YODA Project Status Update

Image Masking Accelerator
Luca Barbas
Lawrence Godfrey
Mahmoodah Jaffer
Matthew Lock

Source: Adapted from [1]

Declaration:

1. We know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. We have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this essay/report/project/ from the work(s) of other people has been attributed, and has been cited and referenced. Any section taken from an internet source has been referenced to that source
3. This essay/report/project/assignment is our own work, and is in our own words (except where we have attributed it to others).
4. We have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work
5. We acknowledge that copying someone else's assignment or essay, or part of it, is wrong, and declare that this is our own work.

Signature : Luca Barbas, Lawrence Godfrey, Mahmoodah Jaffer, Matthew Lock

Table of Contents

Abstract

This document serves to give an update on the progress made thus far in implementation of an Image Mask Accelerator (IMA). This progress update will include a general overview on work completed thus far, work that is planned to be completed in the near future, as well as supporting design documents and code examples to show completed work or proposed plans.

Progress Update

There have been several points of progress made thus far. While we have yet to implement a working rudimentary IMA on the Nexys A7 FPGA, we have validated the proposed concept by means of a golden measure as shown in the list below. We have also implemented a working VGA Adapter Module for the FPGA and will be able to show any processed images. Lastly we have implemented a rudimentary Memory Controller Module which is responsible for loading images from BRAM into the IMA Module. All the progress made thus far has us firmly in line with meeting our targets set out by Plan B of our project proposal, with firm beliefs that we will meet the requirements of Plan A.

List of modules or simulations carried out thus far:

1. C++ Golden Measure Image Masker
2. VGA Adapter Module
3. Basic Memory Controller Module

Golden Measure

A golden measure implementation has been implemented in C++ in order to set a base runtime which we can compare the accelerator against. C++ was chosen due to its high speed and efficiency, as well as the fact that it is similar, syntactically, to Verilog, and will therefore make the comparison easier.

The C++ program reads in three images. Each image has a resolution of 320 x 240, and a bit-depth of 12 bits, as was specified in the brief. The first two images are the images to be overlaid using the mask. The third image is the mask itself, which is made up of only black and white pixels.

The images are read into unsigned char arrays of length $(320 \times 240) \times \frac{12}{8} = 115200$. Then, each unsigned char in the output buffer is calculated using the following equation.

$$O_i = (I_i^1 \bullet \neg M_i) \oplus (I_i^2 \bullet M_i)$$

$$Where$$
$$O_i \text{ is the output image at } i$$
$$I_i^k \text{ is image } k \text{ at index } i$$
$$M_i \text{ is the mask at index } i$$

This logic can be implemented in C++ with the following code.

```
for(int i = 0; i < length; i++) {
      output[i] = (image1[i] & ~mask[i]) ^ (image2[i] & mask[i]);
}
```

The first part of the equation $(I_i^1 \bullet \neg M_i)$ ANDs the inverted mask with the first image. The output of this would be the same image, except that for every white pixel on the mask, the output image will be black.

The second part of the equation $(I_i^2 \bullet M_i)$ does the same thing, but since the mask isn't inverted here, the output will be black wherever the mask is black.

These two outputs are then XOR'd to produce the final output. The whole process is shown conceptually in Figure 1.0.

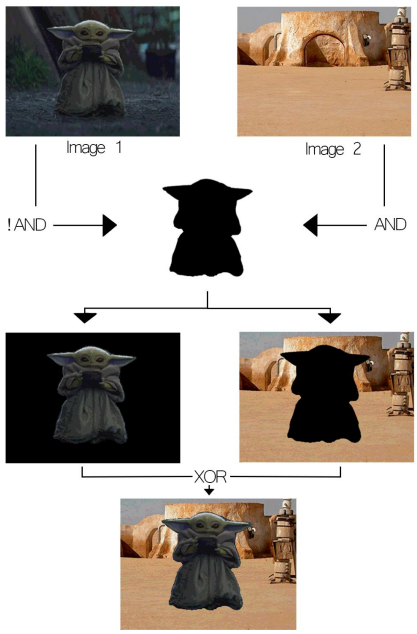

Figure 1.0: Conceptual Diagram of Masking Operation

The following table shows the runtime for the masking operation over 5 runs. Note this does not include the time taken for file input and output.

