Low Latency Parallel Readout of Multiple Trapped Ions Using Field Programmable Gate Array

Submitted by

Adithyan Radhakrishnan

Master Student

Quantum Engineering MSc, ETH Zurich

A report outlining the work carried out during Oct 2020-Feb 2021 as a part of a semester project

Supervised by:

Roland Matt [1]

Dr Abdulkadir Akin [2 3]

Prof Dr Jonathan Home [1]

April 2021

[1] Trapped Ion Quantum Information Lab, ETH Zurich
[2] Quantum Device Lab, ETH Zurich
[3] Quantum Engineering Center, ETH Zurich

Contents

List of Figures

Chapter 1

Introduction and Motivation

In Quantum Information Processing, qubit readout is an essential step for obtaining the state of the qubit for classical post-processing and Quantum Error Correction (QEC) protocols.Useful QEC requires correcting the detection errors (caused due to reading out some part of the qubit register) on timescales which are much shorter than typical physical qubit coherence times (tens of ms) [1]. This demands low latency for readout while keeping the readout fidelity high. In ion-trap applications, Photo-Multiplier Tubes (PMTs) are a good candidate for achieving low latency whilst keeping the fidelity high for single ion systems [2]. However when it comes to multi-ion systems, using a PMT array for parallel readout results in limited flexibility as the associated optics are specific to the setup. There has also been considerable cross-talk observed between PMTs within an array [3] [4]. Another option is using a an Electron Multiplying Charge-Coupled Device (EMCCD). High fidelity readout using an EMCCD array has been achieved before albeit with accompanying high latency [5]. In this case, the image processing had been done on a desktop computer (PC) which led to a limited performance due to the hardware limitations of a PC and a software implementation which was not optimized. If the post processing could be done on a platform which offers more resources for parallel processing like a Field Programmable Gate Array (FPGA) with an optimised software implementation, one could aim for low latency high fidelity parallel readout of multiple ions. Nick Schwegler's work for his Master Thesis explores exactly this where he concluded that an EMCCD based readout setup can achieve parallel readout of more than 50 ions in a linear chain with infidelity less than 10^{-4} in 225 us using an FPGA [6]. For this he interfaces an EMCCD camera (NuVu Hnu128) with an FPGA (VC707 board from Xilinx) using a commercially available frame grabber (FMC422, Abaco Systems) and verifies a state discrimination algorithm using this set-up. My work during for this semester project is built on this work where I improve the state discrimination algorithm and attempt to interface the camera with a different FPGA and Frame Grabber system

((Mars EB1+XU3 from Enclustra)) for reasons that are outlined in the second chapter. The outline of my report is as follows: In the second chapter I give an overview of the readout system followed by the reasons to switch to a new FPGA and the frame grabber system. The third chapter talks about the new image processing algorithm implemented on the new FPGA system which reduces the comparator resource usage such that it does not scale with the number of ions. The fourth chapter talks about setting up the Ethernet Communication between the FPGA and a PC. The fifth and the last chapter talks about the algorithm for detecting the ions and retrieving the regions of interest (ROI) from an overexposed ion array image. These ROI will be crucial for the FPGA image processing algorithm. Finally I conclude this report with a chapter summarizing my work and discussing the outlook and the future aspects of the project.

Chapter 2

Readout Scheme

In the set-up where this readout scheme is implemented, the qubit is encoded in a trapped Ca^+ ion using a linear Paul trap. As shown in fig 2.1, the ground state $|g\rangle$ is the natural ground state S of the ion and the excited state $|e\rangle$ is the dark D state [2]. The reason $|e\rangle$ is called the dark state is because when the decay time from $|e\rangle$ to $|g\rangle$ (or D to S) is almost 1s in a linear Paul trap [7]. This is in stark contrast to the P to S transition (given by a resonant 397 nm transition) where the electron spontaneously decays in the order of ns.

Leveraging this the following readout scheme is performed:

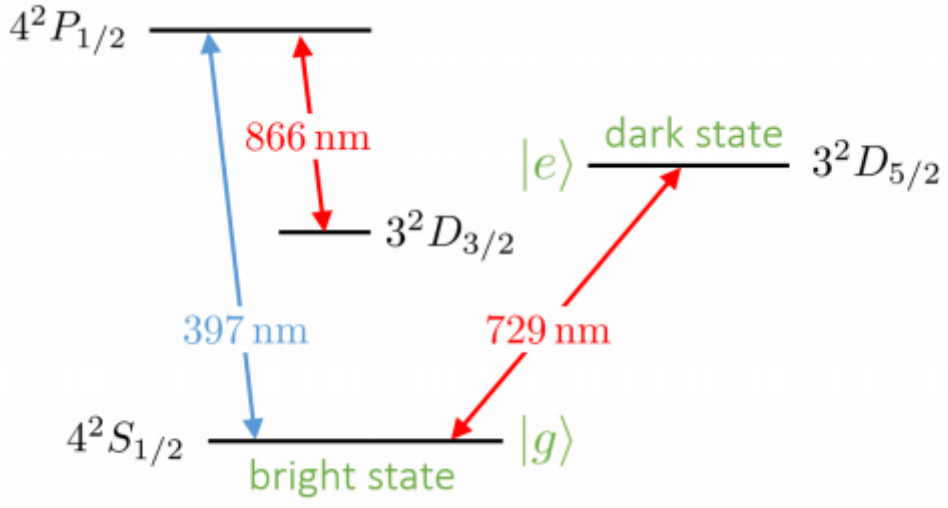


FIGURE 2.1: Energy diagram of the qubit readout scheme. The 397nm is continuously driven to excite the ion from S to P and back to S emitting photons in the process. Whereas the 866nm Red laser is continuously driven to pump any leakage from P $\rightarrow$ D back to P

1. The 397 nm (blue) and 866 (red) lasers are turned on for the duration of the readout.

2. If the qubit is in $|g\rangle$ the 397 nm laser would excite the ion to P and back to S at the end of each Rabi cycle giving out a 397nm photon.

3. Whereas if the qubit is in D, this laser will have no effect and therefore no photon will be emitted.

4. Hence, in a given time window if the number of photons are above a certain threshold, we can conclude that our qubit underwent multiple Rabi cycles and was therefore in a bright ($|g\rangle$) state and a dark state $|e\rangle$ otherwise.

2.1 Readout Set-Up

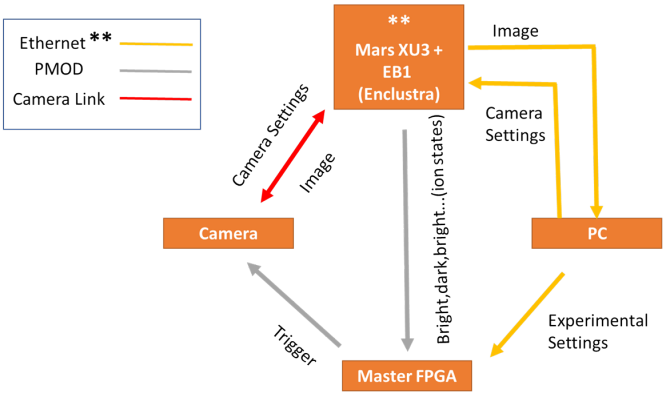


FIGURE 2.2: The integrated readout set-up containing the EMCCD camera, frame grabber, the slave (Mars EB1 + XU3) and the master FPGA (Zynq7 XC72020). The interface between these components are mentioned on the legend box on the top left.

