Photomosaica Report

Andrew Weinfeld
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
andrewjw@mit.edu

Marilyn Meyers
Material Science & Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
mrmeyers@mit.edu

Isaac (Zack) Duitz
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
duitz@mit.edu

***Abstract*—We present a system for real-time generation of photomosaic images. A photographic mosaic, or photomosaic, is a photograph composed of other photographs [1]. Our system uses an OV7670 camera to collect video data, then uses a Spartan 7 FPGA and a DDR3 chip to compare the video data against a database of images on an SD card, and finally displays the output video on an HDMI-compatible display. The system also includes the ability to switch between multiple image libraries and to pause the video feed. We present our design and analyze our completed system.**

***Index Terms*—Digital systems, Field programmable gate arrays, Image processing, Photomosaics**

I. Physical Construction

The physical components of our Photomosaica system include:

- A 2GB microSD card.
- A display with an HDMI cable.
- The Urbana FPGA from RealDigital.
- An OV7670 camera module provided by the course staff.

II. Graphics Pipeline (Andrew)

The graphics pipeline is the main part of the FPGA system. It takes data from the Camera, communicates with the Memory Controller, and produces HDMI output. A block diagram for the graphics pipeline is shown in Fig. 1. The memory subsystem is depicted in Fig. 2.

A. Camera Input Processing (Andrew)

The Camera outputs data one pixel at a time, which are saved to the Input Line Buffer. The Input Line buffer contains sufficient space for 10 lines of input pixels. When the Camera finishes saving 5 lines of input data to the Input Line Buffer, the Image Matcher reads out each (5-pixel-aligned) 5x5 subimage from the Input Line Buffer, computes the address of an image in DDR which best matches that 5x5 subimage, and puts the address in the corresponding location in the Frame Buffer. The output image consists of 16x16 images such that the image at location (i,j) in the output image will be the image at address FrameBuffer[(i,j)] in the DDR.

B. Image Library Algorithm (Zack)

We wanted a way to map chunks of the images directly to the image that best represents them. Therefore we made our own ”color space” in which each color is mapped to the image that best represents it. For example using a 12 bit color space where each RGB color gets 4 bits the color 0000_0000_1111 would map to the image that is the most blue.

In our first iteration we computed the average color of 48x48 images and mapped 16x16 chunks of our input image to match those images. This did not look like the original image because the pictures were too big so we tried using smaller images. We had to balance the trade-off of improving the larger image while the smaller images became increasingly unclear. We decided to use 16x16 images and 5x5 chunks of the overall image we are converting. We found the best way to shrink the images to 16x16 images was using the LANCZOS re-sampling algorithm as part of the Pillow library in python [5]. This was more effective than regular dithering on these images. Then we tried to improve the image matching algorithm by taking into account the gradient from right to left and bottom to top across each image and chunk however this did not seem to improve the image output which seemed to be because there were very few large gradients across the 5 pixel spans.

In our final algorithm we decided just to use average color but when choosing the best image to represent each color we minimized the standard deviation so that it would favor images that were mostly the correct color across all their pixels. We also decided to use the full color space returned from the camera which is 5 bits for red, 6 bits for green, and 5 bits for blue. See Fig. 3 for a pictorial representation of the algorithm. This displays matching the color space to pictures with those colors and then averaging chunks of an image to map onto the color space. Fig. 4 contains a side-by-side comparison of an original image and the algorithm output image.

C. Computing Average of the Image (Zack)

In order to pull pixels from the Input Buffer, one pixel is passed into the Image Matcher module at a time which regulates when it is done processing the chunk using the state machine depicted below Fig. 5. The Image Matcher waits until it is passed the start frame signal at which point it takes in

