Hardware Accelerated Image Masking

Luca Barbas, Lawrence Godfrey, Matthew Lock and Mahmoodah Jaffer
Department of Electrical Engineering
University of Cape Town
Rondebosch, Cape Town, South Africa

***Abstract*—This paper details the design, methodology and results of building a hardware accelerator for image masking applications. The Nexys A7 100T is used for implementation of the accelerator, with the aim of the paper being to investigate the use of an FPGA as a hardware accelerator in order to take full advantage of the FPGA's highly parallelizable hardware, which is naturally extremely well suited to embarrassingly parallel problems such as image masking. The implemented accelerator allows the user to upload images and image masks through simple UART communications protocols, with automatic image masking taking place once the upload has completed. Furthermore, this paper highlights design challenges faced implementing such an accelerator, such as an insufficient number of lookup tables available to synthesise the design, and will assess the effectiveness of such an accelerator against a golden measure implemented in C++.**

I. Introduction

Many image processing techniques are inherently parallel, and image processing, in general, requires an ever-increasing amount of computation as image resolutions, and the complexity of the image processing techniques, increase. The need for edge devices, which can do this processing immediately without needing to send the data to a cluster, is also growing as industries incorporate more automation techniques into their infrastructure. Therefore, it is increasingly important to use hardware which is able to take advantage of the parallel nature of image processing in order to speed up this computation.

Image masking is an image processing technique where two images are overlaid using a mask. This is often used to change the background of an image while keeping an object or figure in the foreground unaltered, which can be seen in Figure 2. This figure also shows that a mask is required for this operation, and that the mask needs to be in the shape of whatever you want to cut out of the original image. This mask essentially acts as an array of zeros and ones which correspond to pixels in the input images, and can be used by simple logic gates to "turn on" or "turn off" certain pixels in the input images.

All code produced for this report can be found in our Git Repository [1].

II. Background

There are two other hardware choices which could be considered for image processing: GPUs and ASICs.

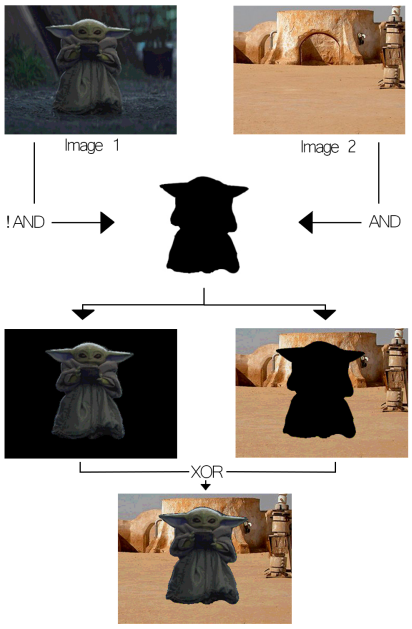

Fig. 1: Conceptual Diagram of Masking Operation

While the GPUs architecture does allow for a high level of parallelization, there are a few reasons why FPGAs could be more desirable: GPUs implement their logic in software, as opposed to the hardware implementation of an FPGA, which is much faster. While this has improved in recent years, GPUs are historically power hungry compared to FPGAs, making them less than ideal for edge devices. GPUs require supporting hardware, namely a host CPU, while an FPGA can send data to and between memory and various other I/O without any supporting hardware, which is a huge benefit for edge devices.

Another option is to implement dedicated hardware for the application. This is known as an application-specific integrated circuit (ASIC). The obvious benefits of using an ASIC is reduced cost, power consumption and size, with an increase in speed if it is required. Kuon and Rose, 2006, estimate that FPGAs require 20 to 40 times more silicon area than an equivalent ASIC, while using between 10 and 15 times more power, and operating 3 to 4 times slower [2].

However, the reconfigurable nature of an FPGA can, in some cases, be a massive advantage over an ASIC, especially in a time where new technologies and techniques in image processing applications are arising more frequently. This could

be a huge factor in certain industries where the problem, and therefore the technology being used to solve this problem, changes frequently, and the overhead of mass-producing a new set of ASICs every time there is a change becomes unsustainable.

III. Methodology

This section outlines the high level approach used to design the system. The methodology used in the system design can be broken down into 3 components. Namely host interaction methodology, masking methodology and display methodology. Each will be discussed in the subsequent subsections,while the block diagram seen in Figure 2 below shows the general flow within the system and how these 3 components make up the system. Furthermore, this section seeks to explore the methods to be used in measuring the results of the investigation, and providing any necessary context behind the metrics used.

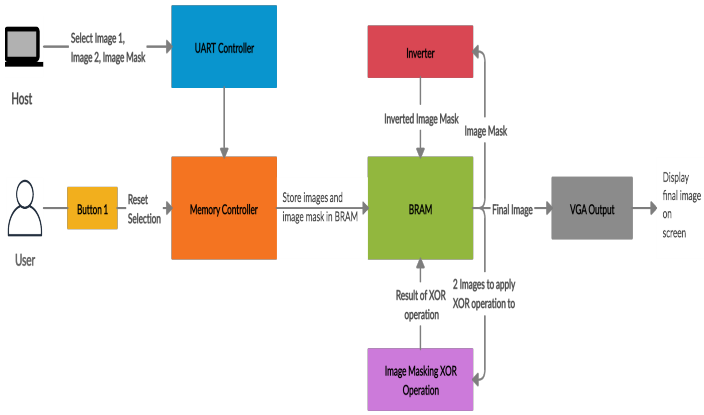

Fig. 2: High Level Block Diagram

A. Host Interaction Methodology

In order for the accelerator to be seen as a practical image masking tool, the user must be able to select the images which are to be masked by the system, as well as the mask that they want to use. Hence, the user needs to be able to load two uncompressed 12-bit RGB colour images and one image mask onto the accelerator, all with a resolution of 320 x 240.
The initial plan was to implement an SD card module that would allow the user to cycle through images that they want to overlay as well as the mask they wanted to use. This solution would have required 3 buttons for the user input:

