DigiSketch Preliminary Report

1st Lydia Patterson
Department of EECS
Massachusetts Institute of Technology
Cambridge, USA
ljewel@mit.edu

2nd Edwin O. Ouko
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
eouko@mit.edu

3rd Fahnmusa J. Edwards
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
fahnmusa@mit.edu

***Abstract*—DigiSketch is a two-player version of Etch-a-Sketch with saving and expanded drawing capabilities. In this design, users can construct multicolored lineographic images using two rotary encoders and view them on a monitor via HDMI. These drawings can then be saved on an SD card and loaded back onto the device at any time. Further, with differential signaling, two FPGAs can draw on the same canvas simultaneously. Finally, this system features a graphical user interface (GUI) that allows users to change the drawing color & stroke width and slide show through the images saved on their SD card.**

I. High-Level Description

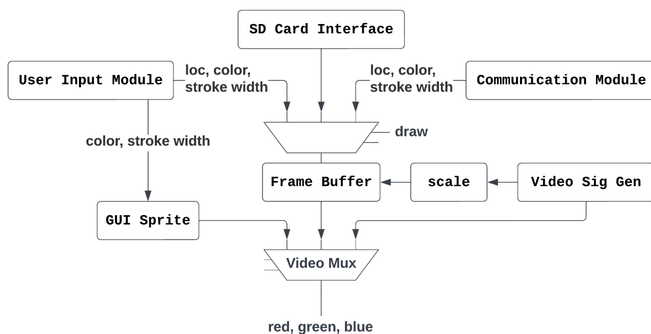

Fig. 1. High-Level Block Diagram

Our project was broken down into four components: the input modules, the communication modules, the video pipeline, and the SD card modules. The user and their collaborator use rotary encoders to provide drawing input signals which are written to the on-board frame buffer (BRAM). The SD storage modules copy images from the frame buffer to the SD card and back as needed allowing slideshow and permanent storage. The video pipeline modules render the image currently sitting in the frame buffer to the screen, allowing the actions of the users, the live sketch, to be visualized.

The input modules handle input from buttons, switches, and rotary encoders and interpret them as cursor directions, drawing color, stroke width, or merely other control signals like reset, draw, or slide-show. The rotary encoders provide directional input and can also be used to change the drawing color and stroke width. An alternative to using the rotary encoders is also supported using a combination of 4 switches for directions and 2 buttons.

To allow collaboration between two sketchers, the communication module sends the user input from one FPGA to the other so that the sketches of one player appear on the second player's screen too. The module uses differential signaling to transfer cursor location and other parameters to and from the second FPGA. The entries are then written to the frame buffer as if they were inputs from the host FPGA.

The storage component extends the on-board storage by allowing some images to be stored in the SD card for use as templates or simply to save progress. Storing images on the SD card also makes it possible to not lose images when the FPGA is unplugged. Once the user and the collaborator have finished drawing, the image is saved to the SD card from where it can be fetched back to the frame buffer and used as if it were always there.

II. The SD Card Interface: Permanent Storage, Using Templates, and Slide-showing Sketches

A. Design Explanation

The SD interface module links the SD controller module (which writes and reads directly from the SD card) and the rest of the system allowing for writes and reads between the on-board block RAM and the SD card. Besides enabling permanent storage beyond power-off cycles, the module also supports slide-showing: displaying previously saved sketches one at a time, usage of old sketches as templates for new sketches, and resetting the permanent storage by deleting old images.

The module takes in $draw$, $slide_show$, $reset_SD_card$, and $manual_slide_show_enabled$ control signals which initiate either drawing, saving to permanent memory, or slide showing of images.

To support permanent storage, the first 4 bytes of the first sector of the SD card is used to store the index where the next new image would be saved. Therefore upon startup, the module reads the first sector to obtain the address so that not only future drawings saved without overwriting currently stored images but also to enable slide show only up to the last stored image. Afterwards, the system goes into an idle state from which drawing, slide-showing, or resetting the SD card could be initiated using the control signals.

