NTNU

Kunnskap for en bedre verden

Department of Computer Science

TDT4295 - Computer Design Project

Fall 2022 - Group B

VGAcentrifuge

Final report

Paolo Celati, Jesper Norsted, Amandus Søve Thorsud, Erik Martinsen
Håvard Krogstie, Jostein Bakkevig, Eirik Lorgen Tanberg, Marius Arhaug

November, 2022

1 Introduction

The VGAcentrifuge is a 2-to-1 video mixer with several mixing modes and options, controllable with an on board keypad and lcd screen. The options decide how the two video inputs get merged, with the possibilites of translating, scaling, clipping, chroma-keying and transparantly blending the foreground with the background. With an SD card, the foreground layer can also be replaced with static images.

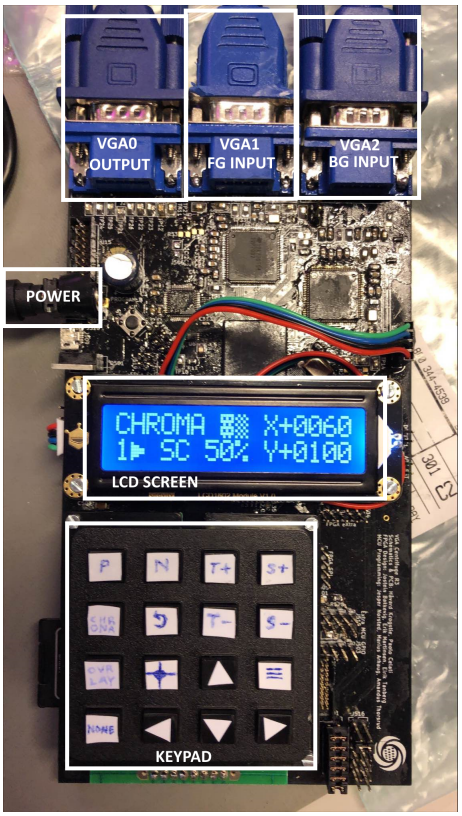

Figure 1: The device turned on

2 User guide

WARNING: Do not connect any VGA-cables before the device is powered and ready. VGA-cables should be disconnected before power is removed. We have had some intermittent problems we believe may stem from voltage back-driving from VGA, so as a safety precaution they should only be connected when the device is ready to receive from them.

Power on the board using a 9V - 16V supply capable of supplying at least 350mA, through the positive core barrel jack connector on the left hand side. Alternatively, use the header just below that.

To get any use out of the device, connect a screen to the leftmost VGA0, and two video sources to VGA1 and VGA2. The middle connector is the foreground layer, while the rightmost connector is the background layer. Both input and output video will be 800x600@60Hz. The connected video sources should be able to tell that they are connected to a "VGAcentrifuge", and know what resolution it supports.

When powered on, the onboard LCD screen shows the current mixing state. Use Figure 2 to decode the different fields displayed on the screen. To interact with the device, use the keypad. See Figure 3 for key names.

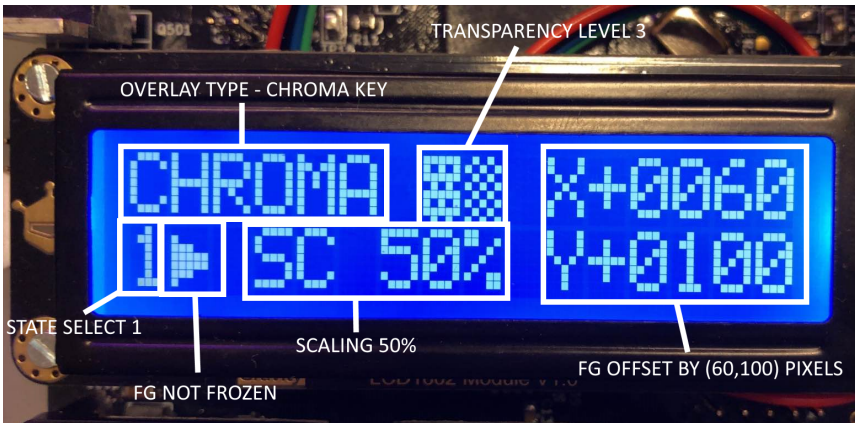

Figure 2: The meaning of different parts of the default mixing ui. The current blending mode is chroma key, the transparency is at 62.5%, state slot is 1, scale is 50%, and offset is (60,100).

