf are not available anymore when the function returns, we need our own buffer to store the data of a message that spreads across multiple calls of `tcp_recv_perf_traffic`. In our implementation, `struct w_job` serves this purpose and we access this struct through the `arg` variable. In Appendix A.1.2, I explain this in more detail and give examples of the source code.

3.4 Implementation - PC

For sending the data to the board, I use the Python module socket. It offers an easy to use API for opening and closing Ethernet connections and sending/receiving data on them. I implemented a Python class called `message` which through its methods creates the header and the payload from e.g. a file (for the mask configuration data) or a Python list (for the threshold values). When we send numbers larger than one byte, one has to send them in little-endian order.

4 Image Processing Algorithm in Programmable Logic

The algorithm which is implemented and tested in this thesis was initially developed by Adithyan Radhakrishnan during a semester thesis [7]. It is functionally equivalent to Nick Schwegler's version but the number of comparators used for detecting whether a pixel belongs to a ROI of an ion does

not scale with the number of ions `N_ion_max` or the ROI size `N_roi_max`.

Every clock cycle new values of `pixdata_i`, `pos_x_i`, `pos_y_i` are provided to the module. The pixels arrive row-by-row from left to right, starting with the top-left corner of the image (this the order dictated by the readout of EMCCD camera, see Chap. 1.3.1 in [6]). Prior to the experiment, the user determines a "region of interest" (ROI) for every ion. The intensity value for pixels in the ROI are summed up (i.e. we "mask" the image) and compared to a threshold. If the accumulated value is larger than the threshold we classify the ion as "Bright", otherwise we classify it as "Dark". The ROI (or mask) configuration data is provided to the algorithm as triplets (x, y, ion number), i.e. a pixel with coordinates (x,y) belongs to the ROI of ion with "ion number". The triplets have to be provided as an ordered list, where the order is the same as the order of the pixels arriving from the camera. As an example, [(26, 2, 1), (40, 2, 2), (25, 3, 1), (41, 3, 2)] would be a correctly ordered list for two ions with ROI size 2. It is important to note here that (in contrast to Schwegler's implementation) one pixel cannot simultaneously belong to the ROIs of two different ions, the reason being the following:

When a new frame starts, the first ROI triplet is stored in a register called `mask_current`. `pos_x_i` and `pos_y_i` are continuously compared to the current x (`mask_current[15:0]`) and y position (`mask_current[31:16]`). If the signals are equal, the next ROI triplet is loaded to `mask_current` and the value of `pixdata_i` is added to one of the `camera_counts` registers.

There are in total `N_ion_max` of these registers and we use `ion_no_hit` = `mask_current[47:32]` -1. The -1 here is important, because currently the ions are indexed using natural numbers starting from 1 in the triplets, but starting from 0 in the hardware. The value of the `camera_counts` registers is compared to the threshold (each ion can have an individual threshold value) and the result (1 for bright, 0 for dark) is a `N_ion_max` bit signal, which can then be sent to the control system of the experiment.

4.1 Structure of the design

Within the top module `system_top`, there are two main parts: The Vivado block design, which contains all the AXI devices needed for exchanging information between PS and PL, and the Verilog module `top_image_processing`, which (as the name suggests) contains the actual image processing algorithm. The module `pixel_generator` simulates the behavior of an actual Camera Link frame grabber (i.e. it provides pixel intensity data and the video sync signals).

The most important block is the Zynq Ultrascale+ processing system, which contains the ARM processor running the Ethernet application. It provides two clocks `Clk50` (at 50 MHz) and `Clk100` (at 100 MHz), where the latter

one is used to clock all other modules. It also includes an AXI Master port, which connects the modules implemented in the PL to the Zynq processing system. Two of these modules are AXI BRAM controllers, used to write configuration data for the image processing algorithm (Sec. 4.2 covers this in more detail). Another two are 1-bit wide GPIOs that are used to signal the image processing algorithm that the configuration data has been written and the algorithm is ready to use.

Appendix A.4 shows a screenshot of the Vivado Block design with all the instantiated IP cores from Xilinx.

4.2 Memory for storing calibration data

We decided to use block memory (BRAM) to store all calibration data that have to be set by the user before the image processing algorithm is ready to use. There are two main reasons for this:

- The data size scales heavily with the number of ions. For example, the mask configuration data for 100 ions assuming a ROI size of 25 pixels amounts $100 \times 25 = 2500$ triplets of the from (x, y, ion number). We represent each of these numbers as a 16 bit integer. This amounts to a data size of $2500 \times 3 \times 16$ bits = 120 Kb. The FPGA in our device (XCZU3EG) only has 141,120 CLB Flip-Flops available[5], so we would use up a significant amount of them. However, the amount of available BRAM is 7.6 Mb, so we can easily store this amount of data.
- To issue read and write operations from the processor, one can use an IP core from Xilinx, the AXI BRAM Controller. This reduces the amount of debugging time necessary, because this IP has been tested by the provider.

The BRAM blocks themselves were added to the Vivado Block design (the IP core is called Block Memory Generator). Usually, for this IP core, one can freely choose the width (i.e. the number of bits stored at one address) and the depth (i.e. the number of data words). If we use the IP core together with an AXI BRAM Controller, there are only certain combinations of width and depth possible. In Fig. 2, we see that the widths have to be a power of 2 with the smallest one being 32 bits (the data bus width of the master AXI port) and the same holds for the depth.

[5] `https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf`

Data Width (Bits)	Memory Depths (Words)
32	512, 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k
64	512, 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k
128	512, 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k
256	512, 1k, 2k, 4k, 8k, 16k, 32k, 64k
512	512, 1k, 2k, 4k, 8k, 16k, 32k
1024	512, 1k, 2k, 4k, 8k, 16k
2048	512, 1k, 2k, 4k, 8k

Figure 2: Supported memory sizes if Block Memory Generator is used in combination with AXI BRAM Controller. For the mask data we chose the combination 64 bits, 4k depth and for the threshold vales 32 bits, 2k depth. Taken from [11].

The reason for this can be found in one peculiarity about this configuration: The Block Memory Generator must have the option "Byte Write Enable" turned on. This means that the BRAM address actually indexes individual bytes and not word, similar to the way how AXI adressing works. This naturally gives rise to an address alignment rule. What I mean by that: For a 64 bit wide BRAM, the address indexing the least significant byte (LSByte) of the n^{th} word needs to be $n \times 2^3$. This is equivalent to saying that if we right shift the address by three bits, we get the address of the data word (i.e. n in the example). Fig. 3 shows a graphical representation of how the bytes are arranged.

Byte Address	n+7	n+6	...	n+1	n
Byte Label	7	6	...	1	0
Byte Significance	MSByte		...		LSByte
Bit Label	63				0
Bit Significance	MSBit				LSBit

Figure 3: Arrangement of bytes and address layout for one data word of a 64 bit BRAM. Taken from [11].

4.2.1 Mask coordinates

As already mentioned in previous sections, the ROI or mask calibration data are triplets (x, y, ion number), identifying which pixels belong to the ROI of a specific ion. We represent each number using 16 bits, so one triplet uses 48 bits. Due to the above mentioned restriction of the data width, we used a 64 bit BRAM to store this data.

Byte Addr	$n+7$	$n+6$	$n+5$	$n+4$	$n+3$	$n+2$	$n+1$	$n+0$
Data	8'd0	8'd0	ion[15:8]	ion[7:0]	y[15:8]	y[7:0]	x[15:8]	x[7:0]

Table 1: Byte arrangement for one mask triplet (x, y, ion number). This represents the n^{th} word for the 64 bit BRAM.