The qubit readout is based on the number of photons detected per unit detection window (< 10 ms). Therefore we need a camera which records frames in this time window and send it to the post processing module which would be responsible for computing the number of photons in each frame. This is achieved as shown in Fig 2.2. The Camera **(NuVu Hnu128)** attached to the frame grabber (**SISO Marathon**) is responsible for recording and sending the frames and the FPGA (**Mars EB1 + XU3**) is responsible for analysing the frame. The components marked in double stars (**) constitute my work for this semester project and is detailed in this report.

2.2 Motivation to Migrate To A New Device and a New Algorithm

The initial implementation of this set-up as described in Nick Schwegler's thesis [6] contains the Virtex VC707 evaluation kit from Xilinx for the image analysis using FPGA and the FMC422 board as the frame grabber. The total cost of this set-up comes out to be around \$3500 + 2700 = \$6200. The replacement that we propose, uses the Mars EB1 board [8] with the Mars XU3 chip [9] in place of the VC707 and the inbuilt Camera Link interface for the frame grabber. The total cost for this set-up comes out to be CHF 760 which is around **8 times cheaper than the VC707 + FMC422 set-up**. Since December 2020, the FMC422 has been discontinued and has entered restricted production. This restricts scaling up the readout set up for multiple experiments in the future. Moreover the Mars XU3 consists of a Zynq Ultrascale + MPSoC chip (XCZU3EG-2SBVA484I) which combines parallel processing of an FPGA with an ARM microprocessor providing additional flexibility towards communication with a PC and also opens doors for software based on-chip image processing applications possibilities in the future. Besides the cost and the hardware utility of the board, the previous implementation is designed to support at most 60 ions utilising all of the available hardware resources. However the same scale of architecture is not available in our low cost FPGA Mars XU3, therefore we implement an image processing circuit which utilizes less hardware resources. The previous implementation lacked the automation of finding the masks/regions of interest (ROI) and required the user to manually input the mask coordinates. During the course of the semester project, we are now able to automate this process upto a certain accuracy (discussed in detail in the fifth chapter).

Chapter 3

Image Processing Using FPGA

As mentioned in the preceding chapter, the FPGA is responsible for analysing the frames to output whether a certain ion is in state $|g\rangle$ or $|e\rangle$. If the cumulative pixel intensity for the corresponding ROI over a number of frames is greater than the threshold, then the ion belonging to that ROI is in $|g\rangle$; and $|e\rangle$ otherwise. The frames are output line-by-line (or row-by-row) pixel-after-pixel (serially) as shown in Fig 3.3. These serialized data is then taken up by the FPGA and deserialized in order to feed it to the state discrimination algorithm which compares the cumulative pixel intensity with the threshold (Fig 3.1) .

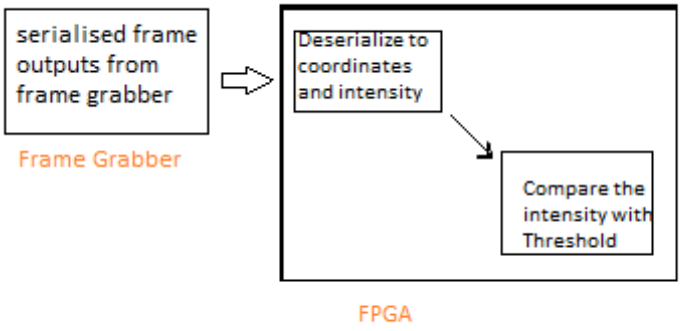


FIGURE 3.1: Flowchart summarising the Frame Grabber + FPGA image processing

3.1 Previously Implemented State Discrimination Algorithm

The initial readout implemented by Nick Schwegler for his master thesis at TIQI uses the following algorithm:

while reset!=1:

- Convert the incoming data from the frame grabber to useful values namely the coordinates (x,y) and the intensity of the pixel.
- Compare the pair coordinates (x,y) with **all the stored mask coordinate pairs.**
- If the pairs match, then update the corresponding mask index of the counter array with the incoming pixel intensity.

finally:

- Compare the cumulative intensity for a certain time window with the threshold for each ion. And if it is above the threshold, the corresponding ion is declared to be in state |1>.

This algorithm utilizes the parallel processing capabilities of an FPGA and hence does the readout on all the ions in the array at once. However, it does so by paying the overhead resource cost at step 2 where it uses comparators to compare the incoming pixel with the stored mask pixels. For a setup which has an array of Ni ions, and each ion having a mask with M pixels, with each of the two pixel coordinates (x,y) being stored as a 16bit number this implementation uses a total number of comparators (Nc): Nc = Ni x M x 16 x 2 For a set up consisting of 10 ions, with each ion having 25 pixels per mask, this already takes up: Nc = 8000 comparators. As the number of ions scale up to 100, the number of comparators required scales up almost a 100000. The available look-up table (LUT) resources in the FPGA is limited to 71000 (Pg 4 of [10]) and each comparator would require several LUTs for its implementation making the previous algorithm not scalable for an array of ions in the hundreds.

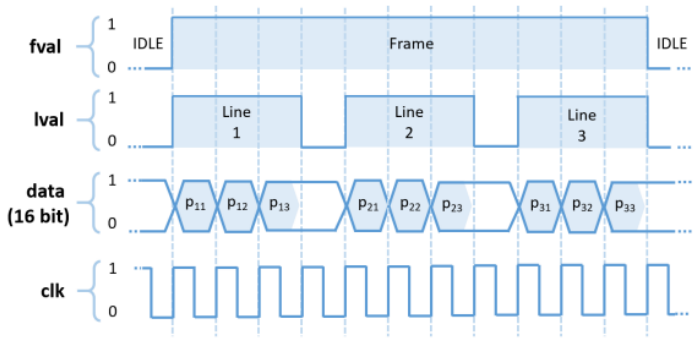


FIGURE 3.3: Timing diagram of the frame grabber output. Here, **fval** switches from 0 to 1 at the start of a frame and remains 1 until it switches back to 0 at the end. **lval** is 1 as long as a particular line is read before switching back to 0 at the end of a line. The 16 bit data consists of the pixel intensity value where p_{xy} denotes pixel value at **line y** and **column x**.

3.2 New Algorithm

The new algorithm that I implemented during the course of this semester project is outlined as follows (**NB**: The variable names in the following pseudo code are NOT the variable names used in the actual code):

- Arrange the mask coordinates in a 1D array of 3 parameter tuple - x coordinate, y coordinate, ion number (the ion which the mask corresponds to). These tuples are arranged in a numerical ordering of y, x coordinate (lower y corresponds to lower index in the array, lower x corresponds to lower index in the array and y takes precedence over x) and store it in **mask_arr**. Set the variable **counter** to 0.

while reset!=1:

- Append **counter** by 1.
- Load the tuple from **mask_arr [counter - 1]** and store it in **mask_tuple.**
- Convert the incoming data from the frame grabber to useful values namely the coordinates (x,y) and the intensity of the pixel.
- Compare the pair coordinates (x,y) with the x,y of **mask_tuple.**
- If the pairs match, then update the corresponding mask index of the counter array with the incoming pixel intensity.

finally:

- Compare the cumulative intensity for a certain time window with the threshold for each ion. And if it is above the threshold, the corresponding ion is declared to be in state |1>.