PREV STATE	NEXT STATE	TRNSPCY UP	SCALE UP
CHROMA MODE	RESET	TRNSPCY DOWN	SCALE DOWN
OVERLAY MODE	RESET OFFSET	UP	MENU
NONE MODE	LEFT	DOWN	RIGHT

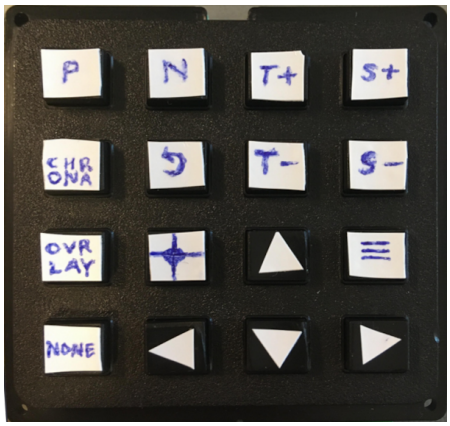

Figure 3: Names of the keys avaliable on the keypad

2.1 General workings

In general, the device works by passing the background signal straight through to the output, while applying the foreground layer on top in some shape or form. The foreground can be scaled down, translated and clipped. Using the different modes allows conditional blending of the foreground, and all modes also support transparency. Replacing the foreground image source with an image is also possible.

2.2 Image loading

The `CHROMA` and `OVERLAY` buttons can be held down to not only switch mode, but also configure a custom foreground source. The list that pops up includes all `.bmp` files on the top level folder of the sd card, as well as normal foreground, called `PLAY LIVE`, and a frozen version of the current VGA foreground input, called `FREEZE`.

2.3 State slots

To allow easy and quick switching between mixing states, the `PREVIOUS STATE` and `NEXT STATE` buttons allow you to circle through 10 states. Changes made in one state will stay in that state. When scrolling through the state slots, the states are displayed on the video output. To move multiple state slots without seeing every state slot between, hold down one of the state slot switching keys until a dialog pops up on the on board screen. Continue pressing the state slot keys to select the state slot you want to move to, and then press `RESET` to jump to that slot. Holding `RESET` will copy the previously selected state into new state slot. Pressing `MENU` will cancel.

2.4 Menu

By pressing the `MENU` button, you are presented with extra options. These include

- **Clipping** enters the clipping menu. See Section 2.5.
- **Chroma key cfg** allows configuring the thresholds for the green screen. The green value is a lower limit, while the red and blue values are upper limits. See Section 2.6.
- **Flash EEPROM1** and **Flash EEPROM2**. Use these when no VGA inputs are connected, and the MCU has been reprogrammed with new EDID, Extended Display Identification Data. See Section 4 for the programming guide.

To pick an option, use the `UP` and `DOWN` keys. Press the `RIGHT` key to select an option. Pressing The `LEFT` key or `MENU` again leaves the menu and goes back to the mixing interface.

2.5 Clipping menu

All mixing modes involving a foreground can optionally clip the foreground, by defining a smaller rectangle within the full 800x600 frame. Images can also be clipped, within the size of the image. Enter the clipping menu from the main menu. The screen now says which border is being adjusted. To change border, move around the keypad in 90 degree increments. Clipping of the current border can be reset with the `RESET OFFSET` key. Removing all clipping can be reset with `RESET`. To leave the clipping menu, press `MENU`.

2.6 Chroma key configuration menu

This menu configures the parameters of the chroma keying mode on the FPGA. The values are used as thresholds for the red, green and blue color channels in order to decide which pixels should be transparent. The green value is a lower limit, while the red and blue values are upper limits. Use the `LEFT` and `RIGHT` keys to choose the color channel threshold to modify, and the `UP` and `DOWN` keys to increase and decrease the value. Use the `MENU` key to return to the menu.