Table 1 shows that the bytes are arranged in little-endian order. This convenient for two reasons: 1. Two consecutive bytes can be naturally interpreted as a 16 bit number just by concatenating the bits. 2. The processor also uses little-endian ordering, so there is no ambiguity on how to pass the numbers to the C function which issues the AXI write command.

The depth of the BRAM was chosen to be 4096, this allows for the storage of up to 163 ions with ROI sizes of 25 pixels each.

Appendix A.1.4 shows the C code which writes the buffered data from `struct w_job` to the BRAM. A detailed description of how the image processing algorithm accesses the BRAM is given in Sec. 4.3.

4.2.2 Threshold values

The threshold values are provided as 32 bit numbers to the algorithm. We decided to use a 32 bit BRAM (with the minimum depth of 2048) to store these values as well, although it is not as suitable here: The algorithm is designed such that the comparison between camera counts and thresholds is continuous and in parallel. So before the first frame arrives, we anyway need to load the threshold values from the BRAM into `N_ion_max` registers (see 4.3).

We nevertheless decided to use BRAM because this approach was rigorously tested with the mask configuration data (both how to write the data using the processor and how to access it from within the algorithm).

4.3 Thresholding algorithm

The module `top_image_processing` contains the actual state discrimation algorithm. It is divided into two submodules: `pixel_positions` and `thresholding`.

The clock for this module should be provided by the Frame Grabber (which essentially forwards the clock from the Camera). This is because the readout speed is dictated by the camera and one pixel is transferred in one clock cycle. Because this design has not been tested with an actual camera, I used the 100 MHz `Clk100` from the PS for both `top_image_processing` and `pixel_generator`, which mocks the behavior of the camera.

`pixel_positions` contains a state machine with 3 states (IDLE, READY, ANALYZE_FRAME, very similar to Schwegler's implementation). The

FSM transitions to READY, once `calibration_valid` goes high (i.e. once mask and threshold values have been successfully written to the BRAMs). If it is in the READY state, the transition to ANALYZE_FRAME occurs when there is a positive simultaneous edge of `frame_valid` and `line_valid`, which marks the beginning of a new frame. During this state, the x and y positions of the pixels are counted in the following way: The x counter is incremented in every clock cycle during which all three video sync signals are high. The y counter is incremented when a positive edge of `line_valid` is detected.

Both counters start counting from one. This means that also the mask triplets (x, y, ion) must index the coordinates starting from one. Also, the `pixel_positions` module introduces one clock cycle of latency to the pixel stream.

When the end of the frame is detected, the signal `ion_states_valid` is high for one clock cycle. Right now, it is used to reset the `thresholding` module. In the future, it can be used to initiate the transmission of the readout result of the master control system.

Fig. 5 shows a more detailed schematic of the circuit layout with emphasis on the part which determines `mask_current`. In the top center of the figure, the inputs `pos_x_i` and `pos_y_i` are compared to `mask_current`. If the pixel coordinates match, the `equal` signal is high in the same clock cycle. The equal signal then triggers several different other events:

1. The register `is_even` is updated. If `equal` is high in one clock cycle, `is_even` will be negated in the next cycle. This signal controls a multiplexer which selects which register (`even` or `odd`) determines `mask_current`
2. The register `mask_count` is incremented by 1. `mask_count_d` constitutes the first 29 bits of the 32 bit address used to access the mask BRAM. The three least significant bits are set to 0 because (as mentioned in Sec. 4.2) for accessing the data word we need to divide the address by #bytes/data word$= 8 = 2^3$.

The architecture using `even` and `odd` in between `mask_current` and the output of the BRAM `mask_dout_i` was initially intended to deal with the one clock cycle latency associated with reading from the memory. This would have caused issues when two consecutive pixels belong to some ROI, because `mask_count` is only incremented at the next positive edge after `equal` is high, i.e. the new mask value only arrives in the cycle **after** the second pixel. This matter was eventually solved by applying `mask_count` +1 to the address port of the BRAM if `equal` is high. This can also be seen in the center right part of Fig. 5. While not strictly necessary, `even` and `odd` are

still useful because they cut the path between BRAM and comparators.

Fig. 4a shows the timing diagram of several signals that are involved in the determination of `mask_current`. It shows the situation in the middle of the analysis of a frame when there is a match for three consecutive pixels. Initially, after the reset period, `even` and `odd` have to be filled with mask triplets #0 and #1, only then the algorithm is ready to use. This is managed by a second counter `config_count`. Fig. 4b shows the associated waveforms.

In the lower part of the schematic one can see the `N_ion_max` registers (`camera_counts`) used for storing the accumulated pixel values. If `equal` is high, the value of `pixdata_i` is added to the register corresponding to `ion_no_hit` = `mask_current[47:32]` −1. The accumulated values are continuously compared to the thresholds. The results of this comparison make up the `N_ion_max` bit signal `ion_par_data_o` which is the result of the state discrimination and will transmitted to the experimental control system.

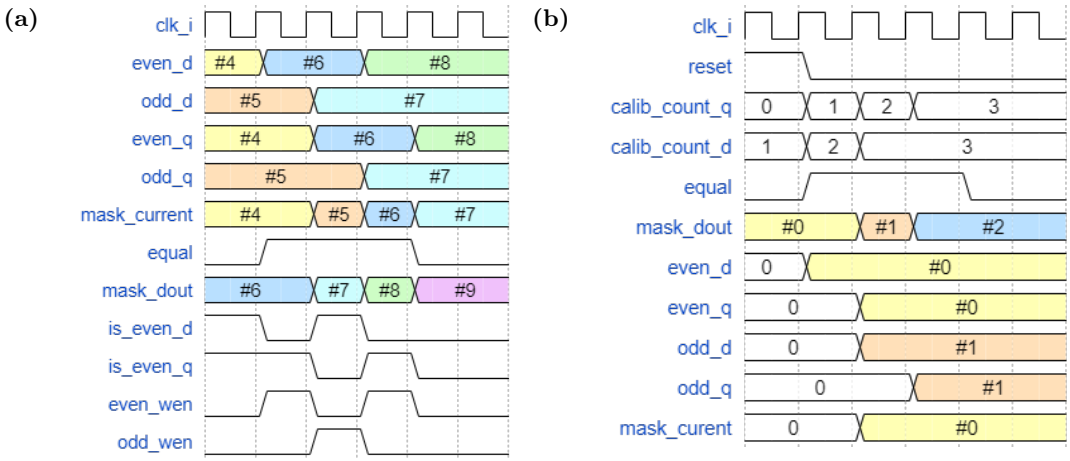


Figure 4: (a) Timing diagram for the signals necessary to determine `mask_current` (b) Timing diagram for the initialization sequence after the reset period. The `even` and `odd` registers are filled with mask triplets #0 and #1.

