Mikkel Jordahn Tom Nugent Marco Lavalle [mj2214@ic.ac.uk](mailto:mj2214@ic.ac.uk) [tn1115@ic.ac.uk](mailto:tn1115@ic.ac.uk) ml7715@ic.ac.uk

CID 00924570 CID 01090230 CID 01046115

Coin Detection Using Catapult C on an FPGA

Interim Report

Contents

[1 Catapult C Exercises 2](#_Toc446878216)

[1.1 Brief Description of Catapult C 2](#_Toc446878217)

[1.2 Dot Product Exercise 2](#_Toc446878218)

[1.3 VGA Blur Exercise 3](#_Toc446878219)

[1.4 Sobel edge detection filter 5](#_Toc446878220)

[1.5 Three-by-three Sobel Filter 6](#_Toc446878221)

[1.6 Five-by-five Sobel Filter 6](#_Toc446878222)

[1.7 Greyscale Sobel Filters 7](#_Toc446878223)

[2 Project 8](#_Toc446878224)

[2.1 Project Introduction 8](#_Toc446878225)

[2.2 High Level Description of System 9](#_Toc446878226)

[2.3 System Schematic 10](#_Toc446878227)

[2.4 Expected Outcomes 10](#_Toc446878228)

[2.5 Suggestions for Future Work 11](#_Toc446878229)

[2.6 Project Planning 11](#_Toc446878230)

[3 References 12](#_Toc446878231)

[4 Appendices 13](#_Toc446878232)

[4.1 Appendix A 13](#_Toc446878233)

# Catapult C Exercises

## Brief Description of Catapult C

Catapult C is a high level synthesis tool, used to create RTL code for use on FPGAs (field programmable gate arrays) and ASICs (application specific integrated circuits) (1). For the purpose of this project, we are creating Verilog code to import into Quartus II and load onto an FPGA. Catapult C allows you to describe the digital logic circuit you want to implement as an algorithm in a high level language (C/C++), which is then compiled down to the actual logic gate configurations used on the FPGA.

The advantage of using such a tool is the ability to create complex digital logic circuits much more quickly, because you only need to describe the algorithm with a language you are probably more familiar with, rather than learning a HDL (high level description language) or even more complicated still, working out the digital logic yourself, which is incredibly time consuming. With Catapult C, you do not have to worry about hierarchy or clocks, but only describe your desired behaviour, making it easier to write and reduces chance of errors occurring.

However, using this tool obviously relies on you knowing the C language, and how well your design is described depends on how familiar you are with the language. You must also be familiar with the Catapult C program itself, and understand how changing the settings will affect the end code generated. There is also the danger of your design not being implemented in the most efficient way, because the actual logic gates used are all computed for you, rather than you working out the most efficient method.

Overall the use of Catapult C during development of a product is very beneficial, purely due to the simplicity and speed of creating something, which would otherwise be very complicated to design just using digital logic gates.

## Dot Product Exercise

Our first task using Catapult C was to compute the dot product of two one-dimensional vectors. We would use the switches as input, and display the result on the LEDs. This involved us learning the main workflow of the program, including changing settings to run on our particular FPGA. We also learnt some differences between timing constraints, and finally how to export the code as a Verilog file, so that it can be used in Quartus II and loaded onto the FPGA.

First we had to set up a new project and imported the relevant source files. Upon viewing the code, it seemed very simple and it did not take long to work out what was going on. The only issue was some unfamiliar syntax, but that was quickly resolved. We then had to use the ‘Setup Design’ window to select the relevant FPGA and its clock speed, and finally we went into ‘Architecture Constraints’ to change the inputs, outputs, and loop unrolling/pipelining.

We also used the scheduling section to simulate the running of the program. This produced a Gantt chart where we could view the timing of the design and compare the differences between loop unrolling and pipelining. Loop unrolling is a process, which increases a program's speed by reducing instructions that control the loop, such as end of loop conditions or arithmetic for each iteration (2). Pipelining consists of a chain of instructions where the output of one instruction feeds into the input of the next, with some amount of buffering in between (3). This allows you to change the input on each clock cycle, while the previous input is still ‘going through’ the pipeline.