3 Design

3.1 Custom PCB

The PCB consists of two main chipsets, the **FPGA**, a Xilinx XC7A100T, and an EFM3200GG **MCU**. These are connected together through a SPI bus marked FPGA_SPI, which was chosen because of its simple implementation, especially on the FPGA, and the high clock frequency achievable means the MCU sending an image to the FPGA would not take too long. There is also a power supply segment to the board, responsible for converting the input power supply to the rails needed by everything else. An overview of the PCB can be seen as an appendix.

3.1.1 FPGA Hardware

The FPGA is connected to the VGA ports through two TVP7002 ADCs and one THS8136 DAC. We picked VGA because there are many straightforward guides describing VGA output online, and it's a relatively simple and debuggable interface. However, VGA meant we'd need ADCs and a DAC, which made the schematic and PCB design more complicated. This lengthened the time needed for soldering because of the large QFPs and both chips requiring lots of passive components.

The ADCs and DAC use parallel interfaces. This avoids needing SIPO/PISO shift registers, but the number of data lines involved meant PCB routing was slower. Our 5-6-5 bit colour depth choice was largely a result of being unable to route more signals due to the limited FPGA I/O, which is unfortunate because the ADCs provide 10b/channel and the DAC does 8b/channel. Both the ADC and DAC are designed for video purposes, and can handle speeds much higher than what we ended up with. The ADC also has extra features, such as generating a pixel clock from HSYNC, which helped the FPGA implementation. While the ADC and DAC would support the pixel clock of 1080p@60Hz, there are other components that make this unfeasible, such as SRAM bandwidth and SRAM size.

The FPGA is connected to the CY7C1372KV33-200A SRAM IC. This is an 18 Mb SRAM with an 18b data bus, and is capable of operating at 200 MHz. Again, we selected an SRAM with a wide data bus because this meant we could get a high throughput for a given clock frequency despite the cost in FPGA I/O. There is also a CY7C1370KV33-200A SRAM IC available, which would've provided us with 18 Mb of capacity, but with a 36b data bus. We bought the CY7C1372 at the start before we had finished our custom PCB design for prototyping, since we didn't know whether we'd be able to route the 36 data lines for the CY7C1370. When designing the PCB we were however able to route the 36b bus. Since the SRAMs have compatible pinouts, our PCB now has traces for either. In hindsight we would've done well to buy the 36b SRAM for our custom PCB because the SRAM controller design at the end would've been simpler, but we mistakenly passed off the opportunity at the last shopping list round.

Originally we planned to use a DRAM IC as well, because they are far faster, and cheaper. In addition, DRAM would've had the advantage of us being able to prototype on the Arty development boards, because those have DRAM but no SRAM. Even then we needed to have an SRAM IC as well on our PCB, because DRAM being harder to use in the FPGA design meant we needed a contingency plan. Finally we got rid of the DRAM idea, simply because we didn't have enough FPGA I/O to accommodate it, an SRAM with a wide data bus and the DAC/ADCs. The DRAM idea is also part of why we had originally bought the 18b wide SRAM, because the wider one would be useless and we'd lose some capacity if we weren't able to route all data lines.

3.1.2 MCU Hardware

The MCU is primarily used as an interface to the human operator, which is why its schematic and PCB design were somewhat simpler than the FPGA section. The MCU is connected to a keypad, LCD and SD card, and to the FPGA section through a SPI bus. However, the MCU is also in charge of configuring the ADCs through their I2C lines. One can also program the DDC EEPROMs on the VGA inputs through the MCU, and it has miscellaneous nice-to-haves such as being able to reset the FPGA.

The ADC chips support a lot of different types of video inputs, hence the need for configuration over I^2C. We disable things like Sync-on-Green, and map the full output range to a 0-0.7V input range, since we want to use the 5-wire VGA signal with HSYNC and VSYNC lines. Settings like clamp, gain, offsets and coast were set, as well as the number of pixels per line and HSYNC length, to allow the ADC to generate the pixel clock for us. Both VGA1 and VGA2 have their DDC pins connected to EEPROMs programmed with a 128 byte EDID-block, to tell the video sources what kind of signal we want. The MCU can connect to the same I^2C lines to flash them.