Run	Time (ms)
1	0.377468
2	0.378788
3	0.383979
4	0.400129
5	0.379729

Table 1.1: Golden measure benchmark

VGA Adapter Module

The VGA Adapter Module was created using the timing principles laid out by Ben Eater [2]. The basic principles of which required a pixel clock to determine not only how long each pixel should be displayed for, but for how long the transitions should be between concurrent vertical lines of pixels. As shown in Figure 1.1, general timing requirements of the VGA output require that there must be a period of “off time” between transitioning between vertical lines, as well as from the end of a frame to the start of the next frame. Furthermore there is a vertical and horizontal sync pulse required to help the connected monitor with timing.

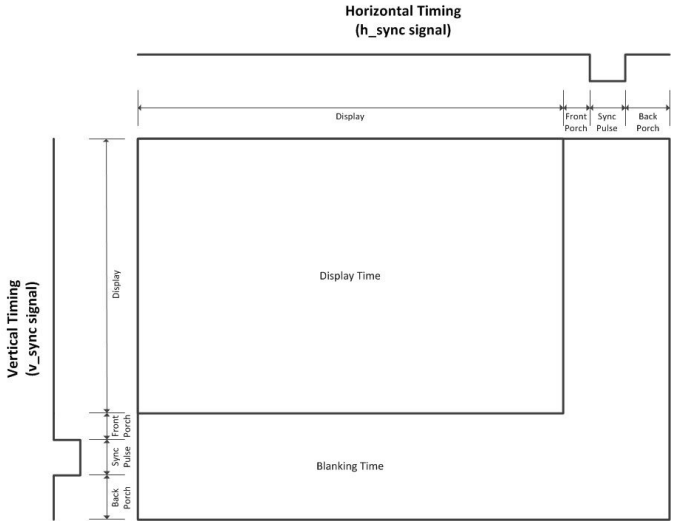

Figure 1.1 : General VGA Timing Diagram [3]

The times shown in these Figure 1.1 are specific to each resolution. Since there are no readily available timing requirements accessible for the required 320x240 resolution, a resolution of 640x480 was selected as it is an even multiple of the required resolution. Specific timing requirements for this resolution is shown in Figure 1.2. As can be seen a pixel clock frequency of 25.175 MHz was required. The Nexys A7 FPGA provides a 100 MHz clock and it was not possible to precisely slow the clock down to the required rate, but was expected to work if it was close enough.This was confirmed in testing as it was found that a pixel clock of 25 MHz worked perfectly to produce the required image. Example code of how the pixel clock was created can be seen in Figure 1.3 while Figure 1.4 shows how the vertical and horizontal sync pulses were created.

VGA Signal 640 x 480 @ 60 Hz Industry standard timing

General timing

Screen refresh rate	60 Hz
Vertical refresh	31.46875 kHz
Pixel freq.	25.175 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Vertical timing (frame)

Polarity of vertical sync pulse is negative.

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.31777557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.0486593843098
Whole frame	525	16.683217477656

Figure 1.2 : VGA Timing Requirements for 640x480 Resolution [4]

```
// ----------------------------------------------------------------
  // Generate Pixel Clock
  // ----------------------------------------------------------------

  always @(posedge CLK100MHZ) begin
      clock_cntr_reg <= clock_cntr_reg+1'b1;
      if (clock_cntr_reg>=2)pixel_clock<=1;
      else pixel_clock<=0;
  end
```

Figure 1.3 : Pixel Clock Generation

```
// ----------------------------------------------------------------

// Generate Horizontal, Vertical counters and the Sync signals

// ----------------------------------------------------------------

// Horizontal counter
always @(posedge pixel_clock) begin
    if( (h_cntr_reg == (H_MAX-1)) ) h_cntr_reg<=0;
    else h_cntr_reg<=h_cntr_reg+1'b1;
end
// Vertical counter
always @(posedge pixel_clock) begin
    if( (h_cntr_reg == (H_MAX-1)) && (v_cntr_reg == (V_MAX - 1)) )
v_cntr_reg<='d0;
    else if(h_cntr_reg == (H_MAX-1)) v_cntr_reg  <=v_cntr_reg+1;
    end
// Horizontal sync
always @(posedge pixel_clock) begin
   if( (h_cntr_reg >= (H_FP + FRAME_WIDTH - 1)) && (h_cntr_reg < (H_FP +
FRAME_WIDTH + H_PW - 1)) ) h_sync_reg<=H_POL;
   else h_sync_reg<=!H_POL;
   End
// Vertical sync
always @(posedge pixel_clock) begin
    if( (v_cntr_reg >= (V_FP + FRAME_HEIGHT - 1)) && (v_cntr_reg < (V_FP +
FRAME_HEIGHT + V_PW - 1)) ) v_sync_reg<=V_POL;
    else v_sync_reg<=!V_POL;
end
```

