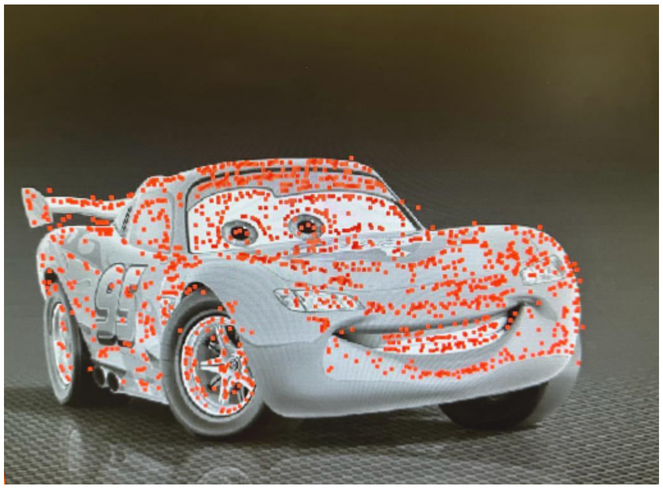
# CSEE 4840 Embedded Systems Design Final Project Report

# Team qSIFT

**Spring 2023**

| Name | UNI |
|---|---|
| Madhav Bhat | mb4989 |
| Khushi Gupta | kg3023 |
| Prathamesh Sahasrabudhe | ps3320 |
| Jeffrey Wolberg | jnw2138 |
| Daniel Seligson | ds3824 |

## Supervisor - Prof Stephen Edwards



## *Table of Contents*

# ***Introduction***

The Scale Invariant Feature Transform (SIFT) is a Computer Vision (CV) algorithm that is widely used in image processing for feature detection and description. It detects and describes image features in scale and rotation invariant encoding, and therefore can be used to correctly register images taken from different distances, angles, and lighting conditions.

To detect the key points, the SIFT algorithm implements a series of steps, namely, creating the scale space pyramid, the difference of Gaussians, and the detection of local extrema. These generated key points are then assigned a unique descriptor based on the magnitude of the nearby pixels and the orientation of the gradient.

The SIFT algorithm can be broadly divided into the following steps:

1. Scale-space peak selection: First, the input image is converted into a grayscale image. Then, multiple scaled-down versions of the image are created. Next, each of these scaled images is convolved with different-sized Gaussian filters. Finally, the Difference of Gaussian (DoG) is computed for each set of the scaled images by subtracting each image from the previous one in the same scale level. This process helps identify potential locations for features.

2. Keypoint localization: The DoG images are examined to accurately locate feature keypoints by identifying the locations with the highest or lowest intensity values. These locations are then refined using an iterative approach.

3. Orientation assignment: An orientation is assigned to each keypoint by analyzing the gradient direction of the nearby pixels.

4. Keypoint descriptor: A descriptor is computed for each keypoint by considering the gradient magnitudes and orientations of the pixels within a certain radius around the keypoint. This results in a high-dimensional vector representation of the key point.

Overall, the SIFT algorithm is a powerful tool for detecting and describing features in images, and it has numerous applications in fields such as computer vision, robotics, and augmented reality. SIFT is used in wide computer vision applications such as image stitching, object recognition, and modeling.

## ***System Overview***

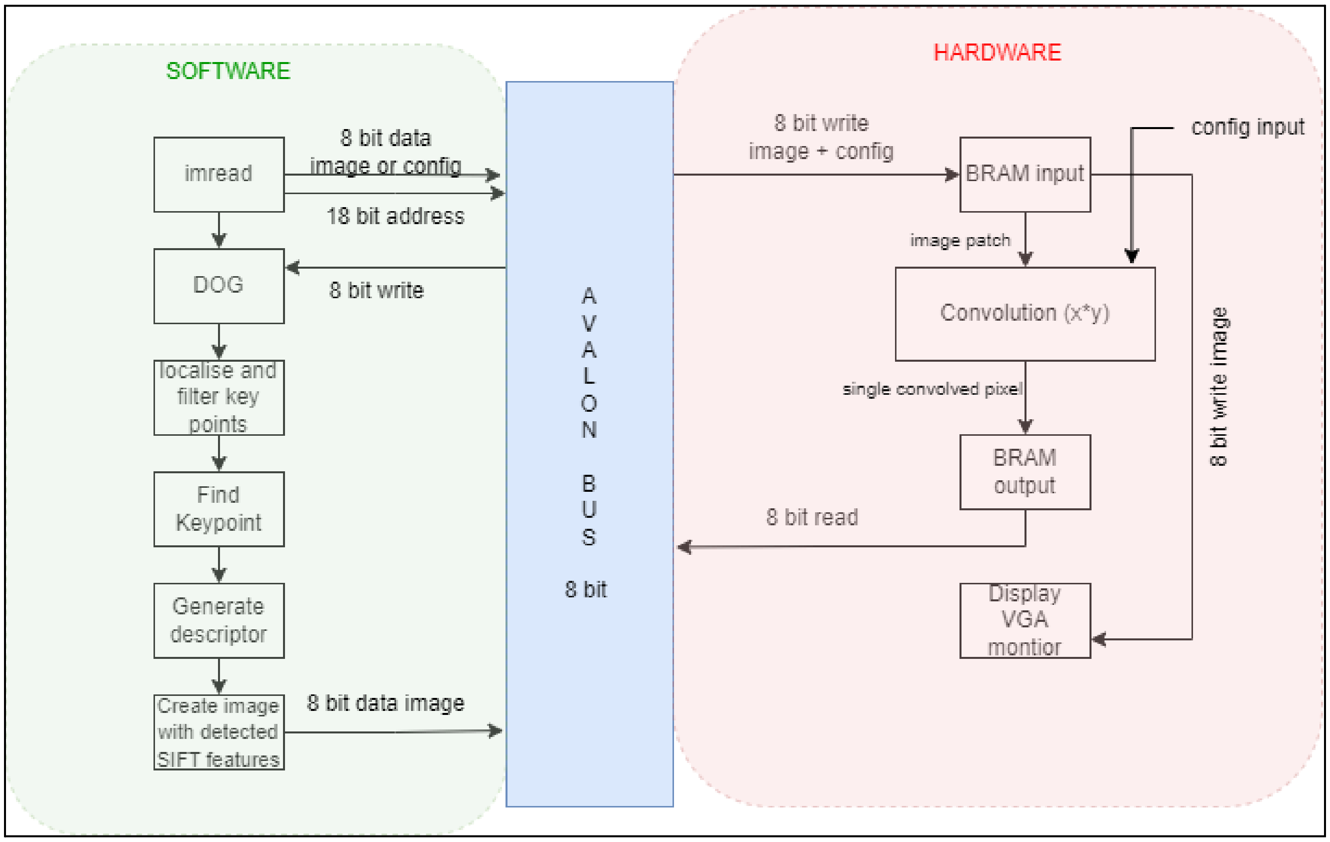


*Fig 01 – System Design & Block Diagram*

1. The SIFT algorithm is divided into five steps as shown in the block diagram above, starting from taking the Difference of Guassians, localizing and filtering the key points, assigning the orientation to the key points, generating the descriptors, and finally creating the output image with the detected SIFT features.
2. In our project we have implemented the first step of the algorithm, the most computationally expensive one, which is performing the convolutions (Gaussian Blurring) on the hardware. The remaining steps of the algorithm are implemented in software.

3. First, the software reads the input grayscale image and sends it through the Avalon Bus to the Block RAM on the hardware.
4. We hardcode the kernel coefficients of all the kernels (9x9, 11x11, 15x15, 19x19, 25x25) and store it on the on-chip memory.
5. The hardware takes the image and performs the convolution using a kernel (the size of the kernel is specified by the software) and stores the output image in another BRAM of the same size.
6. The output image data in the BRAM is then sent via the Avalon bus back to the software from where it can generate the Difference of the Gaussians (DoGs) which are utilized for further processing.
7. Only after the data has been sent back completely we begin the next convolution so as to avoid incorrectly modifying valid data in the BRAM.
8. During this process when the convolution is not being performed and data is being sent from the output BRAM to the hardware to the software, we simultaneously display the original image in the input BRAM on the VGA.
9. Once the software receives the Gaussian Blurred Images from the 5 kernels, and it has executed the remaining algorithm, it sends the final image with the SIFT features back to the input BRAM from where we can display it on the VGA.

## ***<u>Hardware Software Interface</u>***

The image, presumed to be grayscale and binary, is initially read in main.c from the Linux filesystem. It is then loaded into a 1-D array of size 525x350, where each element represents an 8-bit (char) value. This array serves as the storage for the image data.To process the image, the stored data is sent to the hardware module called ‘vga_ball.sv’. This transfer occurs over the Avalon bus, utilizing an 8-bit wide writedata channel and an 18-bit wide address field. The image data is written to the hardware memory through this interface.

Subsequently, the software can read the image data from the hardware memory using the same Avalon bus and address field, but with a separate 8-bit wide readdata channel. This allows the software to access the image data stored in the hardware module. The software controls the convolution parameters directly through two configuration bytes. One byte is dedicated to specifying the kernel, while the other byte determines the octave (image scale) to be used. The first bit of the kernel configuration byte is responsible for triggering the start of the convolution process.

This design simplifies the transfer of image data between the software and hardware modules by leveraging the Avalon bus, which uses the 8-bit writedata, readdata, and 18-bit address fields. By directly controlling the convolution parameters and utilizing this bus interface, more complex memory protocols like Direct Memory Access (DMA) or SDRAM Controllers can be avoided.

## ***Hardware Design***

The hardware is responsible for doing the Gaussian kernel convolutions that blur the image, which is computationally expensive. To give some context, convolving a 600x400 image by a 25x25 kernel requires 150 million multiplications. SIFT requires 20 such blurs at different image and kernel sizes. Therefore, accelerating the convolution in hardware can significantly speed up the SIFT algorithm.

Depending on how blurred the image must be, different standard deviations will be used in the equation $G(x, y, \sigma) = \frac{1}{2\pi\sigma} e^{-\frac{(x^2+y^2)}{\lambda\sigma^2}}$ used to create the Gaussian kernels. These kernels are matrices of different sizes and are used as the weights in a weighted average of the surrounding pixels centered at a given (row, col). A kernel is convolved for each (row, col) in the image, creating a new blurred image. Since these kernels will be reused for all images passed through the algorithm, we will calculate the values for these apriori and hardcode them into the embedded memory.

The final image with the overlaid SIFT key points is then displayed on the VGA screen by sending the data through the same 8-bit write data bus.

## ***Convolution Block in Hardware***

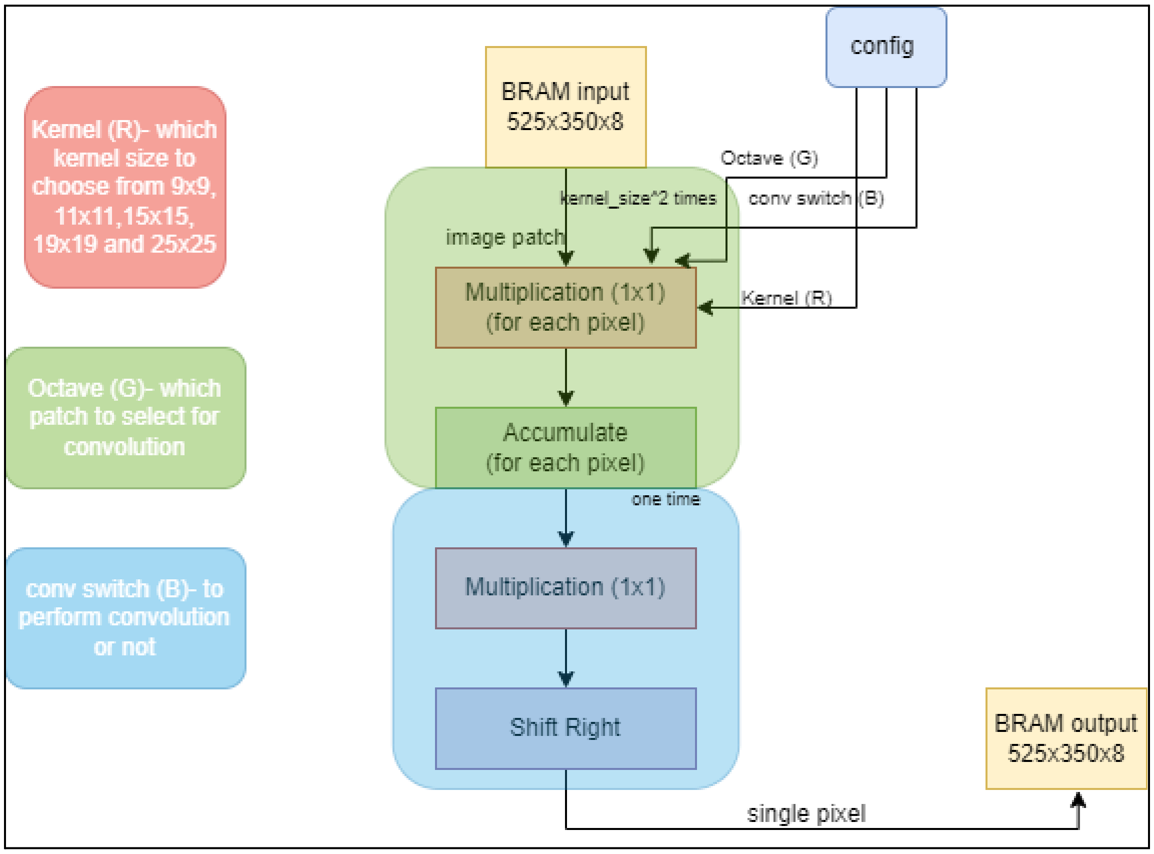

config
BRAM input 525x350x8
Kernel (R)- which kernel size to choose from 9x9, 11x11,15x15, 19x19 and 25x25
Octave (G)
kernel_size^2 times
conv switch (B)
image patch
Multiplication (1x1) (for each pixel)
Kernel (R)
Octave (G)- which patch to select for convolution
Accumulate (for each pixel)
one time
Multiplication (1x1)
conv switch (B)- to perform convolution or not
Shift Right
BRAM output 525x350x8
single pixel

*Fig 02 - Convolution Flow*

In a Gaussian kernel, all values are typically in the range [0,1]. However, floating point operations are not recommended in hardware. Therefore, we scaled the kernel by 1/min(kernel) and saved each value as a 16-bit int. The new kernel is now in the integer range [1, 2^16 -1]. Let's call this *kernel_int*. Since the new values in the kernel are integers, we can easily multiply them with the 8 bit image grayscale pixels. However, the result of the convolution will now be 1/min(kernel) larger than the 8 bit value that is expected by the output image. To correct for this, we could just divide the output convolved value by 1/min(kernel), however division is not recommended in hardware, as division is expensive. Therefore, we came up with the following algorithm to compute the desired 8 bit output value from the output conv value:

1) Consider the maximum value that the scaled convolved value can take on. This will be 255 * sum(kernel_int). Under a normal kernel, which sums to one, the maximum value is guaranteed to be able to be represented in 8 bits. However, our kernel_int's sum depends on the size of the kernel – large kernels have greater sums, as all values of the kernel are scaled up such that the minimum value of kernel_int is 1.
2) Since we predefine our kernel sizes before running SIFT, we know kernel_int's sum and can determine how many bits it would take to represent the largest output of a convolution with this kernel_int.
3) Once we know the number of bits (maxNumBits) required to represent the largest possible conv output, to obtain the desired value, conv_out_8bit, from our scaled conv_out, we can do the following:
4) Define in advance

   multiplier = $2^{maxNumBits}$ / (1 / min(kernel))
5) To convert the scaled conv_out to an 8bit quantity than we can use, compute the following

   conv_out_8bit = (multiplier * conv_out) >> $maxNumBits$

Such a computation has the effect of dividing by (1/min(kernel)), which is our original goal. We cleverly utilize bit shifting and our knowledge of the range of values that our conv_out can take in order to multiply in by $2^{maxNumBits}$ and then shift right by $maxNumBits$, which ultimately has no effect on our final answer. Therefore, an operation that often requires a division in hardware has been converted into one multiplication and right bit shift, which is significantly faster.