- The first button would allow the user to cycle through the images on which the overlay would occur
- The second button would allow the user to cycle through images which would overlay the first image selected
- The third button would allow the user to cycle through the image masks

Due to time constraints it was not possible to implement an SD card module. However, it was possible to implement UART communication between the FPGA and a host. This allows the user to upload new images on which the image masking can be performed by the system.The user can also reset the system by means of a button. The reset button required the implementation of debouncing. The user is therefore still able to select which images they want to overlay together since uncompressed 12-bit RGB colour images and image masks, with a resolution of 320 x 240, can be stored on the host.

B. Masking Methodology

This module required the use of some sort of memory configuration, since the accelerator is intended to work with images preloaded onto the system, or images uploaded by the user. There are various ways in which this memory could be queried so that the appropriate operations could be performed on the images that were loaded onto the system.
The first of which would be to load the images on simulated cache level memory, such as a 2D array structure, shown in the listing below. This would allow the module to make use of simple For-loops to carry out the operations over the range of addresses. These For-Loops would be unrolled during the implementation phase allowing each iteration of the For-Loop to occur concurrently, and essentially creating " a mass of wires" connecting directly between memory addresses and the LUTs needed to carry out the necessary operations. While this implementation method is extremely well suited to the embarrassingly parallel problems, this implementation may come at the cost of heavy resource utilisation. Hence, when the implementation of this option was attempted, we found that it was not possible to do so.

Another way that the memory could be configured is through BRAM modules, where each instantiated module allows for the querying of a single memory address every clock cycle. While this approach would not allow for parallel access of the data structure that is associated with previously mentioned structures, the utilisation cost of such an implementation would be far lower. Furthermore, these BRAM modules can be set up in various configurations. Different options include storing each image on one BRAM module for "sequential" access of the image data, or the instantiation of multiple smaller BRAM modules each holding a portion of each image. This would allow the accelerator "parallel" access of the BRAM modules, making use of the highly parallelizable nature of FPGAs as different parts of each image can be accessed concurrently.

The option that was used in the final implementation of the BRAM module was to use BRAM sequentially. The memory was flattened onto a single bus in the Image Masking Unit, as seen in the listing below.

Once the implementation data structure had been queried, XOR masking operations were performed on each of the pixels of the images and image mask using the following logic:

```
for (i = 0; i < MEM_SIZE - 1; i = i + 1)
begin
    image1[i][0] <= image1[i][0] ^ mask[i][0];
    image2[i][0] <= image2[i][0] ^ (~mask[i][0]);
end
```

Listing 1. 2D Array Structure

```
for (i=0;i < PARTIAL_ADDRESSES; i= i+1) begin
    image1_1D[12*i +: 12] <= image1[i];
    image2_1D[12*i +: 12] <= image2[i];
    mask_1D[12*i +: 12] <= mask[i];
end
```

Listing 2. BRAM Single Bus

$$Result = (I1 \odot Mask) \oplus (I2 \odot !Mask)$$

Once all the masking operations for each pixel are complete and the result is stored to memory, the output image is able to be displayed over the VGA interface of the Nexys A7 via the VGA module. The details of the display methodology are discussed in subsequent sections. .

C. Display Methodology

Since the accelerator is working with images with a resolution of 320×280, the total number of pixels over which the masking operations will be applied is a hefty $76,800$ pixels. While it would be possible to produce some sort of testbench using matlab or c to test the accuracy of the final result produced by the accelerator, a much more effective way would be a visual analysis of the results. By making use of the Nexys A7's VGA interface the accelerator would be able to display the resulting image to any monitor with a VGA port.

While the implementation of a module needed to produce a VGA output is not a difficult endeavor and is most easily explained by the principles laid out by B.Eater [3], there are some strict timing requirements associated with different image resolutions [4]. The basic principles of which require a pixel clock to determine not only how long each pixel should be displayed for, but for how long the transitions should be between vertical lines of pixels. Further signals known as the vertical and horizontal sync pulse used by the monitor to determine the output resolution on the line.

D. Testing

For the golden measure, the image masking operation was implemented in C++ so that we could set a base runtime which we could compare the runtime of the accelerator against. C++ was chosen due to its high speed and efficiency, as well as the fact that it is syntactically similar to Verilog, and therefore made the comparison easier. The golden measure was run for a total of 5 times and an average runtime was obtained. The Verilog implementation was then simulated in Vivado. The time taken for the simulation to run in Vivado will be compared to the runtime of the golden measure, to determine the speed-up of the IMA system. This allows us to determine whether the FPGA is able to perform the image masking operation faster than the golden measure as we expect it to.

IV. Design

In this section, the design aspects of the image masking accelerator and it's implementation on the FPGA will be discussed. The IMC (Image Masking Controller) drives the main function of the FPGA accelerator and has two different modes:

1) A traditional mode: which does the IMA calculation using internal block ram which is uploaded in COE files and displays the resultant image on the VGA adapter.
2) An advanced mode: which utilizes UART communication to upload images through the use of prompts in a python script on a remote laptop. The user can then interchange any of the three images (image 1, image 2 or mask) with a custom image which is then immediately updated and processed on the VGA monitor.