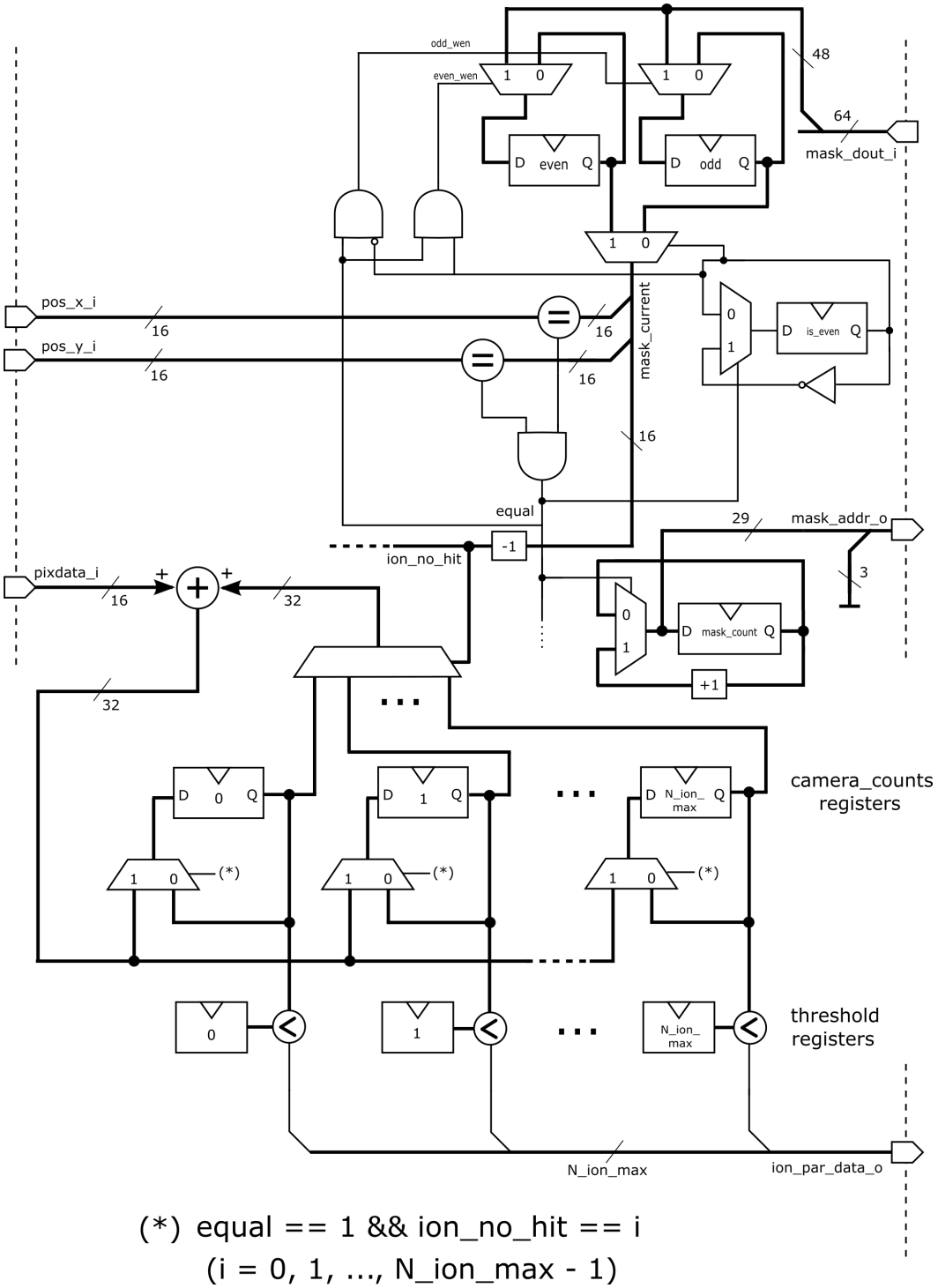


Figure 5: Schematic of the `thresholding` module. See main text for description of the functionality. Some parts have been omitted for readability: The logic which for the initialization sequence after the reset period is over, the logic for filling the threshold registers is not shown.

4.4 Resource utilization and timing analysis

For the following analysis, I synthesized and implemented the design for three different values of `N_ion_max`: 7 (this was the number of ions using in testing), 10 (to compare it with Schwegler's implementation), 100 (to show that it scales). To make sure that the synthesizer does not optimize away parts of the logic, I connected the signals that would eventually go to the transmitter to a tool called Integrated Logic Analyzer (ILA)[6]. It lets you "see" the values of certain signals within the programmable logic. The resources taken up by the ILA are not included in the below analysis.

Fig. 6 shows the resource utilization in Nick's implementation, classified according to modules. I highlighted the row which is most relevant for comparison with our design. `user_if` contains the registers for storing the calibration data, as well as the actual image processing module (`thresholding`). For a fair comparison, we have to substract the resources that `latency_measurement` and `Transmitter` use because this functionality is not (yet) implemented in the new design.

Name	Slice LUTs (303600)	Slice Registers (607200)	F7 Muxes (151800)	F8 Muxes (75900)	Slice (75900)	LUT as Logic (303600)	LUT as Memory (130800)	LUT Flip Flop Pairs (303600)	Block RAM Tile (1030)
vc707_fmc422_axi	19567	21807	1689	796	8844	19364	203	5758	474
axi_cid_ex_0 (axi_cid_ex)	118	99	19	0	38	118	0	66	0
axi_cmd8_mux_0 (axi_cmd8_mux)	114	0	0	0	68	114	0	0	0
axi_fmc422_0 (axi_fmc422)	2785	2614	65	32	1143	2651	134	1415	0
axi_i2c_master_0 (axi_i2c_master)	665	876	5	0	276	665	0	396	0
axi_intrc16_0 (axi_intrc16)	482	754	0	0	211	482	0	338	1
axi_tcpip_engine_vc707_0 (axi_tcpip_engine_vc707)	5722	5895	7	0	2327	5653	69	2362	13
axi_wfm_capture_0 (axi_wfm_capture)	2165	2011	504	252	1250	2165	0	670	456
axi_wh256towh64_0 (axi_wh256towh64)	110	184	0	0	68	110	0	72	4
user_jf_inst (user_jf)	7409	9374	1089	512	3724	7409	0	430	0
i_axi_lite_mux_inst0 (cmdXX_mux_wrapper)	64	147	0	0	56	64	0	29	0
user_axi_wrapper_inst (AXI_Reg_gen_v1_0)	7346	9227	1089	512	3688	7346	0	401	0
AXI_Reg_gen_v1_0_S00_AXI_inst (AXI_Reg_gen_v1_0_S00_AXI)	7346	9227	1089	512	3688	7346	0	401	0
user_top_inst (user_top)	3792	499	1	0	1777	3792	0	356	0
latency_measurement_inst (latency_measurement)	81	179	0	0	61	81	0	51	0
thresholding_inst (thresholding)	3642	298	0	0	1706	3642	0	287	0
Transmitter_inst (Transmitter)	67	20	1	0	20	67	0	18	0

Figure 6: FPGA resource utilization in Nick Schwegler's implementation. The marked module contains registers for configuration data and the actual image processing algorithm. Taken from [6].

Fig. 7 shows the utilization in my implementation for `N_ion_max` = 10. To be able to compare the figures, we have to not only take `top_image_processing` but also the AXI BRAM controllers and AXI GPIOs have to be taken into account. This is because Schwegler's `user_if` also contains logic for the AXI interface.