## ***VGA Block in Hardware***

We are using a 2x1 multiplexer to switch between performing the convolution on the image and displaying output. Due to our BRAM configuration set as 1 read and 1 write for ports we are implementing this switching technique in our system.

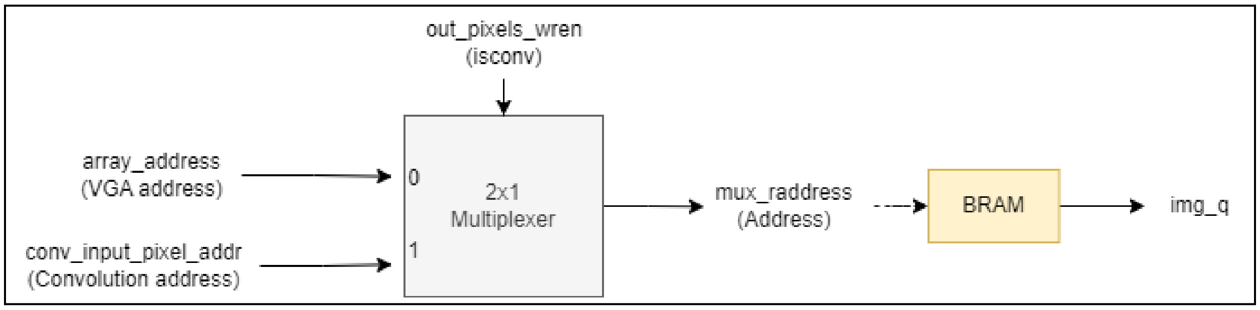
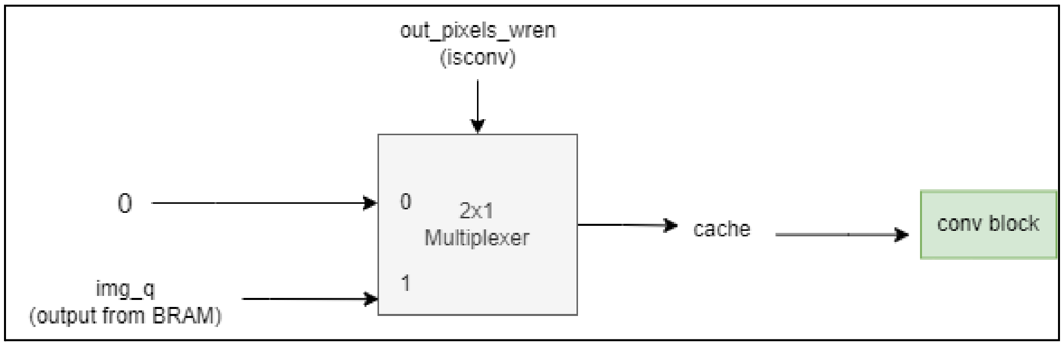


*Fig 03 – Accessing the input BRAM to either load the pixel that we will use during convolution or the pixel that should be displayed on the VGA screen*

The two inputs to the above-shown multiplexer are an array_address (VGA's address) and a conv_input_pixel_addr (Convolution address). The array_address is the pixel's location in the image array that is to be displayed, while the convolution_address is the location of the pixel on which the convolution operation is to be performed. These two address inputs represent two distinct operations that can be performed on a pixel. However, given the single-read port configuration of the BRAM, only one operation can be executed at a time. The multiplexer's function is to select one of these inputs based on a control signal (isconv). In this case, it decides whether to carry out a display or convolution operation. The selected input is then used to index into the BRAM. If the multiplexed address is the array_address, the pixel located at this address is read from the BRAM and sent to the display. On the other hand, if the multiplexed address is the convolution_address, the pixel at this address is read from the BRAM to perform the convolution operation.



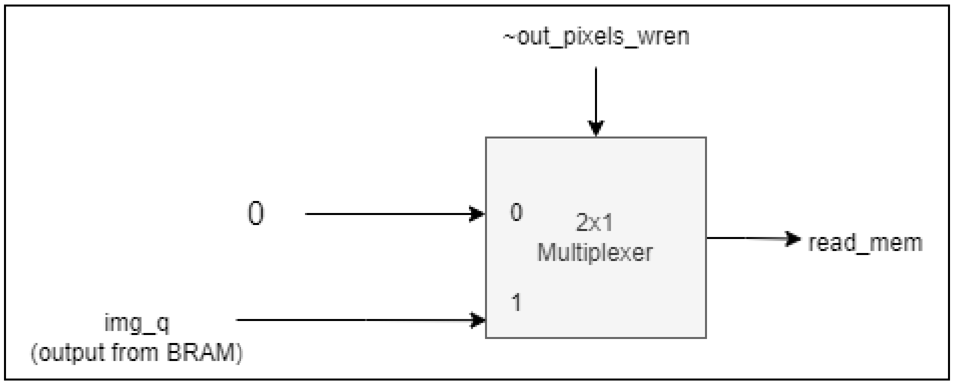


*Fig 04/05 - Logic design for switching between performing convolution and displaying on the VGA.*

The above two multiplexers are the next steps after the pixel is obtained from the BRAM based on the signal input to the multiplexers, if the out_pixels_wren is 1 then the convolution block gets the pixel and if the out_pixels_wren is 0 then the pixel is displayed on the VGA.

## ***Software Design***

The software implements many components of SIFT and functions as an intermediary between the image data and the hardware module responsible for calculating Gaussian blurs. The software begins by reading a specified image from disk. Once the image is loaded, the software transfers this data to the hardware module. This transfer takes place through the Avalon bus, sending the image byte by byte for efficient processing.

Upon the completion of the data transfer, the software signals the hardware to initiate the convolution process. This is done by sending a signal to the Field-Programmable Gate Array (FPGA). The convolution process is integral to computing the Gaussian blur.

In order to monitor the progress and completion of the convolution operation, the software continuously polls a specific flag from the Avalon bus. Once this flag indicates that the convolution process is complete, the software retrieves the image data from the hardware. This retrieval also takes place over the Avalon bus.

It's important to note that this process doesn't occur just once; it is repeated twenty times in total. The reason for this repetition is that there are twenty blurs that need to be computed: five for each of the four different image sizes. After these Gaussian blurs are computed, the software takes over to execute the remaining steps of the SIFT algorithm.

Following the computation of the Gaussian blurs, the software creates the Difference of Gaussians (DoG). This is achieved by subtracting blurred images at adjacent scales. The software then proceeds to extract keypoints from these images. Keypoints are identified as local minima and maxima within the 27 nearest neighbors in the DoG.

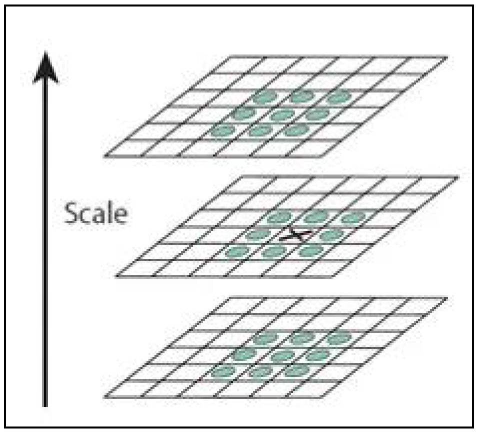


*Fig 06 Difference of Gaussians at different scales*

Once the keypoints are extracted, the software refines them using gradient information from the surrounding area. The purpose of this is to obtain a precise and continuous coordinate for the image keypoint, instead of a discretized one. This is especially important for when keypoints are extracted from images from the smallest scale in the DoG, as the keypoints must be ultimately scaled back up to the full image size.
After refining the keypoint coordinates, the software filters the keypoints based on certain properties, such as whether they are of low contrast. This filtering step is quite computation-intensive, involving numerous linear algebra operations. To handle these operations efficiently, the software employs custom-built linear algebra primitives, thereby eliminating the need for externally written libraries.

After the key points have been filtered, the software marks each valid keypoint on the original image. This marked image is then sent back to the VGA for display. Since the image is in grayscale, each pixel is represented by a single byte that denotes the value for red, green, and blue colors. To effectively highlight the key points on the image, the software uses a pixel value of 0 to denote keypoints, shifting all pixels in the image that originally had a value of 0 to 1. This results in what was once a black pixel being displayed as a slightly less black pixel. Finally, when the hardware sends the image to the VGA for display, it replaces all the marked keypoints (now denoted by a value of 0) with red pixels, thereby highlighting them.

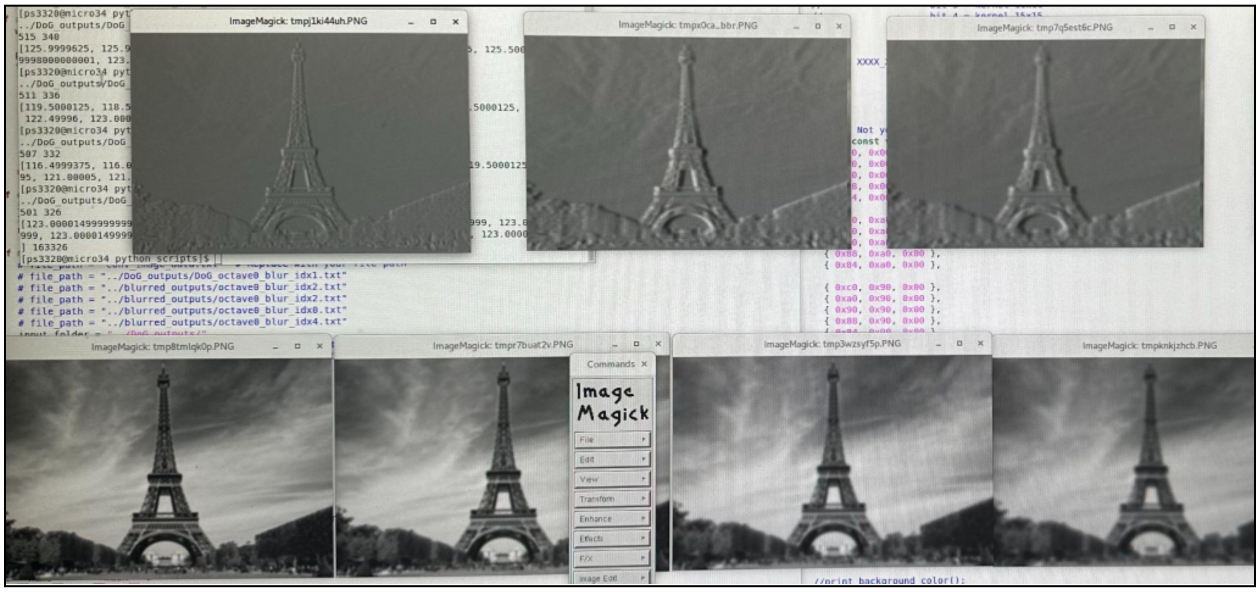
## ***Python Reference***

Before we started writing the C code, we worked on a python implementation of SIFT, which partially used our own code and partially used a pre existing program. This python implementation served as a reference while we were developing the C code.

We also used this implementation as a golden block to verify our intermediate as well as final results from our hardware and C code implementation.

We double checked that the resultant DoGs from our C implementation matched up with the python DoGs, and did the same thing for the number of keypoints generated.

## Results

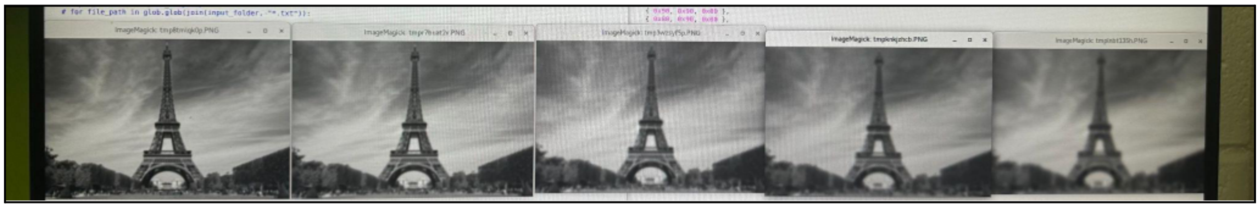

ImageMagick: tmpj1ki44uh.PNG
ImageMagick: tmpx0ca_bbr.PNG
ImageMagick: tmp7q5est6c.PNG
ImageMagick: tmp8tmlqk0p.PNG
ImageMagick: tmpr7buat2v.PNG
Commands
Image Magick
File
Edit
View
Transform
Enhance
Effects
F/X
Image Edit
ImageMagick: tmp3wzsyf5p.PNG
ImageMagick: tmpknkjzhcb.PNG

```
[ps3320@micro34 pyt
../DoG_outputs/DoG_
515 340
[125.9999625, 125.9                                  5, 125.500
9998000000001, 123.
[ps3320@micro34 pyt
../DoG_outputs/DoG_
511 336
[119.5000125, 118.5                                  .5000125,
 122.49996, 123.000
[ps3320@micro34 pyt
../DoG_outputs/DoG_
507 332
[116.4999375, 116.0                                  19.5000125
95, 121.00005, 121.
[ps3320@micro34 pyt
../DoG_outputs/DoG_
501 326
[123.00001499999999                                  999, 123.0
999, 123.0000149999                                  , 123.0000
] 163326
[ps3320@micro34 python scripts]$
```

```
# file_path = "../DoG_outputs/DoG_octave0_blur_idx1.txt"
# file_path = "../blurred_outputs/octave0_blur_idx2.txt"
# file_path = "../blurred_outputs/octave0_blur_idx2.txt"
# file_path = "../blurred_outputs/octave0_blur_idx0.txt"
# file_path = "../blurred_outputs/octave0_blur_idx4.txt"
```

```
bit 4 = kernel 15x15

XXXX_

 Not y
const
0, 0x0
0, 0x0
0, 0x0
8, 0x0
4, 0x0

0, 0xa
0, 0xa
0, 0xa
{ 0x88, 0xa0, 0x00 },
{ 0x84, 0xa0, 0x00 },

{ 0xc0, 0x90, 0x00 },
{ 0xa0, 0x90, 0x00 },
{ 0x90, 0x90, 0x00 },
{ 0x88, 0x90, 0x00 },

//print_background_color();
```

*Fig 07 - Showing the difference in Gaussians (on top)*


ImageMagick: tmp8tmlqk0p.PNG
ImageMagick: tmpr7buat2v.PNG
ImageMagick: tmp3wzsyf5p.PNG
ImageMagick: tmpknkjzhcb.PNG

```
# for file_path in glob.glob(join(input_folder, "*.txt")):
```