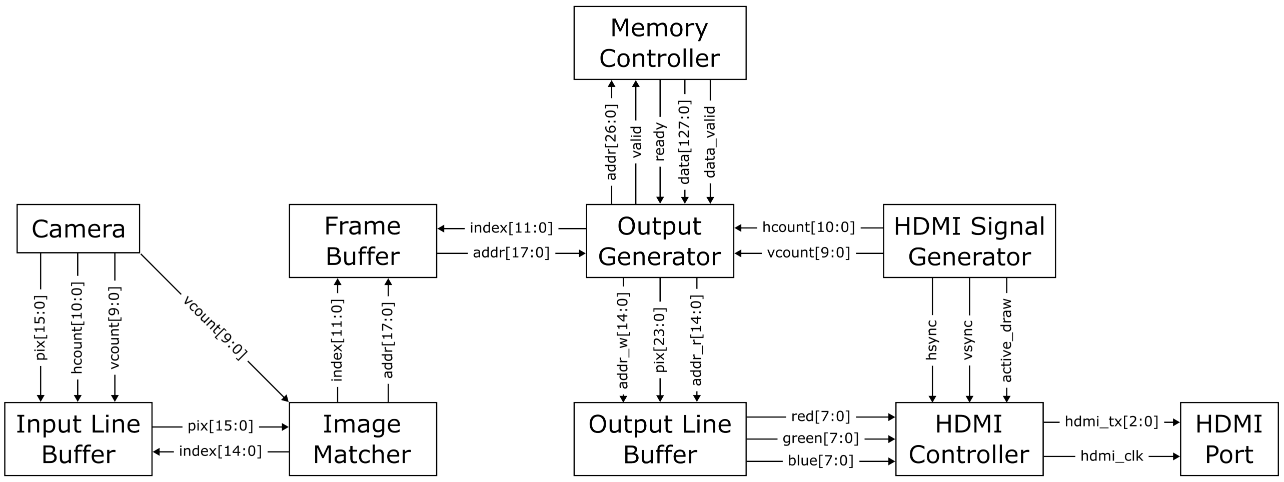


Fig. 1: The overall graphics pipeline.

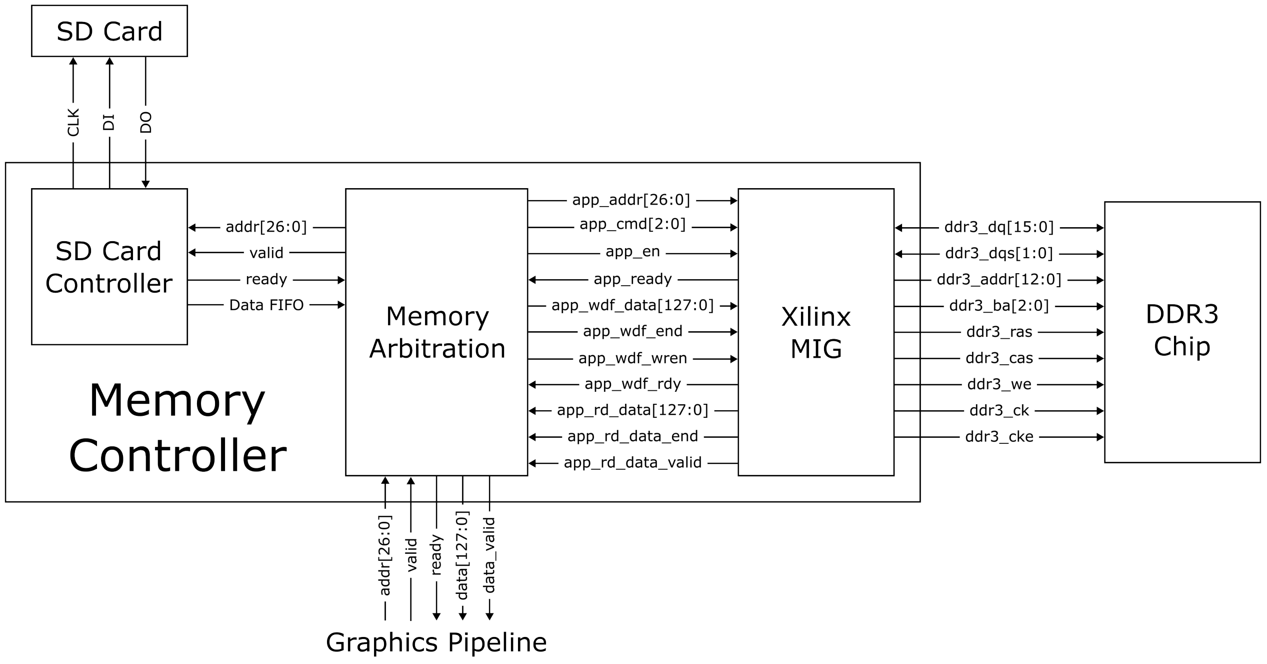


Fig. 2: The memory subsystem.