Although the traditional mode is incorporated into the advanced mode, this section will focus on the functionality of the advanced mode. Figure 3 below shows how the various modules in the system interface with each other.

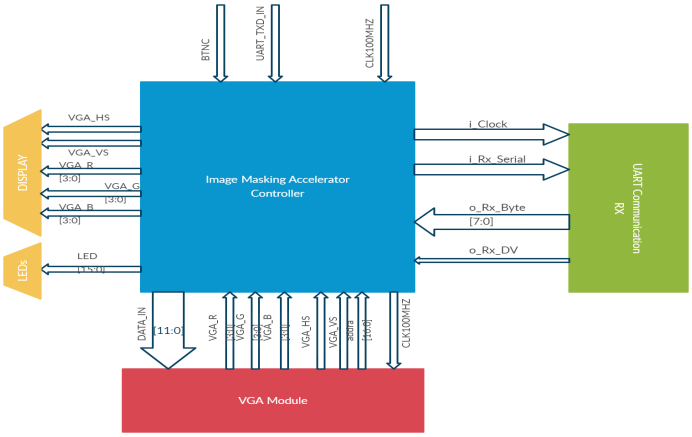

Fig. 3: Interfacing between Modules

A. System Modules

1) Block Memory and Memory Values: Block Memory forms a central part in accessing the pixel data to be manipulated within the program at a pixel level. Storing, accessing and re-storing play a fundamental role in program with respect to the three images (image1, image2 and mask). Figure 4 below is a simplified diagram which explains the logical procedure data travels through before becoming utilized by the program.

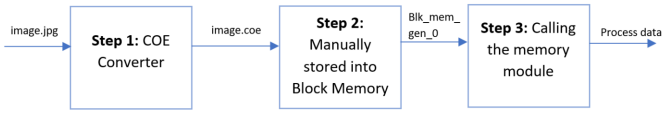

Fig. 4: Logical Procedure Diagram

```
blk_mem_gen_0 bram (     //IMAGE 1
    .clka(CLK100MHZ),      // input wire clka
    .ena(ena),             // input wire ena
    .wea(wea_1),             // input wire [0 : 0] wea
    .addra(addra),         // input wire [16 : 0] addra
    .dina(dina_1),           // input wire [11 : 0] dina
    .douta(data_1)        // output wire [11 : 0] douta
);
```

Listing 3. BRAM Instantiation

COE files are the native format for converting information (pictures, arrays, etc) into a database format. For the image to be stored in memory, the image data must be converted into a COE format using a .coe converter. Once in a COE format, this image data can be loaded into a default IP Catalog Module for Block Memory in Vivado. This image data is now safely stored in memory in an array format which can be easily iterated through in-program and accessed once the block memory module has been instantiated. Instantiating the block memory can be seen in Listing 3 below.

Once the BRAM module has been instantiated, `addra` can be used to iterate through the varying elements within the array and data elements can be accessed through the `data_1` line for the first image.

It should also be expressed the reason for specifically using a 12-bit value for the size of each array element. Each pixel, which is represented by each element in the array, has three channels: R (for Red), G (for Green) and B (for Blue). A combination of the values associated with these channels can create any colour in the colour gamut. Each of these colours hold 4-bits of information about their colour value, with a combined total of 4096 different possible colours, which is suitable for the accuracy of our exercise.

It should also be noted that the `addra` or addresses value is 17-bits long. This is deliberate because the memory should be large enough for each of the pixels to be represented. In this task, the resolution of the output and input images are standardized to 320x240 pixels, which totals to 76800 indexes; this can only be achieved in an array with more than 65536 values which corresponds to the 17-bit length.

2) VGA Adapter Module: VGA (or Video Graphics Array) is a standardized graphical connections protocol to display pixels of information on monitors since the early 2000s. This can be used still today but has been suppressed due to high refresh rate monitors and the technological advancement from original electron-gun based monitors, known as CRT monitors, to LCD monitors. Although technology has developed from CRT monitors, the method of transmission is the same.

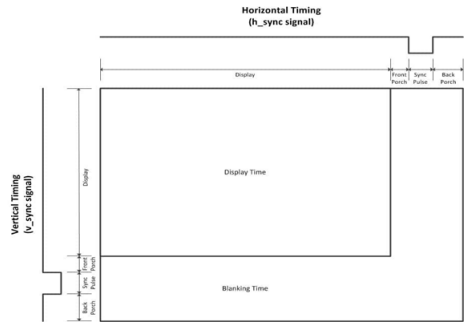

Fig. 5: A diagram of a VGA-based monitor

Figure 5 shows a diagram of a VGA-based monitor. Pixels are graphically represented as a 2D Array by rows (horizontal) and columns (vertical value) when being displayed on the screen. Upon receiving pixel data from the system, the VGA module will, in horizontal fashion, start iterating from the top left to right of the first row before moving onto the second and so forth. This simulates the process of how an electron gun works. The limitation of using an analogue method is that there was is a larger time for the gun to re-position itself from the last pixel in the row, to the first pixel of the next row in comparison to when the pixel was situated directly next to it. This problem was minimized by adding a dark off-screen "blank"-zone which encompassed the delay time it would take to re-position the electron gun. There were two of these, a vertical and horizontal blanking zone. In each of these zones, there are three different segments: Front Porch, Sync Pulse and Back Porch.