*Fig 08 - Showing convolutions with different kernel sizes.*

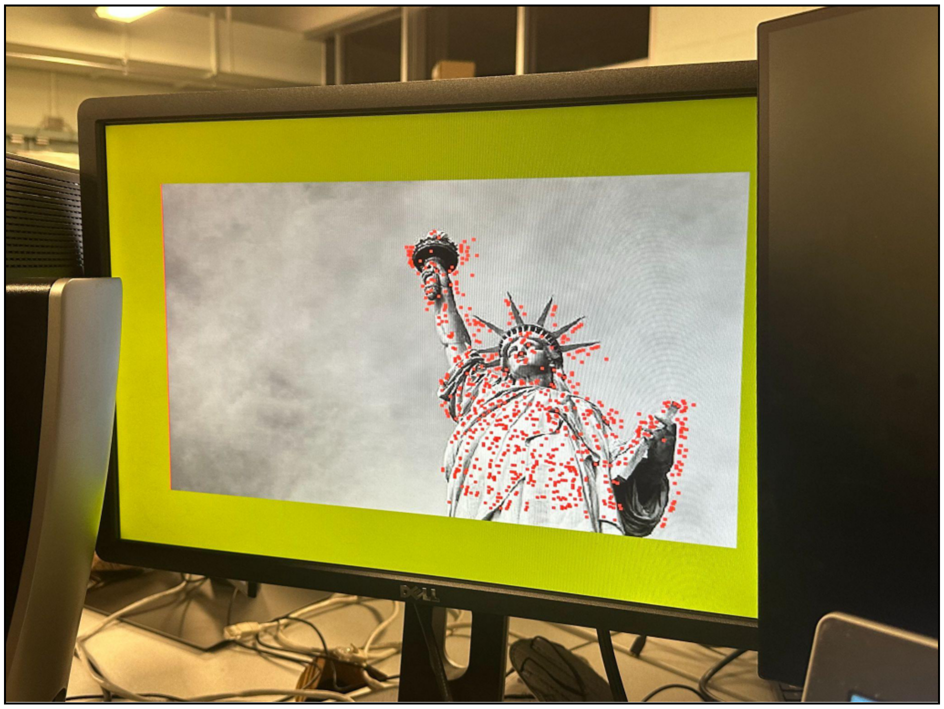


*Fig. 09 - Final Output with the SIFT Annotated Key Points*

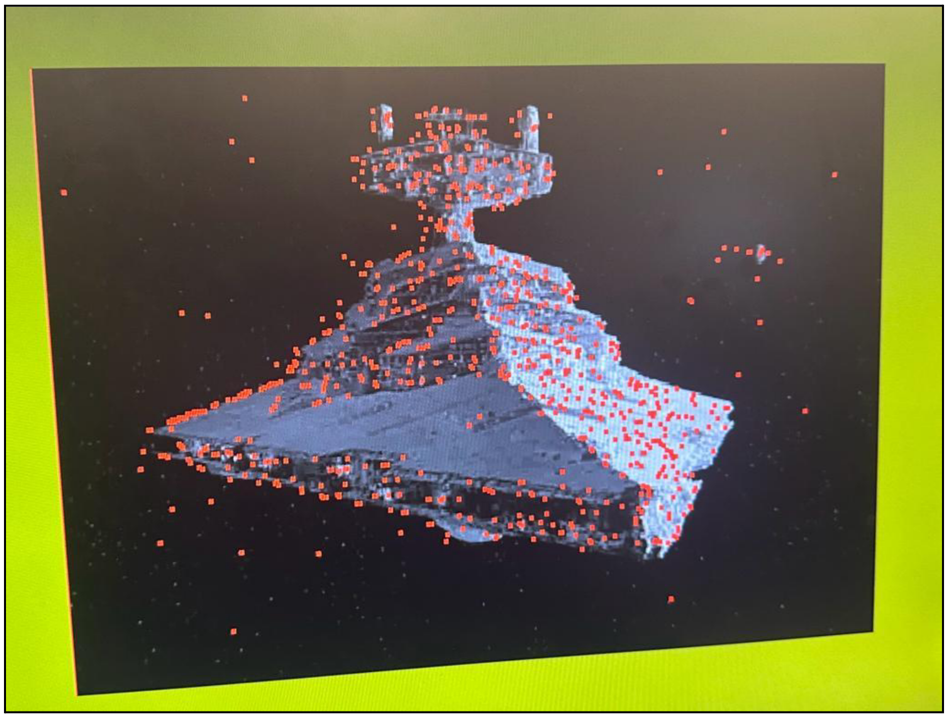


*Fig. 10 - Final Output with the SIFT Annotated Key Points*

# ***Challenges***

## ***Hardware:***

1. At first we intended to store the two images (input and final output) in the 64 KB on chip-built-memory however with the image sizes we were using, (525 * 350 * 8 bits * 2 ) the on-chip memory was insufficient.
2. Thus, to circumvent the issue we created 2 Block RAM blocks one for the input image and one for the output image and reused them in our convolutions. The Block RAM offers larger memory capacity compared to the on-chip memory. It is commonly used for data intensive applications like the one we have implemented by providing faster access times. On the other hand, the on-chip memory is more prominently used to store program data, intermediate variables, etc.
3. Some of the caveats of using the BRAM are:
    a. Reading from the Block RAM takes 2 clock cycles. In the first clock cycle, it interprets the address provided to it on the address port and in the second clock cycle it puts the data on the output port. This doubles the amount of time that our convolution takes (1 vs 2) and is one of the limiting factors.
    b. The Write operation takes one clock cycle. The data on the write port of the BRAM is written to the address specified in the address bus.
    c. This subtle difference between different read and write cycle times was not clear initially and we spent quite a lot of time figuring this out.
4. Using the BRAM modules also solved our problem of consuming over 90% of the available computing resources  and our placement/routing issues of the logic blocks during the synthesis.
5. This was a significant learning point that made us understand why a synthesizable verilog code is essential as against a “simulated” verilog code, as eventually the RTL must be implemented using gates. That’s sort of the advantage of using the FPGA board which maps the verilog to gates by performing synthesis.

## ***Software:***

1. Because we didn't add any padding to the images before the convolution, the output images were slightly different sizes based on the size of the kernel. Therefore, when calculating the difference of gaussians, we had to make sure that the values we were comparing represented the same pixel location in the image. Since we were representing the image in a 1 dimensional array, this involved complicated algebra and bookkeeping.
2. Similarly, when marking the key points on the original image, a similar translation had to be made since each octave shrunk the size of the image by 2. More care also had to be taken to account for the borders that were cropped off during the convolution and DoG calculation.
3. When we first checked the key points that we were producing, the key points practically covered the image, and did not significantly represent important CV locations such as borders to objects. We realized that we needed to adjust specific parameters in the keypoint filtering, such as the contrast and curvature thresholds. After a number of iterations of different values, we finally were able to get good results. These values also vary per image.
4. Given a (row, col) of a keypoint, the idx in a 1D array at which to access this keypoint is defined as row * IM_WIDTH + col. However, when a keypoint coordinate is described as a floating point number, indexing in this manner will not give you the result that you expect. Imagine if the row is 1.5 and col is 0 (the leftmost pixel 1.5 cols down). The .5 in the tens place of the row number will be multiplied by IM_WIDTH, is not the desired intent. The resulting keypoint would be shifted to the right by half the image (or wrapped around to the beginning of the image if it exceeds the width). Therefore, we just needed to define the 1D index at which to access the keypoint as int(row) * IM_WIDTH + col.

## ***Future Work***

The purpose of SIFT is to generate image keypoints that are rotation and scale invariant – meaning that if I take two pictures of the same object at a different angle and zoom, SIFT should be able to find the transformation that maps one image onto the other based on the similarity of keypoints. Since this is an Embedded Systems class and not a Computer Vision class, we did not proceed to write the code to show that such image mapping works. However, showing that two images can be successfully transformed onto each other is the ultimate guarantee of the correctness of our algorithm. We determine correctness by assessing the quality of keypoints, which is not done quantitatively, but by eye. While this is certainly helpful, we cannot conclude with certainty that they are correct.

Configure our BRAM to take in two input reads so that we don't have to make multiplexing logic for displaying and convolution. Instead, both should happen simultaneously.

Use the FPGA to accelerate the convolution by implementing multiple convolution blocks in parallel. However, we would need to be able to read multiple pixel bytes of data at the same time in order to do this.

Use more complex and configurable memory mechanisms, such as SDRAM or DMA in order to run SIFT on images of various sizes

## ***Lessons Learnt and Advice for Future Projects***

This project was surprisingly hard to achieve contrary to our initial expectations. Following are a few of the things we wish we knew earlier and could perhaps help future projects:

- Start with the hardware interfacing first. Although seemingly obvious, and something we thankfully followed ourselves, not doing so at the very beginning would have made our project practically impossible.
- No matter how simple you think implementation on hardware is, things are always way harder than you expect. So unless you are a certified verilog god, try to keep the hardware implementation simple.
- Most projects will probably involve some sort of BRAM implementation. Understand how it works early on.
- Do not shy away from looking at previous projects similar to yours, try using others past sufferings to your advantage while also using your creativity and being original. You will learn a lot more that way.
- Create scripts to configure running and deploying quartus files. It's a small initial investment in time that saves tons later as you inevitably run them hundreds of times.
- The Lab3 code is probably the most useful reference you will have. Understand it like you understand your best friend.
- While the C/linux code is relatively simpler, try to have someone work on it parallelly and test it with dummy code that mimics hardware.
- Speaking of tests, test parts of your code, especially hardware, like you are obsessed with testing. Although system verilog may behave consistently in a simulated environment. The DE1-SoC sometimes adds its own creative inputs so be aware of them earlier rather than later.
- Have fun while learning! Even though you might feel forced to learn things that are relevant only to the specific hardware you are using, a lot of the underlying concepts are super useful to know if you want to be anywhere near an embedded device in your career.

## ***Individual Contributions***

*Madhav* - Focused on hardware implementation. Specifically, worked on getting the interfacing with software to work including the readdata and writedata to be properly interfaced between the hardware and the software and how the software kernel interfaces sends signals and data back and forth and interfacing with BRAM.

*Prathamesh* - Focused mostly on hardware with some implementations in software. Worked on getting the hardware convolution to work, testing with true values on python. Also helped implement some of the software logic on scaling and dealing with the non-padded images coming from hardware.

*Khushi* - Helped in contributing to the overall design flow and the specifics of how the hardware components interact with each other. Also contributed in creating different scripts in python to verify the software aspects of the algorithm.

*Jeffrey* – Had the initial idea to try to run SIFT on the FPGA, and I implemented most of the SIFT algorithm in C. Tested the correctness of my C implementation against a Python implementation of SIFT. Wrote a few python scripts to compute the Gaussian kernel values and display the intermediate outputs of blurring, DoGs, etc. Implemented the logic to convert the scaled conv output to an 8 bit int without floating point operations or dividing.

*Daniel* – Helped with the hardware implementation, specifically the logic flow to figure out which convolution to perform with which kernel. Also contributed some of the software, specifically marking the key points on the final image and verification.

## ***References***

About SIFT:

https://medium.com/data-breach/introduction-to-sift-scale-invariant-feature-transform-65d7f3a72d40

https://www.sciencedirect.com/science/article/pii/S0141933115001921?via%3Dihub

https://www.cs.ubc.ca/~lowe/papers/iccv99.pdf

FPGA Based Parallel Hardware Architecture for SIFT Algorithm

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7784039&tag=1

On keypoints being detected at different scales:

https://stackoverflow.com/questions/75468047/how-does-multiscale-feature-matching-work-orb-sift-etc

On how to compute Taylor expansion:

https://dsp.stackexchange.com/questions/10403/sift-taylor-expansion

Implementation of SIFT in c:

https://github.com/robwhess/opensift/blob/master/src/sift.c

Implementation of SIFT in python:

https://lerner98.medium.com/implementing-sift-in-python-36c619df7945

Performing division via multiplication and shift:

https://stackoverflow.com/questions/171301/whats-the-fastest-way-to-divide-an-integer-by-3?rq=1#:~:text=There%20is%20a,a%20bit%20ROLL

# ***Code Appendix***

Sift.h

```
C/C++
#ifndef _SIFT
#define _SIFT

#include "vga_ball.h"

#define MAX_KERNEL_SIZE 25
#define NUM_OCTAVES 4
#define NUM_BLURS 5
#define SIGMA_ROOT 2.5

// typedef struct Kernel {
//     double kernel[25*25];
//     unsigned int size; // e.g. 19
//     double sigma;
// } Kernel;

typedef struct Img {
  unsigned char *array; // where the actual img is held
  unsigned short height;
  unsigned short width;
  double sigma;
  unsigned char octave;   // either 0,1,2,3
  unsigned char blur_idx; // 0,1,2,3,4
} Img;

typedef struct DoG {
  Img *img1;
  Img *img2;     // more blur than img1 (higher sigma than img1)
  double *array; // where the actual DoG is held
  unsigned char octave;
  int width;
  int height;
} DoG;

#endif
```