If the $draw$ control signal is high, user inputs encoding direction, color and stroke width from the rotary encoders are being written to the frame buffer as the user or the collaborator

draws. Upon switching off the $draw$ signal, the module saves a snapshot of the frame buffer to the SD card and increments the index in the SD card where the next image will be written. The original image on the frame buffer is not modified and should the user desire to draw another image, they would need to erase it by drawing on it with a color consistent with the background color of the image.

If the $slide_show$ signal is switched on, the module reads the address of the last image from the SD card and then starts fetching images from the SD card to the frame buffer. It reads a single image from the SD card and writes it to the frame buffer then it enters a brief dormant state of about 1 second where the current image remains in the frame buffer, allowing the image to be displayed for that long. Once the 1 second of dormancy has elapsed, another image is fetched from the SD card thus overwriting the previous image in the frame buffer. This fetch and rest cycle is repeated until all the images that had been written to the SD card have been displayed.

The module also supports an alternative way to display previous sketches. When $manual_slide_show$ is high, the module allows one to display the next image using a button instead of the automatic rest period. This allows one to quickly scroll to the template of choice or any other sketch that they would want to display. If $slide_show$ or $manual_slide_show$ is switched off or all images have been shown, the module goes to idle state and waits for future signals. To draw on a template (previously saved sketch), use either automatic or manual slide show like described above and turn off the switch responsible for the slide show once the right sketch is reached. One can then activate drawing by turning the $draw$ signal back on again to draw on the template. Once the drawing is complete, the sketch is saved as a new image to the SD card instead of overwriting the template.

One may occasionally want to delete all the previously saved images in the SD card. We support this using $reset_SD_card$ control signal. When $reset_SD_card$ is high, we lazy-delete the images in the SD card by overwriting the address where the next image will be saved (equivalent to the address after the last image saved in the SD card). This ensures that new images are saved the index will be incremented from newly set index and thus overwriting any other image in the SD card.

The module was implemented using a finite state machine with 11 states namely:

- $start_sec_addr_read$: This the entry state. The module waits for a ready signal.
- $read_addr$: The module reads the address from the SD card. If either of the slide-show signals is high, the module transitions to $slide_show_new_sector$. Otherwise it transitions to $idle$ state.
- $idle$: The module is waiting for draw or slide-show signals
- $slide_show_sector$: The module is actively reading a sector from the SD card and writing it to the frame buffer
- $slide_show_new_sector$: The module is waiting to start reading the next sector from the SD card
- $slide_show_next_image$: The module is waiting to start reading the next image from the SD card. If it has reached the last image it transitions to $idle$
- $drawing$: The module is waiting for the user and collaborator to finish drawing
- $saving_sector$ - the module is saving a sketch from the frame buffer to the SD card sector
- $finished_saving_sector$: the module finished saving to a sector of the SD card and is moving on to the next.
- $start_sec_addr_write$: The module is waiting for a ready signal.
- $overwrite_addr$: The module is overwriting the address saved in memory either to increment it or reset it.

The overall state machine diagram is as shown in the figure below:

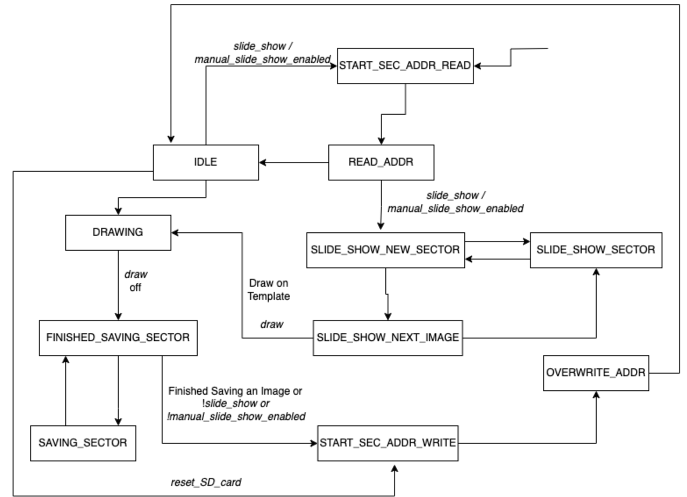

Fig. 2. SD Card Interface Module FSM

B. Design Evaluation

The SD card has a capacity of 2 GB or 2^31 bytes. Each image stored in the frame buffer has 640 x 360 entries where each entry has 4 bits. While copying a single entry from the frame buffer to the SD card, we expand it to a byte so that it occupies 640 x 360 x 8 bits or simply 360 x 640 bytes. While this uses twice as much space in the SD card as we absolutely need to use, we can still store 9320 sketches in the SD card before running out of space. In practice, we are unlikely to get close to this number.