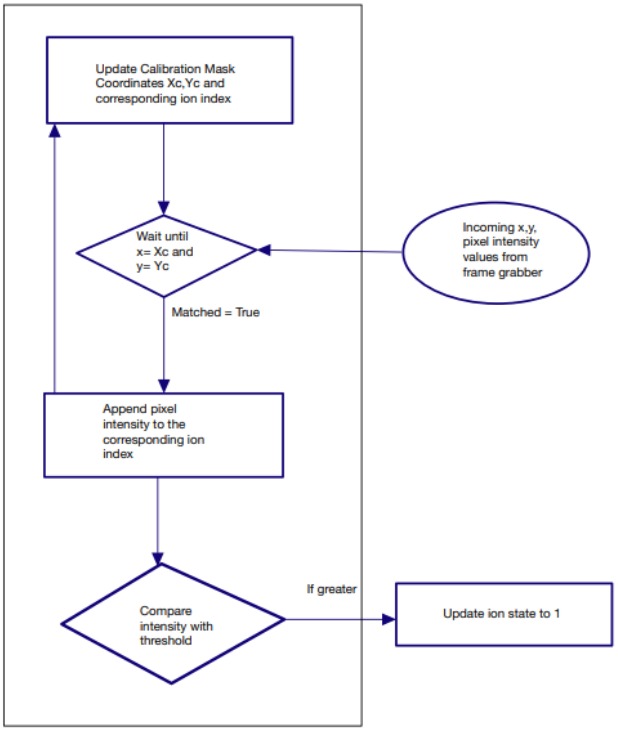


FIGURE 3.5: Simple flowchart of the working of the new algorithm. Fig 2.2 details this flowchart and has a side by side comparison of the new and the old algorithms.

This implementation reduces the number of comparator usages to just two and it does not scale with the number of ions. Instead it uses up the internal memory in order to store the mask coordinates. This is acceptable since the internal memory (7.6 Mb) [10] is far more than the number of comparators and in addition, we could use the inbuilt DDR memory along with an option of extending it with an external SD card.

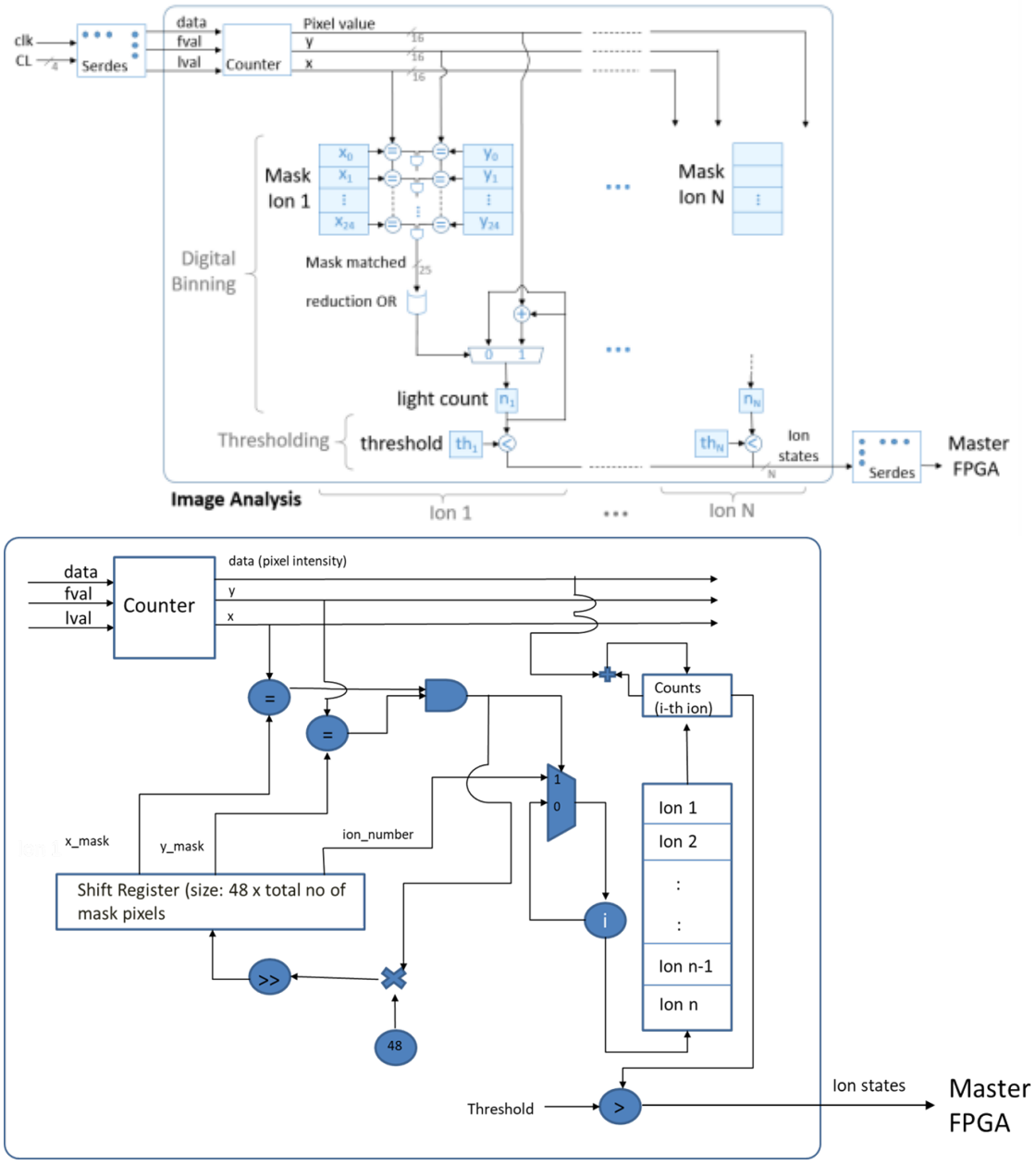


FIGURE 3.6: (from top) Comparison of algorithms of previous implementation (a) and the new implementation (b)

3.2.1 Simulation Results

Fig 2.3 b and c contain the simulation results acquired using a testbench which emulates the frame grabber (provides lval, fval and pixdata; refer to Fig 2.2 for more information.) Fig 2.3a contains the frame which was fed into the testbench which was an overexposed image which was used to generate the masks (as per chapter 4). The final result that we are interested in is contained in the variable par_data_o which is an array of ion state data with the index corresponding to the ion number and as per the input image [1], we

[1]Input image has been taken from Google Images.

expect all the states to be 1 as the input image was overexposed with all bright states. And as one can see from Fig 2.3 b, they are all at state 1 at the end of the simulation. Fig 2.3 a shows a screenshot after a few cycles where some of the ions are still in state 0. To check the validity of these results, I first ran the algorithm on a PC using python and compared the results with that of the simulation results. The values for camera_counts which correspond to the total intensity in the mask of each ion, match exactly with the Vivado simulation results.