main.c

```
C/C++
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define HW_ENABLE 1
#define DOG_PY 0

#define COLORS 8

#include "im_utils.h"
#include "keypoints.h"
#include "sift.h"
#include "sift_utils.h"
#include "vga_ball.h"
#include "vga_ball_utils.h"

int main() {
  vga_ball_arg_t vla;
  vga_ball_value_t cur_val = {0, 0};
  static const char filename[] = "/dev/vga_ball";

  static const vga_ball_color_t colors[] = {
    { 0xc0, 0x00, 0x00 },
    { 0xa0, 0x00, 0x00 },
    { 0x90, 0x00, 0x00 },
    { 0x88, 0x00, 0x00 },
    { 0x84, 0x00, 0x00 },

    { 0xc0, 0xa0, 0x00 },
    { 0xa0, 0xa0, 0x00 },
    { 0x90, 0xa0, 0x00 },
    { 0x88, 0xa0, 0x00 },
    { 0x84, 0xa0, 0x00 },

    { 0xc0, 0x90, 0x00 },
    { 0xa0, 0x90, 0x00 },
    { 0x90, 0x90, 0x00 },
    { 0x88, 0x90, 0x00 },
    { 0x84, 0x90, 0x00 },

    { 0xc0, 0x88, 0x00 },
    { 0xa0, 0x88, 0x00 },
    { 0x90, 0x88, 0x00 },
    { 0x88, 0x88, 0x00 },
    { 0x84, 0x88, 0x00 }

  };

  printf("VGA ball Userspace program started\n");

#if HW_ENABLE
  extern int vga_ball_fd;
  if ((vga_ball_fd = open(filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", filename);
    return -1;

    printf("initial state: ");
    // clear_bram(&cur_val);
    // print_background_color();
    // set_background_color(
    //    &colors[4]); // is this necessary? or just leftover
from lab3
  }
#endif

  // const char *img_name = "eiffel-tower-gray-8-bit.bin";
  //const char *img_name = "eiffel-tower-gray-updated.bin";
  const char *img_name = "statue_of_liberty2.bin";
  //const char *img_name = "greek.bin";
  char array[IMAGE_SIZE] = {0};
  char read_array[IMAGE_SIZE] = {0};

  read_image_data(img_name, array, IMAGE_SIZE);
  printf("Read image data from file\n");
  //for (size_t i = NUM_IOCTL_READS - 20; i < NUM_IOCTL_READS *
  // BUFFER_IMAGE_SIZE; i++) { printf("array[%d] = %d\n", (int)i,
array[i]);
  // }
#if HW_ENABLE
  cur_val.addr = 0;
  for (size_t i = 0; i < IMAGE_SIZE; i++) {

    cur_val.val = array[i];
    if (cur_val.val == 0)
      cur_val.val = 1;
    cur_val.addr++;
    set_value(&cur_val);
  }
#endif

  Img *blurred_ims[20] = {0};
  double sigmas[NUM_BLURS];
  get_sigmas(sigmas, SIGMA_ROOT);
```

```
int heights[NUM_OCTAVES] = {IMAGE_HEIGHT, IMAGE_HEIGHT / 2,
IMAGE_HEIGHT / 4, IMAGE_HEIGHT / 8};
int widths[NUM_OCTAVES] = {IMAGE_WIDTH, IMAGE_WIDTH / 2,
IMAGE_WIDTH / 4, IMAGE_WIDTH / 8};
// conv_out_height = img_height - kernel_size + 18'd1;
#if DOG_PY
int blur_im_heights[NUM_OCTAVES * NUM_BLURS] = {heights[0],
heights[0], heights[0], heights[0], heights[0], heights[1],
heights[1], heights[1], heights[1], heights[1], heights[2],
heights[2], heights[2], heights[2], heights[2], heights[3],
heights[3], heights[3], heights[3], heights[3], };
int blur_im_widths[NUM_OCTAVES * NUM_BLURS] = {widths[0],
widths[0], widths[0], widths[0], widths[0], widths[1], widths[1],
widths[1], widths[1], widths[1], widths[2], widths[2], widths[2],
widths[2], widths[2], widths[3], widths[3], widths[3], widths[3],
widths[3], };
#else
int blur_im_heights[NUM_OCTAVES * NUM_BLURS] = {heights[0]-8,
heights[0]-10, heights[0]-14, heights[0]-18, heights[0]-24,
                                  heights[1]-8 +
check_bit(IMAGE_HEIGHT, 0), heights[1]-10 +
check_bit(IMAGE_HEIGHT, 0), heights[1]-14 +
check_bit(IMAGE_HEIGHT, 0), heights[1]-18 +
check_bit(IMAGE_HEIGHT, 0), heights[1]-24 +
check_bit(IMAGE_HEIGHT, 0),
                                heights[2]-8 +
check_bit(IMAGE_HEIGHT, 1), heights[2]-10 +
check_bit(IMAGE_HEIGHT, 1), heights[2]-14 +
check_bit(IMAGE_HEIGHT, 1), heights[2]-18 +
check_bit(IMAGE_HEIGHT, 1), heights[2]-24 +
check_bit(IMAGE_HEIGHT, 1),
                                  heights[3]-8 +
check_bit(IMAGE_HEIGHT, 2), heights[3]-10 +
check_bit(IMAGE_HEIGHT, 2), heights[3]-14 +
check_bit(IMAGE_HEIGHT, 2), heights[3]-18 +
check_bit(IMAGE_HEIGHT, 2), heights[3]-24 +
check_bit(IMAGE_HEIGHT, 2)
                              };

int blur_im_widths[NUM_OCTAVES * NUM_BLURS] = {widths[0]-8,
widths[0]-10, widths[0]-14, widths[0]-18, widths[0]-24,
                              widths[1]-8 +
check_bit(IMAGE_WIDTH, 0), widths[1]-10 + check_bit(IMAGE_WIDTH,
0), widths[1]-14 + check_bit(IMAGE_WIDTH, 0), widths[1]-18 +
check_bit(IMAGE_WIDTH, 0), widths[1]-24 + check_bit(IMAGE_WIDTH,
0),
                              widths[2]-8 +
check_bit(IMAGE_WIDTH, 1), widths[2]-10 + check_bit(IMAGE_WIDTH,
1), widths[2]-14 + check_bit(IMAGE_WIDTH, 1), widths[2]-18 +
check_bit(IMAGE_WIDTH, 1), widths[2]-24 + check_bit(IMAGE_WIDTH,
1),
                              widths[3]-8 +
check_bit(IMAGE_WIDTH, 2), widths[3]-10 + check_bit(IMAGE_WIDTH,
2), widths[3]-14 + check_bit(IMAGE_WIDTH, 2), widths[3]-18 +
check_bit(IMAGE_WIDTH, 2), widths[3]-24 + check_bit(IMAGE_WIDTH,
2)
                             };

#endif

int dog_heights[NUM_OCTAVES] = {blur_im_heights[4],
blur_im_heights[9], blur_im_heights[14], blur_im_heights[19]};
int dog_widths[NUM_OCTAVES] = {blur_im_widths[4],
blur_im_widths[9], blur_im_widths[14], blur_im_widths[19]};

for (int i=0; i<4; i++) {
     printf("GET BIT(%d, %d) = %d\n", IMAGE_WIDTH, i,
check_bit(IMAGE_WIDTH, i));
     printf("GET BIT(%d, %d) = %d\n", IMAGE_HEIGHT, i,
check_bit(IMAGE_HEIGHT, i));
}
```

```
for (int i=0; i<20; i++) {
      printf("h, w: %d, %d\n", blur_im_heights[i],
blur_im_widths[i]);
}

  //   int widths[NUM_OCTAVES] = {10, 5, 2, 1};
  for (int octave_num = 0; octave_num < NUM_OCTAVES;
octave_num++) {
    for (int blur_idx = 0; blur_idx < NUM_BLURS; blur_idx++) {
      int idx = octave_num * NUM_BLURS + blur_idx;

      Img *blurred_im = (Img *)malloc(sizeof(Img));
      blurred_im->height = blur_im_heights[idx];
      blurred_im->width = blur_im_widths[idx];

      int size = blurred_im->height * blurred_im->width;
      blurred_im->array = (unsigned char *)malloc(size *
sizeof(char));
      //   printf("blurred_im_array#%d: %lx\n", idx,
      //  (unsigned long)blurred_im->array);
#if DOG_PY
#elif HW_ENABLE
      blurred_im->array[0] = set_background_color(&colors[idx]);
      printf("Running conv on octave %d and blur_idx %d...\n",
octave_num, blur_idx);
      // usleep(3000000);
      //usleep(5000000 >> (octave_num));
      for (int i = 1; i < size; i++){
     read_value(blurred_im->array, i);
      }
#else
      for (int i = 0; i < size; i++) {
        blurred_im->array[i] = (char)(i + blur_idx);
      }
#endif
```

```
      blurred_im->blur_idx = blur_idx;
      blurred_im->octave = octave_num;
      blurred_im->sigma = sigmas[blur_idx];
      blurred_ims[idx] = blurred_im;
      char file_path[100] = {0};
      sprintf(file_path,
"blurred_outputs/octave%d_blur_idx%d.txt", octave_num, blur_idx);
      write_image_to_txt_file(blurred_im->array, file_path,
size);
    }
  }

  DoG *diffs[16] = {0};
  for (int octave_num = 0; octave_num < NUM_OCTAVES;
octave_num++) {
    for (int blur_idx = 0; blur_idx < NUM_BLURS - 1; blur_idx++)
{
      int dog_idx = octave_num * (NUM_BLURS - 1) + blur_idx;
      int blur_ims_idx = octave_num * (NUM_BLURS) + blur_idx;
      DoG *diff = malloc(sizeof(DoG));
      diff->img1 = blurred_ims[blur_ims_idx];
      diff->img2 = blurred_ims[blur_ims_idx + 1];
      diff->octave = octave_num;
      // int size = diff->img2->height * diff->img2->width; //
img2s size because img2 is smaller, and DoG must go by smaller im
      int size = dog_widths[octave_num] *
dog_heights[octave_num];
      diff->array = (double *)malloc(size * sizeof(double));
      diff->width = dog_widths[octave_num];
      diff->height = dog_heights[octave_num];
#if DOG_PY
      read_DoG_py(diff->img1, diff->img2, diff);
#else
      compute_DoG(diff->img1, diff->img2, diff);
#endif
      if (size > 5010) {
           for (int i = 5000; i < 5010; i++) {
           printf("octave #%d, DoG #%d, array[%d]: %f\n",
octave_num, blur_idx, i,
                  diff->array[i]);
           }
      }

      diffs[dog_idx] = diff;
      char file_path[100] = {0};
      sprintf(file_path,
"DoG_outputs/DoG_octave%d_blur_idx%d.txt", octave_num, blur_idx);
      write_float_image_to_txt_file(diff->array, file_path,
size);

#if HW_ENABLE
#else
      // To add a random extrema to see if the kp code later on
works
      if (blur_idx % 2 == 0) {
        diff->array[1100] = .234;
        for (int i = 1098; i < 1104; i++)
          printf("octave #%d, DoG #%d, array[%d]: %f\n",
octave_num, blur_idx,
                 i, diff->array[i]);
      }
#endif
    }
  }

  Keypoints keypoints;
  keypoints._max_capacity = MAX_KEYPOINTS;
  keypoints.count = 0;
  keypoints.kp_list = (Kp **)malloc(MAX_KEYPOINTS * sizeof(Kp
**));
  get_candidate_keypoints(diffs, &keypoints);
```

```c
  localize_keypoints(&keypoints, diffs);

  char file_path[100] = {0};
  sprintf(file_path, "keypoint_outputs/keypoints_eifel.txt");
  write_kps_to_txt_file(&keypoints, file_path);

  for (int octave_num = 0; octave_num < NUM_OCTAVES;
octave_num++) {
      mark_keypoints_on_imag(array, &keypoints, octave_num);

#if 1
  cur_val.addr = 0;
  for (size_t i = 0; i < IMAGE_SIZE; i++) {
    cur_val.val = array[i];
    cur_val.addr++;
    set_value(&cur_val);
  }
   usleep(3000000);
#endif

}

 //read_image_data(img_name2, array, IMAGE_SIZE);
  //printf("Read image data from file\n");
  // for (size_t i = NUM_IOCTL_READS - 20; i < NUM_IOCTL_READS *
  // BUFFER_IMAGE_SIZE; i++) { printf("array[%d] = %d\n", (int)i,
array[i]);
  // }
#if 0
  cur_val.addr = 0;
  for (size_t i = 0; i < IMAGE_SIZE; i++) {
    cur_val.val = array[i];
    cur_val.addr++;
    set_value(&cur_val);
  }
#endif
```

```
  printf("VGA BALL Userspace program terminating\n");
  return 0;
}
```

## sift_utils.h

```c
C/C++
#ifndef SIFT_UTILS
#define SIFT_UTILS

#include "sift.h"
#include "keypoints.h"

int check_bit(int num, int i);
int get_octave_im_dim(int orig_dim, int octave_num);
void compute_DoG(Img *imgA, Img *imgB, DoG *dog);
void get_kernel(double *arr, double sigma);
int get_kernel_size_from_sigma(double sigma);
void get_sigmas(double *arr, double root);
void mark_keypoints_on_imag(char *img_array, Keypoints
*keypoints, int octave_num);
void read_DoG_py(Img *imgA, Img *imgB, DoG *dog);

#endif
```

## sift_utils.c

```c
C/C++
#include <stdio.h>
#include <math.h>
```

```
#include "vga_ball.h"
#include "sift.h"
#include "sift_utils.h"
#include <stdlib.h>
#include <assert.h>
#include "keypoints.h"

void compute_DoG(Img *imgA, Img *imgB, DoG *dog) { // assumes
imgA->array and imgB->array are within range [0,255]
    printf("Computing DoG for octave %d blur_idx %d\n",
imgA->octave, imgA->blur_idx);
    int height = dog->height;
    int width = dog->width;

    int img_size = height * width;
    int heightA_diff = imgA->height - height;
    int heightB_diff = imgB->height - height;
    assert(heightA_diff > 0);
    assert(heightA_diff > heightB_diff);
    int widthA_diff = imgA->width - width;
    int widthB_diff = imgB->width - width;
    assert(widthA_diff > 0);
    assert(widthA_diff > widthB_diff);
    if (dog->img2->blur_idx == 4) assert(heightB_diff == 0 &&
widthB_diff == 0);
    assert(widthA_diff % 2 == 0 && widthB_diff % 2 == 0);
    assert(heightA_diff % 2 == 0 && heightB_diff % 2 == 0);

    int idx_startA = imgA->width * (heightA_diff / 2) +
(widthA_diff / 2); // top left pix of cropped A
    int idx_startB = imgB->width * (heightB_diff / 2) +
(widthB_diff / 2); // top left pix of cropped B

    int idxA = idx_startA;
    int idxB = idx_startB;
    for (int i=0; i<img_size; i++) {
        // printf("height_diffA %d, height_diffB %d, widthA_diff
%d, widthB_diff %d\n", heightA_diff, heightB_diff, widthA_diff,
widthB_diff);
        // printf("idxStartA %d, idxStartb %d, idxA %d, idxB
%d\n", idx_startA, idx_startB, idxA, idxB);
        int row_DoG = (i / width);
        dog->array[i] = ((double)(imgB->array[idxB] -
imgA->array[idxA]) / 255);
        if (i % width == (width-1)) {
           idxA += widthA_diff;
           idxB += widthB_diff;
        }
        idxA++;
        idxB++;
    }
    printf("Finished Computing DoG for octave %d blur_idx %d\n",
imgA->octave, imgA->blur_idx);
}

void get_kernel(double *arr, double sigma) {
    int size = get_kernel_size_from_sigma(sigma);
    double constant = 1 / (2 * 3.14159265 * pow(sigma, 2));
    double arr_sum = 0;
    for (int r=0; r<size; r++) {
        for (int c=0; c<size; c++) {
            double val = constant * exp((pow(r - size/2, 2) +
pow(c - size/2, 2)) / (2 * pow(sigma, 2)));
            arr[r * size + c] = val;
            arr_sum += val;
        }
    }
    for (int i=0; i<size*size; i++)
        arr[i] /= arr_sum;

}
```

```
int get_octave_im_dim(int orig_dim, int octave_num) {
      return (orig_dim >> octave_num) + check_bit(orig_dim,
octave_num-1);
}

int check_bit(int num, int i) {
    if (i<0) return 0;
    int mask = 1 << i;
    return (num & mask) != 0;
}

int get_kernel_size_from_sigma(double sigma) {
    return 2 * ceil(3 * sigma) + 1;
}

void get_sigmas(double *arr, double root) {
    for (int i=0; i<NUM_BLURS; i++) {
        arr[i] = pow(2, ((i+1) / root));
    }
}

void mark_keypoints_on_imag(char *img_array, Keypoints
*keypoints, int octave_num) {
    // clear data from the marker value
    // --> this makes the black pixels slightly brighter
    for (int i=0; i < IMAGE_SIZE; i++) {
        if (img_array[i] == 0)
            img_array[i]++;
    }
    // mark the pixels around the kp with the flag to show them
as RED on the VGA board
    for (int i=0; i < keypoints->count; i++) {
        Kp *kp = keypoints->kp_list[i];
        if (!kp->is_valid)
          continue;
        if (kp->DoG_idx / (NUM_BLURS-1) != octave_num) continue;
        int arr_index = (int) ((kp->y+12) * IMAGE_WIDTH +
(kp->x+12));
        /* Do we want to do this for specific DoGs or just for
all found kps? */
        // if (kp->DoG_idx == img->)
        int row = arr_index / IMAGE_WIDTH;
        int col = arr_index % IMAGE_WIDTH;

        img_array[arr_index] = 0;
        if (col < IMAGE_WIDTH-1) // not the last pixel on the
line
            img_array[arr_index + 1] = 0;
        if (col > 0) // not the first pixel on the line
            img_array[arr_index - 1] = 0;

        // not on the last line of the image
        if (row < IMAGE_HEIGHT - 1) {
            img_array[arr_index + IMAGE_WIDTH] = 0;
            if (col > 0)
                img_array[arr_index + IMAGE_WIDTH - 1] = 0;
            if (col < IMAGE_WIDTH-1)
                img_array[arr_index + IMAGE_WIDTH + 1] = 0;
        }

        // not on the first line
        if (row > 1) {
            img_array[arr_index - IMAGE_WIDTH] = 0;
            if (col > 0)
                img_array[arr_index - IMAGE_WIDTH - 1] = 0;
            if (col < IMAGE_WIDTH-1)
                img_array[arr_index - IMAGE_WIDTH + 1] = 0;
        }
    }
}

void read_DoG_py(Img *imgA, Img *imgB, DoG *dog) {
    int img_size = dog->height * dog->width; // imgB size will be
smaller than imgA, since imgA uses a smaller kernel

    for (int octave_num = 0; octave_num < NUM_OCTAVES;
octave_num++) {
      for (int blur_idx = 0; blur_idx < NUM_BLURS-1; blur_idx++) {
          char file_name[100] = {0};
          sprintf(file_name, "../../DoG_out_py/DoG_out_py_%d_%d",
octave_num, blur_idx);

          FILE *file = fopen(file_name, "r");
          if (file == NULL) {
            printf("Cannot open file \n");
            return;
          }

          int i = 0;
          while (i < img_size && fscanf(file, "%lf\n",
&dog->array[i]) == 1) {
            i++;
          }
          fclose(file);
          }
    }
}
```