Figure 1.4 : Vertical and Sync Pulse Generation

Simulations were done to test timing accuracy. Figure 1.5 shows a simulation in which we can determine that the pixel clock has been successfully set to 25 Mhz, while Figure 1.6 shows that the horizontal sync occurs as roughly the correct time of 26 us (visible area time + front porch time).Figure 1.7 then shows a working implementation of the module. Furthermore, while this module has been implemented and tested, it only works as an independent module that implements and displays an image from its own internal BRAM. Work is still to be done to implement an integrated VGA module. Plans for this are listed under “Future Plans”.

Code for the full implementation can be found at our git repository [5].

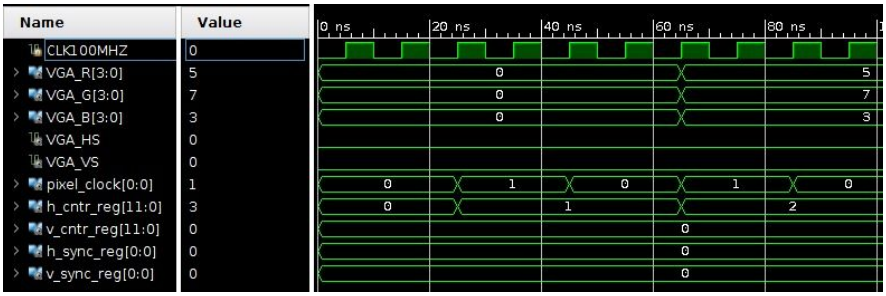

Figure 1.5 : Simulation showing 25 MHz Pixel Clock

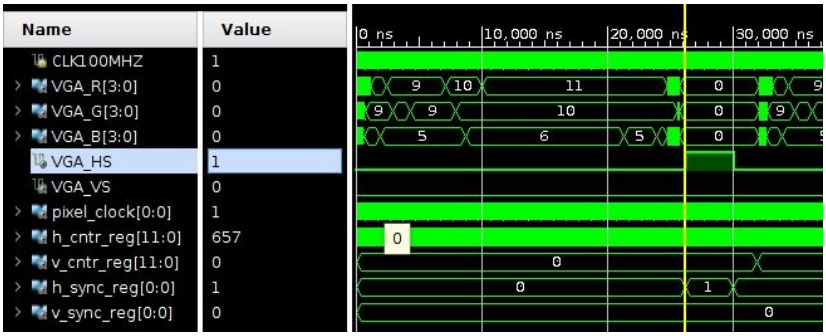

Figure 1.5 : Simulation Showing Horizontal Sync Time

Figure 1.7 : Working VGA Adapter Module

Basic Memory Controller Module

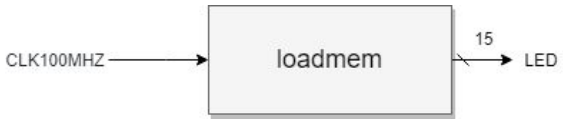

Figure 1.8: Block Diagram showing input and output of loadmem module

The block diagram of the loadmem() module can be seen in Figure 1.8 above. This module is mainly responsible for instantiating the three block ram modules with their corresponding '.coe' files that are going to be operated on – these files are incredibly large, comprising 76800 elements each. The main function parses these large datasets through the bram module to populate functional arrays called image1, image2 and mask in the iamge masking module. These images can then be manipulated on a bit-level in order to produce the desired output and displayed through a VGA module.The flowchart in Figure 1.9 shows how the memory controller module - loadmem() - operates.