3.1.3 Power Supply

We adapted the course's example power supply design, but needed 6 rails. The ADCs needed 1.9 V with a tolerance of +/-50 mV, and we thought it prudent to have a dedicated 3.3 V analog supply for the ADCs. We needed 5 V for the VGA level shifters and EEPROMs, 1 V for the FPGA, 1.8 V for the DAC and FPGA, and a regular 3.3 V supply for most digital logic. We followed advice to use jumpers between the power supply and rest of the board to test the board in isolation. Additionally, we decided to split the 3.3 V in the MCU and FPGA sections and use separate jumpers, in case there were major design flaws in one.

Disconnecting the power supply jumpers also allows easy testing for shorts, by using a multimeter between each rail and ground. This should be done **Between every major component is soldered**, and semi-regularly between support components, to know exactly when shorts are introduced, making it easier to fix. Note: When switched off, some of our rails have a resistance to ground as low as 10Ω. This does not indicate a short, just a case of components ready to pull a lot of current during startup.

3.1.4 Design bugs

Our design turned out to have very few bugs, and most were cleanly fixable.

Schematic:

1. The LSF0102 level shifter schematic was not the manufacturer-recommended one, because we used Nexperia's and their datasheet had very few details beyond the functional diagram. While the PCB was being manufactured we received the level shifters and wanted to check the design with a breakout and breadboard. TI also makes the part, and the design didn't match TI's better datasheet's recommended configuration. Finally we found although the design was not like what TI suggested, we could use the existing design by adjusting pull-up values.
2. The 100 MHz oscillator on the FPGA has an enable pin so one can enable or disable the clock, but we initially used a 10k resistor as a pull-up and this was too weak. When testing

the FPGA clock we found the enable pin's voltage was about 1.4 V, so we replaced the resistor with a blob of solder to pull the pin high and everything worked. Subsequently the board had extremely audible coil whine and we found an input voltage of about 5.5 V caused this, while higher inputs didn't.

3. One of the data pins on the SRAM is not routed. KiCAD does not complain if you use a named wire name in only one location. Luckily this pin is not necessary, since the SRAM is 18 bits wide, and we only used 16 either way.

4. The DDC lines on VGA0 uses the shared I^2C bus. This caused the off-the-shelf I^2C LCD screen to not acknowledge its I^2C packages once a VGA cable was connected. The solution was to de-solder the level shifter and not have any DDC access on VGA0.

Footprints:

1. The footprint for the female header to which the keypad is connected was wrong, as its holes were designed for round pins instead of our square ones. Hence the holes were marginally too small, but we still wanted to use the holes cleanly. We found one could file the header's pins slightly to make the ends sharper and then hammer the header into place, followed by applying solder.

2. The footprint of the MCU DEBUG connector didn't consider the extra width of the cage. We resolved this by cutting off some of the plastic on the connector.

3. While the VGA connectors fit on the PCB, the heads of VGA cables are a bit wider than the ports. A dremel was needed to slim down the middle VGA cable.

4. The VGA connectors have pins arranged in rows, and our footprints had the rows slightly too close together. So we bent the pins slightly and they fit without problems.

Routing:

1. The traces between the SRAM and FPGA are not length matched, and it might be the reason we get artifacts when running at above 80MHz. The SRAM is supposed to be able to run at 200MHz. We were able to mitigate this by changing the phase of the SRAM-clock signal to relax the timing constraints, but running at 200MHz is not possible.

2. We could've made our signal routing simpler by replacing some power planes with layers for signals, and using wide traces instead of planes for some power rails.

Changes we would make if done again:

1. We regret not exposing some ground plane in a corner or more ground headers. It would make it easier to connect alligator probes.

2. We didn't label which pins were which on some frequently used interfaces like the power header, which was inconvenient because you'd need to look at the traces on the bottom to figure out which pin is ground and V_{in}.

3. We regret not adding a reset button for the MCU.

4. We'd like a troubleshooting LED for the FPGA too. It makes debug less cumbersome than connecting an oscilloscope or multimeter to auxiliary I/O. Using a MOSFET to power the LED is probably a bit overkill, since we only need enough current to see some light.