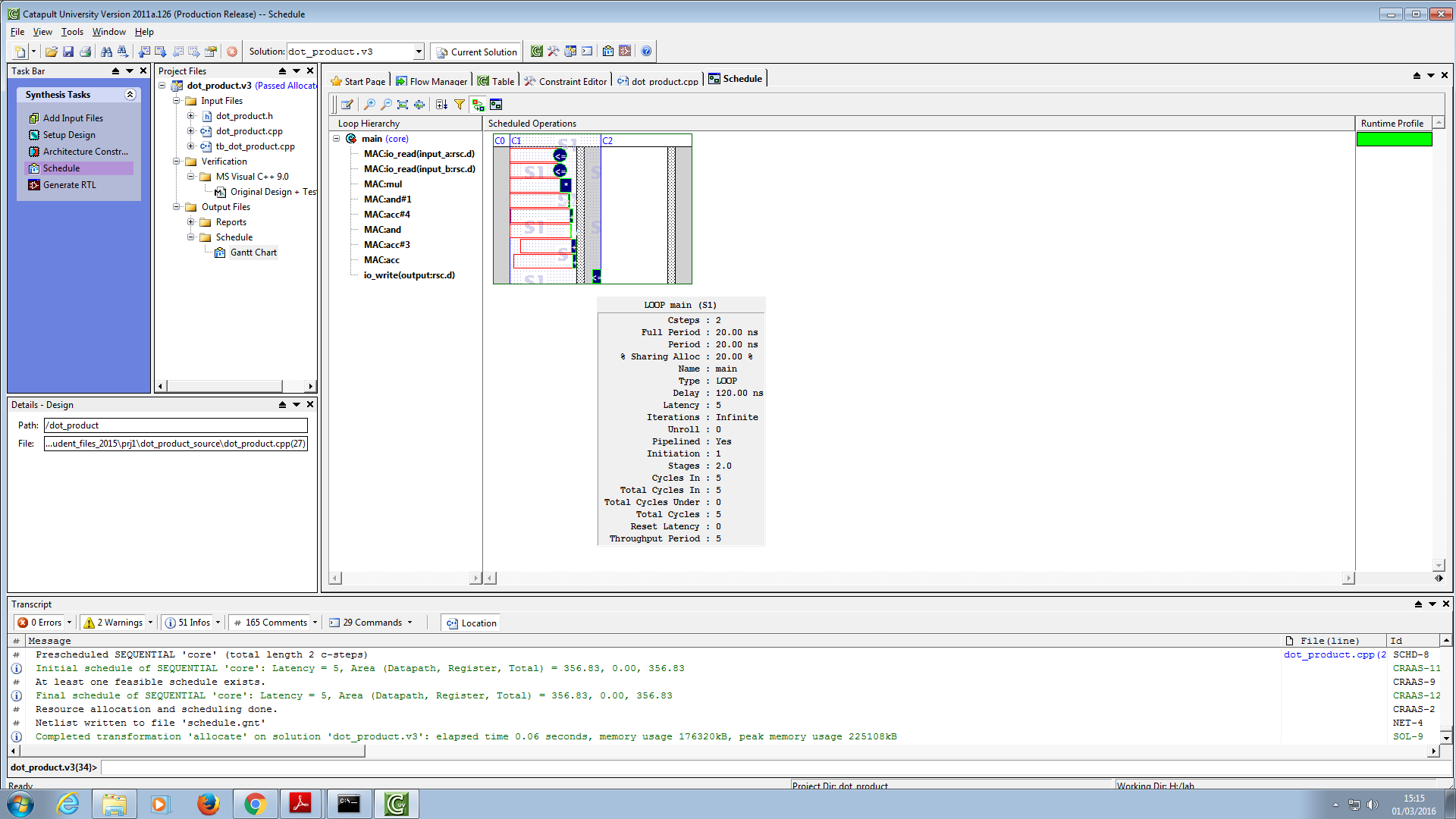


Figure 1 - Gantt chart produced with pipelining enabled

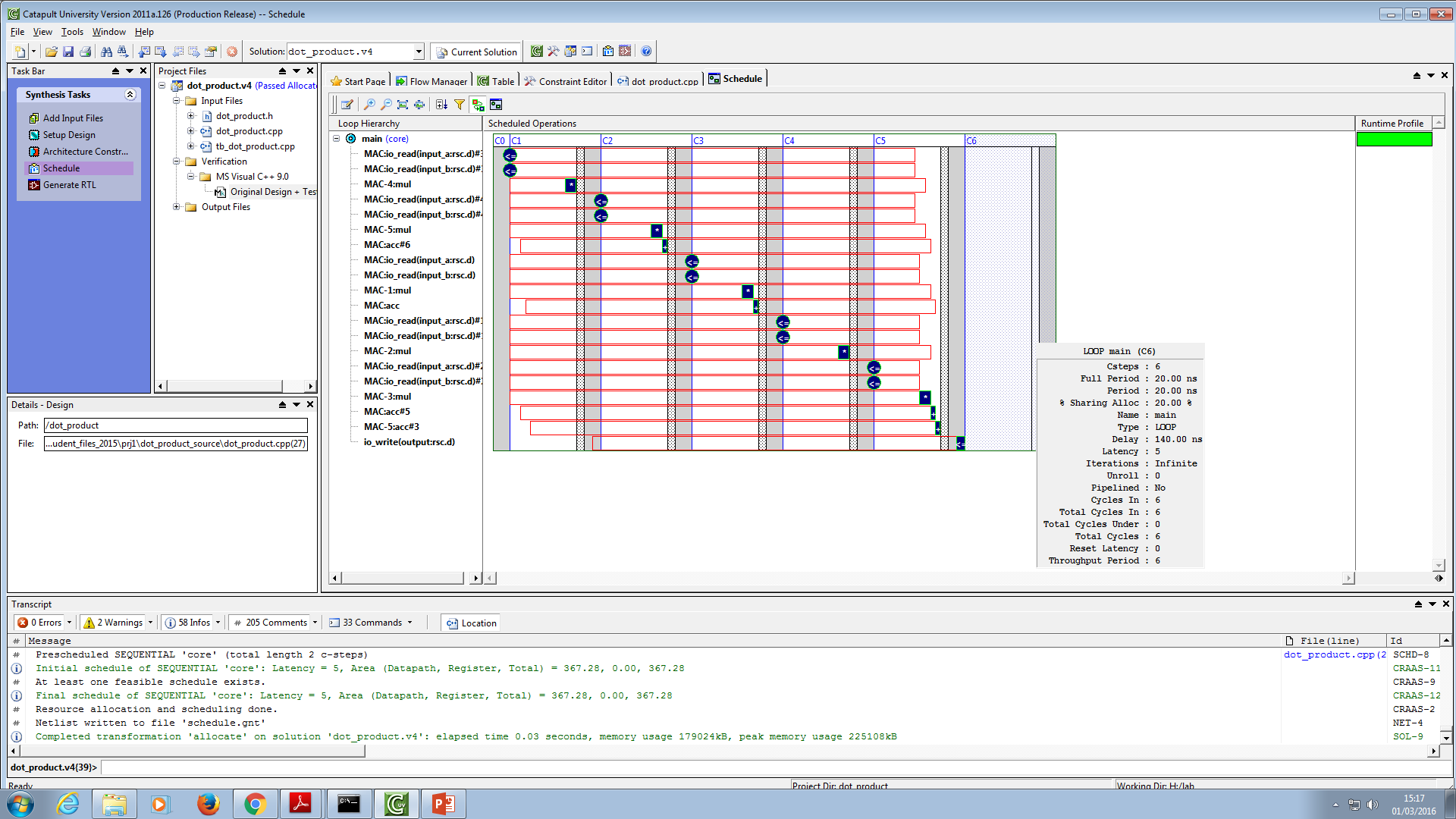


Figure 2 – Gantt chart produced with loop unrolling enabled

Finally, upon generating the Verilog RTL code, we could import this into a Quartus II design project as part of a block diagram, and connect the relevant inputs and outputs. This was then compiled and programmed onto the FPGA, where we saw our vector dot product work for the first time.

## VGA Blur Exercise

The last part of the lab experiment consisted in using Catapult C to generate Verilog files for digital image processing applications. The files provided as part of the lab experiment included the basic building blocks, which we used to develop our solutions.

The Quartus II project made available included the following elements: a line buffer that stores an arbitrary number of pixels, whose values are updated every clock cycle; a VGA mouse block, which superimposes a mask to the pixels inside the area defined by the cursor of the mouse; a VGA blur block that applies a blur filter to the processed image. The source code of the VGA blur and the VGA mouse block were provided in the initial files and were modified and utilized multiple times during the experiment.

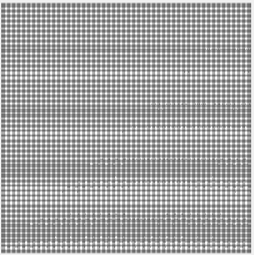
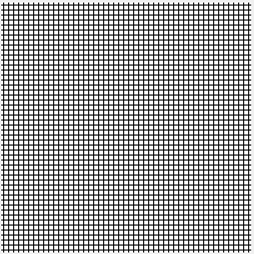
In order understand the behaviour of the blur filter, Matlab was used to apply the filter to a previously defined sample image. The output obtained and the original image are shown in Figure 3.