[6] `https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf`

Name	CLB LUTs (70560)	CLB Registers (141120)	CARRY8 (8820)	F7 Muxes (35280)	F8 Muxes (17640)	CLB (8820)	LUT as Logic (70560)	LUT as Memory (28800)	Block RAM Tile (216)
system_top	9345	10720	95	30	4	2001	8637	708	10.5
dbg_hub (dbg_hub)	444	739	7	0	0	126	412	32	0
MarsXU3_i (Mars_XU3_wrapper)	7835	7997	8	28	4	1586	7236	599	10
pixel_generator_i (pixel_generator)	45	51	2	0	0	14	45	0	0
top_image_processing_i (top_image_processing)	403	841	68	0	0	125	403	0	0
u_ila_0 (u_ila_0)	622	1092	10	2	0	180	545	77	0.5
• • •									
axi_bram_ctrl_0 (Mars_XU3_axi_bram_ctrl_0_0)	286	317	2	1	0	86	286	0	0
axi_bram_ctrl_1 (Mars_XU3_axi_bram_ctrl_1_0)	239	207	2	1	0	66	239	0	0
axi_gpio_0 (Mars_XU3_axi_gpio_0_1)	30	36	0	0	0	7	30	0	0
axi_gpio_1 (Mars_XU3_axi_gpio_1_0)	31	36	0	0	0	7	31	0	0

Figure 7: FPGA resource utilization in our implementation. We have to add up `top_image_processing` and the AXI devices to be able to compare the resources to 6.

Our implementation uses 989 LUTs, whereas Schwegler's uses 7261, i.e. we use about 7x less LUTs. Our register usage is 1,437 whereas for Schwegler it is 9,175. This is a 6x reduction in register usage, which is mainly due to the fact that we use BRAM for storing the calibration data for masks and thresholds. The difference in LUT occupation is partly explained by the fact that we changed the architecture of the image processing algorithm: Instead of $2\times$ `N_roi_max` $\times$ `N_ion_max` comparators we only use 2. Another explanation is the following: If we substract the used LUTs in Schweglers's `thresholding` module from the total ones used in `user_if`, we see that there are 3,617 LUTs used for the logic which translates AXI write and read commands to the registers. I assume that we do not have this overhead in our implementation because the translation from AXI to BRAM requires less logic.

To show that our design can also support parallel readout of up to for several tens of ions, I synthesized and implemented the design for `N_ion_max` $= 100$. Table 2 shows the percentage of utilized resources for the three most important categories for both 10 and 100 ions classified according to modules (only the module `top_image_processing` scales with the number of ions. We see can see that there is enough space left to instantiate missing components such as a transmitter for the ion states (Sec. 5.2) or the frame grabber (Sec. 5.1). According to [8], the frame grabber needs about 0.02 % CLB LUTs and 0.013 % CLB registers.

Module	CLB LUTs	CLB registers	BRAM
system_top (10 ions)	11.73 %	6.31 %	4.63 %
system_top (100 ions)	16.0 %	10.45 %	4.63 %
↳ Block design	11.10 %	5.67 %	4.63 %
↳ pixel_generator	0.06 %	0.04 %	0.0 %
↳ top_image_processing (10 ions)	0.57 %	0.6 %	0.0 %
↳ top_image_processing (100 ions)	4.84 %	4.74 %	0.0 %

Table 2: FPGA resource utilization for the implemented design. The table also shows how the resources are distributed among the different sub modules. `top_image_processing` scales with the maximum number of ions supported, here it is shown for 10 and 100 ions.

The modules `top_image_processing` and `pixel_generator` are both supplied with a 100 MHz clock. All timing constraints are met by the implementation procedure. Eventually, this clock will be replaced by the one provided by the camera. As the maximum operating frequency of the Camera Link interface is 85 MHz, I expect the timing to also hold when the design is used with an actual camera.

4.5 Functional verification

I created the Verilog module `pixel_generator` which simulates the behavior of the frame grabber. Once the reset period is over, the module continuously sends frames with a certain idle time between frames. This idle time is one of the parameters given to the module along with the number of pixels in x direction (width) and y direction (height), and the number of clock cycles between two lines. The outputs of the module are the three video sync signals `frame_valid_o`, `line_valid_o`, `data_valid_o` and the pixel intensity `data_o`. For simplicity, `data_o` has the same value as the x coordinate as the corresponding pixel. Simulating a real image in the programmable logic would be harder and, in fact, unnecessary to verify the correct behavior of the model. What we need to check is, whether the value in `camera_counts` corresponds to what we expect given certain mask calibration values. To have a value for `camera_counts` to compare to (a "golden model"), I wrote a small Python script which calculates the accumulated pixel counts based on the mask values.

As a first step, I tested `top_image_processing` using the Vivado Simulator. This uses a behavioral model for the BRAMs which store the configuration data. Once the correct functioning has been verified, I moved on to test the module after the FPGA has been programmed. To view the `camera_counts` and other signals, I used an ILA core. This way I could verify that the pixel values are indeed correctly summed up and that the configuration data (mask

and thresholds) is correctly accessed by the image processing algorithm.

Fig. 8 shows a screenshot of the ILA waveforms that were generated as part of the verification procedure. I tested the module for `N_ion_max` = 7. The first seven signals are the `camera_counts` registers. The values seen in the figure match with what the "Golden model" predicted. The screenshot also shows how the registers are reset to 0 after a frame has passed. This reset is triggered by the signal `data_valid_o` (second last in the figure), which is high in the clock cycle after a full frame has passed.

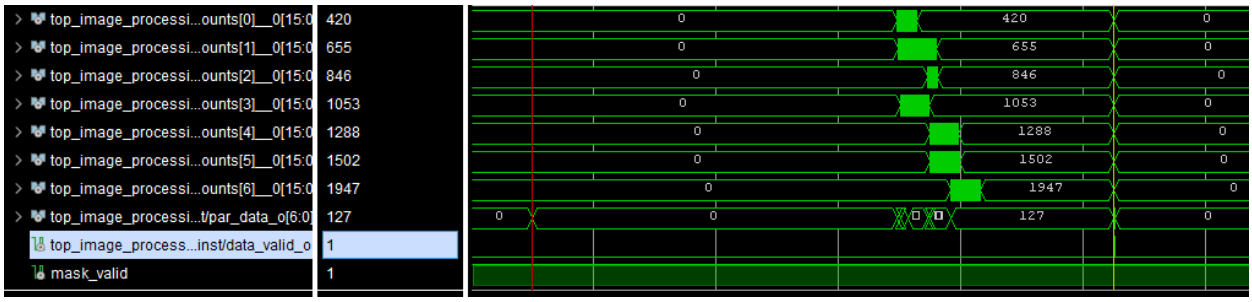


Figure 8: ILA waveforms showing the `camera_counts` registers. The values match with what is predicted by the Python script. One can also see in the right of the screenshot that the registers are reset to zero. This is triggered by the second last signal `data_valid_o`, which is high in the clock cycle after the last pixel of a frame (one clock cycle is too narrow to see in the figure).

5 Possible next steps

5.1 Integrate frame grabber and test with camera

One of the next steps would be to integrate the frame grabber module, developed by Giacomo Bisson, into the design and test it with the camera currently used for readout in the eQual experiment of the TIQI group, the Nuvu HNU128. The first thing we need to see is whether the algorithm correctly detects a new frame and starts the thresholding process. This can be done easily by making use of the ILA. Verifying that the accumulation is done correctly, is a bit more cumbersome: One needs to monitor the pixel intensity values using the ILA, export these values from Vivado to the Python script (the "Golden model") and then compare the accumulated values from the Python script with `camera_counts` from the ILA. Another option is to configure the camera to send a known test image - however not all cameras support this feature.

