6.2050 Final Report
FPGA Video Dithering

Nadia Frieden
nfrieden@mit.edu

Erin Zhang
ewzhang@mit.edu

I. Introduction

The goal of this project was to create a real-time video compressor by dithering the images that make up each frame. "Dithering" is an image compression technique that compresses pixels one by one while spreading the quantization error of this rounding onto a subset of its neighboring pixels. Going from 8-bit grayscale to 1-bit dithered output uses an eighth of the space (quite a significant reduction!) and the FPGA's capability for video signal processing makes it a fantastic candidate for this task.

Dithering images is a well-covered topic, but when it comes to dithered video, we were in relatively uncharted territory. We had no idea whether the quality of the compressed video would be acceptable or useful, or what techniques we could use to improve it if it wasn't.

What's more, the way dithering modifies registers would require a ridiculous amount of inefficient storage if not for the line buffers, which had to be *very* carefully integrated to interact perfectly with the needs of the dither module.

We went into the project with the following checklist:

- *Commitment*: Camera output with Floyd-Steinberg
- *Goal*: Dithered looping gifs off SD Card, improve live-feed visual appeal and try more dithering algorithms
- *Stretch Goal*: Filtering (changed during final stretch meeting to be automatic adjustment instead)

In the end, our commitment was met, along with the live-feed aspect of our goal and an attempt at our stretch goal.

II. System Overview

The system diagram included on the last page of this report shows the path that data must take throughout each module to end up as visual output on the monitor.

Data from the camera are first sent through processing modules that sync them with a 74.25 MHz clock and add corresponding signals detailing their horizontal and vertical locations in the frame (referred to hereafter as "hcount" and "vcount"). The pixel values are converted to a chosen flavor of black and white output and then written to the "line buffer," a module containing four lines of Block Random Access Memory (BRAM, with a finite size and two read/write ports) that efficiently stores the data in the specific format needed by the dither module. Any 1 bit dithering process must be centered around a "threshold" value (above or below which pixels are rounded to be full white or full black): our system has three modules to control this value that are also connected to the FPGA's seven segment display and LEDs for ease of tracking. The dither module will dither each frame with a chosen algorithm, write error-diffusion-affected pixels back to the line buffer, and send dithered pixels off to a second instance of BRAM called a "frame buffer" (of width 1 and depth 320*240 to contain a full frame of dithered pixels). From there, video signal generation will pass them through the scale, rotate, and Transmission Minimized Differential Signaling (TMDS) modules and place them in differential signaling output buffers (OBUFDS). This data can then be read by the HDMI receivers and displayed on our 720p monitors at 60 fps.

We also left the original width-8 frame buffer in place and muxed it in alongside the other to provide users the ability to compare the dithered and non-dithered input.

III. Components

We will now embark on a more detailed dive into each aspect of our system.

A. Camera & Recover

The input data for our dithered video is a live camera feed through the OmniVision OV7670/OV7171 CameraChip Sensor. This camera outputs 320 * 240 pixels at 30 frames per second in a raster pattern using all the vsync, hsync, and href signals we are familiar with.

This output is then sent through the camera and recover modules, which sync the signals coming out at 24MHz with our 74.25 MHz clk_pixel and assign them hcount and vcount values to be used throughout the rest of the module.

B. Color Modification

To get the camera's 16-bit RGB565 output to the 8-bit grayscale format needed for this task, we created a module that outputs 7 different types of 8-bit grayscale based on user input to switches [4:2]: the average of R G B, just R, just G, just B, Y, Cr, and Cb. Certain types will show up better in different environments, so we wanted to have as many options for visual clarity as possible. This black and white output value is saved into the critical 8-bit register **bw**.

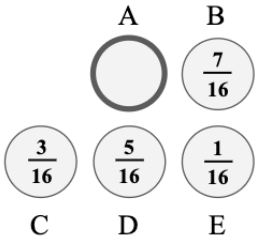
C. Dither module

Although data must be sent through the line buffer before the dither module, we will explain dithering first because the line buffer's behavior doesn't make sense on its own.

This project includes two options for dithering algorithms: Floyd-Steinberg and minimized average error dithering by Jarvis, Judice, and Ninke (JJN). In both algorithms, an individual pixel will be dithered as **0** if its 8-bit-grayscale value is under some given threshold, and **1** if above. But what makes dithering dithering is that this quantization error *(original value − rounded value)* will be scaled and pushed onto the neighboring, upcoming pixels.

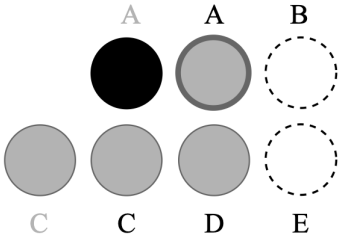
To demonstrate how this was done on the FPGA, we will analyze the process for Floyd-Steinberg in depth and then show how this methodology was extended to the more complicated JJN.