pixels for 25 consecutive clock cycles. After these 25 cycles it begins calculating the average of the red, green, and blue colors separately using the division module. This finishes in a variable number of cycles however given that our input size is a max of ten bits it finishes quickly. When all three values are done being computed the Image Matcher module updates the addr and index signals with the new color values and pixel location, which stores the image address in the appropriate location in the Frame Buffer.

D. Output Generator and HDMI Output (Andrew)

In order to produce the output image, the Output Generator copies images from the Memory Controller to the Output Line Buffer just before the corresponding lines sent over HDMI. The system operates as follows:

- The Output Line Buffer holds 32 lines of output pixels, corresponding to two rows of 16x16 output images.
- At any given moment, one row of 16x16 output images is being read out to the HDMI controller while the Output Generator updates the other row of 16x16 output images with the next images.

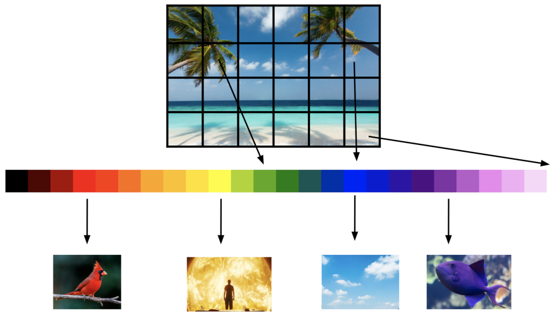
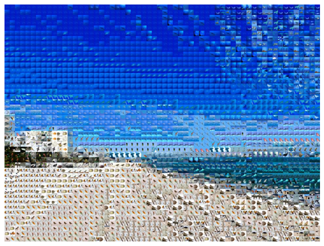


Fig. 3: Image Algorithm Description

(a) Original Image



(b) Converted Image

Fig. 4: Sample image algorithm output

- When the vcount signal is an exact multiple of 16, the Output Generator knows that the previous row of 16x16 images has been fully read out over HDMI and can be replaced.
- In order to perform this replacement, for each 16x16 output image in the upcoming row, the Output Generator:
  1) Looks up the corresponding DDR address in the Frame Buffer.
  2) Requests the pixel data at that address from the Memory Controller.
  3) Writes the pixel data from the Memory Controller to the Output Line Buffer at the appropriate location.

  Getting the data from the Memory Controller requires 16*16*16/128 = 32 DDR transactions, all of which are on the same row, which allows the transactions to proceed as consecutive read commands which easily keep up with the HDMI output.
- The pixel data in Output Line Buffer is padded from 16 bits of color to the 24 bits expected by the HDMI before being passed to the HDMI.

In order to efficiently read data out of the Xilinx MIG, the Output Generator contains two linked state machines: one reads from the Frame Buffer and sends read requests to the Memory Controller, while the other puts read data into the Output Line Buffer. Once one state machines complete a 16x16 output block, it waits for the other state machine to finish handling the block before moving on to the next block.

In addition to displaying the output image, we also display the original camera feed in the upper right corner of the screen. We do so by saving pixels to a 320x240 BRAM as the come in from the Camera and reading them off directly into the HDMI data.

III. Memory Controller (Andrew)

The Memory Controller block diagram is depicted in Fig. 2. The Memory Controller is responsible for managing the image libraries and providing image data to the Graphics Pipeline. The Memory Arbitration system, which is implemented in the Memory Controller module, is responsible for sending read and write requests to the Xilinx MIG, sending read requests to the SD Card Controller, and handling read requests by the Graphics Pipeline. A more detailed state description of the state machine is as follows:

0) Initial state after a reset. Automatically goes to state 6.
1) Put the data from the `saved_write_data` register into the MIG data FIFO, then go to state 2 once the data FIFO signals ready.
1) Send a write commend to the Xilinx MIG. If there are more bytes remaining in the requested image, go to state 7. If the current image is complete but there are more images to load from the SD card, go to state 6. If all images have been loaded from the SD card, go to state 3.
3) Wait for a read request from the Output Generator. On a read request, transition to state 4.
4) Pass the Output Generator's read request on to the MIG command FIFO. When the FIFO signals ready, return to state 3.
5) Unused.
6) Send a read request to the SD card, then enter state 7.
7) Wait until we have received 16 bytes from the SD card and save them in one larger 128-bit register called `saved_write_data`. Then proceed to state 1.