Front Porch: A blank time for the left – This takes into account the amount of time used for the electron gun to shift through the blanking zone on the left side of the image.
Sync Pulse: A separate signal line which communicated to the electron gun to start moving into the position of the next line.
Back Porch: A blank time for the right - This takes into account the time used for the electron gun to be shifted through the blanking zone on the right side of the image.

Due to the nature of the front and back porch, the full image resolution of the image will be significantly larger in order to accommodate for the blank zones around the image. In the case of the IMA VGA module, this is twice the value, bringing the resolution to 640x480.

The values in Table I were calculated with a lowered clock speed of 25.175Hz in order to output at the correct refresh rate. The VGA interface can be seen in Listing 4 below.

Scanline/Frame Part	Pixels: (Horizontal)	Lines: (Vertical)
Visible Area	640	480
Front Porch	16	10
Sync Pulse	96	2
Back Porch	48	33
Whole Frame	800	52

TABLE I: VGA Timings

```
vga vga_controller  (
    .CLK100MHZ(CLK100MHZ),
    .data_in(vga_data),
    .VGA_R(VGA_R_out),
    .VGA_G(VGA_G_out),
    .VGA_B(VGA_B_out),
    .VGA_HS(VGA_HS_out),
    .VGA_VS(VGA_VS_out),
    .addra(vga_addra)
);
```

Listing 4. VGA Interface

As it can be seen, the VGA outputs can be divided into three channels, namely: VGAR, VGAG and VGAB. In addition to these, there is a horizontal sync and vertical sync named VGAHS and VGAVS respectively. The pixel information from the program can be sent into the VGA module through the vgadata line.

B. State Machine

The program works in the logic of a state machine and utilizes five different cases/states. Each of these states will be examined and explained in order of the processing.

Diagram 6 is a diagram expressing the different states and their relationship between one another.

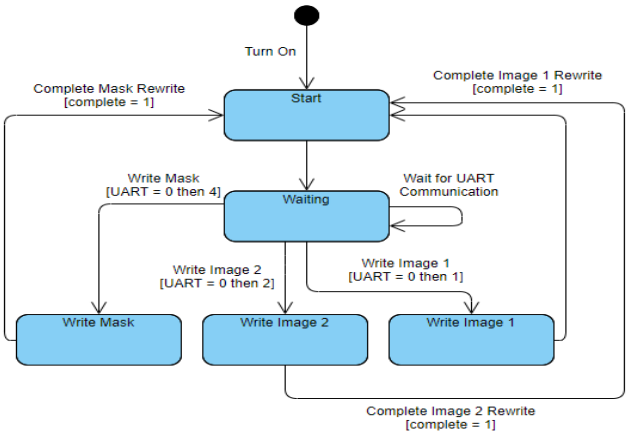

Fig. 6: State Machine Diagram

1. The Start State

The first state, *start*, runs the masking as soon as the system is booted, which processes the images currently stored in Block Ram. The program does this by iterating through the addresses, and independently calls the values stored in block ram (called data_1, data_2 and data_3) to calculate the result. This result, which can be seen in Listing 10, uses the bitwise logic mentioned previously to achieve the pixel value for the final image.

Once this data has been stored in the result_data_in variable and the result_wea goes high, the information will be written to the result memory space.

2. The Waiting State

During the waiting state, the variables pertaining to the start state are reset to be completed again if there is a prompt on the remote laptop as specified in the advanced mode. If there is no prompt, the data stored in result_data_in will be displayed to the VGA and then await a further command.

3. Writing Images States

The next three states are only accessed in the advanced mode. The user is prompted on a laptop by a Python Script whether they would like to interchange one of the images for a custom image. Depending on the prompt value, the image stored in the required directory will replace either image1, image2 or the mask. This new image will then overwrite the block memory values of the image it has just replaced.

The code for the next three states is similar in nature and can be found in Listing 6, with the only difference being that each state accesses a different memory block - for the first image, second image or the mask - which will be written over. The code below is from the first image process but will be explained in a general way as to understand the other cases as well.

Receiver Image Logic

If the state is within either of the write_image_1, write_image_2 or write_mask states, it will use the UART Communication Protocol to receive the transmitted bits of the image from the laptop. The protocol will firstly check whether the transmission of data has begun, and if it has, a pixel counter will be implemented. The purpose of such a counter is that there is a limitation of 8-bits that can be sent through the protocol at any one time. Unfortunately, all the pixel data values are 12 bits long (split into three, 4-bit values of R, G and B). This is overcome by only sending a single 4-bit pixel channel at a time. In order to keep track of the pixel value, this counter will count to three before storing the value in data. This process will continue until all the pixels have been sent through the protocol. At that time, the o_RX_DV line will be raised, which simply acts as an acknowledgment bit that the receiver has acquired all the data. The code can be seen in Listing 7

C. Experimented Builds

1) Mass of Wires Alternative: An alternative method that was optioned and initially tested was the 'Mass of Wires' method. This was substantially easier to implement and used a parallel system but was eventually discarded due to the limitation on our resources within the Nexys A7's

```
if(addra < (ADDRESSES-1)) begin
    addra <= addra + 1;  //reads pixel data from memory
    result_data_in<= (data_1 & data_3) ^ (data_2 & ~data_3);
```