Figure 3 – Original image (left), Blur filter applied with Matlab (right)

As we can see in the sample catapult C code, Figure 4, the blur filter sums the components of each pixel and divides the value obtained by the total number of pixels. In this case, there were five rows and five columns of pixels analysed during each clock cycle, and therefore a total of twenty-five pixels were processed concurrently.

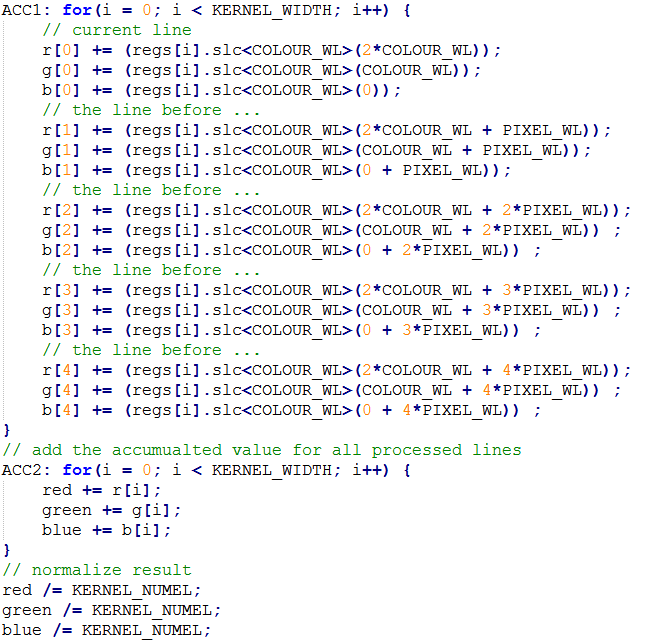


Figure 4 – VGA blur source code

Each pixel in the output is therefore the average of the pixels composing the corresponding five-by-five mask in the original image. As we can see in the code illustrated in Figure 4, the components (red, green and blue) of each line of pixels are summed together and stored in a five-entry array. The five entries are then summed together and assigned to the variables red, green and blue. Finally, the values stored in the three variables are normalized, divided by twenty-five (the number of pixels processed each clock cycle), and fed to the output pixel through an output signal.

## Sobel edge detection filter

The Catapult C source code of the VGA blur block was used as a starting point to generate a Sobel edge detection filter. The Sobel filter utilizes either a three-by-three (Figure 5) or a five-by-five (Figure 6) matrix operator to generate an edge-detection effect. We created both versions in the experiment.

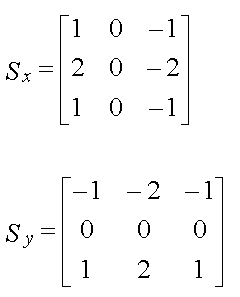
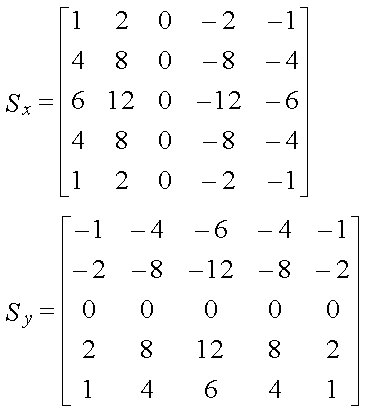
 

Figure 5 - 3x3 Sobel matrix operator (4) Figure 6- 5x5 Sobel matrix operator (4)

In order to apply the Sobel filter, each set of pixels provided by the line buffer was convoluted both with the x-matrix and with the y-matrix of the filter. The convolution of two bi-dimensional matrices is defined in this case as the product of the kernel matrix flipped both vertically and horizontally and the entries overlapped on the sample data matrix (5). The two values obtained by convoluting the data with the x and the y matrix were then summed (after taking the absolute value of both) and normalized. The absolute value provides a good approximation of the edge gradient (Figure 7), because we could not access the square root function in Catapult C.



Figure 7 – Edge gradient calculation (6)

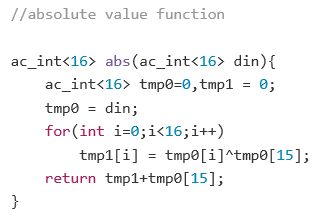
In order to calculate the absolute value, the corresponding function was found in the high-level-synthesis blue book, provided as part of the experiment, and imported in the main program (Figure 8). The function was modified to operate on sixteen bit signed binary numbers.