Since the Xilinx MIG runs off of a different clock from the SD Card Controller and the Output Generator, the Memory Arbitration system includes asynchronous FIFOs on its communications with the SD Card Controller and the Output Generator. The Xilinx MIG runs off of an 81.25 MHz clock signal, the Output Generator runs off of a 74.25 MHz clock signal, and the SD Card Controller runs off of a 25 MHz clock.

A. DDR SDRAM (Andrew)

The Xilinx MIG handles startup and refresh of the DDR memory for us. It gives an interface consisting of a FIFO for write data, an AXI bus for read data, and an AXI bus for sending read and write commands. There are a few other inputs, including manual refresh and data masking functionality, which we will not be using. With our current configuration of the MIG, read and write requests consist of straightforward 128-bit bus transactions. Full details of the interface may be found in [3].

The Memory Arbitration system is written to run synchronously with the Xilinx MIG since of interfaces between the three adjacent clock domains, the Xilinx MIG has the most complicated interface, and skipping the async FIFOs for the MIG simplifies the design.

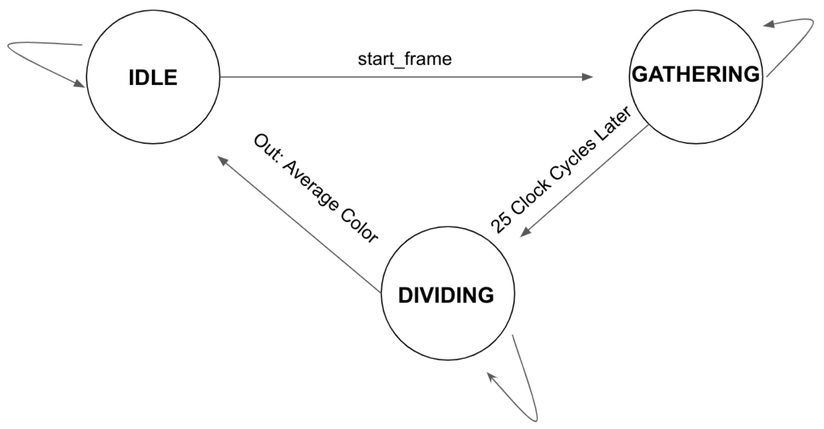


Fig. 5: Image Matcher State Diagram.

B. SD Card (Marilyn)

The SD Card is pre-loaded with the Hexadecimal encoding of the compressed images that make up the image libraries. This has been done by copying 2^{16} images comprised of 512 bytes of information for each image library.

In the system, the communication with the SD Card will be controlled by the SD Card Controller [2]. Using this controller a block of 512 bytes can be requested from the SD card, which is sent one byte at a time. Each 512 byte block represents one image. This system operates on a 25 MHz clock cycle which was derived from the system's 100 MHz clock cycle using the clock wizard IP.

For our project, we loaded two libraries onto the SD card, taking up approximately 67 MBs of the SD's available 2GBs. When the FPGA is flashed, the DDR3 loads images from the first library of images, which begins at address 0. When the switch, SW0, is turned on, the DDR3 reloads from the second library by incriminating the address requests by $2^{16} * 512$ (the size of the first library). The two libraries are structured identically but contain different image sets.

The SD Card Controller communicates with the rest of the system via the Memory Arbitration system, which is implemented in the Memory Controller module. The modules will communicate via an AXI bus and a FIFO; the Memory Arbitration system sends read request addresses to the SD card through the AXI bus and the SD Card Controller responds by placing data read into the FIFO. Both communications use asynchronous FIFOs (specifically, the `xpm_fifo_axis` primitive) to convert between the different clocks of the two modules.