Listing 5. Masking Operation

```
write_image_1: begin
    wea_1<=1;           // Enable write for image 1
    dina_1<=rx_data;
    addra<=rx_address;
    if( !(rx_address < (ADDRESSES-1)) ) begin
        write_address_1<=1; // Write operation complete
        wea_1<=0;
        addra<=0;
        addra_delayed_2<=0;
    end
end
```

Listing 6.

```
pixel_counter<=pixel_counter+1;
 if(pixel_counter==0) pixel_R=uart_data_rx[3:0];
 else if(pixel_counter==1) pixel_G=uart_data_rx[3:0];
 else if(pixel_counter==2) begin
     pixel_B=uart_data_rx[3:0];
     if(! write_address_1) begin
          rx_address <= rx_address+1;
          rx_data<={pixel_R,pixel_G,pixel_B};
     End
end
```

Listing 7. Receiver Image Logic

design, specifically Look-Up Tables (LUTs). These LUTs are simulated at cache-level memory. An example of an operation that heavily depends on LUTs is a For-Loop – in Verilog, For-Loops are unravelled into copies of the body of the For-Loop, in this case a counter and executed concurrently. This is an example of embarrassingly parallel systems.The code of this can be seen in Listing 9.

2) Parallel BRAM Access Alternative: Another method which was discarded was a parallel alternative. Due to the limited amount of Look-Up Tables, each image would be divided into multiple different parts, and stored in different Block Memory segments. This would complicate much of the implementation when designing the IMA around the multiple BRAM segments.

D. Notable Functions

1) Increasing the Speed using a Phase Locked Loop: Phase Locked Loops (PLLs) are a technique that can be used to increase the clock frequency. In theory, the higher the clock frequency, the faster the computations can take place and the faster the benchmark of the IMA would be able to be. Within Vivado, there is a built-in PLL Clock Speed which can be changed to increase the Clock Speed from 100Mhz to 400Mhz. This clock speed was altered using the Xilinx Clocking Wizard which provides two options of whether to drive PLLs or MMTMs. After such a present is chosen, the wizard also informed us about the effective clocking rate which can be achieved. This was used during testing.

2) Debounce Module: Another notable function added was a debounce module that was created for the RESET button. The debounce module was not necessary but brought further

```
parameter ADDRESSES = 78600 ;
 reg [11:0] image1[0:ADDRESSES-1];          //image 1 register
reg [11:0] image2[0:ADDRESSES-1];          //image 2 register
reg [11:0] mask[0:ADDRESSES-1];            //mask register
integer i;

...

for (i=0;i < ADDRESSES; i= i+1) begin
    image1[i] <= (image1[i] ^ mask[i]) & image1[i];
    image2[i] <= (image2[i] ^ mask[i]) & image2[i];
    image1[i] = image1[i] ^ image2[i];
end
```

Listing 8. Mass of Wires Implementation

practical element to the project. The interface for the RESET button within the IMA Controller can be seen in Listing 9.

V. Proposed Development Strategy

As discussed in the introduction, there is an increasing need for edge devices that can perform complex image processing techniques in parallel to speed up the computation of these processing requirements. Currently, our implementation of the IMA does the following:

- Allows the user to select images to be overlaid using UART communication with the FPGA
- Performs XOR operation on image pixels and produces the overlaid output image
- The output image is displayed on a monitor using the VGA interface of the FPGA

In order for the IMA to be developed into a commercial product, a GUI application could be developed to run on the host computer for ease of use. The user would be able to select which images and image mask they would like the image masking operation to be performed on.

Alternatively, instead of using UART communication with the FPGA to load the images onto the system Ethernet could be used instead. The Nexys A7 board includes an SMSC Ethernet PHY that uses the RMII interface and supports 10/100 Mb/s. Therefore, using Ethernet to load the images onto the system would be a lot faster than using UART. The SD card module that we were unable to implement due to time constraints could also be implemented. This would prevent the user from having to store the images and masks that the IMA uses on the host computer.

During the development of the current IMA implementation, it was difficult to find images that were in RGB 12-bit colour (4-bit per component) uncompressed format at a resolution of 320 x 240. If the IMA were to be developed into a commercial product, it would have a wider range of use if user were able to overlay images with various resolutions. This could be possible if the memory available on the FPGA for the image processing was increased, which could be achieved by including a memory controller and physical layer (PHY) interface in the FPGA design so that the DDR2 SDRAM can be used.

```
wire RST_BTN;
Debounce RST (
    .clk(CLK100MHZ),
    .button(BTNC),
    .debouncedBtn(RST_BTN)
);
```

Listing 9. Debounce Module Instantiation

```
time = high_resolution_clock::now() - begin2;
...
std::cout << "Time taken :" << duration<double, std::milli>(time).count() << ".\n";
```

Listing 10. Golden Measure Timing Measurements

VI. Planned Experimentation

As described in the earlier methodology section, there are several implementation options available when approaching the design of necessary masking operations. These approaches include the more simple means sequentially querying and performing operations of individual pixels stored in BRAM, as well as more complex approaches of simulating cache level memory and following the "mass of wires" approach. This section will thus detail the methodology followed when testing such implementations, as well as methods used in generating a golden measure result to which the performance of the accelerator can be compared.

A. Golden Measure Experimentation

In order to access the effectiveness of the accelerator, some sort of bench mark was needed for which the run time results and implementation cost could be compared against. The solution to this was to create a sequential C++ image masking algorithm that completed the same masking operation as the accelerator. While the run time required to read the image and complete the masking is to be measured, an emphasis will be placed on the actual image masking component. The run time is to be measured using the C++ high resolution clock, with the commands shown in Listing 10 to measure run time over particular lines of code. The C++ code will also be compiled with optimization flags set to "O3" is order to produce the best possible run time.