5. We have holes for standoffs for the keypad and LCD, but holes in the corner for securing the PCB to a 3D-printed case would have been nice.

6. The microUSB port for power is kind of useless, since we need more than 5V input to get a 5 V rail for the VGA interfaces. If we had an easy way to disable the 5 V supply and connect microUSB 5 V to the board's 5 V with a jumper that would be useful.

7. We'd merge the schematics and route the PCB much earlier, because the board could be made much smaller and consequently cheaper.

8. We'd use 0603 passives where possible, because they are still easy to solder but would let us have more decoupling capacitors and/or more flexibility in placement with respect to vias.

3.1.5 Design wins:

There are some aspects of the design we are happy we did.

1. The small PCB size of about 18x10 cm is cute.

2. Giving the SD card and FPGA completely separate SPI buses. In theory you can share lines, and only use the CHIP SELECT line to select device, however the platform SDK halted on startup if we asked for manual control of CS. Remember that we had to de-solder level shifters from the main I^2C bus, to make the LCD work again. Don't try to save some pins by putting too much on the same bus.

3. Except the large ICs like DAC, ADCs, SRAM, MCU and FPGA, we considered large pin pitch and size very important when selecting components such as level shifters, ESD diodes and the EEPROMs. Given these were some of the most difficult to solder, any smaller would've been beyond our capabilities.

4. Placing many test pads, making it much easier to debug signals on an oscilloscope. We found small trapezoid-shaped loops to solder, which meant mini grabbers could be attached securely and the added cost/BOM was worth it.

5. Placing many solder jumpers on the board, because these took little space but were useful for troubleshooting problems like the LCD drawing from the VGA 5V pin.

6. Fitting the design onto a 6 layer board, because originally it was looking like an 8 layer PCB would be needed for the FPGA routing.

3.2 FPGA

The FPGA performs several crucial tasks. Most visible to the end user is handling of input, output and image processing. It is responsible for accepting input from both sources through ADCs, synchronizing the inputs such that they can be ran through an image processing pipeline, and then outputting the result. The image processing pipeline can be configured using SPI and a custom protocol. Read more about the protocol in Section 3.4.

The basic layout of the FPGA can be seen in Figure 4, which shows the most important components. The diagram is somewhat simplified, since the outputs and inputs are passed through ADCs and DACs so that we can work with digital data within the FPGA. Internally the FPGA is divided into a few main components. We have some processing modules for handling the input from the ADCs, and for writing output to the DAC. Additionally we have an arbitration module that handles writing and reading to/from the SRAM. Another module handles receiving SPI commands from the MCU and keeping track of settings for the image processing pipeline.

3.2.1 Life of a pixel

The best way to explain how the FPGA operates is to follow the lifecycle of a pixel as it passes through the FPGA. There are two different sources for input, as can be seen in Figure 4, which differ slightly. Input 1 is what we consider the foreground, while input 2 is what we consider the background. Regardless of which input source the pixel comes from, it is added to a FIFO queue for that source. The foreground pixels are read from the queue, and written to SRAM to be used later.

The background pixels are directly passed to the image processing pipeline (See Figure 5), along with its' position in the image. The pipeline's foreground scaling manager then determines what position in the foreground corresponds to this background position, if any. For example, if scaling is set to 50%, the corresponding pixel for background (100, 100) is (200, 200) in the foreground. This is then passed to the foreground clipping manager, which checks if the request (if any) made by the scaling manager is within the bounds. For example, if we are clipping the left side of the foreground by 100 pixels, a request for a pixel at (50, 100) will be denied. The output of this is then passed back through the pipeline, which makes a request to read the foreground pixel with the given position from SRAM. This request takes a few cycles to process, so the background pixel data is stored in a set of shift registers while we wait.