The system reports the SD load progress on LEDs 4-15; when all of the LEDs are lit, the load is complete. The load takes roughly one minute.

LEDs 0-3 are tied to various internal signals inside of the system. When the system is successfully generating an output video, these LEDs should be partially on due to them flickering on and off; the LEDs will be either all the way on or all the way off when data is being loaded from the SD card.

IV. Results and Evaluation (Marilyn)

In this project, we were able to successfully achieve our commitment and goal. We have produced a real-time, high quality photomosaic video feed. Our commitment was to create "A project that can take a still image from our OV7670 camera module, compare portions of the image with images stored on an SD card, generate a reasonable-looking output, and show the output as a still image over HDMI." Our goal was to create "A photomosaic video feed which can use Joe's DDR interface to generate output images in real time, with the ability to pause the video feed." As seen from the output here and the video, we have achieved both of these objectives. Furthermore, we used our own MIG setup instead of Joe's DDR interface.

Our system does not include the capability to save an output image to the SD card, which would likely require only a modest amount of additional state logic.

The overall system outputs recognizable images and functions just as well as we had hoped. An example of the full system running is in Fig 6, and a close-up of the overall system is in Fig 7.

A. Resource Utilization and Timing (Andrew)

Our design uses 19.07% of slice logic, which is reasonably small.

Our design uses 95.33% of BRAM, of which 64% is due to buffering the original video feed and 31.33% is the actual photomosaic system. Recall that we buffer the original video feed so that we can display it in parallel with the photomosaic, where it can be seen in the upper right corner of Fig 6.

We also use a handful of special blocks, including 80% of the MMCME2_ADV blocks. It is likely possible for us to reduce this number by reorganizing the clock generation, since the 25 MHz SD card clock can probably be merged with the

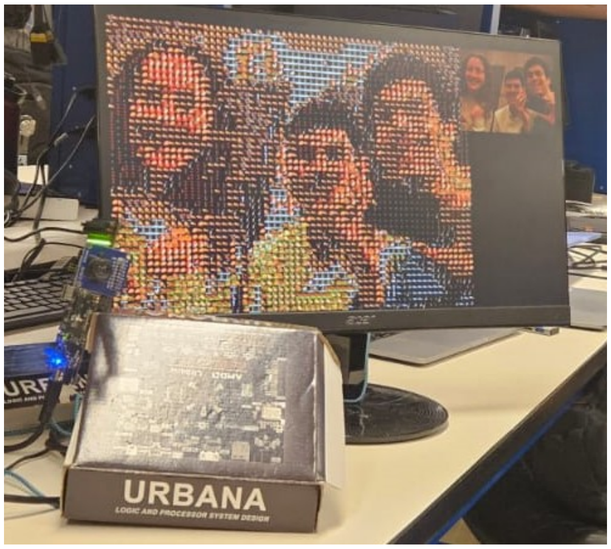


Fig. 6: The full system running.

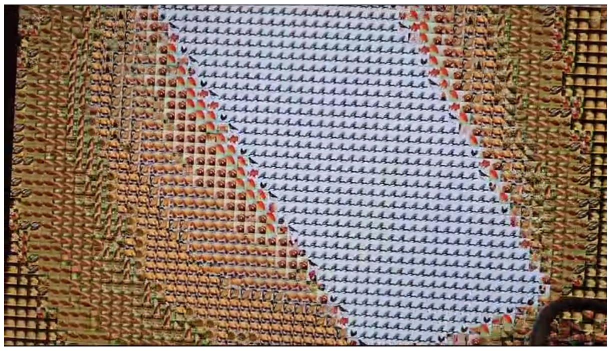


Fig. 7: A zoom in on a single frame.

DDR clock generator. Our design utilizes 2.50% of the DSP blocks, and we are not quite sure what this block is used for–perhaps the reduction modulo 10 or the division by 5 which are used in a couple of locations.

The design meets timing slack by 1.249ns, which is plenty, and the critical path is inside the Xilinx MIG, so there is not much to do as far as optimizing timing.