5.2 Integration into experimental control system

As mentioned before, there is currently no option to retrieve the results of the algorithm, i.e. the bit string of length `N_ion_max` containing the states bright (= 1) and dark (= 0), except using the ILA. As a first step towards integrating our platform into the current experimental control system, one can take the `Transmitter` module used in Schwegler's version. It implements a fast serial communication protocol for sending the ion states (see Chap. 2.1 in [6]). Similar to our new image processing scheme, it was designed to be scalable to an arbitrary number of ions, so there are no compatibility issues with the newly designed platform. Also, by using an interface that is already successfully used, the integration into the control system will need less debugging.

For determining the ideal threshold prior to the actual experiment, we need to transfer the counts and ion states to the control PC. This can be either done by adding additional AXI GPIO modules from which the processor can read the result or by storing the counts inside another dual port BRAM.

5.3 Send and receive commands for the camera

To control the camera, set parameters such as (amongst others) readout mode or exposure time, and receive answers from the camera, we can use ASCII commands which are sent via a UART interface which is integrated in Camera Link (CL): There are two dedicated LVDS pairs in the CL interface for the RX and TX signals. As mentioned in 2.2, Giacomo Bisson used the UARTlite IP core from Xilinx for this and tested the functionality with a camera [8]. I have already added the relevant IP cores and IO buffers in the block design available. There are, however, three things to be done, such that this can be used in our design:

1. Our setup uses a different camera than what Bisson tested the module with. So as a first step, one needs to verify that we can also send and receive ASCII characters with the Nuvu camera.

2. The implementation of Xilinx' UART Lite module uses a FIFO of length 16 to buffer the data it receives [12]. Because the answers from the camera can be (and usually are) longer than 16 bytes, one needs to write C code which handles this, e.g. by continuously checking whether the FIFO is non-empty and reading from it, until the character sequence "OK" is received (this marks the end of the camera's answer).

3. Currently, the C code does not support sending messages back to the PC. However, I did test (with dummy data) how one can do this in general, so I can give a quick overview on which modifications need to be done in order to make this work:

- The function which is responsible for sending data on a TCP connection is called `tcp_write`. The function signature is shown in Listing 3. The first argument `pcb` is a struct containing all the information about the current connection, `arg` is a pointer to the actual data that is to be sent, `len` is the length of the data in bytes, using `apiflags` one can change settings. `tcp_write` only enqueues the data for sending, it does not send it right away. lwIP automatically groups data together in larger groups because this is more efficient. If one wants to send the data right away, one has to call `tcp_output` following `tcp_write`.
- One should enclose the call to `tcp_write` with an if statement as shown in Listing 3. This ensures that we only enqueue data for sending if the send buffer is large enough (otherwise data might be lost).
- As described in Sec. A.1.3, the Ethernet application actually processes the data when `execute_job` is called. Currently, the only argument of this function is of type `struct w_job`. Because `tcp_write` also needs the current value of `struct tcp_pcb`, one would need to pass this down from the receive callback function to `execute_job`.
- Information on how to interact with the UART device drivers and examples are given at Xilinx' GitHub[7].

```
1 err_t tcp_write (struct tcp_pcb* pcb,
2                  const void* arg,
3                  u16_t len,
4                  u8_t apiflags
5                 )
6
7 if (tcp_sndbuf(pcb) > len) {
8     err = tcp_write(pcb, arg, len, apiflags);
9 }
```

Listing 3: Function signature of `tcp_write`

[7] https://github.com/Xilinx/embeddedsw/tree/master/XilinxProcessorIPLib/drivers/uartlite

6 Conclusion

The main purpose of this project was to adapt and test a quantum state discrimination algorithm on the new hardware platform Mars XU3 + EB1, based on real-time processing of an image from an EMCCD camera. The algorithm was designed such that it makes use of the FPGA resources more efficiently. The necessary configuration data is supplied by a custom Ethernet application, which exchanges data with a PC running a simple Python script also created during this project.

It was demonstrated that this new platform can provide the same image processing capabilities required for quantum information processing as the platform which is currently in use while using approximately 6x less hardware resources compared to the current solution deployed in the TIQI group. It was also shown that the new design can support the parallel readout of up to 100 ions while only using about 16 % of the available hardware resources. This leaves room for possible augmentations of the algorithm in the future, as well as other additional functionality. Despite extensive changes in the hardware design, no additional latency was introduced which makes this platform also suitable for experiments which include feedback based on the measured quantum states of the qubits. Also, because the Ethernet application was entirely written by us, it can be easily customized by the user. The on board processor also opens the way for CPU based image processing applications.

As of now the platform is not in its final state (i.e. it cannot be used in quantum information experiments now), but there is a clear road map on how to get there (see Sec. 5). The frame grabber developed by Bisson [8] needs to be implemented in our design and tested with the Nuvu camera. We need to interface the new platform with the current experimental control system. This also requires our setup to be able to send commands to the camera. Sec. 5.3 explained in detail which steps remain to be taken in order to achieve this.

References

[1] P. W. Shor. "Scheme for reducing decoherence in quantum computer memory". In: *Physical Review A* 52.4 (Oct. 1, 1995). Publisher: American Physical Society, R2493–R2496.

[2] J. Roffe. "Quantum Error Correction: An Introductory Guide". In: *Contemporary Physics* 60.3 (July 3, 2019), pp. 226–245. arXiv: 1907.11157.

[3] S. Haroche and J.-M. Raimond. *Exploring the Quantum: Atoms, Cavities, and Photons*. Publication Title: Exploring the Quantum. Oxford University Press, Aug. 2006.

[4] H. Häffner, C. F. Roos, and R. Blatt. "Quantum computing with trapped ions". In: *Physics Reports* 469.4 (Dec. 1, 2008), pp. 155–203.

[5] A. H. Burrell. "High fidelity readout of trapped ion qubits". PhD thesis. Oxford University, UK, 2010.

[6] N. Schwegler. "Towards Low-Latency Parallel Readout of Multiple Trapped Ions". Master thesis. Zürich: ETH Zürich, Oct. 16, 2018.

[7] A. Radhakrishnan. "Low Latency Parallel Readout of Multiple Trapped Ions Using Field Programmable Gate Array". Semester thesis. Zürich: ETH Zürich, Apr. 2021.

[8] G. Bisson. "FPGA based Fast Camera Readout and Laser Beam Profiling". Semester thesis. Zürich: ETH Zürich, June 2021.

[9] "Transmission Control Protocol Darpa Internet Program Protocol Specification". Information Sciences Institute, University of Southern California, Sept. 1981.

[10] A. Dunkels. "Design and Implementation of the lwIP TCP/IP Stack". Thesis. Swedish Institute of Computer Science, Feb. 20, 2001.

[11] Xilinx. *AXI Block RAM (BRAM) Controller v4.1: LogiCORE IP Product Guide*. May 22, 2019.

[12] Xilinx. *AXI UART Lite v2.0: LogiCORE IP Product Guide*. Apr. 5, 2017.

A Appendix

A.1 Code for Ethernet application

This section contains details about the software implementation of the Ethernet application. It presents the source code of the most important functions, as well as pseudocode to support the understanding of the working principle.