After the specified delay, the background pixel is read from the shift registers and processed along with the fetched foreground pixel. If no foreground pixel was requested, no processing is done and the background pixel is outputted from the pipeline. Otherwise, the foreground and background pixel data is passed to all the different modules that handle the different modes which select which one to use. This may sound complicated, but all that is really happening here is determining whether to use the available foreground pixel or ignore it. If the foreground pixel is ignored, for example when using chroma key mode and the foreground pixel is within the thresholds for being considered a green screen, we simply output the background pixel. If the foreground pixel is selected for use, it will be merged with the background pixel to apply transparency, then this result is outputted.

The output of the pipeline is then passed into a FIFO queue, which the DAC reads from.

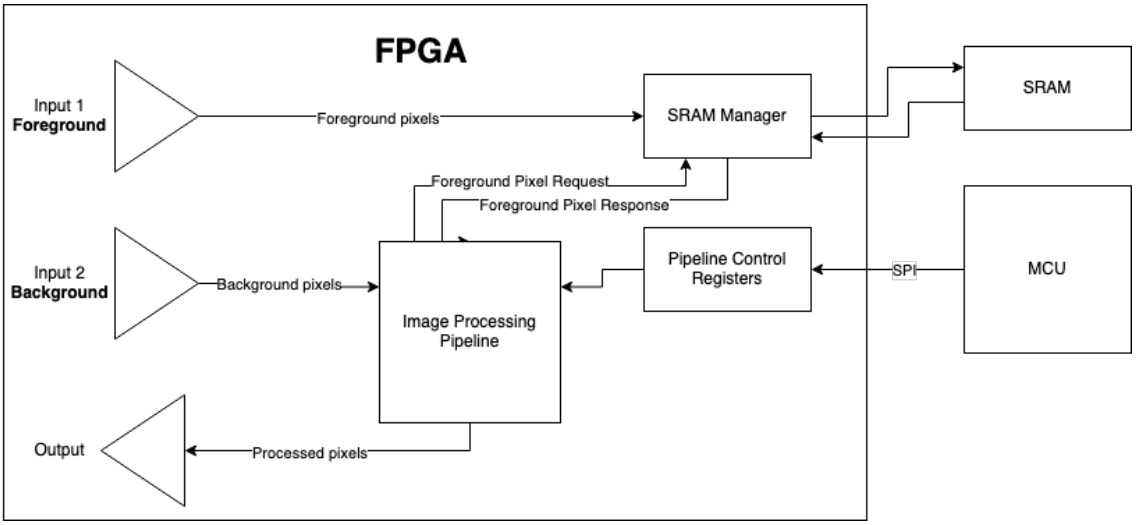

Figure 4: FPGA simplified layout

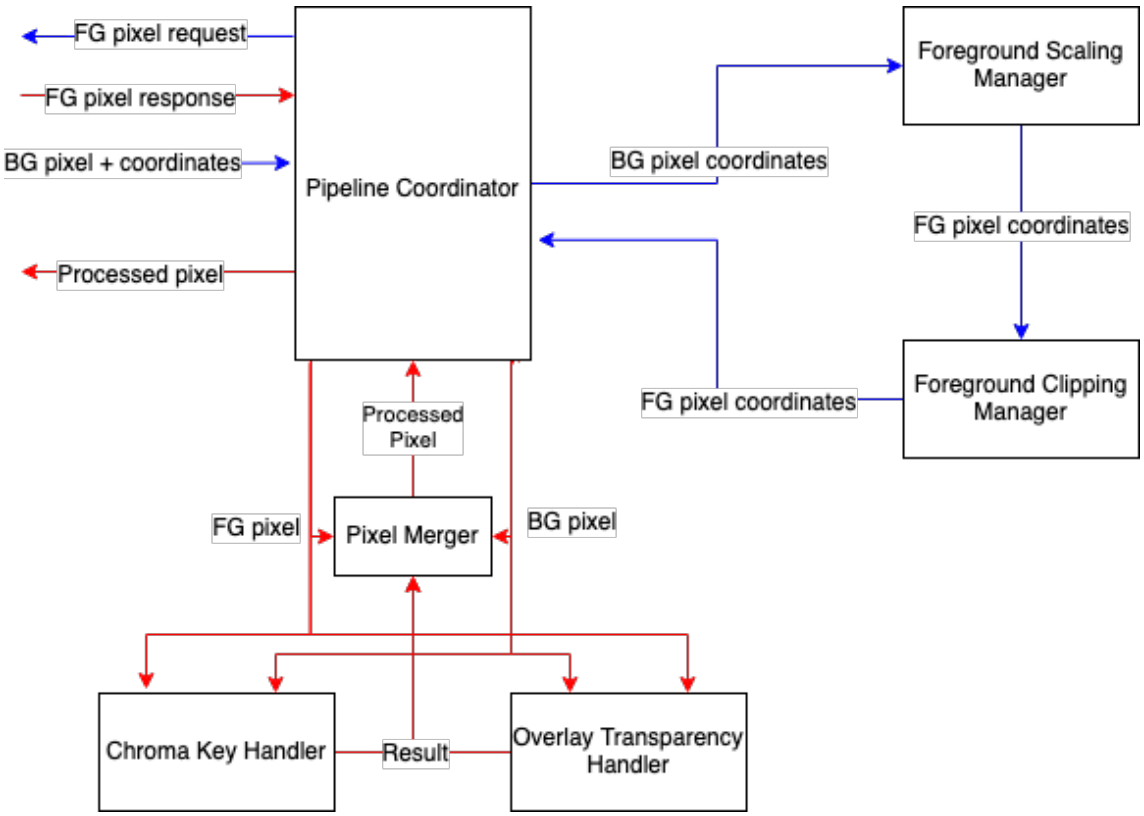

Figure 5: FPGA image processing pipeline. The blue flow describes the initial scaling process when a background pixel is ready. The red flow describes the processing once the foreground pixel request has been processed.

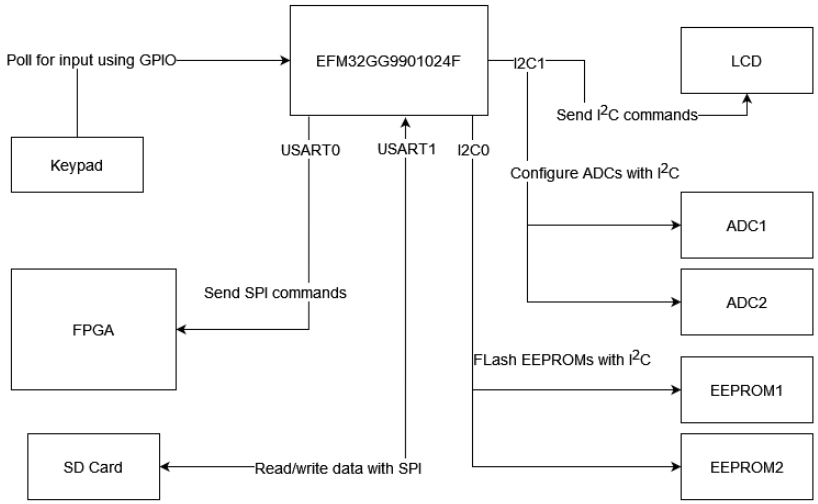

Figure 6: An overview of the MCU and it's peripherals.

Figure 7: The MCU simulator program, currently showing the chroma key configuration menu.

3.3 MCU

The MCU acts as the main controlling unit for handling the various modes and settings we want to apply to our pipeline. It handles user input from a 4x4 keypad, and also displays various menus and configuration options on its own LCD panel. Based on these configurations it tells the FPGA what actions needs to be done, using our custom SPI protocol (see Section 3.4). It also supports reading from the SD card over SPI, where you can store images for overlay effects. Finally, the MCU can flash the EEPROMs which stores the configuration for the DDC part of the VGA ports, and configure the ADCs. For an overview of the MCU and its peripherals, see Figure 6.

The MCU has a quite extensive amount user interface code, so a simulation program was created within the MCU repository. It compiles a mix of actual MCU code with simulations of the hardware libraries, letting us test UI changes quickly. See Figure 7 for an example.

3.3.1 Takeaways

1. Using Simplicity Studio and the SDK was a good idea, since it provided easy access and setup of software components, such as SPI, I2C and timers.

2. Using Simplicity Studio with Git was sometimes difficult, since Simplicity Studio had a tendency to create merge conflicts with its project files.