III. Drawing Controls and Video Pipeline

A. Design Explanation

This section highlights both the user input modules and the connection to the video pipeline. The user's input on the FPGA device is interpreted by the $user_input.sv$ module as the cursor's x and y location, the cursor color, and the stroke width. Currently the x and y location are being controlled through two rotary encoders similar to a real life etch a sketch. Pressing down and turning one of the rotary encoders will

cycle through stroke width. Pressing down and turning the other will cycle through the 16 different colors. The pinout for the rotarty encoders is displayed in the figure below.

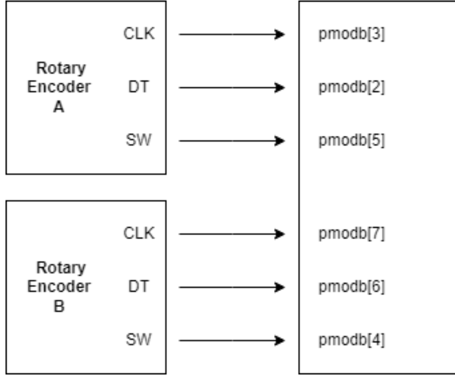

Fig. 3. Rotary Encoder Pinout

The user input module then passes this information along to the frame buffer module which acts as a sort of canvas. The frame buffer is a dual port block RAM with a depth of 360x640 corresponding to the canvas size and a width of 4 corresponding to the amount of bits for color.

The frame buffer uses the hcount and vcount signals that are being created by $video_sig_gen.sv$ along with the x and y coordinates of the cursor and stroke width to determine whether or not to color the pixel. If the hcount and vcount are within the stroke width's radius of the x and y coordinates, the current color is written into the frame buffer. Writing happens to both ports of the BRAM, one for the brush coming from the user input module, and one for the brush coming from the communication module.

At the same time, the frame buffer is being read from the same hcount and vcount. The 4-bit color is then transformed into 24-bit color using a switch case and this is output by the frame buffer module.

The GUI sprite is a module separate from the frame buffer which displays the users current color and stroke width choice on the left side of the screen. The GUI sprite module takes in the current cursor color and stroke width, along with the hcount and vcount to render this static image on top of the canvas.

The canvas is scaled up by a factor of 2 to match the 720x1280 resolution and the color from the frame buffer is passed along the rest of the video pipeline where it is merged with the GUI sprite and passed along to the TMDS encoders and serializers.

B. Design Evaluation

The total amount of BRAM Memory available is 2,760,000 bits. Space is saved by making the frame buffer 360x640 and scaling the image up. Four bits of color are written to the frame buffer thereby giving it a total size of 921,600 bits which is 33% of the total available storage.

It takes one clock cycle to write to the frame buffer and two cycles to read to the frame buffer. The frame buffer module is currently unpipelined which may be leading to some aliasing issues on screen.

IV. Communication Module

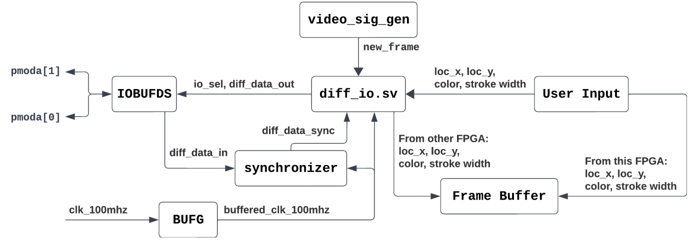

Fig. 4. Communication Protocol

The communication module facilitates DigiSketch's multiplayer feature. The goal is for drawings created by either FPGA to appear on both displays simultaneously. Since the controls to video pipeline works by using the cursor's location along with the selected color and stroke width to write to the appropriate pixels in the frame buffer, that information is also sent to the other FPGA as a 26-bit data packet so that frame buffer can be updated as well.