Figure 8 – Absolute value function

## Three-by-three Sobel Filter

As can be seen in the Figure 9, in order to apply the filter using a three-by-three matrix, each component of the pixels analysed was stored in a nine-entry array and convoluted with the x and the y operator, summing the corresponding partial products together. The final values were then divided by three in order to apply an appropriate scaling and reduce the noise in the final image and then they were sent to the VGA output through an output signal.

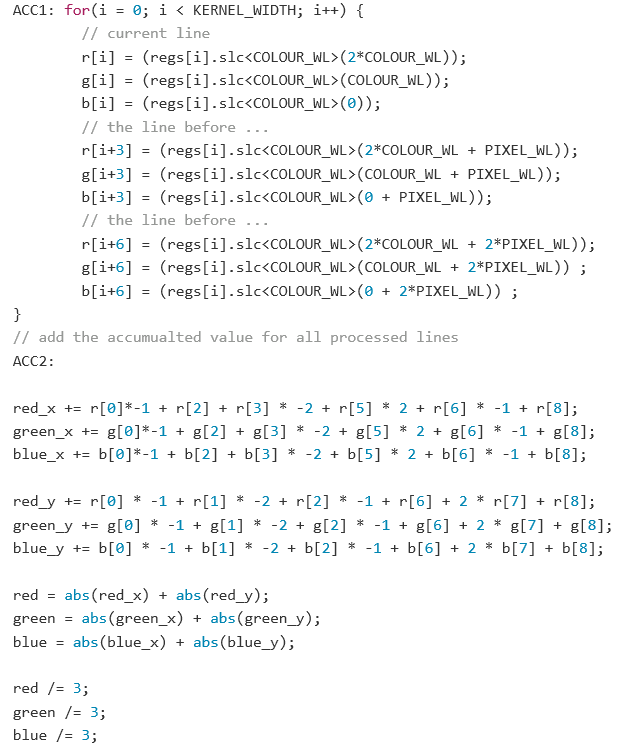


Figure 9 – Three-by-three Sobel filter colour code

Figure 10 - Three-by-three Sobel filter colour effect

The filter was then compiled, placed in the original Quartus II project and sent to the FPGA device. The effect produced by the filter can be seen in Figure 10.

## Five-by-five Sobel Filter

A similar approach was used to create the Sobel filter using a five-by-five matrix (Figure 11). This time the convolution was executed utilising the five-by-five matrix operators; the pixel values were stored in an array. In order to normalize the results obtained after applying the convolution a scaling factor was applied to reduce the values obtained to a usable range; each component was multiplied by 3/66 (7). The effect produced by the filter can be seen in Figure 12.