B. VGA Experimentation

One of the challenges involved in implementing the VGA module is the precise timing requirements of outputting over the VGA interface. In order to ensure that these timing requirements are met, a simplified version of the VGA module was implemented without the functionality of receiving data from an external module. This VGA module instantiated an internal BRAM module from which image data would be pulled. A test bench module was then created to instantiate the VGA module, where variable factors within the test bench such as clock speed were set to match the hardware clock speed of the Nexys A7. A behavioural simulation is then run on this test bench module, where the time of the visible area, front porch, synch and such are compared against the industry standard timing for a 640x480 resolution image at 60 Hz.

Once the timings have been confirmed to comply within tolerable amounts of the standard timing, the simplified VGA module undergoes synthesis, implementation and bit stream generation in order to test the output of the VGA module on the Nexys A7 to an actual monitor. The output image that has been preload into BRAM is a bright, colorful image of two birds acquired from shutterstock [bird], and then downscaled to a 16 bit color space.

C. "Mass of Wires" Experimentation

The major concern regarding the mass of wires implementation will be the number of LUTs available on the Nexys A7's. With the ultimate goal of the mass of wires implementation being to perform the required masking operation parallelly over all pixels, we can expect the number of LUTs on the system to at least exceed the number of pixels in the images we plan to work with. This is unfortunately not the case as the Nexys A7 specifications [digilentic] shows that the 100T model variant only has 15,850 Programmable logic slices, each with four 6-input LUTs, for a total of 63400 LUTs. This would ultimately not provide enough head room to simulate cache level memory to store even one of the required images, let alone provide for the logic required to implement the necessary unrolled loops and to control the rest of the system.

While these limitations prevent the system from implementing a solution that is embarrassingly parallel, it was decided that experimentation could be done with varying sizes of simulated cache memories to find out precisely how much of the image could be stored and operated on concurrently. To achieve this, exhaustive testing was carried out whereby memory modules with varying sizes of simulated cache memory and equally sized for loops were synthesised and implemented on the Vivado v2017.6 Design Suite. The design runs of these implementations will be analysed to consider how much simulated memory can be implemented before 100% LUT usage is reached.

D. Parallel BRAM Access Experimentation

Similar to the mass of wires, there was a concern regarding resource utilization on the Nexys A7. While the goal of implementing several BRAM modules, and concurrently reading and writing to such modules seems more achievable than the mass of wires implementation, there is still some hesitation as to how much BRAM can be instantiated on the Nexys A7 100T variant. As shown in the specifications [digilentic], the Nexys A7 100T has 1,188 Kbits. This at first glance may only allow for the equivalent of 1.3 images of the required size to be stored in BRAM.

Once again, exhaustive testing was carried out using synthesis and simulation in the v2017.6 Design Suite. This

testing procedure simply involved instantiating as many BRAM modules as possible of the required size, without exceeding the 100% BRAM usage shown in design runs. Further tests were conducted to find out how many different BRAM modules could be instantiated to hold one complete image, with each BRAM module containing $\frac{76800}{N}$ addresses, where N is the number of instantiated BRAM modules. This will allow us to test exactly how many BRAM modules could be operated concurrently.

E. Sequential BRAM Access Experimentation

This approach was the simplest implementation approach for the accelerator, and hence called for equally as simple experimentation. As described in earlier sections, testing would be carried out in the v2017.6 Design Suite by analysing design runs and resource utilization percentages to confirm that the Nexys A7 100T could comfortably instantiate 4 separate BRAM modules, each capable of storing images of the required size.

While this approach was considered "sequential", there is still a parallel nature to the solution in the sense that each BRAM module containing the two images and the image mask are read from concurrently, while the masking result between two pixels is then stored back into the results BRAM module on the next clock cycle. Hence experimentation here also needs to capture the potential run time of such an implementation. In order to do this, a test bench module will be set up to mimic the normal operation of the image masking accelerator, with variable factors within the test bench such as clock speed to be set to match the hardware clock speed of the Nexys A7. This simulation run will not include upload of images to the accelerator by the user, and will instead focus on measuring time to mask the images preloaded into the BRAM modules. These results will be compared against that of the golden measure.

Furthermore this module will then undergo synthesis, implementation and bitstream generation so that the final result can be viewed for accuracy on a monitor via the VGA interface. Once the masking operation is deemed to have the correct upload, different images and images masks will be uploaded to the accelerator via UART communications [script] and the displayed image checked for accuracy.

VII. Results

A. Golden Measure Results

Table II confirms our expectations the run-time of the C++ code set to compile with optimisation flags set was significantly shorter than that without the optimisation flags set. The final results of the two images were compared visually and found to have both produced the expected output.

While we would generally take the benchmark time to be that of the unoptimised code, we will take our golden standard measurement to be the optimised code as their was no additional effort needed for its development, and was simply an optimisation flag. Thus the final golden measure run-time against which we will compare the image masking accelerator results is $0.028\ ms$.

B. VGA Results

Simulation results for the VGA module are shown in figures 1 - 3. In these simulations we tested the timing constraints of the VGA module against the industry standard timings for a 640 x 480 image at 60 Hz. Particularly, paying attention to the pixel frequency, the duration of time before that passes before the vertical and horizontal sync pulses are activated, as well as the duration of the vertical and horizontal sync pulses. All figures have markers placed in the appropriate locations, showing the time's at which the markers occur, as well as showing the time difference shown at the bottom of each simulation.