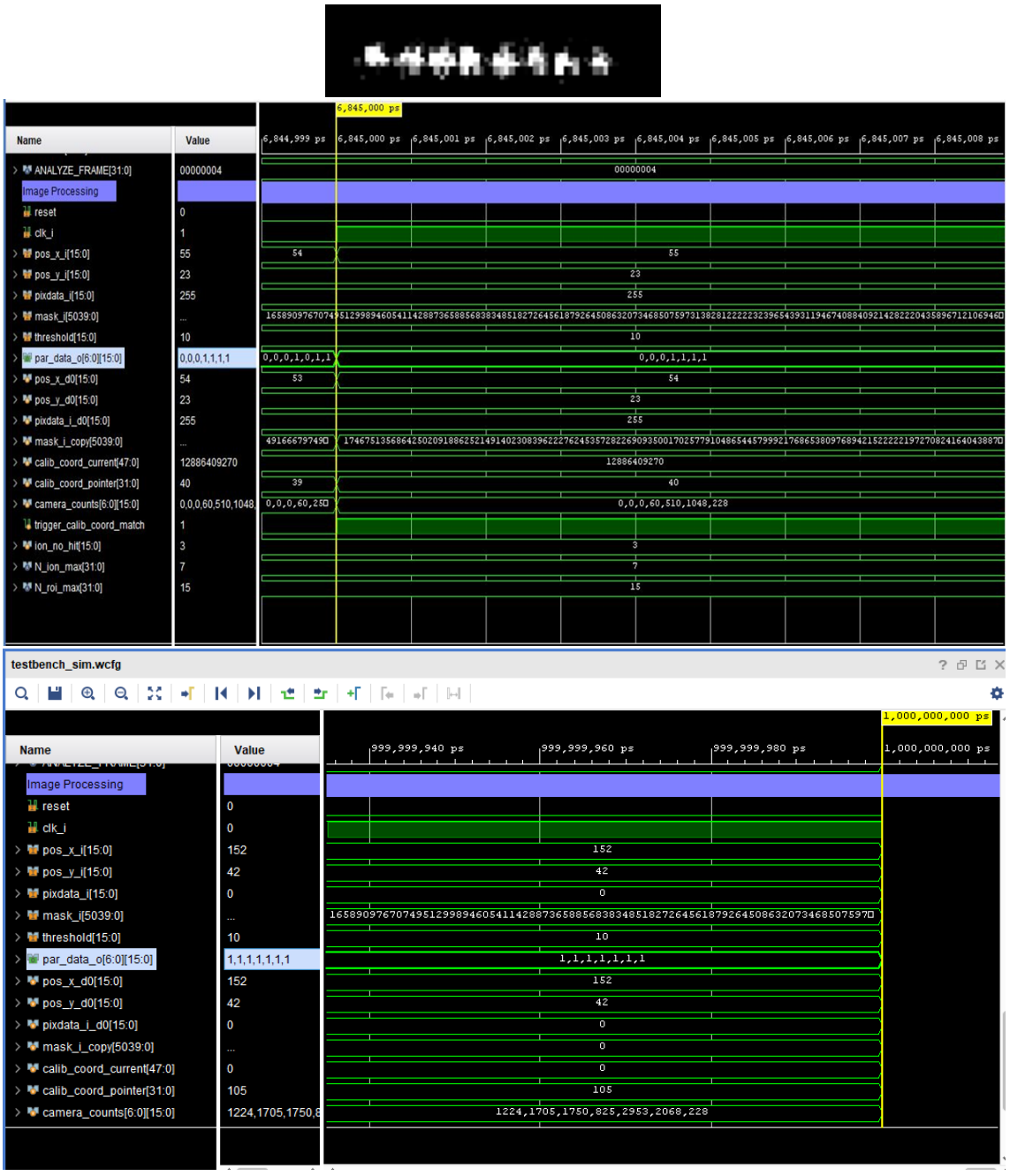


FIGURE 3.7: (from top) a) Image fed into the simulation b) Screenshot of the simulation at around 1 percent of the total time c) Screenshot of the simulation at the end of 1 ms.

The following table elaborates on the variables shown in the simulation

Variable Name	Usage/Function
reset	Resets the process, sets all the registers to 0
clk_in	Input clock to the verilog module
pos_x_i, pos_y_i	16 bit x, y coordinates of the incoming pixel
mask_i	1D array of the stored mask coordinates
threshold	threshold for comparing with the input pixel intensity
par_data_o	List of ion output states. Index corresponds to the ion number. 0 = dark state (excited), 1 = bright state (gnd)
pos_x_d0, pos_y_d0	16 bit x, y coordinates of the incoming pixel delayed by 1 clock cycle
pixdata_i_d0	Incoming pixel intensity delayed by 1clock cycle
mask_i_copy	Copy of mask_i
calib_coord_current	Current calibration mask coordinates which would be compared with incoming coordinates (pos_x_d0, pos_y_d0)
calib_coord_pointer	Counter to keep track of which set of mask coordinates are being loaded from mask_i
camera_counts	Total camera counts corresponding to each ion. Index corresponds to ion number
trigger_calib_coord_match	Wire which triggers when the incoming pixel pos_x_d0, pos_y_d0 matches with calib_coord_current
ion_no_hit	Corresponds to which ion's data was updated most recently
N_ion_max	Total number of ions in the array
N_roi_max	Total number of pixels in each ROI

FIGURE 3.8: Table summarizing all the variables in the simulation results shown in Fig 3.7

Chapter 4

Ethernet Communication Using FPGA

4.1 Introduction

As detailed in the third chapter, image masks are an integral component of the algorithm. The masks are generated on a PC and are supposed to be sent to the FPGA to be stored in its local memory before the experiment. For this purpose, communication between PC and FPGA is rather important and is carried out via an ethernet connection. In this chapter, I go over the important settings to be done on the FPGA side in order to establish an ethernet communication followed by a simple demonstration of a Transmission Control Protocol (TCP) based data transfer over the ethernet between the PC and FPGA. I use Lightweight Internet Protocol (LWIP) to carry out the TCP/IP implementation [11].

4.2 Setting Up An Ethernet Connection

An ethernet interface consists of two components- the **PHY** (short for physical layer) and the **Media Access Controller (MAC)**. The MAC is the device which one would like to communicate over the ethernet and PHY enables the interface between our MAC and the ethernet cable. PHY does this commonly through the Gigabit Media Independent Interface **(GMII)** and its more efficient equivalent the Reduced Gigabit Media Independent Interface **(RGMII)**. The MARS XU3 module has an inbuilt PHY chip which is the **Micrel KSZ9031RNX** chip. And our MAC can be one of the four Gigabit Ethernet Modules **(GEM0...GEM3)** which are inbuilt in the Xilinx FPGA

chip (XCZU3EG-2SBVA484I). The **GEM3** inside MARS XU3 are internally connected (hardwired) via the MIO pins (64..75) [8] to the Micrel PHY chip. This PHY connection allows the GEM3 to send data over the ethernet cable to the desired client/server.

4.2.1 Enabling The MAC (GEM3)

To enable the Gigabit Ethernet Module (GEM3) of the Zynq ultrascale MPSoC chip, carry out the following steps:

1. Open the block design in Vivado as per section 3.3-3.4 of the Mars XU3-EB1 reference design manual [12].
2. Double Click on the Zynq Ultrascale+ component of the block design.
3. Click on I/O configuration and type 'GEM' in the search bar.
4. Under the list of high speed I/O peripheral section tick 'GEM3'.

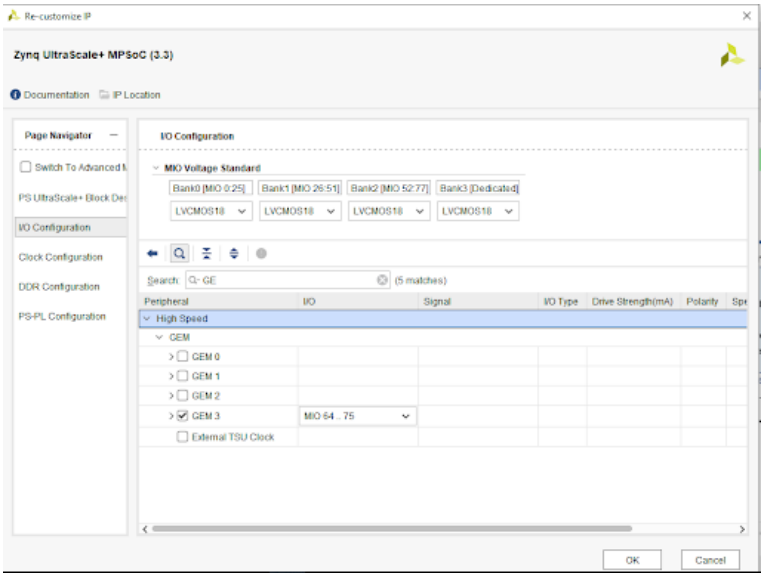


FIGURE 4.1: Enabling the MAC

4.2.2 Setting Up the PHY

The PHY layer is managed by the Micrel chip as mentioned in the introduction and is internally connected to the GEM3 on the Mars XU3 board. We however have to set up the RGMII delays in order to get it working [13]. The following steps enable you to do that:

1. Once you have enabled the GEM3 according to the previous section, you can generate the bitstream file for this project and open it in Vitis.