A.1.1 Starting the application

Listing 4 shows parts of the function `start_application`, which configures the application to act as a server. The error handling parts ([...]) have been omitted for better readability. The variables of type `struct tcp_pcb` contain all the relevant information about the current state of the TCP application. The function `tcp_listen_with_backlog` (line 16) tells the software to listen for incoming connections and to accept at most 1 at a time.

```
1  void start_application(void** current_job)
2  {
3    err_t err;
4    struct tcp_pcb *pcb, *lpcb;
5
6    /* Create Server PCB */
7    pcb = tcp_new_ip_type(IPADDR_TYPE_ANY);
8    [...]
9
10   err = tcp_bind(pcb, IP_ADDR_ANY, TCP_CONN_PORT);
11   [...]
12
13   /* Set connection queue limit to 1 to serve
14    * one client at a time
15    */
16   lpcb = tcp_listen_with_backlog(pcb, 1);
17   [...]
18
19   /* We pass the pointer to the current job as an argument */
20   tcp_arg(lpcb, current_job);
21
22   /* specify callback to use for incoming connections */
23   tcp_accept(lpcb, tcp_server_accept);
24
25   xil_printf("The␣application␣has␣been␣started.\r\n");
26
27   return;
```

```
28 }
```

Listing 4: Function `start_application`

The functions `tcp_arg` and `tcp_accept` serve an important purpose in terms of the callbacks mentioned in Sec. 3.1. `tcp_accept` (line 23) tells the application to call the function `tcp_server_accept` (see Listing 5) when the event "New connection accepted" happens. Every callback function (such as `tcp_server_accept`) has as its first argument `void *arg`. Using the function `tcp_arg` (line 20) we specify that `current_job` will be passed to every callback function. (Note that `current_job` is of type `void**` whereas it is passed as `void*` to the callback function. We will need to take care to correctly interpret this pointer! More details in Appendix A.1.2)

As an example, consider Listing 5. This function is called whenever a new connection is established. Its main purpose is to specify new callback functions (e.g. in line 14: when data is received on **this** connection, call `tcp_recv_perf_traffic`).

```
1  static err_t tcp_server_accept(void *arg, struct tcp_pcb *newpcb,
2      err_t err)
3  {
4    if ((err != ERR_OK) || (newpcb == NULL)) {
5      return ERR_VAL;
6    }
7    /* Save connected client PCB */
8    c_pcb = newpcb;
9
10   print_tcp_conn_stats();
11
12   /* setup callback functions for the current connection */
13   tcp_arg(c_pcb, arg);
14   tcp_recv(c_pcb, tcp_recv_perf_traffic);
15   tcp_err(c_pcb, tcp_server_err);
16
17   return ERR_OK;
18 }
```

Listing 5: Accept callback function

A.1.2 Code for receive callback function

To access the same instance of `struct w_job` across multiple calls of `tcp_recv_perf_traffic`, we use the variable `arg` which is passed to every callback function and has to be interpreted as `void**`. The reason for this is that the variable of type `struct w_job` will only be dynamically allocated from within `tcp_recv_perf_traffic`. `malloc` will return a pointer to the new instance of `struct w_job` and we make `arg` point to this pointer. Listing 6 shows how this is done in the function `process_pbuf`, which is called from within `tcp_recv_perf_traffic`. We first have to cast arg to type `void**` and then check whether `*arg_point` is a NULL pointer. If so, we have to create a `struct w_job`, otherwise we interpret `*arg_point` as a pointer to `struct w_job`.

```
1  err_t process_pbuf(void* arg, u16* payload_len, char** payload_point){
2    [...]
3    void** arg_point = (void**)arg;
4    struct w_job* current_job;
5
6
7    if(*arg_point == NULL) {
8        //This means there exists no struct w_job yet
9        // -> beginning of a new message
10       *arg_point = malloc(sizeof(struct w_job));
11         [...]
12
13       current_job = *((struct w_job**)arg_point);
14       [...] //actual copying of the data from pbuf to w_job
15
16   } else {
17       //parts of the message have been read in previous calls of
18       //tcp_recv_perf_traffic
19       current_job = *(struct w_job**)arg_point;
20       [...] //actual copying of the data from pbuf to w_job
21   }
```

Listing 6: Parts of function `process_pbuf`

To illustrate the idea behind the algorithm of `tcp_recv_perf_traffic` consider Pseudocode 1. The small right arrow means we access a variable inside a struct (as in C syntax). The names of the variables in the pseudo code are the same as in the actual C code. `payload_point` points to the **first un-**

read byte in the current pbuf, `payload_len` indicates how many bytes **have not been** read yet in the current pbuf. The first if statement distinguishes the case where the pbuf chain is longer than one or not. Then we enter a while loop, which repeatedly executes a function to buffer the contents of the payload until everything has been buffered (i.e. `payload_len` is zero).

Pseudocode 1 tcp_rcv_perf_traffic

```
Input: void *arg, struct pbuf *p
  if p→next ≠ NULL then
    current_pbuf ⟵ p
    while current_pbuf ≠ NULL do
      payload_point ⟵ current_pbuf→payload
      payload_len ⟵ current_pbuf→len
      while payload_len ≠ 0 do
        execute process_pbuf(arg, adr(payload_len), adr(payload_point))
      end while
      current_pbuf ⟵ current_pbuf→next
    end while
  else {p→next = NULL}
    payload_point ⟵ p→payload
    payload_len ⟵ p→len
    while payload_len ≠ 0 do
      execute process_pbuf(arg, adr(payload_len), adr(payload_point))
    end while
  end if
```

In Pseudocode 2 the overall working principle of `process_pbuf` is explained (heavily simplified: Most variables are member variables of `current_job`, which is of type `struct w_job`. I omitted writing "`current_job->`" for easier readability.) The reason why this function is so long and convoluted is because we don't know how many bytes will be presented to us in the pbuf. It may well be (however rather unlikely) that, for example, not all 11 bytes of the header are contained within on pbuf. This is why we check in line 5 whether `payload_len` is smaller then the header size. If not, we can interpret the bytes of the header in line 15 and start reading the actual payload in line 16. Then, we check whether we have already received the whole message (i.e. compare `recv_bytes` with what we expect, `num_bytes`). If this is true, we call the function `execute_job`, which actually performs the operation that we intended to do with the message we sent. More on this in the next section.

A.1.3 Levels

The first byte of the header carries the information on how the payload of the message should be interpreted. We call it the `level` variable, which is a part of `struct w_job` (see Listing 1). The function `execute_job` essentially consists of a switch statement, which depending on `level` executes a function specific to the required functionality. For example, if `level` is

Pseudocode 2 process_pbuf

```
Input: void* arg, u16* payload_len, char** payload_point
    if arg is a NULL pointer (see Listing 6) then
        current_job ⟵ allocate memory for struct w_job
        header_pos ⟵ 0
        recv_bytes ⟵ 0
 5:     if payload_len < HEADER_LEN (i.e. 11) then
            copy payload_len bytes from payload_point to header
            header_pos ⟵ payload_len
            payload_len ⟵ 0
        else
10:         copy HEADER_LEN bytes from payload_point to header
            header_pos ⟵ HEADER_LEN
            payload_len ⟵ payload_len - HEADER_LEN
            payload_point ⟵ payload_point + HEADER_LEN

15:         resolve_header(current_job) {Filling the variables from Listing 1}
            read_payload(current_job, payload_len, payload_point)
        end if
        if num_bytes = recv_bytes then
            execute_job(current_job)
20:     end if
    else
        current_job ⟵ arg
        if header_pos ≠ HEADER_LEN then
            if payload_len < HEADER_LEN - header_pos then
25:             copy payload_len bytes from payload_point to (header + header_pos)
                header_pos ⟵ header_pos + payload_len
                payload_len = 0
            else
                copy HEADER_LEN - header_pos bytes from payload_point to (header +
                header_pos)
30:             payload_len ⟵ payload_len - (HEADER_LEN - header_pos)
                payload_point ⟵ payload_point + HEADER_LEN - header_pos

                resolve_header(current_job)
                read_payload(current_job, payload_len, payload_point)
35:         end if
        else
            read_payload(current_job, payload_len, payload_point)

            if num_bytes = recv_bytes then
40:             execute_job(current_job)
            end if
        end if
    end if
```

equal to 1 (meaning it is ROI/mask data), the function `execute_mask` is executed (defined in `levels.c`). This function interprets six consecutive bytes as three 16 bit numbers (x, y, ion) and writes them to the BRAM as will be explained in Sec. 4.2.1.