A. Data Packet Structure

Those data packets are structured as follows:

- `x-loc [25:16]`: The position of the cursor in the x direction. Since there are 640 pixels in the x direction, this field is 10 bits wide.
- `y-loc [15:7]`: The position of the cursor in the y direction. Since there are 360 pixels in the y direction, this field is 9 bits wide.
- `color [6:3]`: The "ID" of the color that is currently selected by the user. Since our system supports up to 16 colors, this field is 4 bits wide.
- `sw [2:0]`: The line width that is currently selected by the user. Since our system supports up to 8 stroke widths, this field is 3 bits wide.

B. Communication Protocol

To minimize the number of wires in our design, we've employed differential signaling. In the final state of our project, one differential pair (two wires total) facilitates bidirectional communication. This module uses a custom messaging protocol (modeled after the IR lab [1]) that involves varying the duty cycle for each piece of information sent. By default, the

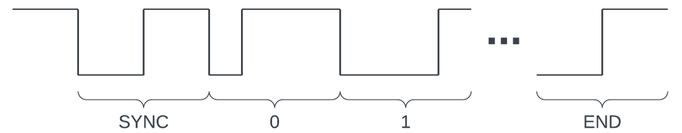

Fig. 5. Communication Protocol

line is held high. The transmission begins with a sort of sync period where the duty cycle is 50%. Afterwards, the 26-bit

message is sent—starting at the MSB. Zeroes are represented by a duty cycle of 25%, and ones by 75%. To prevent any ambiguity with the last bit, the transmission ends with another sync period.

C. Finite State Machines

The transmission and reception modules each operate with an FSM. The transmission module (diff_tx.sv) attempts to transmit the 26-bit data packet on each new frame. The receiving module (diff_rx.sv) reads the synchronized output of the differential signaling input buffer (IBUFDS) and reconstructs the message to be sent to the frame buffer. Their FSMs are outlined in the following figures. As a note, but the receiving module FSM uses the following format: signal_in / signal_counter_range / data_index.

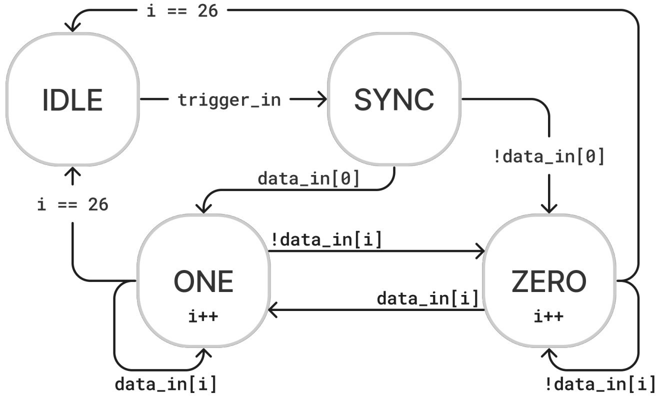

Fig. 6. Transmission Module FSM

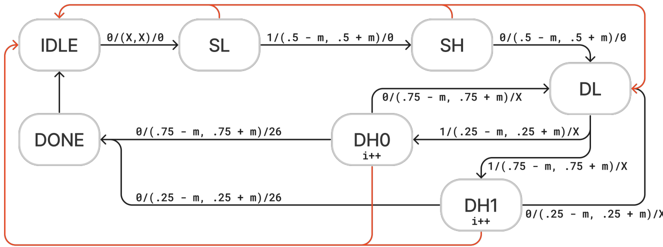

Fig. 7. Receiving Module FSM

Reducing the number of differential pairs from two to one required that some logic be added to determine whether the differential I/O buffer (IOBUFDS) was transmitting or receiving. The transmission and receiving modules are wrapped in a simple major FSM (diff_io.sv) that receives by default and transmits outgoing messages when the line is quiet. This FSM is outlined in the following figure:

Further, to keep the line high when both FPGAs are idling, an internal pullup resistor is added to one of the lines (and a pulldown to the other).