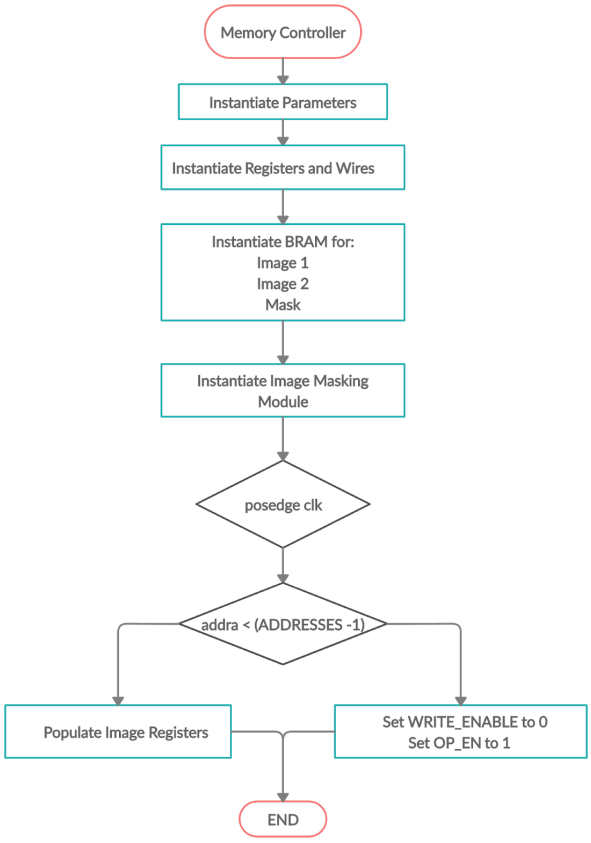

Figure 1.9: Flowchart for loadmem module

The .coe files used for the project are of a 320x240 resolution with a 12-bit pixel value (R-4, G-4, B-4). The result of this results in the image resolution is 76800, the number of pixels that will be displayed on-screen.

```
1.      //instantiate parameters
2.      parameter ADDRESSES = 76800 ;
```

Although .coe files store the information in a suitable format for the block ram module, this is unable to be manipulated on a pixel level unless loaded into an array local to the program. In order to do this, we call the bram module and parse the .douta as “data_1” to populate the image registers in the BRAM module. Below is simply the code from a single block ram module but there are three in total.

```
1.     blk_mem_gen_0 bram (   //IMAGE 1
2.       .clka(CLK100MHZ),    // input wire clka
3.       .ena(ena),           // input wire ena
4.       .wea(wea),           // input wire [0 : 0] wea
5.       .addra(addra),       // input wire [7 : 0] addra
6.       .dina(dina),         // input wire [10 : 0] dina
7.       .douta(data_1)        // output wire [10 : 0] data_1
8.     );
```

The code for the main function in this module works as follows: Once all the data values have been parsed from the COE file in Block Ram to the arrays in Bram Module - when the *addra* == *ADDRESSES* – the WRITE_ENABLE pin will go low and OP_EN will go high, signalling the bitwise operations to commence in the slave.

```
1.      //Main function
2.      always@(posedge CLK100MHZ)
3.      begin
4.          // Step 1: Populate the image registers with BRAM data
5.          if(addra < (ADDRESSES-1)) begin
6.              WRITE_ENABLE=1;
7.              addra <= addra + 1;
8.          end
9.          // Step 2: Perform image masking process using bit operations
10.         else begin
11.         WRITE_ENABLE<=0;        //stops triggering image register
population
12.         OP_EN<=1;               //triggers bit operations
13.         end
14.   end
```

Future Plans

In order to fulfil the requirements set out by Plan B of our project proposal , some work still needs to be done. This includes modifying some existing modules to accomodate for integration with other modules, as well overcoming the synthesis difficulties associated with some modules. These plans are outlined below according to their specific modules.

VGA Adapter Module

The VGA Adapter Module needs to be modified in order to display image data provide by other modules. Currently the module works by displaying an image stored on an internally instantiated BRAM module. This BRAM module will hence be removed and two additional input and output lines will be created for the module. One shall output the current pixel to a connected module to indicate which pixel is needed in order for display. The calculation of this pixel count is shown below:

```
Pixel_count <= ( ((v_cntr_reg_dly/2)*(FRAME_WIDTH/2)) +(h_cntr_reg_dly)/2 )
```

Furthermore a data_in line will be added to receive the display data for the specified pixel. Since the pixel clock is slowed down by a factor of 4, this currently proposed plan for the VGA controller will work as long as the connected modules will be able to pass in the required data for the specified pixel_count within 4 clock cycles of the main system clock. Figure 2.1 Shows the proposed design.