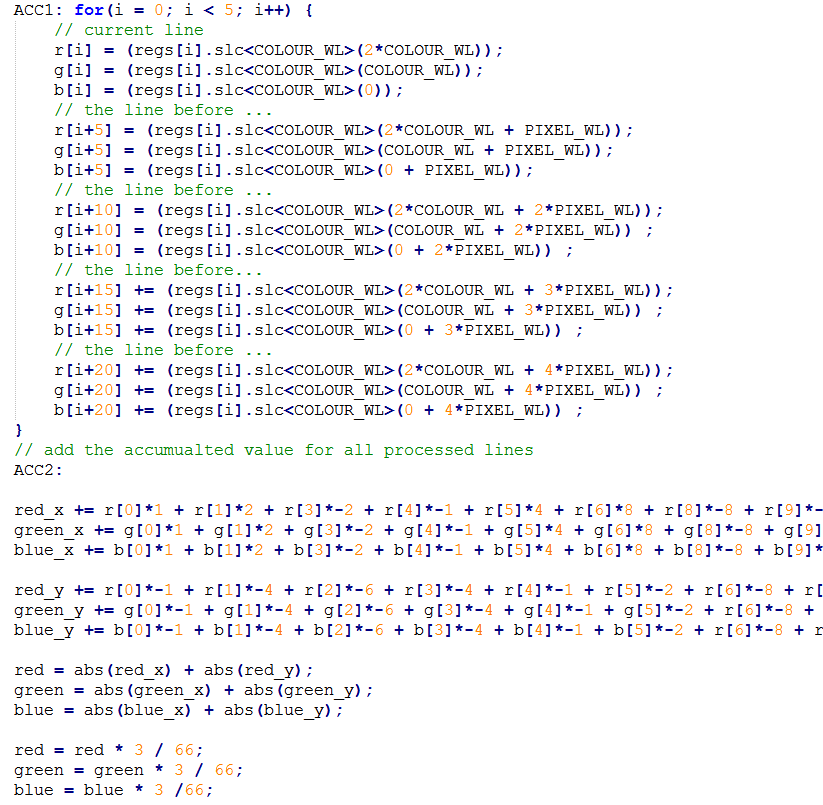


Figure 11 – Five-by-five Sobel filter colour code

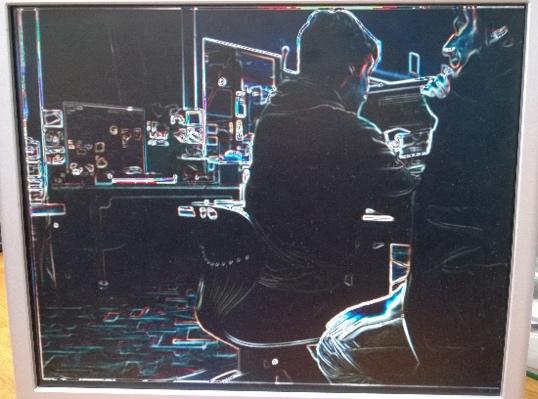
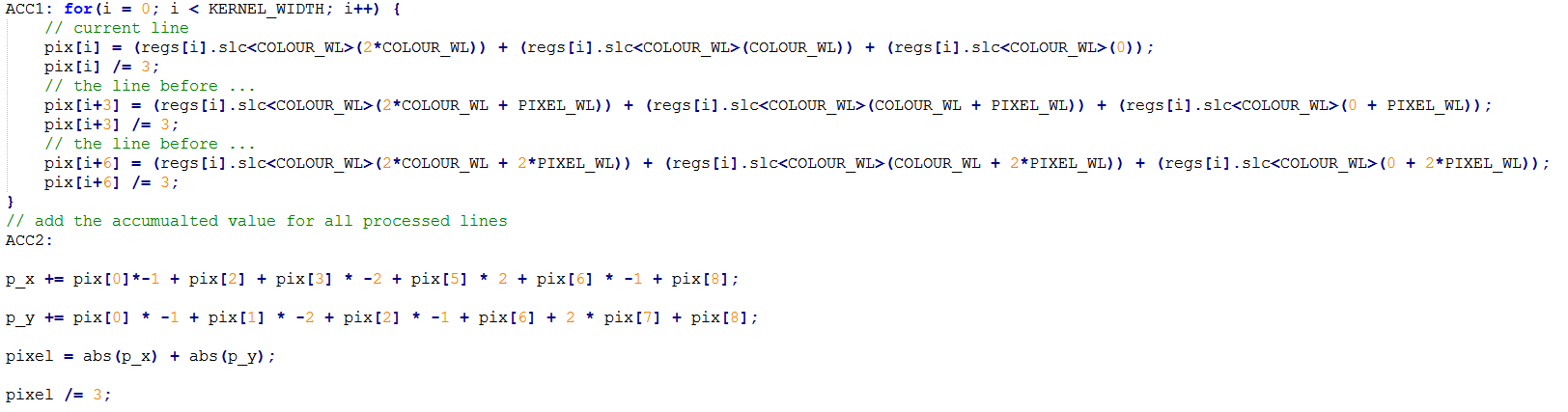


Figure 12 – Five-by-five Sobel filter colour effect

## Greyscale Sobel Filters

A greyscale version of both filters was obtained applying small modifications to the original code. As can be seen in Figure 13, a single array was used to store the average value of each pixel analysed (the three colour components were summed together and averaged). The matrix operators were then applied as usual and the final values normalized. A single value was fed to each of the colour components of the output pixel. The effect produced by the two filters is shown in Figures 14 and 15.



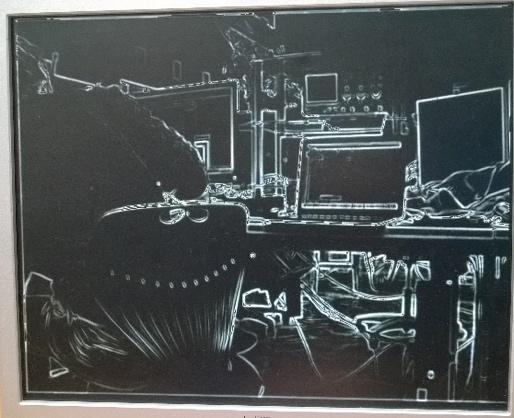
Figure 13 – Three-by-three Sobel filter greyscale code



Figure 14 – Three-by-three Sobel filter greyscale effect

Figure 15 – Five-by-five Sobel filter greyscale effect

# Project

## Project Introduction

Having completed our first exercises with the FPGA in the spring labs, we have seen how image processing can be used to convey new information with regards to an image. In our project the main aim is to use the FPGA and Image Processing algorithms to create a coin detector.