2. In Vitis, create the board support package as per the steps detailed in step 2 of section 3.4 of the reference design.

3. Go to board the support package settings, enable LWIP.

4. Inside LWIP settings, change two things under the temac_adapter_options: The value of emac_number to 3 (since we use GEM3) and phy_link_speed to 1000 Mbps. (See fig 4.2).

5. Build the project. Post building, you can see the binary files which are generated as part of LWIP.

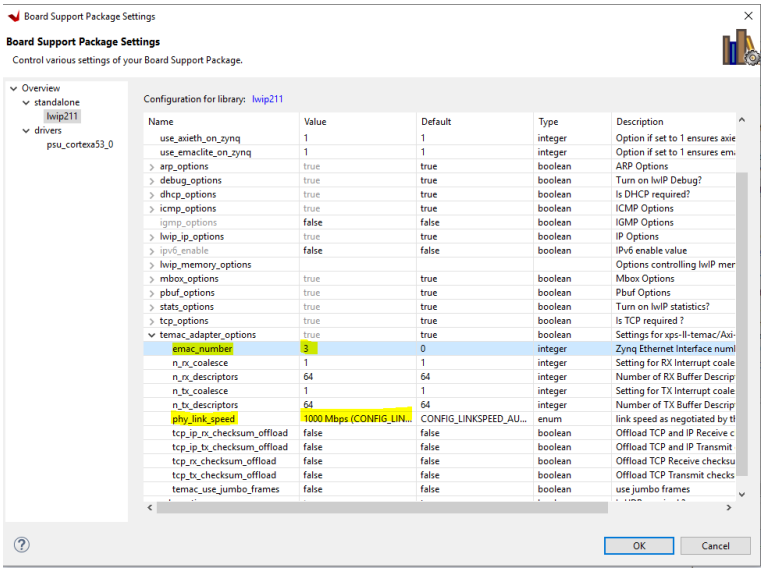


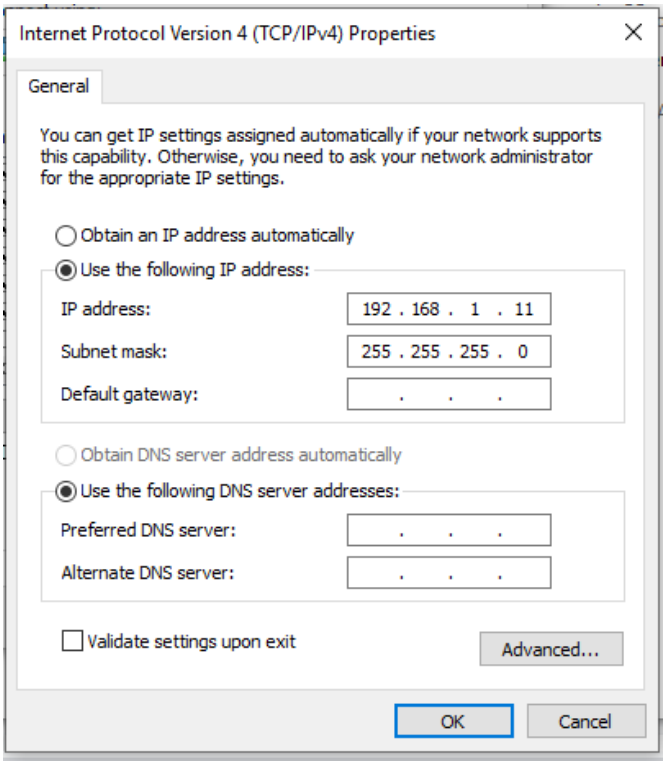
FIGURE 4.2: Setting up PHY : The settings that need to be changed are highlighted

4.3 Testing the Ethernet Connection

4.3.1 Running an LWIP Echo Server

Once you have followed the previous two sections, the GEM3 and the PHY should hopefully be enabled and working. In order to test the ethernet connection, we can run a simple Echo server which echos the data sent to it from the client (PC) via the ethernet connection. The following steps outline the process:

1. Connect the board to your PC via an ethernet cable



2. Go to the network settings of your PC, find the ethernet connection with the board and open the properties window.

3. Set the IP address of your PC for this connection as 192.168.1.11 (you can use 11 for the last digits or any number except 10 as this is the default address given to the board by the BSP).

4. Open PuTTy (or any serial port communication tool of your choice) and connect to the board using the settings mentioned on section 2.1.7 of the reference design.

5. Open Vitis with the board support package containing all the LWIP settings mentioned in the last section. Go to file → New → Application Project.

6. Type in an appropriate name, select the appropriate bitstream (.xsa) files and click next.

7. From the list of example servers, choose LWIP Echo Server

8. Click finish. The Echo server should now be visible on the left-hand side section listing the projects.

9. Before building this application project, go to the Binary Files section of the Echo server project, open the file xadapter.c (./Mars_XU3_EB1/psu_cortexa53_0/standalone_domain/bsp/psu_cortexa53_0/include/netif/xadapter.h).

10. Comment the lines:
    if ((eth_link_status == ETH_LINK_UP) && (!phy_link_status))
    eth_link_status = ETH_LINK_DOWN;
    This is done to prevent a future error while running the server which could lead to an infinite looping between switching the adapter on and off (more on this error can be found here https://github.com/Xilinx/embeddedsw/issues/70).

11. Build the project and run the application on the board. The serial console output should look something like Fig 3.3a upon successful connection.

12. Open a telnet connection to the board (192.168.1.10) via a separate terminal of PuTTy.

13. Send random characters over this terminal and see if they are echoed back.

4.3.2 Board - PC Communication via TCP

4.3.2.1 Running TCP Server Example

Once you have verified according to the previous section that your ethernet connection works, you can proceed to set up a TCP server on the board and use the PC as a client to communicate with the board.