## keypoints.h

```
C/C++
#include "im_utils.h"
#include "sift.h"
#include "sift_utils.h"
#include "vga_ball.h"

#ifndef _KEYPOINTS
```

```
#define _KEYPOINTS

//#define CONTRAST_THRESHOLD 0.03
#define CONTRAST_THRESHOLD 0.06
#define PRINCIPLE_CURVATURE_THRESHOLD 12.1 // (r+1)^2 / r, with
r=10
//#define PRINCIPLE_CURVATURE_THRESHOLD 5

#define MAX_KEYPOINTS 10000

typedef struct Kp_precise {
  double x;
  double y;
  double real_x;
  double real_y;
  double DoG_idx;
} Kp_precise;

typedef struct Kp {
  unsigned short x;
  unsigned short y;
  DoG *DoG;
  int DoG_idx;
  int is_valid;

  // used for keypoint localization
  double *jacobian; // should point to 3 * 8 bytes (dx, dy, ds)
  double *hessian; // should point to 4 * 8 bytes (only care
about dxx, dxy, dxy, dyy, actual mat is 3x3)
  double *offset; // should point to 3 * 8 bytes (offset_row,
offset_col, offset_sigma)
  Kp_precise *kp_precise;
} Kp;

typedef struct Keypoints {
  Kp **kp_list;
  unsigned short count;
  int _max_capacity;
} Keypoints;

void get_candidate_keypoints(DoG **DoGs, Keypoints *keypoints);
void get_candidate_keypoints_for_octave(DoG **DoGs, Keypoints
*keypoints, int octave_num);
void localize_keypoints(Keypoints *keypoints, DoG **diffs);
void localize_kp(Kp *kp, DoG **diffs);
double compute_kp_contrast(Kp *kp, DoG **diffs);
void print_keypoints_stats(Keypoints *keypoints);
void write_kps_to_txt_file(Keypoints *keypoints, const char
*file_path);

#endif
```

keypoints.c

```
C/C++
#include "keypoints.h"
#include "im_utils.h"
#include "lin_alg.h"
#include "sift.h"
#include "sift_utils.h"
#include "vga_ball.h"
#include "vga_ball_utils.h"

#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>

double compute_kp_contrast(Kp *kp, DoG **diffs) {
  DoG *cur_DoG = diffs[kp->DoG_idx];
  int width = cur_DoG->width;

  double dot_output;
  dot(kp->jacobian, kp->offset, &dot_output, 1, 3, 1);

  double contrast = cur_DoG->array[kp->y * width + kp->x] + 0.5 *
dot_output;
  return contrast;
}

int get_argmin(double *arr, int size) {
  int argmin = 0;
  double min = 10000;
  int idx = 0;
  while (idx < size) {
    if (arr[idx] < min) {
      argmin = idx;
      min = arr[idx];
    }
    idx++;
  }
  return argmin;
}

int get_argmax(double *arr, int size) {
  int argmax = 0;
  double max = -10000;
  int idx = 0;
  while (idx < size) {
    if (arr[idx] > max) {
      argmax = idx;
      max = arr[idx];
    }
    idx++;
  }
  return argmax;
}

// find the 3x3 neighborhood around pix_idx. pix_idx is indexing
into a 1-D
// array, we must find the 2D 3x3 window.
void get_neighborhood(DoG *cur_DoG, double *neighborhood, int
pix_idx) {
  int width = cur_DoG->width;
  int row = pix_idx / width;
  int col = pix_idx % width;
  for (int r = row - 1; r <= row + 1; r++) {
    for (int c = col - 1; c <= col + 1; c++) {
      *neighborhood = cur_DoG->array[r * width + c];
      neighborhood++;
    }
  }
}

// finds the local min maxima for each DoG w.r.t its DoG
neighboring pixels in
// different blurs
void get_candidate_keypoints_for_octave(DoG **DoGs, Keypoints
*keypoints,
                                        int octave_num) {
  for (int blur_idx = 1; blur_idx < NUM_BLURS - 1 - 1;
blur_idx++) {
    int cur_DoG_idx = octave_num * (NUM_BLURS - 1) + blur_idx;
    DoG *left_DoG = DoGs[cur_DoG_idx - 1];
    DoG *cur_DoG = DoGs[cur_DoG_idx];
    DoG *right_DoG = DoGs[cur_DoG_idx + 1];

    int im_height = cur_DoG->height;
    int im_width = cur_DoG->width;

    int start_pix_idx = im_width + 1;
    int ending_pix_idx = (im_height - 1) * im_width - 1;
    for (int pix_idx = start_pix_idx; pix_idx < ending_pix_idx;
pix_idx++) {
      //   printf("DoG octave, blur_idx, array: %d, %d, %lx\n",
cur_DoG->octave,
      //   blur_idx, (unsigned long) cur_DoG->array);
      double neighborhood[27] = {0};
      get_neighborhood(left_DoG, neighborhood, pix_idx);
      get_neighborhood(cur_DoG, neighborhood + 9, pix_idx);
      get_neighborhood(right_DoG, neighborhood + 18, pix_idx);

      int argmax = get_argmax(neighborhood, 27);
      int argmin = get_argmin(neighborhood, 27);
      // printf("argmin: %d, argmax: %d\n", argmin, argmax);
      if (argmin == 13 ||
          argmax == 13) { // center pix in cur_dog is the min/max
        keypoints->kp_list[keypoints->count] =
malloc(sizeof(Kp));
        Kp *cur_kp = keypoints->kp_list[keypoints->count];
        cur_kp->DoG = cur_DoG;
        cur_kp->DoG_idx = cur_DoG_idx;
        cur_kp->x = pix_idx % im_width;
        cur_kp->y = pix_idx / im_width;
        cur_kp->is_valid = 1; // set all of them as valid for
now, until localization later on;
        keypoints->count++;
        if (keypoints->count > keypoints->_max_capacity) {
          printf("[DEBUG] MAX KEYPOINTS EXCEEDED,
REALLOCING!\n");
          keypoints->kp_list = (Kp **)realloc(keypoints->kp_list,
(keypoints->_max_capacity + MAX_KEYPOINTS) * sizeof(Kp **));
       keypoints->_max_capacity += MAX_KEYPOINTS;
       printf("[DEBUG] MAX KEYPOINTS NOW EQUALS %d\n",
keypoints->_max_capacity);
        }
        // printf("Keypoint x, y, count: %d, %d, %d\n",
cur_kp->x, cur_kp->y,
        //       keypoints->count);
      }
    }
  }
}

void get_candidate_keypoints(DoG **DoGs, Keypoints *keypoints) {
  assert(keypoints->count == 0);
  for (int octave_num = 0; octave_num < NUM_OCTAVES;
octave_num++) {
    printf("Getting candidate keypoints for octave %d\n",
octave_num);
    get_candidate_keypoints_for_octave(DoGs, keypoints,
octave_num);
  }
}

void localize_kp(Kp *kp, DoG **diffs) {
  int row, col, width, DoG_idx;
  row = kp->y;
  col = kp->x;
  width = kp->DoG->width;
  DoG_idx = kp->DoG_idx;

  assert(DoG_idx % 4 == 1 || DoG_idx % 4 == 2);

  int pix_idx = row * width + col;
  double dx = (diffs[DoG_idx]->array[pix_idx + 1] -
               diffs[DoG_idx]->array[pix_idx - 1]) /
              2;
  double dy = (diffs[DoG_idx]->array[pix_idx + width] -
               diffs[DoG_idx]->array[pix_idx - width]) /
              2;
  double ds = (diffs[DoG_idx + 1]->array[pix_idx] -
               diffs[DoG_idx - 1]->array[pix_idx]) /
              2;

  double dxx = diffs[DoG_idx]->array[pix_idx + 1] -
               2 * diffs[DoG_idx]->array[pix_idx] +
               diffs[DoG_idx]->array[pix_idx - 1];

  double dxy = (diffs[DoG_idx]->array[pix_idx + 1 + width] -
                diffs[DoG_idx]->array[pix_idx - 1 + width]) -
               ((diffs[DoG_idx]->array[pix_idx + 1 - width] -
                 diffs[DoG_idx]->array[pix_idx - 1 - width]));

  double dxs = (diffs[DoG_idx + 1]->array[pix_idx + 1] -
                diffs[DoG_idx + 1]->array[pix_idx - 1]) -
               ((diffs[DoG_idx - 1]->array[pix_idx + 1] -
                 diffs[DoG_idx - 1]->array[pix_idx - 1]));

  double dyy = diffs[DoG_idx]->array[pix_idx + width] -
               2 * diffs[DoG_idx]->array[pix_idx] +
               diffs[DoG_idx]->array[pix_idx - width];

  double dys = (diffs[DoG_idx + 1]->array[pix_idx + width] -
                diffs[DoG_idx + 1]->array[pix_idx - width]) -
               ((diffs[DoG_idx - 1]->array[pix_idx + width] -
                 diffs[DoG_idx - 1]->array[pix_idx - width]));

  double dss = diffs[DoG_idx + 1]->array[pix_idx] -
               2 * diffs[DoG_idx]->array[pix_idx] +
               diffs[DoG_idx - 1]->array[pix_idx];

  dxy /= 4;
  dxs /= 4;
  dys /= 4;

  double J[3] = {dx, dy, ds};
  double HD[9] = {dxx, dxy, dxs, dxy, dyy, dys, dxs, dys, dss};
```

```
  double HD_inv[9];
  if (inv_3x3(HD, HD_inv) == -1) {
    printf("There is no inv for hessian!\n");
    print_3x3(HD);
    kp->is_valid=0;
    return;
  }
  double *offset = malloc(3 * sizeof(double));
  dot(HD_inv, J, offset, 3, 3, 1);
  for (int i = 0; i < 3; i++)
    offset[i] *= -1;

  double *dx_dy_ds = malloc(3 * sizeof(double));
  memcpy(dx_dy_ds, J, 3 * sizeof(double));

  double *dxx_dxy_dyx_dyy = malloc(4 * sizeof(double));
  dxx_dxy_dyx_dyy[0] = dxx;
  dxx_dxy_dyx_dyy[1] = dxy;
  dxx_dxy_dyx_dyy[2] = dxy;
  dxx_dxy_dyx_dyy[3] = dyy;

  kp->jacobian = dx_dy_ds;
  kp->hessian = dxx_dxy_dyx_dyy;
  kp->offset = offset;
}

void localize_keypoints(Keypoints *keypoints, DoG **diffs) {
  int idx = 0;
  printf("Localizing %d keypoints...\n", keypoints->count);
  while (idx < keypoints->count) {
    //printf("idx=%d\n",idx);
    Kp *cur_kp = keypoints->kp_list[idx];
    localize_kp(cur_kp, diffs);
    if (cur_kp->is_valid=0) {
      // printf("Cur kp has already been deemed invalid (no inv
for hessian), y: %d, x: %d\n", cur_kp->y, cur_kp->x);
```

```
    }
    else if (fabs(compute_kp_contrast(cur_kp, diffs)) <
CONTRAST_THRESHOLD) {
       // printf("Discarding kp because contrast is too low, y:
%d, x: %d\n",
       //      cur_kp->y, cur_kp->x);
      cur_kp->is_valid = 0;
    } else if (get_det_2x2(cur_kp->hessian) < 0) {
      // printf("Discarding kp because hessian determinant is
negative, y: %d, "
      //        "x:%d\n",  cur_kp->y, cur_kp->x);
      cur_kp->is_valid = 0;
    } else if (pow(get_trace_2x2(cur_kp->hessian), 2) /
                    get_det_2x2(cur_kp->hessian) >=
                PRINCIPLE_CURVATURE_THRESHOLD) {
      // printf("Discarding kp bc principal curvature is too
large, y: %d, "
      //       "x: %d\n", cur_kp->y, cur_kp->x);
      cur_kp->is_valid = 0;
    } else if ((cur_kp->x + cur_kp->offset[0] >
cur_kp->DoG->width) ||
               (cur_kp->y + cur_kp->offset[1] >
cur_kp->DoG->height)) {
      // printf("Discarding keypoint because localized x or y is
too large, x: "
      //       "%f, y: %f\n", cur_kp->x + cur_kp->offset[0],
cur_kp->y + cur_kp->offset[1]);
              cur_kp->is_valid = 0;
    } else if ((cur_kp->x + cur_kp->offset[0] < 0) || (cur_kp->y
+ cur_kp->offset[1] < 0)){
      // printf("Discarding keypoint because localized x or y is
negative, x: "
      //       "%f, y: %f\n", cur_kp->x + cur_kp->offset[0],
cur_kp->y + cur_kp->offset[1]);
              cur_kp->is_valid = 0;
    }
```

```
      else {
      Kp_precise *kp_prec = malloc(sizeof(Kp_precise));
      kp_prec->x = cur_kp->x + cur_kp->offset[0];
      kp_prec->y = cur_kp->y + cur_kp->offset[1];
      kp_prec->DoG_idx = cur_kp->DoG_idx + cur_kp->offset[2];

      cur_kp->is_valid = 1;

      int octave_num = cur_kp->DoG_idx / (NUM_BLURS-1);
      int blur_idx = cur_kp->DoG_idx % (NUM_BLURS-1);

      int width = cur_kp->DoG->width;
      int height = cur_kp->DoG->height;
      int width_diff = get_octave_im_dim(IMAGE_WIDTH, octave_num)
- width;
      int height_diff = get_octave_im_dim(IMAGE_HEIGHT,
octave_num) - height;

      kp_prec->real_y = (height_diff / 2) * (1 << octave_num) +
kp_prec->y * (1<<octave_num);
      kp_prec->real_x = (width_diff / 2) * (1<<octave_num) +
kp_prec->x * (1<<octave_num);

      assert(height_diff == 24);
      assert(width_diff == 24);

      if ((double)rand() / RAND_MAX < 0.001) {
          printf("get_octave_im_dim(IMAGE_WIDTH, octave_num): %d,
width: %d\n", get_octave_im_dim(IMAGE_WIDTH, octave_num), width);
          printf("get_octave_im_dim(IMAGE_HEIGHT, octave_num) %d,
height: %d\n", get_octave_im_dim(IMAGE_HEIGHT, octave_num),
height);
          printf("Localized kp!\n");
          printf("Octave: %d, Blur: %d\n", octave_num, blur_idx);
          printf("DoG width, height: %d, %d, width_diff,
height_diff: %d, %d\n", width, height, width_diff, height_diff);
```

```
          printf("Orig x, y: %d, %d\n", cur_kp->x, cur_kp->y);
          printf("real_x: %lf, real_y: %lf, kp_prec->x: %f,
kp_prec->y: %f\n", kp_prec->real_x, kp_prec->real_y, kp_prec->x,
kp_prec->y);
      }

      cur_kp->kp_precise = kp_prec;

    }

    idx++;
  }
  print_keypoints_stats(keypoints);
}

void print_keypoints_stats(Keypoints *keypoints) {
     int idx = 0;
     int num_valid = 0;
     while (idx < keypoints->count) {
          if (keypoints->kp_list[idx]->is_valid) {
               num_valid++;
          }
          idx++;
     }
     printf("Total keypoints: %d. Total valid: %d, Total invalid:
%d\n", keypoints->count, num_valid, keypoints->count-num_valid);
}

// format: row col octave_num blur_idx is_valid
void write_kps_to_txt_file(Keypoints *keypoints, const char
*file_path) {
    printf("Writing %d keypoints to file %s...\n",
keypoints->count, file_path);
    FILE *file = fopen(file_path, "w");
    if (!file) {
        perror("Error opening file for writing");
        return;
    }

    for (int i = 0; i < keypoints->count; i++) {
      Kp *cur_kp = keypoints->kp_list[i];
      if (!cur_kp->is_valid) continue;
      fprintf(file, "%f %f %d %d", cur_kp->kp_precise->real_y,
cur_kp->kp_precise->real_x, cur_kp->DoG_idx/(NUM_BLURS-1),
cur_kp->DoG_idx % (NUM_BLURS - 1));
      fprintf(file, "\n");
    }
    printf("Wrote %d keypoints to file %s!\n", keypoints->count,
file_path);

    fclose(file);
}
```