We will attempt to use the different inputs and outputs to create several modes for the FPGA coin detector. The frames generated from the FPGA camera will be used to detect coins, which in this case will be Sterling Pound coins. Building on top of the basic coin detector, we will create two modes.

For the general detection of coins, “blob” detection (8) will be used. Using blob detection, we can find the area of each of these blobs; the area of each coin will then be used to determine its value, depending on preset values. Building on top of this, the selective mode will let the user hover over a single coin using the mouse connected to the FPGA, and thus only display the value of the selected coin. For the total value mode, all coins in the current image will be processed and their values will be added together and displayed. This would obviously include the single coin mode, as if only one coin is present that will represent the total value. In all cases, the values will be displayed on the 7-segment display on the FPGA. The switches will be used to determine which mode the FPGA is in.

The expectations are to fulfill all of the above requirements, as they have been deemed feasible. We expect the hardest and most time-consuming part of the project to be to create a working blob detector. Once this part of the project is done, it should be rather simple, although not trivial, to develop the other modes, using the blob detector as our basic building block.

## High Level Description of System

According to Kiran et al (9) before detecting the blobs as required, it is first necessary to preprocess the image provided by the FPGA camera. First the image will be converted to greyscale in order to make pixel comparisons simpler. Median filtering will then be used to remove any noise from the image stream. The image will then be converted to a binary image, after which complementing will be performed. Once this has been done, the blobs will be read as segments of matrices in which several adjacent values are the same. From here it should be simple to calculate area, as area will just be considered the amount of connected pixels of the same value.

In order to then determine which coin is being processed in the image, the area of the coin will be compared to predetermined coin sizes. This way, it is possible to determine what coin is in the image, and based on a pre-programmed value, the FPGA will display this value on the 7-segment display. These areas will be predetermined based on the amount of pixels the coin “covers” when placed at a standard distance from the camera.

Once the blob detector and area comparator is working, it should be simple to create the remaining modes. The single-coin detector will be used as the basic building block. For the mouse functionality, the user will be able to hover over a certain coin, at which point the value of that specific coin will be displayed. The basic idea is to limit the processed part of the image, so that only the part covered by or nearest to the mouse is processed, rather than the entire image. For the total value mode, the full image will be processed, and for every coin detected, the value will be added to a total sum, which then will be displayed on the 7-segment display, thus showing the total value.

For the basic coin detection, a separate block will be made in Quartus. A separate block will also be made for the image pre-processing, which also will take in input the mouse, which will only be used if the FPGA is in selective mode. A state-machine block will also be made in order to control the entire circuit and which modes the FPGA should be in.

For obvious reasons, it is a requirement that the conditions are the same every time the FPGA is run. It will therefore be necessary to determine a certain distance from which the coins will be detected; otherwise the area values will not match accordingly. An issue that may arise is whether some of the coin areas are of the same size. Should this be an issue, it may be necessary to define further specifications for which the coin detection works, such as colour. In addition to this, it may be necessary to have the background be completely white, i.e. using a piece of paper as background, as it otherwise would be hard to standardize pixel conditions. It may also be necessary to control the lighting, as we noticed during the exercises that the camera had poor dynamic range, meaning it worked better in certain light conditions.