1. Follow steps 1-6 of the previous section.

2. From the list of example projects, choose TCP server.

3. Build it. And run it on the board.

4. Upon successful implementation, you should see the output as per Fig 3.3 b.

5. You can connect to the server using a suitable client (for e.g iperf client or the python socket module).

4.3.2.2 Handling Packets From The Stream

The example project for TCP server on the board sets up the connection by setting up various components required for TCP communication as can be seen from the functions in the main.c file of the project (Fig 3.2 a). The data that the client or server sends to the stream is stored as a special structure which consists of a buffer, the buffer len, the pointer to the recently received data among other things. The libraries of Xilinx SDK

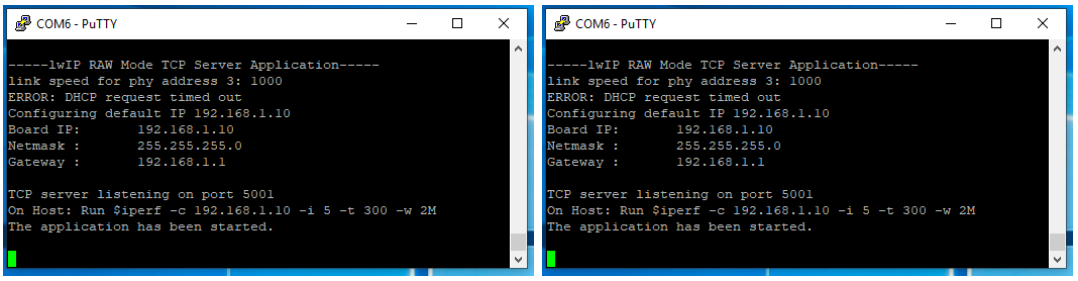


FIGURE 4.3: (From left) Outputs from the console (PuTTY) of echo server (a) and TCP Server (b)

handles the receiving of packets using a series of callback routines whose hierarchy is outlined in Fig 3.2 b. One can edit these callback functions to customise a lossless reading of the data from the stream. Sending of data is handled by the function tcp_write().

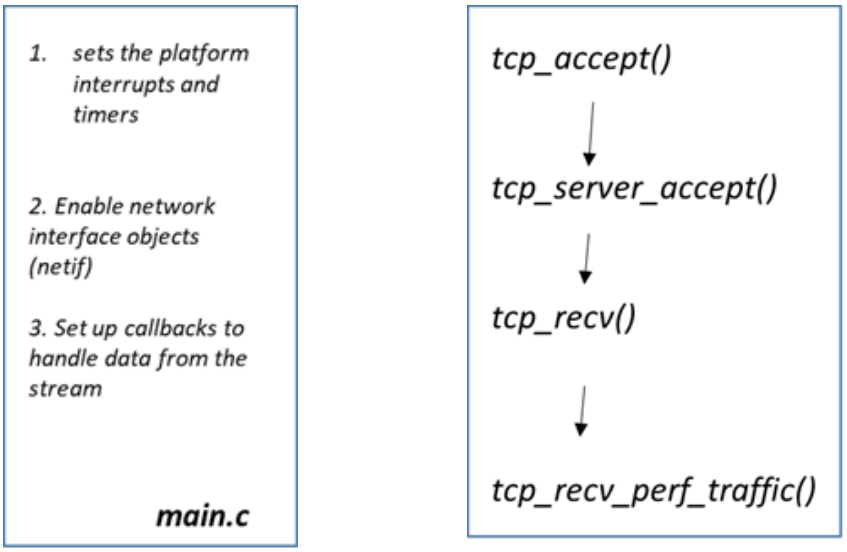


FIGURE 4.4: (From left) a) Sequence of events as it happens in main.c of TCP server example project. b) Callback hierarchy for receiving the data from the stream.

Chapter 5

Mask Retrieval Using Python

5.1 Introduction

As elaborated in the third chapter, the incoming pixel coordinates are compared against the stored mask coordinates in order to do the readout. For this, we need to be able to retrieve the masks from a given (overexposed) image of an array of ions (Fig 3.1). For this purpose, I wrote a python code which takes in an image of an ion array and outputs the masks by means of a simple image processing algorithm. This automates the entire process of finding the masks and serves to improve upon the previous implementation where the masks were fed manually to the FPGA. The previous implementation had the masks grouped per ion. During the readout process, the FPGA compares the incoming pixel value with the masks of all the ions in parallel to look for a match. In my implementation, I order the masks not with respect to the ions but with respect to the row (y) and column (x) values. To keep track of the ion number belonging to a certain mask pixel, the information is stored as a 3-tuple consisting of (x coordinate,y coordinate, ion number) for each mask pixel. These 3-tuples are stored in a python list where the highest y and lowest x pixel of a mask gets index 0 and so on. For eg a mask-pixel at (y,x) = (19,35) will be arranged before (18,56); and (18,30) will be arranged before (18,50). An example of such a list would [(18,30,1), (18,40,1),(18,70,2) (21,20,1),(21,40,1)....].

FIGURE 5.1: (From left) a) Overexposed image of a ion-chain (courtesy: Google Images) b) Same image with Masks marked on it obtained from the algorithm

5.1.1 Algorithm

The pseudocode for the algorithm is as follows:

```
Passed: Image of ions (img_in), threshold, Number_of_pixels_masks (N)

#Windowing Algorithm
dim_x, dim_y = dim(img_in)
ROI_window = rectangular window
Initialize: ROI_window_size =15, step_size, current_position =0,0

for current_position < dim_x,dim_y:
    for pixel in ROI_window:
        if pixel_intensity (pixel) > 230:
            area_ROI.append(pixel)

    if size(area_ROI)/ROI_window_size > threshold:
        ROI_list.append(current_position)

    current_position += window_size * window_resolution

for ROI in ROI_list:
    mask = N brightest pixels in ROI
    mask_list.append(mask coordinates, ion number)

returns mask_list
```

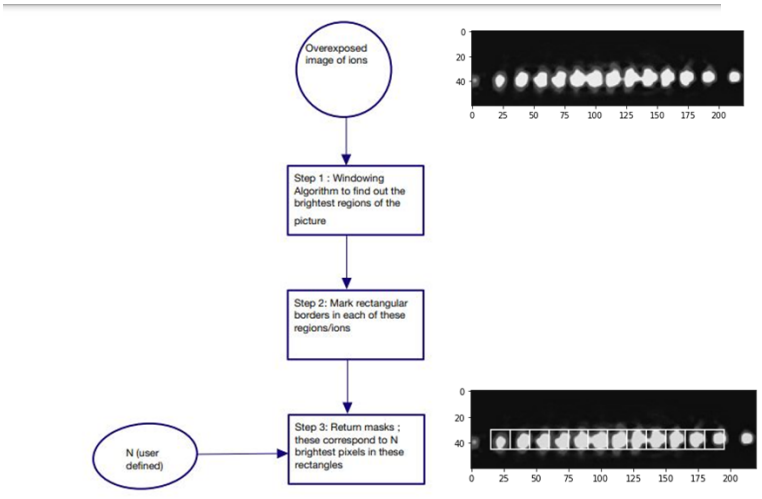


FIGURE 5.2: Flowchart describing the algorithm

5.2 Results, Caveats and Shortcomings

5.2.1 Results

The algorithm has various parameters that one can control in order to increase the accuracy of detection of the masks namely the threshold,window_resolution and window_size (section 5.2). The window_size parameter controls the dimensions of the enclosing rectangular box that scans over the image to detect the masks; the window_resolution is the number of pixels moved by the box in each iteration – in other words, the step size. The threshold parameter determines how much of the enclosing box needs to be filled (in percentage) in order for the area of interest to be detected as an ion. This section outlines the results for the windowing algorithm for different thresholds and window_resolution parameters since the window_size parameter is image/frame dependent and is set at the beginning of the experiment. For this purpose I used a dummy ion chain image (courtesy:Google Images). Fig 5.3 illustrates the results for a scan of threshold from 0 to 0.75 for a window_resolution of 1. The algorithm yielded zero masks for threshold ¿ 0.75 for this particular window size which makes sense as it becomes increasingly less probable to find a space with more than 75% region filled with bright pixels. The bottom image describes the Accuracy plotted against different thresholds. Accuracy here is defined as $\frac{\text{no of masks detected}}{\text{actual no of ions}}$.