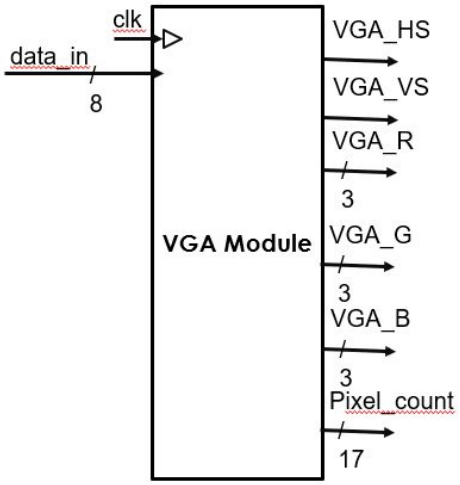

Figure 2.1 : Proposed Modified VGA Adapter Module

IMA Module

The block diagram of the bram() module can be seen in Figure 2.2 below. The flowchart in Figure 2.3 shows how the image masking module - bram() - operates.

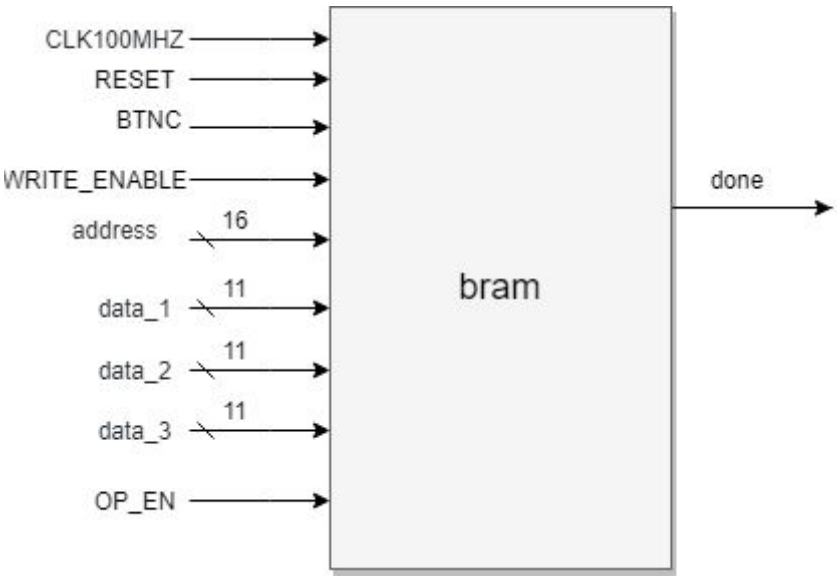

Figure 2.2: Block Diagram showing inputs and output of bram module

When WRITE_ENABLE has been triggered high from the last program, the values from the BRAM modules are loaded into the i^{th} element of the image1, image2 and mask arrays. Upon completion, the bitwise operations are commenced. These operations are the same for both image 1 and 2 – firstly an XOR operation with the mask (or inverse mask for image2), and then an AND operation with itself. The result of these two final images are XOR'd again with each other to result in the image mask originally desired.