To add additional functionality to the application, the user has to follow these steps:

1. Add a new level to the `level_t` enum type in `w_job.h`
2. Write a function in `levels.c` which interprets the raw bytes from the payload and processes them (similar to `execute_mask`)
3. Add a new case to the switch statement in `execute_job` which calls the new function if level happens to have that value.

A.1.4 Writing mask data to BRAM

Listing 7 shows a part of the function `execute_mask` that writes the mask calibration data to the BRAM. The code expects that the mask data is sending a sequence [x, y, ion number, x, y, ion number, x, ...] with $3 \times$ number of ions $\times$ size of ROI numbers.

```
  //j is a pointer of type struct w_job (see Listing 3)
char* point = j->data_point; //points to the first byte in the payload

u16 num[4];
u32 offset = 0;

  //j->write_point points to the location behind the last byte
while (point != j->write_point) {

  //the masks come as [x, y, ion_no, x, y, ion_no, ...]
  //where ion_no, x and y are all 16-bit unsigned integers
  for (int i = 0; i < 3; i++) {
    //interpret 2 consecutive bytes as 16-bit unsigned integer
    num[i] = (u16)*point | (u16)*(point + 1) << 8;
    point = point + 2;
  }
  num[3] = 0;

  u32 num1 = *(u32*)num;
  u32 num2 = *(u32*)(num + 2);
  //bram.Config.MemBaseAddress is the AXI base address of the
  //AXI BRAM Controller
```

```
23      XBram_Out32(bram.Config.MemBaseAddress + offset, num1);
24      XBram_Out32(bram.Config.MemBaseAddress + offset + 4, num2);
25
26      offset = offset + 8;
27    }
```

Listing 7: Part of the function `execute_mask` which writes the mask triplets (x, y, ion) to the BRAM

A.2 Recreate Vivado project from GitLab

The Vivado project can be found in the TIQI GitLab within the "Parallel Readout Vivado" repository. The directory structure looks as follows:

```
/
├── python
├── scripts
└── src
    ├── bd
    │   └── Mars_XU3
    ├── constr
    ├── hdl
    │   └── tb
    └── ip
```

The `scripts` directory contains a tcl script, which recreates the Vivado project based on the files in the directories above. I decided to also include all files associated to the Block design and the IP cores in the repository (in contrast to only including a script which recreates the block design), following Xilinx' recommendation for using Vivado with revision control systems.[8] The advantage of this approach is that using this approach the project does not need to be recreated with the same version of Vivado which it was created with.

The `python` repository contains a Python script ("Golden model") which simulates the behavior of the thresholding module (see Sec. 4.5).

A.3 Recreate Vitis project from GitLab

The Vitis project can be found in the TIQI GitLab within the "Parallel Readout Vitis" repository. It only contains the "Application Project" (which includes the C source files for the user, but no hardware information, device

[8] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug892-vivado-design-flows-overview.pdf

drivers or build files). So to recreate the entire Vitis workspace, one has to follow these steps.

(The application project was created with Vitis 2020.1, but I tested the following procedure with Vitis 2020.2)

1. Open Vitis and create a new Platform Project (I called it MARS_EB1 in this example). Select the `.xsa` file that was created by Vivado when exporting the hardware platform (use options "Fixed" and "Include bitstream, when exporting the hardware).
2. Keep the default options for "Operating system", "Processor", "Architecture" and "Boot components".
3. Clone the GitLab repository to you local machine (**do not** clone it into the Vitis workspace, it is easier to keep the source files and the workspace seperate)
4. Back in Vitis select "File" -> "Import...". The "Import Projects" will pop up. Select "Import projects from Git". On the next page select "Existing local repository". Add the path to the git repository and hit next. Make sure that "Import existing Eclipse projects" is selected. If everything is done right, the last page will look as in Fig. 9a. Hit "Finish".

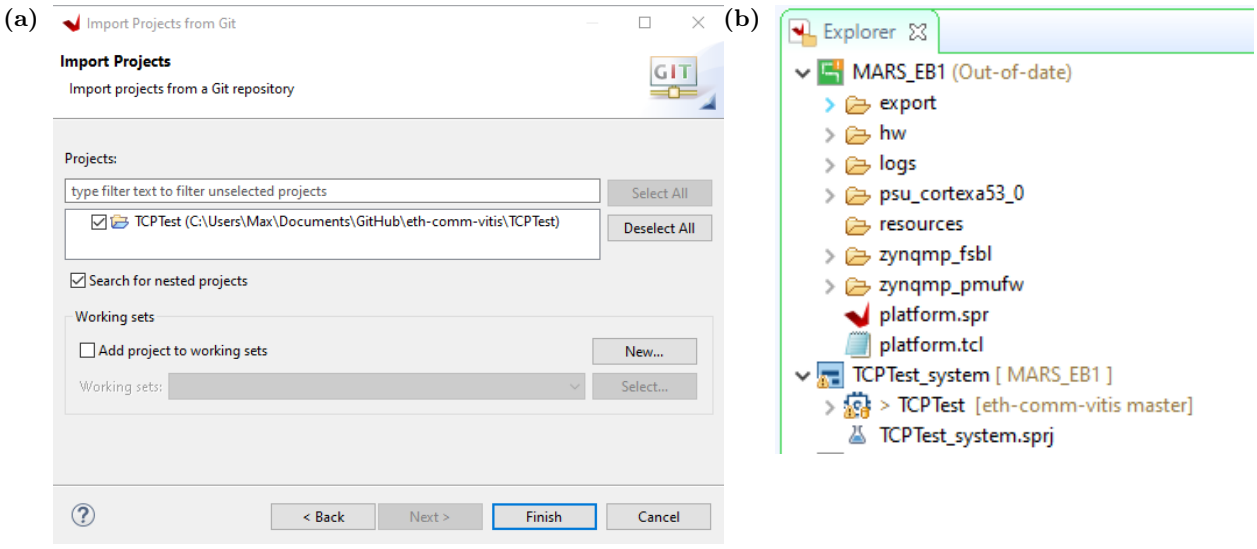


Figure 9: (a) Importing the git project into the current Vitis workspace. (b) How the Vitis explorer should look like after steps 1-5 are completed.

5. There might be an error message "Platform '...' could not be resolved to a valid platform. [...]" Select "Change referred platform" and ignore

the following "Failed to read the system project settings for 'TCPTest'" message.

6. In the explorer (see Fig. 9b) on the left, double click on "TCPTest_system.sprj". Make sure that the selected "Platform" is the Platform Project that we have created in step 1. If not, click on the three dots and select it.