Each step of Floyd-Steinberg can be implemented with 5 main registers: **A** and **B** represent neighboring pixels on the same line, while **C**, **D**, and **E** are three pixels on the next line down that will all be affected by dithering **A**.



When the dithering of **A** is complete, pixels **B** through **E** will take on a new value equal to their previous value plus the quantization error scaled by the fraction they are labeled with above.

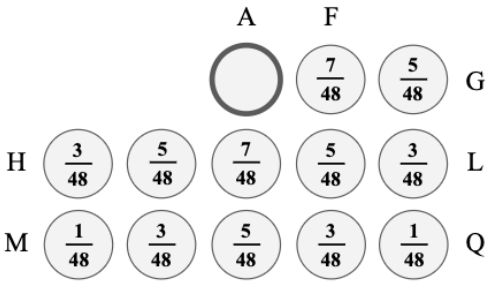
We dither pixels in a frame from left-to-right, top-to-bottom, so the next pixel up to be dithered is **B**. To do this, we must bring in two more pixels (the next one to the right on both relevant lines). This will be the "new **B**" and the "new **E**," represented in the diagram below with dotted circles.



When the line buffer has sent along these two new values, the "old **B**" becomes the "new **A**" to be dithered, "old **D**" becomes "new **C**", and "old **E**" becomes "new **D**". In the diagram below, all pixels shaded in with grey are now updated values that do not match what came out of the camera/SD Card and was written into the line buffer.

The "old **A**" is now a 1-bit value and is sent off to the frame buffer. The "old **C**" won't be needed by the dither module until it gets read in as **B** many cycles later: its updated value will be written back to the line buffer. Thanks to the nature of this rotation, the new **C**, **D**, and **E** values will all eventually be written back with their (repeatedly) updated values, and then later be read back out to be dithered themselves as **A**. This cycle continues until every line in the frame has been dithered, then restarts at the top of the next frame.

The process for JJN is very similar, except that the quantization error gets pushed onto far more pixels, forcing the process to involve three lines. Instead of 5 primary registers, it requires 13, labeled below with the scaling factor for the quantization error.

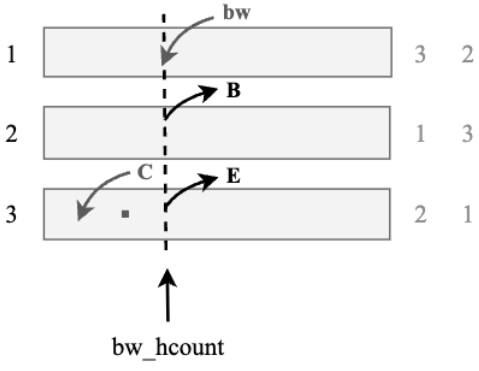


JNN results in fewer visual artifacts than Floyd-Steinberg, but on our 320-by-240-pixel frames, it is a noticeably courser dithering.

D. BRAM line buffer

We created a line buffer module to abstract away the complications of all the pixels coming in and out of the dithering process. The line buffer module consists of four individual lines of BRAM, all of width 8 (to store the 8-bit-grayscale values coming from the camera or SD card) and depth 320 (for our default image size of 240 x 320 pixels).

When a user flips switch [13] to select Floyd-Steinberg dithering, only three of the four lines will need to be used. Each line rotates through three different roles, visualized in the diagram below:

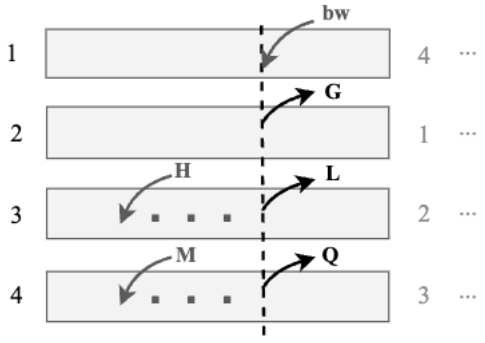


- *Role 1* : Write in pixel **bw** at location **bw_hcount**, which will be read out when the line is later assigned a different role.
- *Role 2* : Read out pixel **B** at location **bw_hcount** for the ditherer, which was written in as **bw** at an earlier time when the line had the previous role.

- *Role 3* : Read out the pixel **E** at location **bw_hcount** for the ditherer, written in as **bw** at an earlier time. Write in updated pixel **C** at location **bw_hcount** - 2.

Each time **bw_hcount** traverses a full line (goes from 0 to 320), the lines rotate roles. Since every value from the line with role **2** has been read into the dither module and sent to the frame buffer in its new 1-pixel form, these values no longer need to be stored in the line buffer and can be written over by assigning this line role **1**. The line with role **3** that was writing in updated **c** values is next up to be dithered, so it takes on role **2**. Finally, the next line of pixels just written in by role **1** will naturally take on role **3**.