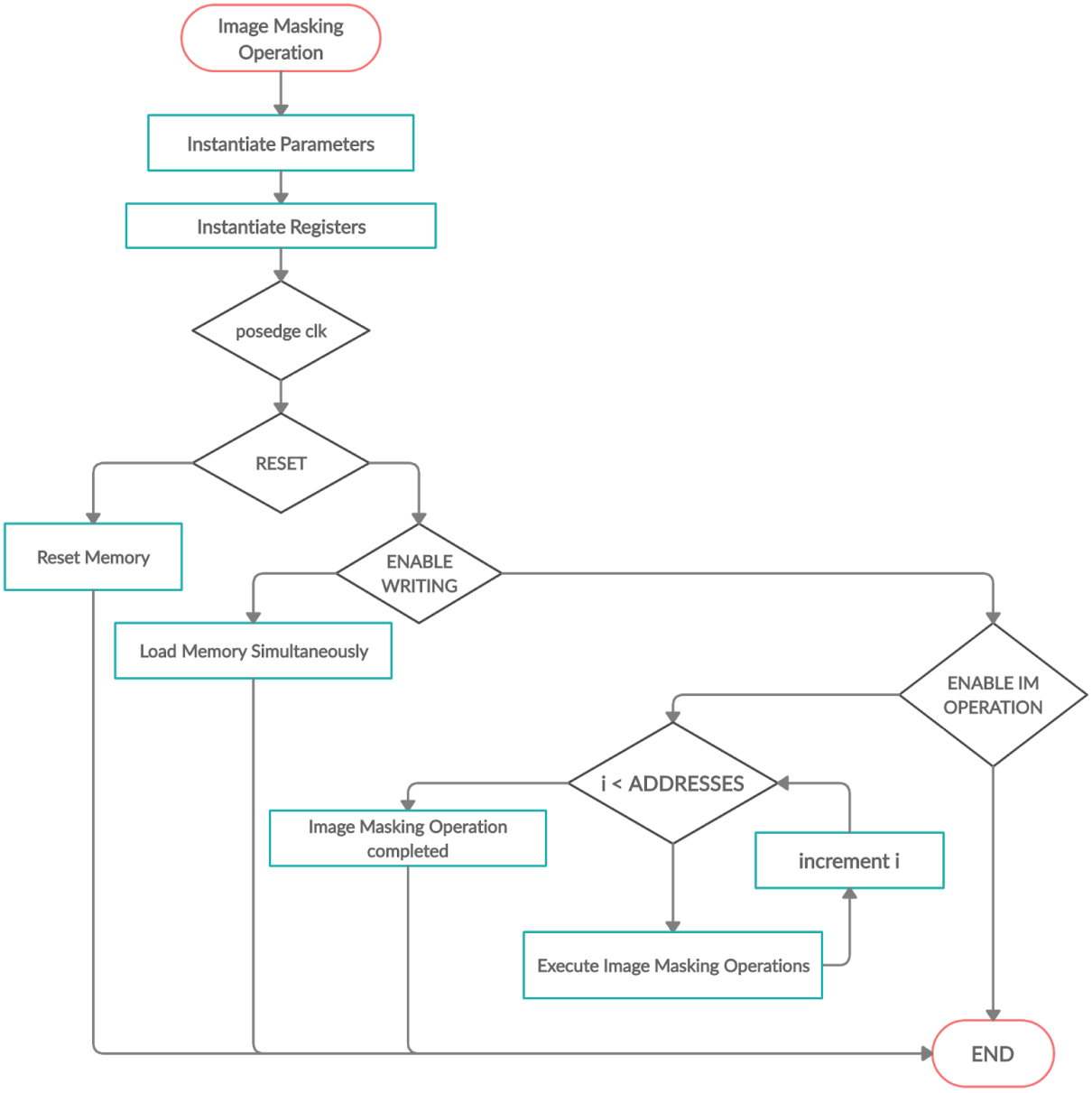

Figure 2.3: Flowchart for bram module

```
1.      always@(posedge CLK100MHZ) begin
2.          if(RESET) begin
3.              done<=0;
4.              // clear memory
5.          end
6.          else if (WRITE_ENABLE) begin
7.              // Load each memory item into address i simultaneously
8.              image1[i]<=data_1;
9.              image2[i]<=data_2;
10.         mask[i]<=data_3;
11.         end
12.         else if(OP_EN) begin
13.         // Execute operation
14.         for (i=0;i < ADDRESSES; i= i+1) begin
15.               image1[i] <= (image1[i] ^ mask[i]) & image1[i];
16.               image2[i] <= (image2[i] ^ (~mask[i]) ) & image2[i];
17.               image1[i] = image1[i] ^ image2[i];
18.         end
19.         //operation has finished and awaiting VGA call
20.         done <= 1;
21.         end
22.   end
```

Various considerations and challenges resulted in the way the above method was implemented, these relate to the constraints in the size of block memory as well as the limitation on the amount of look-up tables – which limits the methods of calculation and the speed at which the FPGA can process the image operation.

It should be noted that the current code is still a work in progress and changes may occur before the final version.

References

[1] Clipart Library. 2020. [online] Available at: <http://clipart-library.com/clipart/65715.htm> [Accessed 10 June 2020].

[2] B. Eater, “The world’s worst video card?” Jul 2018. [Online]. Available: https://www.youtube.com/watch?v=l7rce6IQDWs

[3] Tolerances, V. and Frost, P., 2020. VGA Decoding - Dealing With Tolerances. [online] Electrical Engineering Stack Exchange. Available at: :https://electronics.stackexchange.com/questions/92900/vga-decoding-dealing-with-tolerances> [Accessed 10 June 2020].

[4 Tinyvga.com. 2020. VGA Signal 640 X 480 @ 60 Hz Industry Standard Timing. [online] Available at: <http://tinyvga.com/vga-timing/640x480@60Hz> [Accessed 14 June 2020].]

[5] L.Barbas, L.Godfrey, M. Jaffer, M.Locl ”, “Ima project proposal,” May 2012. [Online]. Available: https://github.com/matthew-william-lock/Image-Masking-Accelerator/blob/master/vga_controller/picture_display/vga.srcs/sources_1/new/vga.v