## lin_alg.h

C/C++
```
#ifndef _LIN_ALG
#define _LIN_ALG

int inv_3x3(double *arr, double *output);
void dot(double *arr1, double *arr2, double *out, int i, int j,
int k);
void print_3x3(double *arr);
void get_eigvals_2x2(double *arr, double *output);
double get_trace_2x2(double *arr);
double get_det_2x2(double *arr);

#endif
```

## lin_alg.c

C/C++
```
#include "lin_alg.h"
#include <stdio.h>
#include <string.h>
#include <math.h>

double get_trace_2x2(double *arr) {
    return arr[0] + arr[3];
}

double get_det_2x2(double *arr) {
    return arr[0] * arr[3] - arr[1] * arr[2];
}

// λ = (-b ± sqrt(b^2 - 4ac)) / 2a, where a=1, b=-(a+d), and c =
ad-bc
void get_eigvals_2x2(double *arr, double *output) {
    double a = 1;
    double b = -(arr[0] + arr[3]);
    double c = arr[0] * arr[3] - arr[1] * arr[2];

    output[0] = (-b - sqrt(b*b - 4*a*c)) / (2 * a);
    output[1] = (-b + sqrt(b*b - 4*a*c)) / (2 * a);
}

void print_3x3(double *arr) {
    for (int i=0; i<9; i++) {
        if (i % 3 == 2)
            printf("%f\n", arr[i]);
        else
            printf("%f, ", arr[i]);
    }
}

int inv_3x3(double *arr, double *output) {
    double det = arr[0] * (arr[4]*arr[8] - arr[7]*arr[5]) -
arr[1] * (arr[3]*arr[8] - arr[6]*arr[5]) + arr[2] *
(arr[3]*arr[7] - arr[6]*arr[4]);
    if (det == 0) return -1;
    output[0] = arr[4] * arr[8] - arr[7] * arr[5];
    output[1] = arr[7] * arr[2] - arr[1] * arr[8];
    output[2] = arr[1] * arr[5] - arr[4] * arr[2];
    output[3] = arr[5] * arr[6] - arr[8] * arr[3];
    output[4] = arr[8] * arr[0] - arr[2] * arr[6];
    output[5] = arr[2] * arr[3] - arr[5] * arr[0];
    output[6] = arr[3] * arr[7] - arr[4] * arr[6];
    output[7] = arr[6] * arr[1] - arr[0] * arr[7];
    output[8] = arr[0] * arr[4] - arr[3] * arr[1];
    for (int i=0; i<9; i++) {
        output[i] /= det;
    }
    return 0;
}

// Matrix with dims ixj dot product with matrix with dims jxk
void dot(double *arr1, double *arr2, double *out, int i, int j,
int k) {
    memset(out, 0, i*k*sizeof(double));
    for (int _i=0; _i<i; _i++) {
        for (int _j=0; _j<j; _j++) {
            for (int _k=0; _k<k; _k++) {
            out[_i * k + _k] += arr1[_i * j + _j] * arr2[_j * k +
_k];
            }
        }
    }
}

// int main() {
//     double arr[9] = {1,2,3,4,5,6,7,8,9};
//     double arr2[9] = {2,3,4,5,6,7,8,9,10};
//     double out[9] = {0};
//     dot(arr, arr2, out, 3, 3, 3);
//     print_3x3(out);
// }

// int main() {
//     double arr[9] = {1,1,0,1,0,1,1,1,1};
//     // output should be {1,1,-1,0,-1,1,-1,0,1};
//     double output[9] = {};
//     inv_3x3(arr, output);
//     print_3x3(output);
// }
```

## Im_utils.h

```
C/C++
#ifndef IM_UTILS
#define IM_UTILS

void read_image_data(const char* filename, char* image_data, int
img_size);
void write_image_to_txt_file(const char image[], const char
*file_path, int img_size);
void write_float_image_to_txt_file(const double image[], const
char *file_path, int img_size);

#endif
```

## Im_utils.c

```
C/C++
#include "vga_ball.h"
#include "keypoints.h"
```

```c
#include <stdio.h>
#include <stdlib.h>

int vga_ball_fd;

void read_image_data(const char* filename, char* image_data, int
img_size) {
    FILE* file = fopen(filename, "rb");
    if (file == NULL) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    size_t elements_read = fread(image_data, sizeof(char),
img_size, file);
    if (elements_read != img_size) {
        perror("Error reading file");
        fclose(file);
        exit(EXIT_FAILURE);
    }

    fclose(file);
}

void write_image_to_txt_file(const char image[], const char
*file_path, int img_size) {
    FILE *file = fopen(file_path, "wb");
    if (!file) {
        perror("Error opening file for writing");
        return;
    }

    for (int i = 0; i < img_size; i++) {
      fprintf(file, "%d ", (unsigned char) image[i]);
      fprintf(file, "\n");
    }
    printf("Wrote image to file %s!\n", file_path);

    fclose(file);
}

void write_float_image_to_txt_file(const double image[], const
char *file_path, int img_size) {
    FILE *file = fopen(file_path, "wb");
    if (!file) {
        perror("Error opening file for writing");
        return;
    }

    for (int i = 0; i < img_size; i++) {
      fprintf(file, "%f ", image[i]);
      fprintf(file, "\n");
    }
    printf("Wrote image to file %s!\n", file_path);

    fclose(file);
}
```

vga_ball_utils.h

C/C++
```
#ifndef VGA_BALL_UTILS
#define VGA_BALL_UTILS

#include "vga_ball.h"

int read_background_color(vga_ball_arg_t *vla);
void print_background_color();
char set_background_color(const vga_ball_color_t *c);
void set_value(vga_ball_value_t *v);
void clear_bram(vga_ball_value_t *v);
void read_value(char *image_data, int start_index);

#endif
```

vga_ball_utils.c

```
C/C++
#include "vga_ball.h"
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int vga_ball_fd;

int read_background_color(vga_ball_arg_t *vla) {
  if (ioctl(vga_ball_fd, VGA_BALL_READ_BACKGROUND, vla)) {
    perror("ioctl(VGA_BALL_READ_BACKGROUND) failed");
    return -1;
  }
  return 0;
}

/* Read and print the background color */
void print_background_color() {
  vga_ball_arg_t vla;

  if (ioctl(vga_ball_fd, VGA_BALL_READ_BACKGROUND, &vla)) {
    perror("ioctl(VGA_BALL_READ_BACKGROUND) failed");
    return;
  }
  printf("%02x %02x %02x\n", vla.background.red,
vla.background.green,
         vla.background.blue);
}

/* Set the background color */
char set_background_color(const vga_ball_color_t *c) {
  vga_ball_arg_t vla, temp;
  vla.background = *c;
  char t = 1;
  if (ioctl(vga_ball_fd, VGA_BALL_WRITE_BACKGROUND, &vla)) {
    perror("ioctl(VGA_BALL_SET_BACKGROUND) failed");
    return t;
  }

  while (t == 1) {
     //printf("%d \n",vla.image[0]);
    if (ioctl(vga_ball_fd, VGA_BALL_READ_VALUE, &temp)) {
      perror("ioctl(VGA_BALL_READ_VALUE) failed");
      return t;
    }
    t = temp.image[0];
    //printf("t=%d\n",t);
    usleep(100000);
  }
  return t;
}

/* Set the value */
void set_value(vga_ball_value_t *v) {
  vga_ball_arg_t vla;
  vla.value = *v;
  if (ioctl(vga_ball_fd, VGA_BALL_WRITE_VALUE, &vla)) {
```

```c
    perror("ioctl(VGA_BALL_WRITE_VALUE) failed");
    return;
  }
}

/* Clear BRAM memory */
void clear_bram(vga_ball_value_t *v) {
  vga_ball_arg_t vla;
  vla.value = *v;
  if (ioctl(vga_ball_fd, VGA_BALL_MEM_CLEAR, &vla)) {
    perror("ioctl(VGA_BALL_MEM_CLEAR) failed");
    return;
  }
}

/* Read and print the value from hardware */
void read_value(char *image_data, int start_index) {
  vga_ball_arg_t vla;

  // vla.value.addr = address;
  if (ioctl(vga_ball_fd, VGA_BALL_READ_VALUE, &vla)) {
    perror("ioctl(VGA_BALL_READ_VALUE) failed");
    return;
  }
  // usleep(200000);
  // printf("value at address %u is %u\n",address,
vla.value.val);
  for (int i = 0; i < BUFFER_IMAGE_SIZE; i++) {
    // printf("%d \n",vla.image[0]);
    image_data[start_index] = vla.image[i];
  }
}
```

vga_ball.h

C/C++
```c
#ifndef _VGA_BALL_H
#define _VGA_BALL_H

#include <linux/ioctl.h>
#define IMAGE_HEIGHT 350
#define IMAGE_WIDTH 525
# define IMAGE_SIZE IMAGE_HEIGHT * IMAGE_WIDTH
# define BUFFER_IMAGE_SIZE 1
//# define NUM_IOCTL_READS 10000
# define NUM_IOCTL_READS IMAGE_SIZE/BUFFER_IMAGE_SIZE
typedef struct {
      char red, green, blue;
} vga_ball_color_t;

typedef struct {
      char x, y;
} vga_ball_pos_t;

typedef struct {
      char val;
        unsigned int addr;
} vga_ball_value_t;

typedef struct {
  vga_ball_color_t background;
  //vga_ball_pos_t position;
  vga_ball_value_t value;
  char image[BUFFER_IMAGE_SIZE];
} vga_ball_arg_t;


#define VGA_BALL_MAGIC 'q'

/* ioctls and their arguments */
```

```
#define VGA_BALL_WRITE_BACKGROUND _IOW(VGA_BALL_MAGIC, 1,
vga_ball_arg_t *)
#define VGA_BALL_READ_BACKGROUND  _IOR(VGA_BALL_MAGIC, 2,
vga_ball_arg_t *)
//#define VGA_BALL_WRITE_POSITION   _IOW(VGA_BALL_MAGIC, 3,
vga_ball_arg_t *)
#define VGA_BALL_WRITE_VALUE      _IOW(VGA_BALL_MAGIC, 3,
vga_ball_arg_t *)
#define VGA_BALL_READ_VALUE       _IOR(VGA_BALL_MAGIC, 4,
vga_ball_arg_t *)
#define VGA_BALL_MEM_CLEAR        _IOR(VGA_BALL_MAGIC, 5,
vga_ball_arg_t *)

#endif
```