If the user flips switch [13] to select JJN dithering, all four lines will be in play. The pattern of roles is essentially the same, but each time a pixel is dithered, you are reading out three pixels and updating two (instead of two and one, respectively).



The interaction of these roles is the perfect partner to the dither module, efficiently dealing data out of a small amount of structured BRAM so that the dither module only has to hold values in five/thirteen 8-bit registers at any given time.

Although of course the line buffer may spit out garbage on the edge cases, its steady-state behavior results in a clear enough HDMI output that agonizing over the math for starting/ending each frame was not a top priority.

E. Thresholding

A large amount of the work since the preliminary report was concentrated on everything related to thresholds: how to edit them, display them, and even algorithmically choose them live.

The "threshold buttons" module is the first method of control. It uses buttons [2] and [3]; pressing one will increment the threshold by the amount represented on switches [11:5] (mirrored on the seven segment display for easier viewing), pressing the other will decrement it by that same amount.

Using the manta module, you can move the threshold value around by simply running a Python script that assigns it any given value. One cool application of this is using time.sleep(x) to systematically adjust the threshold every x seconds, allowing the user to view what many different thresholds' outputs would look like without having to manually move them around.

Finally (and least reliably), the threshold calibrator. The adjusted stretch goal of this project was to come up with a module that would output the best threshold for the environment at a push of a button. The only reasonable measurement we could think of was to count bit transitions per frame, thinking that the higher the transitions, the more defined the dithered output. When the project is flashed or the reset button is pressed, the threshold calibrator runs through x frames, incrementing the threshold for each frame by $256/x$ and keeping track of which one had the most transitions. You can see whether it made a good choice by pressing button [1] and setting the dither module's threshold to that value.

Sometimes this module works pretty well, and sometimes it works horribly (which is why we allow the user to not upload the module's threshold suggestion: sometimes you can tell from the LEDs that it output garbage). As of project turn-in I have not yet managed to get it to where I wanted it to be, but at least most of the time, it gets you to a point close enough to the correct threshold that you can use the buttons to make small adjustments from there. It should also be taken into account that the best threshold is not always where the most transitions occur, so you'd have to find some other, more complicated heuristic to get it working perfectly.

Luckily, these modules can all be used at the same time; for example, you can set a threshold value using manta, move it around from that point with the buttons, decide you want to re-calibrate to a different value, and continue to adjust with the buttons from there. Dithering has a narrow threshold "sweet spot" where the output will look as expected, so being able to adjust to the environment is incredibly important.

F. BRAM frame buffers

The two frame buffers we have both work very similarly to how they were used in labs 4 and 5. They both have depth $240*320$, but one has width 8 (to hold values directly from the b&w module) and the other has width 1 (to hold the dithered version of the image). A switch-controlled mux determines which of these frame buffers will be used to generate the HDMI output. Since pixels come out of the camera/SD card at a different rate than the video_sig_gen module sends them to HDMI output, the frame buffer acts as a critical layer of separation.

G. Path to HDMI output

The pipeline from the frame buffer to the monitor is also a simplified version of what we used in labs 4 and 5: the rotate, scale, and video signal generation modules, along with the TMDS encoders/serializers and OBUFDSs, are all unchanged. The only key difference is that because we are doing everything in black in white instead of RGB, either all 0s or all 255s will be fed into the TMDS modules for any given pixel.

IV. Happy Accidents

Due to a bug in the line buffer module where line roles were rotated even when a new pixel wasn't ready, the 1-bit output

for most of the project wasn't actually dithering, just another unique type of compressed video. Because of its novelty and cool artifacts, this version is still easy to access within the project (set switch [14] to high).

V. Design Evaluation

Because there's no need to reinvent the wheel, many of our timing constraints arose from our desire to reuse many of the modules from the video generation pipeline. What felt especially important to maintain was the path of hcount and vcount throughout the system, driven by the recover and video signal generation modules.

Both the line buffer and dither modules work very closely with the hcount value coming out of recover: as long as they are able to do all their work for a pixel at a given hcount before that hcount changes, no data will be lost and the image will display as intended. However, since both modules do their work for each pixel in one cycle (reading it in or writing it out for the line buffer, editing the pixel value and assigning it to its new destination register for dither), there is no risk of pixel information being lost. Even if hcount was changing as fast as it could (every cycle, which it does not), both modules are designed to keep up.