Fig 5.4 describes algorithm outputs for different window resolution inputs for a fixed threshold. I chose the fixed threshold at the value 0.3 owing to the median value giving the maximum accuracy from the previous plot. The accuracy sharply drops beyond a value of 1.6 owing to the problem overcounting the same pixels given small step sizes as can be seen from the Accuracy vs Window Resolution plot below.

5.2.2 Caveats

Ideally, the algorithm is supposed to work for any given image and the parameter ROI_window_size as described in the pseudocode should adapt to the image dimensions. However, as it stands currently, the algorithm only works for a ROI_window_size = 15 which limits the dimensions of the image input from the user. The algorithm works well for input sizes of 150*40px (roughly 4:1 size) for a ROI_window_size = 15 and threshold = 0.5. Though this limits the use cases, one can either resize the input image to the aforementioned dimensions or scan through the parameter space to optimise the result. Since this is done before the experiment, the parameter space scanning would not be a bottleneck in terms of the time and thus would be a one time procedure. And this aspect is certainly something that the future versions of the code could improve upon.

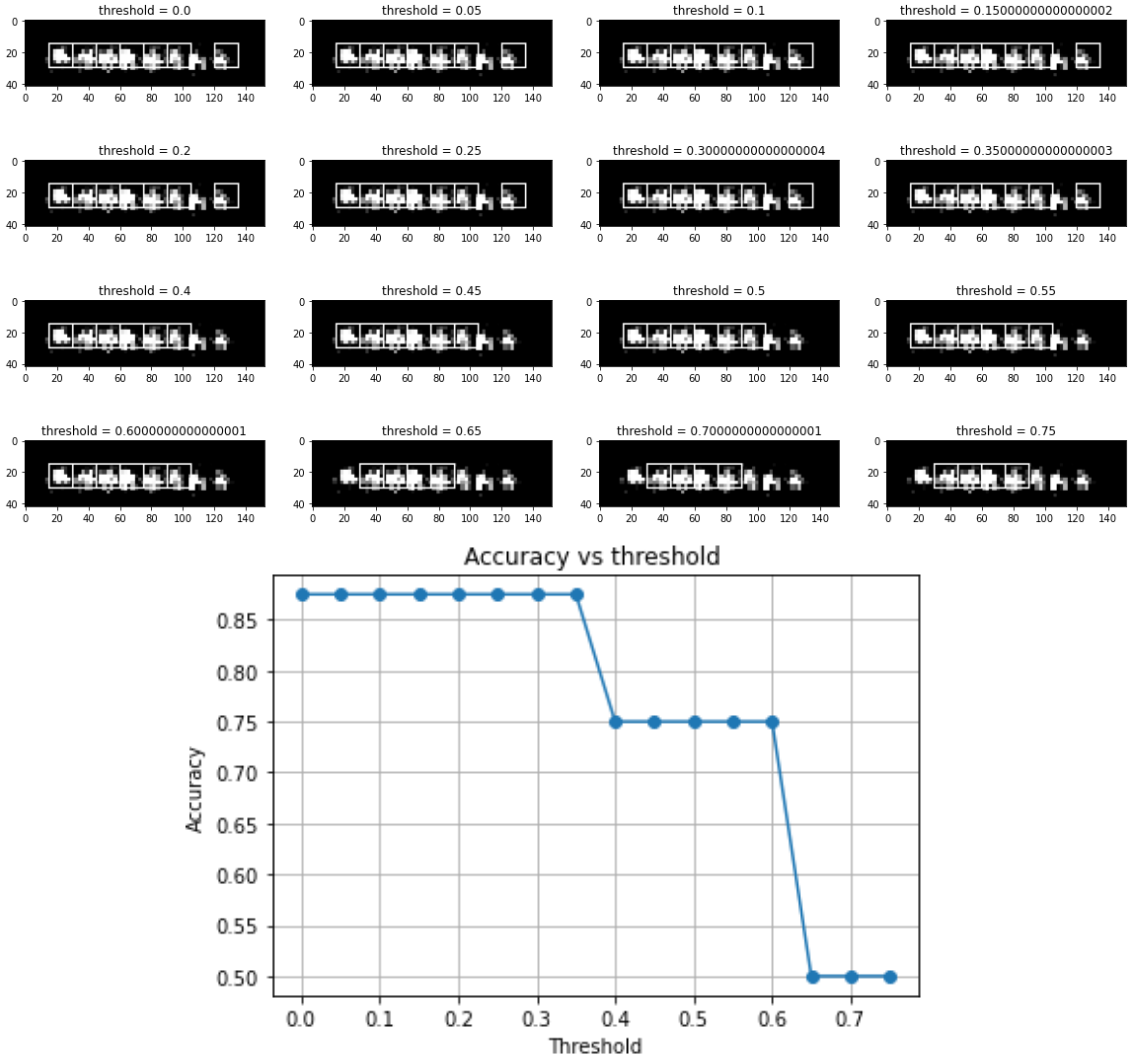


FIGURE 5.3: Images describing algorithm outputs for different thresholds for a fixed window resolution (top) and Accuracy vs Threshold plot (bottom)

5.2.3 Shortcomings

There are a few visible shortcomings that the user may come across while using this algorithm:

1. **Overlap of ROI** : This happens when the step_size is too little. As a result two closely located ROIs are allocated to the same ion giving redundant masks (Fig 5.5 a).

2. **Sharing of ROI** : If the ions are too closely located, there is a probability that one ROI/mask is shared by two ions. This results in a single mask coordinate being allocated to both ions (Fig 5.5 b).

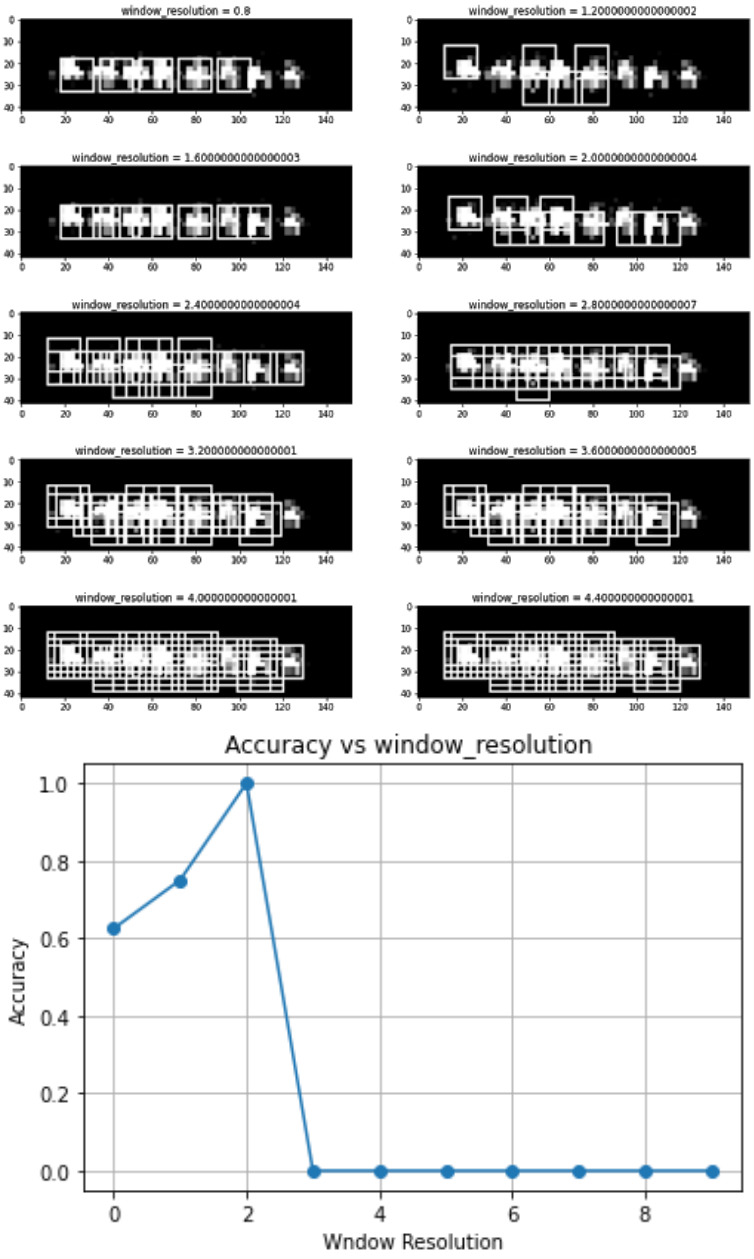


FIGURE 5.4: Images describing algorithm outputs for different window resolutions for a fixed thresholds (top) and Accuracy vs Window Resolutions plot (bottom)