Figure 1 shows that the pixel clock period is $40ns$, hence we can determine that the pixel frequency produced by the VGA module is $\frac{1}{40\times10^{-9}} = 25MHz$. While this is not perfectly in line with the industry standard of $25.175MHz$, this is deemed acceptable as many monitors are a means of dealing with slight inaccuracies due to timing. Furthermore, the method used for implementation whereby counters to slow down the pixel clock restricts our ability to slow down the system clock by positive integer factors, meaning achieving a frequency of $25.175MHz$ is simply not possible.

Figure 8 shows the recorded results for the horizontal sync timing. From the figure we are able to see that the sync pulse starts at $26.23\ \mu s$. From the industry standard, this sync pulse occurs after the visible area and front porch times have both passed, hence starting at $25.42\ \mu s + 0.64\ \mu s = 26.06\ \mu s$. Thus the sync pulse starts within a tolerable time frame. Furthermore the industry standard time for which the sync pulse remains high is $3.81\ \mu s$, where our sync pulse period is $3.84\ \mu s$. This is within a tolerable amount of the industry standard.

Run	**Unoptimised Benchmark (ms)**	**Optimised Benchmark (ms)**
1	0.377	0.030
2	0.378	0.035
3	0.384	0.025
4	0.400	0.025
5	0.380	0.025
Average Time (ms)	0.383	0.028

TABLE II: Golden Measure Results

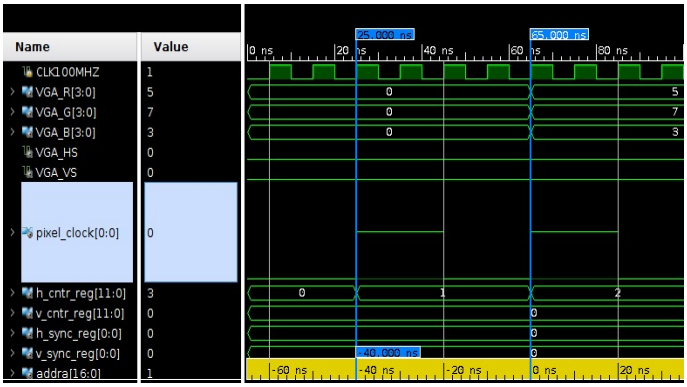

Fig. 7: VGA Pixel Clock Timing

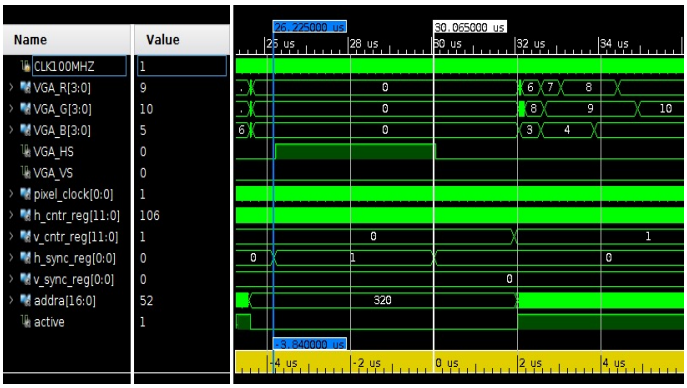

Fig. 8: VGA Horizontal Sync Timing

Figure 9 shows the recorded results for the vertical sync timing. From the figure we are able to see that the sync pulse starts at $15.65\ \mu s$. From the industry standard, this sync pulse occurs after the visible area and front porch times have both passed, hence starting at $15.25ms + 0.32ms = 15.57ms$. Thus the sync pulse starts within a tolerable time frame. Furthermore the industry standard time for which the sync pulse remains high is $0.06ms$, where our sync pulse period is also $0.06ms$. This is within the tolerable amount of the industry standard.

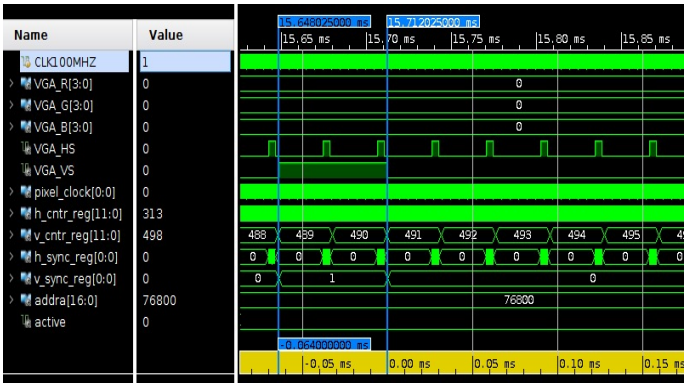

Fig. 9: VGA Vertical Sync Timing

Figure 10 shows the result of displaying an image to a monitor over the VGA interface. As we can see from the photogram, the displayed image is in line with the expected result of the image obtained from shutterstock. While the image does not contain the smooth colour gradients visible in the background of the image, this was expected as the colour space has been dramatically reduced and is no longer able to show the fine transitions between similar colours.