The other timing constraint that our device must follow is that the DDR must be able to keep up with the output video feed. We performed rudimentary benchmarking using a DDR3-based implementation of the Sieve of Eratosthenes, which indicated that the DDR3 has a read latency of just under 300 ns for 16 bytes; this is already sufficient for our system to function.

Furthermore, the DDR3 protocol allows both multiple reads to be sent simultaneously to the same row (see [4], Figure 70) and multiple reads to be sent simultaneously to different banks. Since we are sending batches of 32 consecutive reads to the DDR3 chip at a time, we believe that the DDR is easily keeping up with our system workload.

If the DDR did not keep up with the system workload, this would manifest as visual glitches on screen even when the output image is paused (since the output video is generated continuously from the DDR just-in-time, line by line, and pausing the video feed only pauses the Image Matcher), and we did not observe any such visual glitches in testing.

One final point of timing to consider is the time that it takes to load data from the SD card to the DDR3. This is not precisely a timing constraint, since the system will run correctly regardless of how long it takes to load the data, but we would like the data to load reasonably quickly. Currently, the data takes just over a minute load from the SD card, and increasing the SD card clock speed only slightly improved the SD card load speed; since the DDR3 chip should be able to process all of our data (32 MB per image library) in under a second (at 300 ns per 16 bytes), we expect that the main slowdown is in the internals of the SD card.

B. Challenges (Marilyn, Zack)

While we have made our project sound straightforward, we had a few bumps during our work to implement it.

When writing the image matcher module originally most of the screen showed up black and it was not updating the color based on the camera. After a few hours of debugging we noticed that although we thought only the final color we calculated was being written to the output buffer. Really the intermediate updates to the final color during the averaging were being written to the output. The wrong color was constantly being written to the output. We fixed this by putting another variable to hold the final color so that only the correct color would be constantly written to the output.

One challenge we faced was setting up the SD card. We struggled to get the original SD module working due to a typo in which we set the inouts to be inputs. This took a lot of debugging and office hours to get sorted out. Thank you Joe. We also struggled to program the SD card. One of our computers couldn't write to the SD card for an unknown reason. We also had to re-structure the SD card by writing to it through the FPGA so that the data format was compatible with the hex editor.

Another challenge that we faced was formatting the image libraries to be placed on the SD card. When we added the SD card to our FPGA, we saw pixelated green and blue images. We spent hours trying to locate the source of the problem. We eventually found that the python code that we used to generate our library was outputting the ASCII hex encoding for the images, instead of the bit encoding. This resulted in the output image consisting of the "colors" 0x30-0x39 (0-9 in ASCII) and 0x41-0x46 (a-f in ASCII), which correspond to a blue and green output.

V. Future work (Marilyn)

This project could be expanded and improved in a number of ways.

1. We could improve the SD card library switching. This could be done by implementing a way to continue the live video stream while concurrently loading a new library of images from the SD card to the DDR3. We imagine this

improvement would result in a video stream photomosaic that has some images from the old library and some images from the new library corresponding to what has and hasn't been replaced in the DDR3.

2. We could continue improving the video stream quality. First, having a better camera module would help a lot with this goal. Second, we could continue researching and testing ways to improve the image selection/replacement algorithms to improve the quality of our output image.

3. We could speed up the SD to DDR3 loading process. Currently, we are only using single block reads which results in a minute long load time. Altering the SD card clock speed does not seem to change the load time significantly, which indicates that the limiting factor in the loading is the internals of the SD card; issuing a multi-block read when loading data might reduce the SD card's internal overhead.

References

[1] https://en.wikipedia.org/wiki/Photographic_mosaic

[2] B. Gross, J. Matthews, N. Rodman "Live-Action RC Mario Kart", 2014.

[3] Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v4.2, User Guide (UG586). https://docs.xilinx.com/v/u/en-US/ug586_7Series_MIS

[4] DDR3 SDRAM x4, x8, x16 Component Data Sheet. Micron. https://mm.digikey.com/Volume0/opasdata/d220001/medias/docus/588/MT41J256M4,128M8,64M16.pdf

[5] https://pillow.readthedocs.io/en/stable/releasenotes/2.7.0.html