3. The PCB's 48MHz crystal is not used, since the MCU never managed to switch clock.

3.4 FPGA - MCU communication protocol

The communication uses SPI to send commands from the MCU to the FPGA. These commands alter the configuration for the image processing pipeline. An overview of commands and their purpose are shown in Table 1.

Command ID	Argument	Description	Values
0x0	None	Reset all fields	Sets all fields to 0, except for green screen filter which is set to `16'b0010010110001100`
0x1	uint8	Set foreground mode	0 = none, 1 = overlay, 2 = chroma
0x2	uint8	Set foreground flags	Currently unused
0x3	uint8	Set foreground scale	0 = 100%, 1 = 50%, 2 = 25%
0x4	int16	Set foreground X offset	$(-800, 800)$
0x5	int16	Set foreground Y offset	$(-600, 600)$
0x6	uint8	Set foreground transparency	0 = fully opaque, 7 = max transparency
0x7	uint16	Foreground clipping left	0 = no clipping
0x8	uint16	Foreground clipping right	0 = no clipping
0x9	uint16	Foreground clipping top	0 = no clipping
0xA	uint16	Foreground clipping bottom	0 = no clipping
0xB	uint8	Set foreground freeze	0 = not frozen, 1 = frozen
0xC	uint16	Start foreground image upload on given row	(0, 599)
0xD	uint8	Flip fg/bg for chroma key	0 = use foreground as green screen, 1 = use background as green screen
0xE	uint16	Set green screen filter	Arranged R, G, B with 5, 6, 5 bits, where R and B are upper limits and G is a lower limit

Table 1: Protocol for communication between MCU and FPGA

4 Programming Guide

Design files for the project can be found at the VGA centrifuge GitHub organization: https://github.com/vgacentrifuge. The files are split into three repositories.

1. PCB - the KiCAD design files, as well as a large README detailing the work that was done picking components.

2. MCU - The C source code for the MCU, including the values we ended up using for ADC configuration. Includes simulator as well.

3. FPGA - The Verilog source for the FPGA, as well as some verification and simulation scripts.

There is also a video showing some of the troubleshooting done while configuring the ADCs, and trying to interface with the synchronous SRAM correctly: https://youtu.be/O-lGtJfA8RE.

For programming the MCU or the FPGA, the board needs to be powered first. Ensure the jumpers are also placed so the power supply powers the board.

To write code to the MCU, use the 2x10 pin, 1.27mm pitch, Cortex Debug+ETM Connector called `DEBUG`. Connect it to a Giant Gecko Starter Kit using a ribbon cable, and use the starter kit as a programmer. By opening Simplicity Commander, debug SWO output from the board can also be read. More details are found in the repository README.

You may have to manually reset the board, by pulling the reset pin of the DEBUG connector to ground. Also remember the simulator inside the `/sim/` directory in the MCU repository.

To flash the FPGA, use a Digilent JTAG programmer, connected to the 14 pin 2mm pitch JTAG next to the power connector. A lot of tests have been written for the FPGA "code" using Verilator, which can be found in the `/test/` directory. The included Makefile providers phony targets to run the simulations. For example, running `make sim_pipeline` will generate many images in the `/output/` directory for different settings of the image processing pipeline.

5 Conclusion

The board works great, with very little distortion and jitter. The resolution is of course a bit reduced, and the color depth is not as good as 24-bit color, but that is often not noticeable.

Takeaways:

1. Providing VGA output is easy. VGA input is a fair bit worse. The ADC has a lot of configuration, and the EDID data must be defined to make video sources know what resolution we support. It's honestly hard to tell if HDMI in would be easier.
2. Debugging external components like the SRAM is paramount, far too much time was spent in the last few weeks fixing small timing problems and incorrect signals. Running the SRAM on a very slow clock($\sim$ 1 Hz) through the devboard allowed for debugging simple reads and writes, but the underlying timing problems were not apparent before we started testing on the PCB. Since these components are difficult to unit test, making use of real-time debugging like ILAs could have allowed us to find problems faster.

VGA centrifuge PCB quick reference v1