D. Timing/Latency Evaluation

The user input is written to the frame buffer on each new frame, so the communication module has that amount of

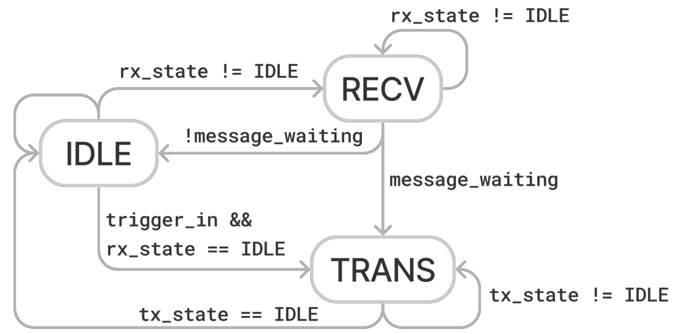

Fig. 8. Major FSM

time to send and receive the data packets. It takes $\sim 5.5\mu s$ (550 clock cycles ÷ 100MHz clock) to send one message. In the worst case, an FPGA has to receive an entire incoming message before transmitting. Thus, data is made available to the other FPGA in at most $2 \times 5.5 \simeq 11\mu s$, which is more than enough time given that our frame speed is 30 fps ($\sim 33.3ms$ per frame). The current state of the communication module satisfies our timing constraints and meets one of the two stretch goals described in the project checklist. From here, we have room to experiment with implementing CRC.

V. Implementation Insights

Here are a couple of implementation insights that have come up by this point in the project.

- Encode colors as a 4 bit number to write to frame buffer saving space.
- Instead of only writing to the frame buffer where the cursor is, using the already incrementing *hcount* and *vcount* to write to the frame buffer.
- A lot of time was lost to trying to get the differential signal output buffer (OBUFDS) to work. However, it was eventually discovered that OBUFDS isn't explicitly necessary for outputting differential signals. Throwing the output and its negation on two different lines is sufficient.
- When running a clock through a clock manager and also using it to power multiple modules, the clock must first be buffered.
- Another thing to consider during the design process is the IOSTANDARD configuration of the pins. We were lucky to have everything work out, but in future projects it would be good to consider that IOSTANDARDs aren't always compatible with each other. For example, BLVDS_25 requires that all other pins in the same bank also be 2.5V, and TMDS_33 isn't always happy to share a bank with LVCMOS33.

VI. External Repo & Contribution Statements

A link to our external repository is as follows: https://github.com/oukoedwin/6205project.

A. *Lydia Patterson*

Lydia designed, implemented, tested, and evaluated the communication interface end-to-end. She also contributed to the high-level design of the system—particularly the GUI and the user input to video display pipeline. With respect to the report, she created the high-level diagram and wrote the summary, the communication section, and a few implementation insights.

B. *Edwin O. Ouko*

Edwin worked on interfacing the SD card with the rest of the system to allow saving and fetching of images from the SD card, usage of previous sketches as templates, and supporting permanent storage on the SD card. He also implemented the slide-show functionality of the system. For the report, he worked on the *High-Level Description* and the *SD Card Interface* sections.

C. *Jordan Edwards*

Jordan implemented the module that interprets the user input and passes it along to the frame buffer. He also implemented the module that uses the cursor location, color, and stroke width to write to the frame buffer. He designed and implemented the basic GUI sprite. Jordan also set up the video pipeline and put things together in the $top_level.sv$ file. He also performed qualitative evaluation of the rough implementation of the system.

D. *Acknowledgements*

6.205 course staff were consulted in the implementation of this project, particularly Joseph Steinmeyer, Adrianna Wojtyna, and Joseph Feldman.

REFERENCES

[1] Author: 6.205 Fall 2023 Teaching Staff, “Lab 03: Catching Some Rays”, 6.205 Website October 2023. https://fpga.mit.edu/6205/F23/assignments/week03/checkoff01

[2] Author: 6.205 Fall 2023 Teaching Staff, “Lab 05: Checkoff 01: An PopCat By Any Other Name”, 6.205 Website October 2023. https://fpga.mit.edu/6205/F23/assignments/week05/checkoff01