## System Schematic

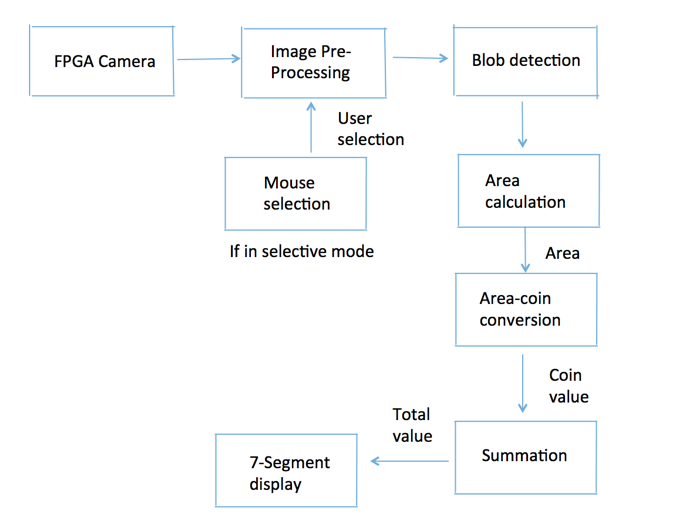


Figure 16 – Expected steps required for project

## Expected Outcomes

The first, and perhaps most difficult part of the project is going to be to get the blob detection to work. In order for the blob detection to work, it is a requirement to have the correct pre-processing of the image done and correctly manipulate the binary image. However, once the blob detection is working, the rest of project should be rather simple to get to work. The process of creating blob detection will require a large amount of simulation to be done, which will have to be completed in Catapult C, because exporting to Quartus and running on the FPGA can be time consuming. A difficult aspect will also be to predetermine the amount of pixels corresponding to each coin. This is a key aspect in the project, as it otherwise will be impossible to differentiate between the different coins.

However, the expected outcome is to get the full project to work. It has been deemed feasible and therefore should be possible. Several algorithms can be found online for blob detection, and therefore there should be sufficient material and resources to help with certain difficulties that may be encountered throughout the project. The general idea that the area detection works on, is also commonly used throughout other scientific papers, and therefore works in theory and practice.

## Suggestions for Future Work

If successful, the work could be extended to several things. The first one could be general money processing, so that notes also could be included in the project. This should not be difficult as the notes will have their own size and area and therefore easily could be identified from the coins. Another option is to extend the work to different currencies, so that different modes could be selected depending on which currency is being analyzed. This would only require separate discrete coins and notes sizes for each of the currencies. This could be done simply by creating a new block in Quartus which determines which currency currently is being analyzed, so that only the area-coin conversion block seen in Figure 16 would have to be changed.

## Project Planning

It is suspected that the blob detection will be the hardest part to get to work, and therefore it has been devoted the most time. It has also been found that what takes the longest is in fact debugging of written code. Therefore, several days have been devoted to debugging at the end of each segment of the project. A Github repository has been made, which should allow easy sharing and editing of code throughout the group.  
  
The two days allocated at the end of the project time, have been allocated for unforeseen time that may be required. This is generally always the case with projects, as some aspects always will appear during the project period. A group member will on a daily basis update the Gantt chart, in order to ensure that the project is on track with regards to time. The total value mode and the selective mode editions have specifically been placed at the end of the project, to ensure that the group will at least have made a functioning project with the basic coin detector before attempting to extend the project.

*Please see Appendix A for Gantt Chart.*

# References

1. Wikipedia, the free encyclopaedia. Catapult C. <https://en.wikipedia.org/wiki/Catapult_C> [Accessed March 26th 2016]
2. Wikipedia, the free encyclopaedia. Loop Unrolling. <https://en.wikipedia.org/wiki/Loop_unrolling> [Accessed March 26th 2016]
3. Wikipedia, the free encyclopaedia. Pipeline (software). <https://en.wikipedia.org/wiki/Pipeline_(software)> [Accessed March 26th 2016]
4. Centre For Intelligent Machines. Sobel 3x3 and 5x5 Masks. <http://www.cim.mcgill.ca/~image529/TA529/Image529_99/assignments/edge_detection/references/sobel.htm> [Accessed March 26th 2016]
5. Song Ho Ahn. Convolution. [http://www.songho.ca/dsp/convolution/convolution.html - convolution\_2d](http://www.songho.ca/dsp/convolution/convolution.html) [Accessed March 26th 2016]
6. Wikipedia, the free encyclopaedia. Sobel Operator. <https://en.wikipedia.org/wiki/Sobel_operator> [Accessed March 26th 2016]
7. Yves Daoust. Sobel filter algorithm thresholding. <http://stackoverflow.com/questions/23830097/sobel-filter-algorithm-thresholding-no-external-libs-used> [Accessed March 27th 2016]
8. Wikipedia, the free encyclopaedia. Blob detection[.](https://en.wikipedia.org/wiki/Blob_detection) <https://en.wikipedia.org/wiki/Blob_detection> [Accessed March 26th 2016]
9. Divya Kiran, Abdul Imran Rasheed & Hariharan Ramasangu. FPGA Implementation of Blob Detection Algorithm for Object Detection in Visual Navigation. Scientific paper. M.S. Ramaiah School of Advanced Studies; 2013.

# Appendices

## Appendix A