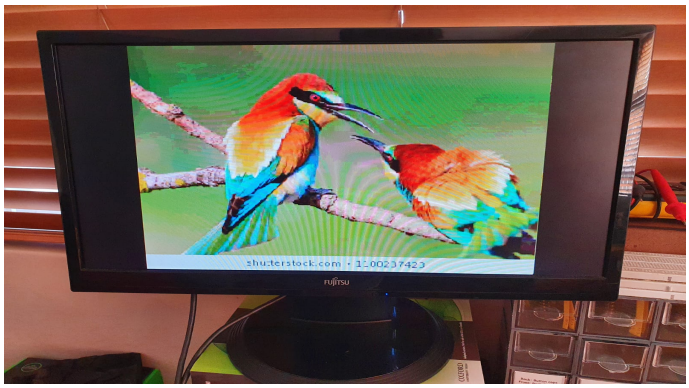

Fig. 10: VGA Display Test

C. Sequential BRAM Access Experimentation

Due to time limitations, the only working image masking accelerator we managed to implement was one that made use of sequential BRAM access. As mentioned in earlier sections, while BRAM addresses could only be accessed every clock cycle, this was able to be done for each image concurrently. Hence we were able to access all necessary memory addresses and perform the image masking operations of each pixel in a single clock cycle, where the result was able to be stored within the separate BRAM module instantiated specifically for storing results.

Test bench simulations showed that using the $100\ MHz$ internal clock of the Nexys A7 resulted in a total time of $0.768\ ms$ being needed to perform image masking over the entire image. This is in line with our predicted value of $0.76\ ms$, where the extra time can be associated with the transitioning within the state machine. This $100\ MHz$ internal clock was then used to drive the a PLLs generated by Xilinx Clocking Wizard IP. Preliminary simulation results showed that run time needed for masking decreased linearly with the increased clock speed produced by the PLL. This worked up to a limit of $1000\ MHz$ where the Xilinx Clocking Wizard warned that we were unable to achieve the required clock speed. We were thus able to reduce the total time needed for masking in simulation to $0.077\ ms$. Unfortunately these results proved to be futile as once this was implemented on the Nexys A7, the results displayed on the monitor was subject to visible artifacts.

In the end we were only able to increase the internal clock speed by a factor of two while still retaining accurate results, ultimately bringing our run-time down to $0.384\ ms$.

The results produced by this experimentation can be seen in Figures 11 - 13 where Figures 11 and 12 show the images and mask uploaded to the FPGA. Figure 13 shows the final result seen displayed on the monitor.

D. "Mass of Wires" Results

In testing the mass of wires implementation, we quickly ran into problems. While simulation results showed the execution time of the masking functionality to be absolutely superb, generally completing in 2 - 4 clock cycles (30 ms), there were severe limitations placed on the number of pixels that could be stored and concurrently operated on due very demanding utilisation of BRAM by this approach. Table III shows this more clearly, where 98 $\%$ LUT usage was reached for the Nexys A7 100T with a memory configuration size of only 14,000. This means that we would only be able to store 6.08 $\%$ of each of the three required images.

While working on small fractions of the image and incrementally building the final result still held potential for incredibly quick computation of a masked image, these options were not explored as they are incredibly costly in terms of development time.

E. Parallel BRAM Access Results

Much like the sequential BRAM access implementation, the parallel BRAM implementation called for sequential access of the BRAM modules, with the difference being that in this implementation many smaller memory modules would be made to hold different portions of each image. While this seemed like an promising approach, we were faced with similar issues to that faced with mass of wires approach. With the golden run-time measure being at 0.028 ms and the sequential access run-time being measured at 0.384 ms, the sequential access result would have to be improved by a factor of 14 to compete with the golden measure. The implication of this being that each of the 3 images as well as the result would need to be manually split into 14 different BRAM modules, for a total of 56 BRAM modules. While implementation of such a solution was deemed to be possible (the amount of BRAM utilization is essentially the same as storing 3 images and a result image), the amount of development time required was deemed for too high considering the time constraints faced by the team.

Fig. 11: Images Used for Image Masking

Fig. 12: Images Used for Image Masking

Fig. 13: Images Used for Image Masking

VIII. Conclusion

While this investigation proved that implementation of a hardware accelerated image masking solution might be possible using an approach such as the "mass of wires" approach discussed, such an implementation was not achieved. Instead only a partially parallel implementation was achieved whereby we were unable beat the run-time achieved by the C++ golden measure.

We can easily acknowledge that FPGAs such as the Nexys A7 100T hold massive potential to solve embarrassingly

Size of Memory Structure and Number of Iteration in Unrolled For Loop	**LUT Usage (n)**	**LUT Usage (%)**
1000	4,469	7.05
2000	8,903	14.04
7000	31,123	49.09
10,000	45,513	70.21
14,000	62,25	98.2

TABLE III: Mass of Wires Resource Utilization

parallel problems due to their parallel nature, though the harsh limitations placed on the amount of data that can be stored on such FPGAs end up creating unique problems themselves, driving up the development time needed to develop strategies around such limitations. In retrospect, we could have split up the work more evenly into investigating the different potential implementations. In particular we would like to have focused on the ”mass of wires” implementation to see how much of a performance increase we could have achieved.

REFERENCES

[1] L. M. M.Lock, L.Barbas. Image-masking-accelerator. Git. [Online]. Available: https://github.com/matthew-william-lock/Image-Masking-Accelerator

[2] J. R. Ian Kuon, “Quantifying and Exploring the Gap Between FPGAs and ASICs.” Springer, 2010, pp. 39–58, ISBN 978-1-4419-0739-4.

[3] B. Eater. The world’s worst video card? Youtube. [Online]. Available: https://www.youtube.com/watch?v=l7rce6IQDWs

[4]