Of course, even if the throughput is unaffected by our modifications to the system, latency certainly is. The worst case delay for any given pixel traveling through the modules we've added to the system are as follows:

- 3 cycles for color modification module (integrates the 3-cycle rgb_to_ycrcb module from lab 5)
- $\approx$ 14 500 cycles between when a pixel gets written into line buffer and when it is next up to be read out to be dithered (3 full lines for JNN, Tline from camera specs = 65072 ns = 4834 cycles at 74.25 MHz).
- 2 cycles for it to be read out of BRAM and sent to dither
- 2 cycles to get through dither (1 to transition from b to a, 1 to be dithered)

Each pixel in the dithered output has to go through this pattern, so they will come out the other side at the same rate they came in. And although 14 500 cycles seems like a lot, this is just over a ten-thousandth of a second at clk_pixel, and the remaining modules in the pipeline that we built our project off of will hardly affect this. There will be no perceptible difference to the human eye; when you shake your hand in front of the camera, you will see its movement in "real time."

When it comes to space utilization, our design is quite light: we only have a couple instances of block RAM that are as small as they can be to hold all the pixels in a line or a frame.

In terms of project checklist, the live feed dithering easily met our goal of solid visual quality with multiple dithering algorithms, and even foraged into the stretch goal region with the automatic threshold adjuster.

For additional use cases, it wouldn't be too difficult to get video from other sources up and running as well: if you made a module that matched the output format of the recover module to release pixels from pre-saved video, no changes would need to be made to any other part of the pipeline.

Overall, I am quite happy with how everything related to live feed dithering went; the visual quality of the dithered output turned out far better than I expected, especially taking camera quality into account. If I were to do it again, I think I'd just start thinking about thresholding earlier so I could have really nailed that automatic adjuster down.

VI. SD Card Integration

Another way our project could have taken in input is through a series of images placed on an SD Card, strung together to form a several-frame gif. Although this part of our original goal was never realized, we'll include a short run-through below.

{*Erin*}

We use a Python script to get images from a computer into 8-bit grayscale by taking the average of each pixel's R, G, and B values. To get the data on the SD card using MacOS, we can save it by first unmounting the disk using diskutil, and then flashing the data onto the card with the line "sudo dd if="imagedata" of="diskpath" bs=1m."

With the SD card plugged into the FPGA, we use a modified version of Fischer Moseley's SD card controller (read only capability) from course documentation to read that data onto the FPGA. The sd_controller module reads in 512 bytes at a time, and the specified read address (addr) must be a multiple of 512. It sends a trigger through byte_available whenever the sd_out output has a valid new byte.

To handle the timing, we use the FIFO IP. It outputs high on s_axis_tready (ready_for_sd_data) when there are 512 available bytes to be written to. Reading from the FIFO syncs with when the camera recover module outputs data (data_valid_rec) so as to maintain a similar frequency to the camera output. With every read from the FIFO, a counter (read_count) stores the location of the pixel out of all frames in order to derive the hcount and vcount.

Finally, a mux gives our system the capability of switching between video and SD output on the same build. Based on a switch, the inputs to the line buffer change from data either from the camera or the SD card.

Work to do: The SD card integration hasn't been tested, and the timing of the very start of the card read might take some work – reading from the FIFO must wait until there is sufficient data written to it. The switch for muxing between the camera and SD card must also be included.

VII. Repository

https://github.com/nfrieden25/fpgifa

References

[1] https://en.wikipedia.org/wiki/Floyd-Steinberg_dithering
[2] https://en.wikipedia.org/wiki/Error_diffusion#minimized_average_error
[3] https://web.mit.edu/6.111/www/f2016/tools/OV7670_2006.pdf
[4] https://www.haoyuelectronics.com/Attachment/OV7670%20+%20AL422B(FIFO)%20Camera%20Module(V2.0)/OV7670%20Implementation%20Guide%20(V1.0).pdf

VIII. Glossary

Switches:

- sw[15] ⟶ 0 = regular b&w output, 1 = dithered output
- sw[14] ⟶ 0 = correct dithering, 1 = wrong dithering
- sw[13] ⟶ 0 = Floyd-Steinberg, 1 = JJN
- sw[12] ⟶ 0 = live camera feed, 1 = freeze frame
- sw[11:5] ⟶ amount that threshold will increment/decrement on button press (also shown in hexadecimal on the seven segment display)
- sw[4:2] ⟶ type of black and white output (0-6 in order are RGB, R, G, B, Y, Cr, Cb)
- sw[1:0] ⟶ HDMI output size (same as lab 5)

Buttons:

- btn[0] ⟶ reset (also triggers threshold calibration)
- btn[1] ⟶ set threshold to the value calculated by the threshold calibrator
- btn[2] ⟶ increment threshold
- btn[3] ⟶ decrement threshold

LEDs:

- led[7:0] ⟶ what threshold calibration thinks the threshold should be
- led[15:8] ⟶ what the threshold currently is