vga_ball.c

```
C/C++
#if HW_ENABLE

/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_ball.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_ball.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_ball.h"

#define DRIVER_NAME "vga_ball"

/* Device registers */
#define BG_RED(x) (x)
#define BG_GREEN(x) ((x)+1)
#define BG_BLUE(x) ((x)+2)
#define MEM_CLEAR(x) ((x)+3)
#define WRITE_VALUE(x) ((x)+4)

/*
 * Information about our device
 */
struct vga_ball_dev {
	struct resource res; /* Resource: our registers */
	void __iomem *virtbase; /* Where registers can be accessed
in memory */
```

```
        vga_ball_color_t background;
        //vga_ball_pos_t position;
        vga_ball_value_t value;
} dev;


static void write_background(vga_ball_color_t *background)
{
      iowrite8(background->red, BG_RED(dev.virtbase) );
      iowrite8(background->green, BG_GREEN(dev.virtbase) );
      iowrite8(background->blue, BG_BLUE(dev.virtbase) );
      dev.background = *background;
}

static void write_value(vga_ball_value_t *value)
{
        iowrite8(value->val,
WRITE_VALUE(dev.virtbase)+value->addr);
      dev.value = *value;
}

static void mem_clear(vga_ball_value_t *value)
{
        iowrite8(value->val, MEM_CLEAR(dev.virtbase));
      dev.value = *value;
}

static void read_im_from_hw(char *image)
{
    int i = 0;
    for (; i < BUFFER_IMAGE_SIZE; i++) {
        image[i] = ioread8(dev.virtbase);
        //image[i] = ioread8(dev.virtbase);
    }
}
```

```
/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_ball_ioctl(struct file *f, unsigned int cmd,
unsigned long arg)
{
	vga_ball_arg_t vla;

	switch (cmd) {
	case VGA_BALL_WRITE_BACKGROUND:
		if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
				   sizeof(vga_ball_arg_t)))
			return -EACCES;
		write_background(&vla.background);
		break;

	case VGA_BALL_READ_BACKGROUND:
		vla.background = dev.background;
		if (copy_to_user((vga_ball_arg_t *) arg, &vla,
				 sizeof(vga_ball_arg_t)))
			return -EACCES;
		break;

	case VGA_BALL_WRITE_VALUE:
		if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
				   sizeof(vga_ball_arg_t)))
			return -EACCES;
		write_value(&vla.value);
		break;
	case VGA_BALL_READ_VALUE:
		read_im_from_hw(vla.image);
		if (copy_to_user((vga_ball_arg_t *)arg, &vla,
sizeof(vga_ball_arg_t)))
```

```
               return -EACCES;
           break;
     case VGA_BALL_MEM_CLEAR:
           if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                          sizeof(vga_ball_arg_t)))
                return -EACCES;
           mem_clear(&vla.value);

     default:
           return -EINVAL;
     }

     return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_ball_fops = {
     .owner          = THIS_MODULE,
     .unlocked_ioctl = vga_ball_ioctl,
};

/* Information about our device for the "misc" framework -- like
a char dev */
static struct miscdevice vga_ball_misc_device = {
     .minor          = MISC_DYNAMIC_MINOR,
     .name      = DRIVER_NAME,
     .fops      = &vga_ball_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_ball_probe(struct platform_device *pdev)
{
        vga_ball_color_t beige = { 0x00, 0x00, 0xff};
```

```
      int ret;

      /* Register ourselves as a misc device: creates
/dev/vga_ball */
      ret = misc_register(&vga_ball_misc_device);

      /* Get the address of our registers from the device tree */
      ret = of_address_to_resource(pdev->dev.of_node, 0,
&dev.res);
      if (ret) {
            ret = -ENOENT;
            goto out_deregister;
      }

      /* Make sure we can use these registers */
      if (request_mem_region(dev.res.start,
resource_size(&dev.res),
                          DRIVER_NAME) == NULL) {
            ret = -EBUSY;
            goto out_deregister;
      }

      /* Arrange access to our registers */
      dev.virtbase = of_iomap(pdev->dev.of_node, 0);
      if (dev.virtbase == NULL) {
            ret = -ENOMEM;
            goto out_release_mem_region;
      }

      /* Set an initial color */
         write_background(&beige);

      return 0;

out_release_mem_region:
      release_mem_region(dev.res.start, resource_size(&dev.res));
```

```
out_deregister:
      misc_deregister(&vga_ball_misc_device);
      return ret;
}

/* Clean-up code: release resources */
static int vga_ball_remove(struct platform_device *pdev)
{
      iounmap(dev.virtbase);
      release_mem_region(dev.res.start, resource_size(&dev.res));
      misc_deregister(&vga_ball_misc_device);
      return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree
*/
#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
      { .compatible = "csee4840,vga_ball-1.0" },
      {},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

/* Information for registering ourselves as a "platform" driver
*/
static struct platform_driver vga_ball_driver = {
      .driver    = {
            .name = DRIVER_NAME,
            .owner     = THIS_MODULE,
            .of_match_table = of_match_ptr(vga_ball_of_match),
      },
      .remove    = __exit_p(vga_ball_remove),
};

/* Called when the module is loaded: set things up */
```

```
static int __init vga_ball_init(void)
{
      pr_info(DRIVER_NAME ": init\n");
      return platform_driver_probe(&vga_ball_driver,
vga_ball_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit vga_ball_exit(void)
{
      platform_driver_unregister(&vga_ball_driver);
      pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA ball driver");

#endif
```

## Makefile

C/C++
```
CC = gcc
CXX = g++
INCLUDES=
ccflags-y = -mfloat-abi=soft -g -Wall $(INCLUDES)
LDFLAGS = -g
LDLIBS = -lm

ifneq (${KERNELRELEASE},)
```

```
# KERNELRELEASE defined: we are being compiled as part of the
Kernel
        obj-m := vga_ball.o

else

# We are being compiled as a module: use the Kernel build system

     KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
        PWD := $(shell pwd)

default: module hello

module:
     ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

main: sift_utils.o im_utils.o vga_ball.o vga_ball_utils.o
keypoints.o lin_alg.o descriptors.o main.o
     $(CC) $(LDFLAGS) $^ -o $@ $(LDLIBS)

main.o:  main.c
     $(CC) $(CFLAGS) -c $^ -o $@

hello: hello.o vga_ball.o
hello.o: hello.c

vga_ball: vga_ball.o vga_ball.h
vga_ball.o: vga_ball.h vga_ball.c

sift_utils.o: sift_utils.h sift_utils.c
im_utils.o: im_utils.h im_utils.c
vga_ball_utils.o: vga_ball_utils.h vga_ball_utils.c
keypoints.o: keypoints.h keypoints.c
lin_alg.o: lin_alg.h lin_alg.c
descriptors.o: descriptors.h descriptors.c
```

```
.PHONY: clean_main
clean_main:
	rm *.o main

.PHONY: run
run: clean_main main
	./main

clean:
	${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
	${RM} hello

TARFILES = Makefile README vga_ball.h vga_ball.c hello.c
TARFILE = lab3-sw.tar.gz
.PHONY : tar
tar : $(TARFILE)

$(TARFILE) : $(TARFILES)
	tar zcfC $(TARFILE) .. $(TARFILES:%=lab3-sw/%)

endif
```

vga_ball.sv

```
Unset
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module vga_ball(input logic      clk,
           input logic      reset,
       input logic [7:0]  writedata,
       output logic [7:0] readdata,
       input logic    write,
       input          chipselect,
       input logic [17:0]  address,

       output logic [7:0] VGA_R, VGA_G, VGA_B,
       output logic        VGA_CLK, VGA_HS, VGA_VS,
                   VGA_BLANK_n,
       output logic        VGA_SYNC_n);

   parameter IMG_WIDTH  = 525;              //Reduced 600x400
image size to this value as FPGA runs out of computational
resources.
   parameter IMG_HEIGHT = 350;
   parameter IMG_SIZE   = IMG_WIDTH * IMG_HEIGHT;

   logic [10:0]      hcount;          //Gives column number
   logic [9:0]  vcount;          //Gives row number
   logic [7:0]       background_r, background_g, background_b;

   logic [17:0] array_address;
   logic        print_pixel;
   logic [7:0]       readmem = 1;

   logic [17:0] address_minus_4;

   logic [17:0] address_in_cache = 0;
   logic [7:0]  cache = 8'b0000_0000;
   logic [7:0]  out_pixel_cache = 8'b0000_0000;
   logic        async_clr = 0;
   logic        read_flag = 0;
   logic        write_flag = 0;
   logic [17:0] conv_input_pixel_addr_start = 18'd0;

   logic [17:0] kernel_size_sq;
   logic [17:0] kernel_size;
   logic [17:0] img_width_orig = 18'd525;
   logic [17:0] img_height_orig = 18'd350;
   //logic [17:0]     img_width_orconv_input_pixel_addrig =
18'd262;              //What if the image is actually smaller?
   //logic [17:0]     img_height_orig = 18'd175;

   logic [17:0] img_width;                        //When you
resize, your image width and height will change
   logic [17:0] img_height;
   logic [17:0]  conv_input_pixel_addr = 18'd0;
   logic [17:0]  conv_row_change_offset_kernel;
   logic [17:0]  conv_row_change_offset_image;

   logic [15:0] kernel[1412:0];

   logic [31:0] conv_out = 0;
   logic [7:0] pixel_out;

   logic [2:0]   count = 0; // counter used to wait for every nth
clock cycle for convolution (we use every 8th cc) (on 3 it reads
a pixel from memory and on 7 it does the convolution for that
pixel)

   logic [17:0] idx_offset = 18'b0000_0000_0000_0000_00; // idx
into kernel
   logic   conv_done = 0;

   /*
   logic [17:0]  row_change_idx_2 = 17;
   logic [17:0]  row_change_idx_3 = 26;
   logic [17:0]  row_change_idx_4 = 35;
   logic [17:0]  row_change_idx_5 = 44;
   logic [17:0]  row_change_idx_6 = 53;
   logic [17:0]  row_change_idx_7 = 62;
   logic [17:0]  row_change_idx_8 = 71;
   logic [17:0] conv_out_next_row_end = 18'd517;
//img_width-kernel_size_9+1
   logic [17:0] conv_out_next_row_end = 18'd515;
//img_width-kernel_size_11+1
   logic [17:0] conv_out_next_row_end = 18'd501;
//img_width-kernel_size_25+1
   */

   logic [17:0] out_pixels_idx = 0;
   logic out_pixels_wren = 1'd0;

   logic [17:0] conv_out_width;                   //Output sizes,
depends on image size and kernel
   logic [17:0] conv_out_height;
   logic [17:0] conv_out_end;
   //logic [17:0] conv_out_next_row_end;
   logic [17:0] kernel_select_offset;
   //logic [17:0] conv_out_next_row_end_start;
   logic [40:0] conv_out_scaled;
   logic [17:0] step_size;

   logic [17:0] mux_rdaddress;
   logic [7:0] img_q;
   logic [17:0] width_gap;
   logic [17:0] height_gap;

   always_comb begin
```

```
      // Configs for kernels
      if (background_r[6]) begin // kernel 9x9
      kernel_size = 18'd9;
      kernel_size_sq = 18'd81;
      kernel_select_offset = 18'd0;
      //conv_out_next_row_end_start = 18'd517;
      conv_out_scaled = conv_out * 9'd314;
      pixel_out = conv_out_scaled[32:25];          //shift by 25
bits. Dividing by 2^25
      end
      else if (background_r[5]) begin // kernel 11x11
      kernel_size = 18'd11;
      kernel_size_sq = 18'd121;
      kernel_select_offset = 18'd81;
      //conv_out_next_row_end_start = 18'd515;
      conv_out_scaled = conv_out * 9'd463;
      pixel_out = conv_out_scaled[32:25];    //shift by 25 bits.
Dividing by 2^25
      end
      else if (background_r[4]) begin // kernel 15x15
      kernel_size = 18'd15;
      kernel_size_sq = 18'd225;
      kernel_select_offset = 18'd202;
      //conv_out_next_row_end_start = 18'd511;
      conv_out_scaled = conv_out * 9'd377;
      pixel_out = conv_out_scaled[34:27];    //shift by 27 bits.
Dividing by 2^27
      end
      else if (background_r[3]) begin // kernel 19x19
      kernel_size = 18'd19;
      kernel_size_sq = 18'd261;
      kernel_select_offset = 18'd427;
      //conv_out_next_row_end_start = 18'd507;
      conv_out_scaled = conv_out * 9'd347;
      pixel_out = conv_out_scaled[34:27];    //shift by 27 bits.
Dividing by 2^27
      end
      else if (background_r[2]) begin // kernel 25x25
      kernel_size = 18'd25;
      kernel_size_sq = 18'd625;
      kernel_select_offset = 18'd788;
      //conv_out_next_row_end_start = 18'd501;
      conv_out_scaled = conv_out * 9'd331;
      pixel_out = conv_out_scaled[35:28];    //shift by 28 bits.
Dividing by 2^28
      end
      else begin
      kernel_size = 18'd9;
      kernel_size_sq = 18'd81;
      kernel_select_offset = 18'd0;
      //conv_out_next_row_end_start = 18'd517;
      conv_out_scaled = conv_out * 9'd314;
      pixel_out = conv_out_scaled[32:25];    //shift by 25 bits.
Dividing by 2^25
      end
```

```
     conv_out_width = img_width - kernel_size + 18'd1;  //By
default kernel size is 18'd9 and img_width = 525 then,
conv_out_width = 525 - 9 + 1 = 517

     // Configs for octaves
     if (background_g[6]) begin // octave original
     img_width = img_width_orig; // 525
     img_height = img_height_orig; // 350

     conv_row_change_offset_kernel = conv_out_width;    //
     conv_row_change_offset_image = kernel_size;    //
     step_size = 18'd1;
     end
     else if (background_g[5]) begin // octave divde by 2
                         //To understand this, read line 249
     img_width = (img_width_orig>>1) + img_width_orig[0]; // 263
- 8                            //525 >> 1 = 262 + 1 = 263
//From where is the -8 coming?
     img_height = (img_height_orig>>1) + img_height_orig[0]; //
175 - 8                  //350 >> 1 = 175 + 0 = 175

     conv_row_change_offset_kernel = (img_width_orig -
kernel_size + 1)<<1;                         //What is the value
of kernel_size?
     conv_row_change_offset_image = (kernel_size<<1) +
img_width_orig - (1-(img_width_orig-1)%2);        // 18'd1;
     step_size = 18'd2;
     end
```

```
      else if (background_g[4]) begin // octave divde by 4
      img_width = (img_width_orig>>2) + img_width_orig[1];
                             //525 >> 2 = 131 + 0 = 131
      img_height = (img_height_orig>>2) + img_height_orig[1];
                             //350 >> 2 = 87 + 1  = 88

      conv_row_change_offset_kernel = (img_width_orig -
kernel_size + 1)<<2;
      conv_row_change_offset_image = (kernel_size<<2) +
(img_width_orig*3)+1;
      step_size = 18'd4;
      end
      else if (background_g[3]) begin // octave divde by 8
      img_width = (img_width_orig>>3) + img_width_orig[2]; //
66-24
      img_height = (img_height_orig>>3) + img_height_orig[2]; //
44-24

      conv_row_change_offset_kernel = (img_width_orig -
kernel_size + 1)<<3;
      conv_row_change_offset_image = (kernel_size<<3) +
(img_width_orig*7)-3;
      step_size = 18'd8;
      end
      else begin
      img_width = img_width_orig;
      img_height = img_height_orig;
```

```
        conv_row_change_offset_kernel = conv_out_width;
        conv_row_change_offset_image = kernel_size;
        step_size = 18'd1;
        end

        conv_out_height = img_height - kernel_size + 18'd1;
        conv_out_end = conv_out_width*conv_out_height;

        if (VGA_BLANK_n) begin
        if (print_pixel) begin
        if (readmem)
             {VGA_R, VGA_G, VGA_B} = {readmem, readmem, readmem};
        else
             {VGA_R, VGA_G, VGA_B} = {8'd255, 8'd0, 8'd0};
        end
        else
        {VGA_R, VGA_G, VGA_B} = {background_r, background_g,
background_b};
        end
        else
        {VGA_R, VGA_G, VGA_B} = {8'd128, 8'd128, 8'd128};

     end
     assign width_gap = (18'd640-img_width_orig)>>1;
     assign height_gap = (18'd480-img_height_orig)>>1;

     //assign print_pixel   = ((hcount[10:1] < (conv_out_width)) &&
(vcount < conv_out_height));
```

```
   assign print_pixel   = ( (hcount[10:1] > width_gap+1) &&
(hcount[10:1] < (img_width_orig+width_gap)) && (vcount >
height_gap) && (vcount < (img_height_orig+height_gap)) );

   //assign array_address = (print_pixel ? (((conv_out_width) *
vcount) + hcount[10:1]+1) : 0);
   assign array_address = (print_pixel ? ( ((img_width_orig) *
(vcount-height_gap)) + (hcount[10:1]-width_gap) ) : 0);

   //assign {VGA_R, VGA_G, VGA_B} = (VGA_BLANK_n ? (print_pixel ?
({readmem, readmem, readmem}) : ({background_r, background_g,
background_b})) : {8'h0, 8'h0, 8'h0});

   assign address_minus_4 = address-18'd4;

   assign mux_rdaddress = out_pixels_wren ?
conv_input_pixel_addr+18'd1 : array_address;

   assign readmem = ~out_pixels_wren ? img_q : 8'b0;

   assign cache = out_pixels_wren ? img_q : 8'b0;


     `include "hardcoded_kernels/kernel_all.sv"

   mem img_memory(
     .aclr(async_clr),
     .clock(clk),
//.rdaddress(array_address), // address to read
     //.rdaddress(conv_input_pixel_addr+1),
     //.rdaddress(array_address),

     .rdaddress(mux_rdaddress),
     .wraddress(address_minus_4),          // address to write
     //.q(readmem),
     .q(img_q),                            // result from
reading is stored here
     //.q(cache),
     .wren(chipselect && write),           // write enable
     .data(writedata)                      // data to write
   );

   mem out_pixels(
     .aclr(async_clr),
     .clock(clk),

//.rdaddress(array_address), // address to read
     .rdaddress(address_in_cache),
     .wraddress(out_pixels_idx),      // address to write
                                       //.q(readmem),
     .q(out_pixel_cache),                         // result from
reading is stored here
     .wren(out_pixels_wren && conv_done),         // write enable
     .data(pixel_out)                      // data to write
```

```
                                   //.data(conv_out[7:0])
     // data to write
   );

   vga_counters counters(.clk50(clk), .*);

   always_ff @(posedge clk) begin

     if (conv_done == 0 && out_pixels_wren == 1) begin
     count <= count+1;
           if (idx_offset == 0) begin
                 conv_input_pixel_addr <=
conv_input_pixel_addr_start;
     end
           if (count == 1) begin             // calculate
conv_out += kernel_val * pixel
                 count <= 0;
      conv_out <= conv_out + cache *
kernel[kernel_select_offset+idx_offset]; // do the multiplcation
and accum the result        //Kernel offset is required to start
reading the correct kernel as all kernels are stored serially in
memory. idx_offset traverses the elements of the kernel
      if (idx_offset == kernel_size_sq-18'b1) begin
// reset the idx that idxes into kernel
           idx_offset <= 0;
           conv_done <= 1;
      end
                 else begin
           idx_offset <= idx_offset+1;
                if ((idx_offset%kernel_size) == (kernel_size-1))
begin
                     conv_input_pixel_addr <=
conv_input_pixel_addr + conv_row_change_offset_kernel;
                end
                else
           conv_input_pixel_addr <= conv_input_pixel_addr +
step_size;
           end
      end
      end
      else if (out_pixels_wren == 1) begin

      if (out_pixels_idx == (conv_out_end-18'd1)) begin
      out_pixels_wren <= 1'd0;
      out_pixels_idx <= 0;
      conv_input_pixel_addr <= 0;
      conv_input_pixel_addr_start <= 0;
      //conv_out_next_row_end <= conv_out_width;
      end
      else if ( (out_pixels_idx%conv_out_width) ==
(conv_out_width-18'd1) ) begin // 500, 500+501, 500+501*2
      //end else if ( out_pixels_idx ==
(conv_out_next_row_end-18'd1)) begin
      out_pixels_idx <= out_pixels_idx + 18'd1;
      //conv_input_pixel_addr_start <= conv_input_pixel_addr_start
+ kernel_size;
```

```
       conv_input_pixel_addr_start <= conv_input_pixel_addr_start
+ conv_row_change_offset_image;
      //conv_out_next_row_end <= conv_out_next_row_end +
conv_out_width;
      end
      else begin
      //if (out_pixels_idx == 18'd0)
            //conv_out_next_row_end <= conv_out_width;
      out_pixels_idx <= out_pixels_idx + 18'd1;
      conv_input_pixel_addr_start <= conv_input_pixel_addr_start +
step_size;
      end

      conv_out <= 0;
      conv_done <= 0;
      count <= 0;
      end


      if (async_clr == 1'b1) begin
      async_clr <= 1'b0;
      address_in_cache <= 18'b0;
      out_pixels_wren <= 1'd0;
      out_pixels_idx <= 0;
      conv_input_pixel_addr <= 0;
      conv_input_pixel_addr_start <= 0;
      //conv_out_next_row_end <= conv_out_width;
      end
```

```
    if (address_in_cache > (conv_out_end-18'd1))
    address_in_cache <= 18'b0;
    if (reset) begin
   background_r <= 8'h0;
   background_g <= 8'h0;
   background_b <= 8'h80;
    async_clr <= 1'b1;
    address_in_cache <= 18'b0;
    out_pixels_wren <= 1'd0;
    out_pixels_idx <= 0;
    conv_input_pixel_addr <= 0;
    conv_input_pixel_addr_start <= 0;
    //conv_out_next_row_end <= conv_out_width;
    end else if (chipselect && write) begin
    case (address)
   3'h0 : begin
     background_r <= writedata[7:0];
              if (writedata[7] && ~write_flag) begin
              out_pixels_wren <= 1'd1;
       out_pixels_idx <= 0;
       conv_input_pixel_addr <= 0;
       conv_input_pixel_addr_start <= 0;
              address_in_cache <= 18'b0;
       //conv_out_next_row_end <= conv_out_width;
              end
    end
   3'h1 : background_g <= writedata[7:0];
   3'h2 : background_b <= writedata[7:0];
```

```
     3'h3 : async_clr <= 1'b1;
      endcase
      write_flag <= 1'b1;
      end else if (chipselect && ~write) begin
      //readdata <= pixel_out[7:0];
      //readdata <= out_pixels_reg[address_in_cache];
      if (out_pixels_wren)
      readdata <= 8'b1;
      else begin
      readdata <= out_pixel_cache;
      if (~read_flag)
           address_in_cache <= address_in_cache + 18'b1;
      read_flag <= 1'b1;
      end
      end
      else if (read_flag == 1'b1)
      read_flag <= 1'b0;
      else if (write_flag == 1'b1)
      write_flag <= 1'b0;

   end

endmodule

module vga_counters(
 input logic          clk50, reset,
 output logic [10:0] hcount,  // hcount[10:1] is pixel column
 output logic [9:0]  vcount,  // vcount[9:0] is pixel row
 output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other
cycle
 *
 * HCOUNT 1599 0           1279       1599 0
 *             _______________            ________
 * ___________|  Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *    _______________________    _____________
 * |____|       VGA_HS          |____|
 */
   // Parameters for hcount
   parameter HACTIVE       = 11'd 1280,
           HFRONT_PORCH = 11'd 32,
           HSYNC      = 11'd 192,
           HBACK_PORCH  = 11'd 96,
           HTOTAL     = HACTIVE + HFRONT_PORCH + HSYNC +
                            HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE       = 10'd 480,
           VFRONT_PORCH = 10'd 10,
           VSYNC      = 10'd 2,
           VBACK_PORCH  = 10'd 33,
           VTOTAL     = VACTIVE + VFRONT_PORCH + VSYNC +
                               VBACK_PORCH; // 525

   logic endOfLine;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)       hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else             hcount <= hcount + 11'd1;

   assign endOfLine = hcount == HTOTAL - 1;

   logic endOfField;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)       vcount <= 0;
     else if (endOfLine)
     if (endOfField)   vcount <= 0;
     else             vcount <= vcount + 10'd1;

   assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DFconv_input_pixel_addr
(0x57F)
   // 101 0010 0000 to 101 1101 1111
   assign VGA_HS = !( (hcount[10:8] == 3'b101) &
           !(hcount[7:5] == 3'b111));
   assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) /
2);
   assign VGA_SYNC_n = 1'b0; // For putting sync on the green
signal; unused

   // Horizontal active: 0 to 1279    Vertical active: 0 to 479
   // 101 0000 0000  1280        01 1110 0000  480
   // 110 0011 1111  1599        10 0000 1100  524
   assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) )
&
            !( vcount[9] | (vcount[8:5] == 4'b1111) );


   /* VGA_CLK is 25 MHz
    *           __    __    __
    * clk50    __|  |__|  |__|
    *
    *           _____      __
    * hcount[0]__|   |_____|
    */
   assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge
sensitive

endmodule
```