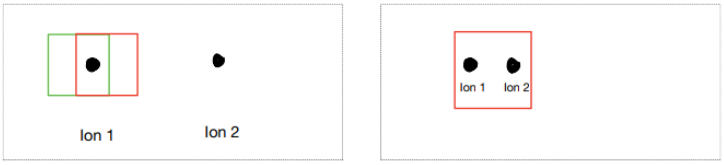


FIGURE 5.5: Illustrations showing overlap error (a) and sharing error (b).

Chapter 6

Summary and Outlook

The main aim of my project has been to investigate the possibility of running a state discrimination algorithm of multiple trapped ions with high fidelity and low latency that uses less lookup-table resources than the previously implemented algorithm. In addition, an auxiliary goal was to automate the calibration mask retrieval using a python script. Both of these goals were achieved but up-to varying degrees.

The state discrimination algorithm was conceived, written and tested up-to the computational simulation stage; moving the synthesis and netlist optimization and programming the FPGA with this firmware, to a future project. In order to have a complete system for experimental evaluation of the readout scheme, we would also require a camera-link interface within the set-up and implementing this would be an immediate future task. In terms of reducing the resource usage further, one can look into storing the masks in Block RAMs (BRAMs) available in Mars XU3 taking the memory load away from the flip-flops of the FPGA. And during the experimental evaluation phase, the ethernet communication between the FPGA and the PC would be an important component in the experimental control system. I hope the experience gained in terms of understanding the ethernet system including some additional non-trivial changes (with respect to setting the RGMII delays, section 4.2.2) to the templates provided by Xilinx software development kit (SDK) would be helpful in the future for setting up a TCP based robust ethernet communication system between the FPGA and the PC.

The mask retrieval algorithm works well with constraints on the dimensions of the input image with fairly good accuracy leaving room for improvements in both accuracy and adaptations to all image sizes and resolutions. Currently, the mask shapes are strictly square with fixed lengths. In a future work, this implementation can be modified

to have the ROIs to be a shape which gives the best signal to noise ratio [2]. To improve the accuracy in estimating the ion positions, We can leverage ion-trap physics knowledge. For eg, the separation of ions in a string is completely regular and this information could be used when setting the masks. In addtion, imaging aberrations should be used and characterized for setting the optimal masks.

[2]For a more detailed discussion, refer to Section 1.4 of Nick Schwegler's thesis [6]

Appendix

Reference Design Manual

For the sake of completeness, I attach the reference design manual of Mars EB1 + Mars XU3 provided by Enclustra Inc which is also publicly available online at: (https://github.com/enclustra/Mars_XU3_EB1_Reference_Design/blob/master/reference_design/doc/Mars_XU3_EB1.pdf). The latest version of the reference design manual as per 23 Mar 2021 for which the algorithm in chapter has been tested can be found with other important backup files at the J folder of Trapped Ion Quantum Information (TIQI) Lab network drive.

MCT

The Module Configuration Tool (MCT) is a software provided by Enclustra for configuring the device and also use it as a medium to check if the board is powered on as per the user's need. The 'Scan for Device Changes shows the number of FTDI devices and the related settings with which it has been powered on.

Useful Links

Enclustra

1. Mars EB1 : https://www.enclustra.com/en/products/base-boards/mars-eb1 .
2. Mars XU3: https://www.enclustra.com/en/products/system-on-chip-modules/mars-xu3/
3. Enclustra Github Link: https://github.com/enclustra/

Xilinx

1. Zynq UltraScale+ MPSoC Product Tables and Product Selection Guide (xilinx.com): https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf

2. Xilinx Synthesis and Simulation Design Guide https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/sim.pdf

3. Vivado Design Suite User Guide: Synthesis (xilinx.com): https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug901-vivado-synthesis.pdf

4. Xilinx XAPP1026 LightWeight IP (lwIP) Application Examples, v5.1, Application Note: https://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf

Appendix

References

[1] Julia Cramer, Norbert Kalb, M Adriaan Rol, Bas Hensen, Machiel S Blok, Matthew Markham, Daniel J Twitchen, Ronald Hanson, and Tim H Taminiau. Repeated quantum error correction on a continuously encoded qubit by real-time feedback. *Nature communications*, 7(1):1–7, 2016.

[2] AH Myerson, DJ Szwer, SC Webster, DTC Allcock, MJ Curtis, G Imreh, JA Sherman, DN Stacey, AM Steane, and DM Lucas. High-fidelity readout of trapped-ion qubits. *Physical Review Letters*, 100(20):200502, 2008.

[3] Colin D Bruzewicz, John Chiaverini, Robert McConnell, and Jeremy M Sage. Trapped-ion quantum computing: Progress and challenges. *Applied Physics Reviews*, 6(2):021314, 2019.

[4] Shantanu Debnath, Norbert M Linke, Caroline Figgatt, Kevin A Landsman, Kevin Wright, and Christopher Monroe. Demonstration of a small programmable quantum computer with atomic qubits. *Nature*, 536(7614):63–66, 2016.

[5] Alice Burrell. *High Fidelity Readout of Trapped Ion Qubits*. PhD thesis, Exeter College Oxford, 2 2010. An optional note.

[6] Nick Schwegler. Towards low-latency parallel readout of multiple trapped ions. Master's thesis, ETH Zurich, ETH Zurich, 10 2018. An optional note.

[7] Peter Staanum. *Quantum Optics with Trapped Calcium Ion*. PhD thesis, Institute of Physics and Astronomy University of Aarhus, Denmark, Danish National Research Foundation Center for Quantum Optics – QUANTOP Institute of Physics and Astronomy University of Aarhus, Denmark, 2 2004. An optional note.

[8] Mars EB1 datasheet. https://www.enclustra.com/en/products/base-boards/mars-eb1/, . Accessed: 2021-06-09.

[9] Mars XU3 datasheet. https://www.enclustra.com/en/products/system-on-chip-modules/mars-xu3/, . Accessed: 2021-06-09.

[10] Zynq Ultrascale+ MPSOC datasheet. https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf. Accessed: 2021-06-09.

[11] LightWeight IP Application examples. https://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf. Accessed: 2021-06-09.

[12] Mars EB1 + XU3 reference design. https://github.com/enclustra/Mars_XU3_EB1_Reference_Design/tree/master/reference_design/doc. Accessed: 2021-06-09.

[13] RGMII constraints. https://github.com/enclustra/GigabitEthernetAppNote/blob/master/Chapter-2-RGMII_timing_constraints.md. Accessed: 2021-06-09.