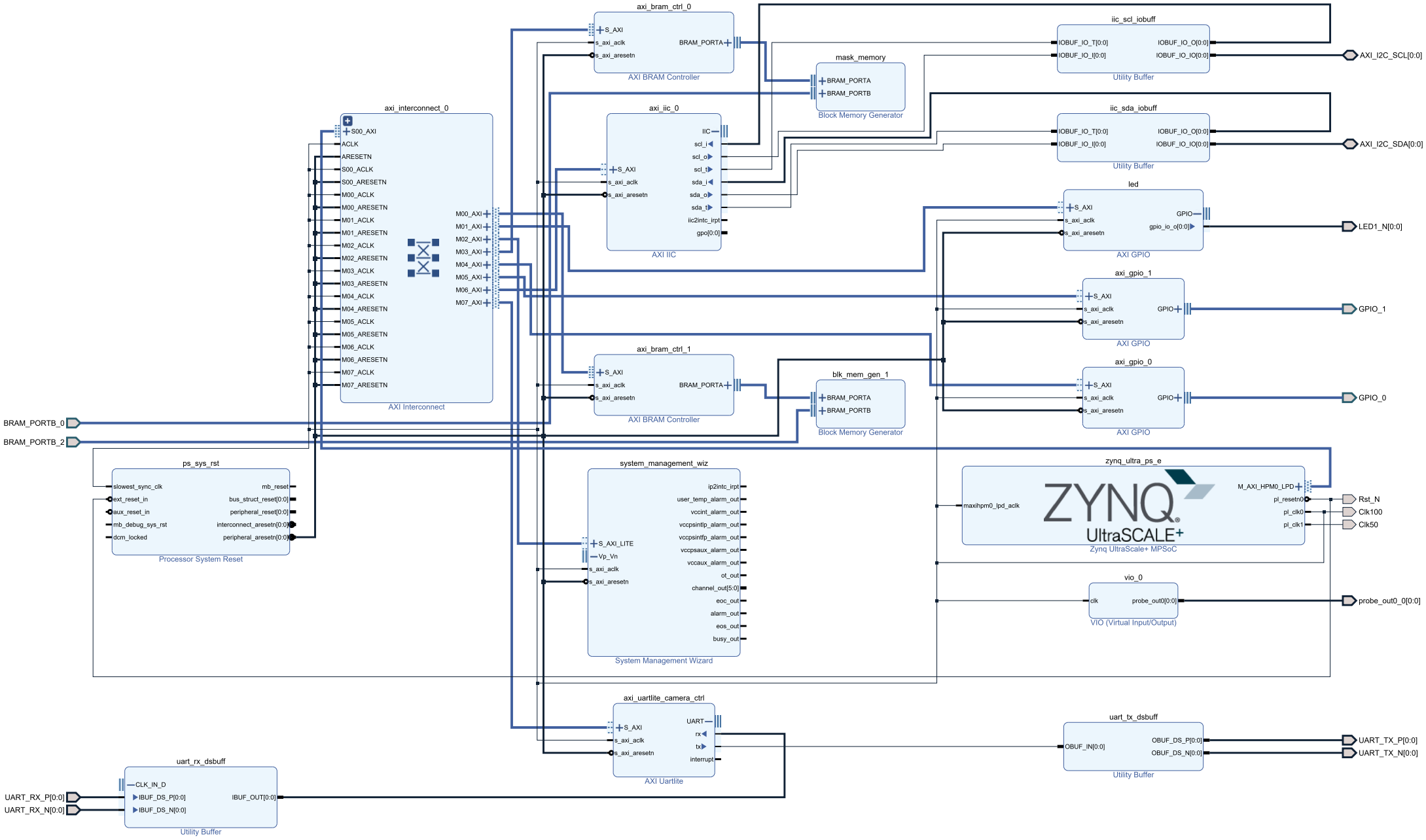
7. Double click on "platform.spr" in the explorer and then on "Board Support Package" beneath "standalone on psu_cortexa53_0". Click on "Modify BSP Settings...", and select "lwip211". In the same window on the left, go to lwip211, and modify the following settings under "temac_adapter_options": emac_number = 3 and phy_link_speed = 1000 Mbps.

8. If you use version 1.2 of lwip, there should be a file called "xemacpsif_physpeed.c". Go to `https://github.com/enclustra/GigabitEthernetAppNote` and replace this file with the file provided in this GitHub repository.

9. Build the project. If the building finishes without any errors, the application is ready to run on the board.

All changes to the C source files in the "TCPTest" are now tracked by git but not the build files (which are easily recreated once you have all the source files, the building process takes max. 1 min).

A.4 Vivado Block design


axi_bram_ctrl_0
S_AXI
s_axi_aclk
s_axi_aresetn
BRAM_PORTA
AXI BRAM Controller
mask_memory
BRAM_PORTA
BRAM_PORTB
Block Memory Generator
iic_scl_iobuff
IOBUF_IO_T[0:0]
IOBUF_IO_I[0:0]
IOBUF_IO_O[0:0]
IOBUF_IO_IO[0:0]
Utility Buffer
AXI_I2C_SCL[0:0]
axi_interconnect_0
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M01_ACLK
M01_ARESETN
M02_ACLK
M02_ARESETN
M03_ACLK
M03_ARESETN
M04_ACLK
M04_ARESETN
M05_ACLK
M05_ARESETN
M06_ACLK
M06_ARESETN
M07_ACLK
M07_ARESETN
M00_AXI
M01_AXI
M02_AXI
M03_AXI
M04_AXI
M05_AXI
M06_AXI
M07_AXI
AXI Interconnect
axi_iic_0
IIC
scl_i
scl_o
scl_t
sda_i
sda_o
sda_t
iic2intc_irpt
gpo[0:0]
AXI IIC
iic_sda_iobuff
AXI_I2C_SDA[0:0]
led
GPIO
gpio_io_o[0:0]
LED1_N[0:0]
AXI GPIO
axi_gpio_1
GPIO_1
axi_bram_ctrl_1
blk_mem_gen_1
axi_gpio_0
GPIO_0
BRAM_PORTB_0
BRAM_PORTB_2
ps_sys_rst
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
system_management_wiz
S_AXI_LITE
Vp_Vn
ip2intc_irpt
user_temp_alarm_out
vccint_alarm_out
vccpsintlp_alarm_out
vccpsintfp_alarm_out
vccpsaux_alarm_out
vccaux_alarm_out
ot_out
channel_out[5:0]
eoc_out
alarm_out
eos_out
busy_out
System Management Wizard
zynq_ultra_ps_e
maxihpm0_lpd_aclk
M_AXI_HPM0_LPD
pl_resetn0
pl_clk0
pl_clk1
Rst_N
Clk100
Clk50
Zynq UltraScale+ MPSoC
vio_0
clk
probe_out0[0:0]
probe_out0_0[0:0]
VIO (Virtual Input/Output)
axi_uartlite_camera_ctrl
UART
rx
tx
interrupt
AXI Uartlite
uart_tx_dsbuff
OBUF_IN[0:0]
OBUF_DS_P[0:0]
OBUF_DS_N[0:0]
UART_TX_P[0:0]
UART_TX_N[0:0]
uart_rx_dsbuff
CLK_IN_D
IBUF_DS_P[0:0]
IBUF_DS_N[0:0]
IBUF_OUT[0:0]
UART_RX_P[0:0]
UART_RX_N[0:0]

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Low Latency Implementation of a Resource Utilization Optimized State Discrimination Algorithm

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Glantschnig	Max

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 01.07.2021	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.