mem.v

```
Unset

// megafunction wizard: %RAM: 2-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: mem.v
// Megafunction Name(s):
//               altsyncram
//
// Simulation Library Files(s):
//               altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 21.1.0 Build 842 10/21/2021 SJ Lite Edition
// ************************************************************


//Copyright (C) 2021  Intel Corporation. All rights reserved.
//Your use of Intel Corporation's design tools, logic functions
//and other software and tools, and any partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Intel Program License
//Subscription Agreement, the Intel Quartus Prime License
Agreement,
//the Intel FPGA IP License Agreement, or other applicable
license
//agreement, including, without limitation, that your use is for
//the sole purpose of programming logic devices manufactured by
//Intel and sold by Intel or its authorized distributors.  Please
```

```
//refer to the applicable agreement for further details, at
//https://fpgasoftware.intel.com/eula.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module mem (
     aclr,
    clock,
    data,
    rdaddress,
    wraddress,
    wren,
    q);

     input      aclr;
    input      clock;
    input    [7:0]  data;
    input    [17:0]  rdaddress;
    input    [17:0]  wraddress;
    input      wren;
    output    [7:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
     tri0      aclr;
    tri1      clock;
    tri0      wren;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [7:0] sub_wire0;
    wire [7:0] q = sub_wire0[7:0];
```

```
altsyncram    altsyncram_component (
             .address_a (wraddress),
             .address_b (rdaddress),
             .clock0 (clock),
             .data_a (data),
             .wren_a (wren),
             .q_b (sub_wire0),
             .aclr0 (aclr),
             .aclr1 (1'b0),
             .addressstall_a (1'b0),
             .addressstall_b (1'b0),
             .byteena_a (1'b1),
             .byteena_b (1'b1),
             .clock1 (1'b1),
             .clocken0 (1'b1),
             .clocken1 (1'b1),
             .clocken2 (1'b1),
             .clocken3 (1'b1),
             .data_b ({8{1'b1}}),
             .eccstatus (),
             .q_a (),
             .rden_a (1'b1),
             .rden_b (1'b1),
             .wren_b (1'b0));
defparam
  altsyncram_component.address_aclr_b = "NONE",
  altsyncram_component.address_reg_b = "CLOCK0",
  altsyncram_component.clock_enable_input_a = "BYPASS",
  altsyncram_component.clock_enable_input_b = "BYPASS",
  altsyncram_component.clock_enable_output_b = "BYPASS",
  altsyncram_component.intended_device_family = "Cyclone V",
  altsyncram_component.lpm_type = "altsyncram",
  //altsyncram_component.numwords_a = 240000,
  //altsyncram_component.numwords_b = 240000,
  altsyncram_component.numwords_a = 183750,
  altsyncram_component.numwords_b = 183750,
      altsyncram_component.operation_mode = "DUAL_PORT",
      altsyncram_component.outdata_aclr_a = "CLEAR0",
      altsyncram_component.outdata_aclr_b = "CLEAR0",
      altsyncram_component.outdata_reg_b = "CLOCK0",
      altsyncram_component.power_up_uninitialized = "FALSE",
      altsyncram_component.read_during_write_mode_mixed_ports =
"DONT_CARE",
      altsyncram_component.widthad_a = 18,
      altsyncram_component.widthad_b = 18,
      altsyncram_component.width_a = 8,
      altsyncram_component.width_b = 8,
      altsyncram_component.width_byteena_a = 1;



endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: ADDRESSSTALL_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLRdata NUMERIC "0"
// Retrieval info: PRIVATE: CLRq NUMERIC "0"
// Retrieval info: PRIVATE: CLRrdaddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRrren NUMERIC "0"
// Retrieval info: PRIVATE: CLRwraddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRwren NUMERIC "0"
// Retrieval info: PRIVATE: Clock NUMERIC "0"
// Retrieval info: PRIVATE: Clock_A NUMERIC "0"
// Retrieval info: PRIVATE: Clock_B NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_REG_B NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_B"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING
"Cyclone V"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MEMSIZE NUMERIC "1920000"
// Retrieval info: PRIVATE: MEM_IN_BITS NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING ""
// Retrieval info: PRIVATE: OPERATION_MODE NUMERIC "2"
// Retrieval info: PRIVATE: OUTDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: OUTDATA_REG_B NUMERIC "1"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_MIXED_PORTS
NUMERIC "2"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC
"3"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_B NUMERIC
"3"
// Retrieval info: PRIVATE: REGdata NUMERIC "1"
// Retrieval info: PRIVATE: REGq NUMERIC "1"
// Retrieval info: PRIVATE: REGrdaddress NUMERIC "1"
// Retrieval info: PRIVATE: REGrren NUMERIC "1"
// Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
// Retrieval info: PRIVATE: REGwren NUMERIC "1"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: USE_DIFF_CLKEN NUMERIC "0"
// Retrieval info: PRIVATE: UseDPRAM NUMERIC "1"
// Retrieval info: PRIVATE: VarWidth NUMERIC "0"
// Retrieval info: PRIVATE: WIDTH_READ_A NUMERIC "8"
// Retrieval info: PRIVATE: WIDTH_READ_B NUMERIC "8"
// Retrieval info: PRIVATE: WIDTH_WRITE_A NUMERIC "8"
// Retrieval info: PRIVATE: WIDTH_WRITE_B NUMERIC "8"
// Retrieval info: PRIVATE: WRADDR_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: WRADDR_REG_B NUMERIC "0"
// Retrieval info: PRIVATE: WRCTRL_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: enable NUMERIC "0"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: LIBRARY: altera_mf
altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: ADDRESS_ACLR_B STRING "NONE"
// Retrieval info: CONSTANT: ADDRESS_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_B STRING
"BYPASS"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING
"Cyclone V"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "240000"
// Retrieval info: CONSTANT: NUMWORDS_B NUMERIC "240000"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "DUAL_PORT"
// Retrieval info: CONSTANT: OUTDATA_ACLR_B STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING
"FALSE"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_MIXED_PORTS
STRING "DONT_CARE"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "18"
// Retrieval info: CONSTANT: WIDTHAD_B NUMERIC "18"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_B NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: data 0 0 8 0 INPUT NODEFVAL
"data[7..0]"
// Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL "q[7..0]"
// Retrieval info: USED_PORT: rdaddress 0 0 18 0 INPUT NODEFVAL
"rdaddress[17..0]"
// Retrieval info: USED_PORT: wraddress 0 0 18 0 INPUT NODEFVAL
"wraddress[17..0]"
// Retrieval info: USED_PORT: wren 0 0 0 0 INPUT GND "wren"
// Retrieval info: CONNECT: @address_a 0 0 18 0 wraddress 0 0 18
0
// Retrieval info: CONNECT: @address_b 0 0 18 0 rdaddress 0 0 18
0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: @data_a 0 0 8 0 data 0 0 8 0
// Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
// Retrieval info: CONNECT: q 0 0 8 0 @q_b 0 0 8 0
// Retrieval info: GEN_FILE: TYPE_NORMAL mem.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL mem.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL mem.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL mem.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL mem_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL mem_bb.v TRUE
// Retrieval info: LIB_FILE: altera_mf
```