HardwareX 6 (2019) e00064

Contents lists available at ScienceDirect

HardwareX

journal homepage: www.elsevier.com/locate/ohx

# Triple frame buffer FPGA implementation

James Williams *, Ilya Mikhelson

*Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, United States*

ARTICLE INFO

*Article history:*

*Keywords:*
Verilog
FPGA
Hardware description
Frame buffer
Camera
Education

ABSTRACT

This article demonstrates a Verilog-based triple frame buffer capable of buffering arbitrary data, such as camera frames, between any two asynchronous processes. The frame buffer modules consume 143 logic elements and use a simple, intuitive design. Herein, we discuss the overall implementation of the design as well as practical uses such as in a small camera, or for use as an educational tool.


## 1. Specifications Table

| | |
|---|---|
| **Hardware name** | General Purpose Triple Frame Buffer |
| **Subject area** | • Signal Processing<br>• Imaging Hardware<br>• Educational Tools<br>• Open Source Alternatives to Existing Infrastructure<br>• General Data Buffer |
| **Hardware type** | • Imaging tools<br>• Electrical engineering and computer science<br>• Data Science |
| Open source license | GNU General Public Licence Version 3 |
| **Cost of hardware** | Test Hardware: $734.69 USD, price may differ significantly depending on implementation. |
| **Source file repository** | https://github.com/fluorine21/Triple-Buffer |

## 2. Hardware in context

This hardware can primarily serve as a frame buffer for a simple camera setup. As the camera transmits frames, the frame buffer reads the frames into a buffer, and later writes the frames out to another piece of hardware such as a microcontroller

* Corresponding author.
*E-mail addresses:* james.an.williams@gmail.com (J. Williams), i-mikhelson@u.northwestern.edu (I. Mikhelson).

https://doi.org/10.1016/j.ohx.2019.e00064
2468-0672/

or CPU. The flexibility of FPGA fabric allows our buffer design to be tailored to and integrated with a user's project while adding as little overhead as possible.

Frame buffers have been in use since Noll at Bell laboratories [1] used a frame buffer to implement a graphical display on a Honeywell DDP-224 mainframe computer. As the need for graphics processing and graphical user interfaces increased, double and triple frame buffers were implemented. Double buffers have the advantage of allowing both processes to access at least one buffer at a time, however one process (i.e a camera or a display) will need to wait until the other has finished with the buffer before a swap can occur. Triple buffers avoid this waiting problem by always having an extra buffer available so that one process can still use one buffer uninterrupted while the other swaps buffers. This effectively decouples the processes from one another and allows them to run independently. Double buffering can also be used in applications which are not strictly graphical such as DMA to meet the addressing requirements of a device. These buffers are referred to as "bounce buffers" in UNIX [2]. One example of a proprietary double buffer can be seen in Nvidia's patent [3] where two logical buffers are used to buffer image data between a system bus and a display. Oracle also holds a patent [4] on a frame buffer system which utilizes a similar double buffer scheme. Nvidia has used triple buffering to implement a software-side rendering speedup known as "Fast Sync" [5]. All of these examples demonstrate special purpose buffers built to suit the needs of a specific system. Without using re-programmable logic, it is difficult to build an adaptable buffer using off-the-shelf components. By using an FPGA to implement the buffer, the design can be customized to fit within the confines of a given project. To the extent of the knowledge of the authors, a generalized open-source verilog-based triple frame buffer has yet to be presented.

Many examples of FPGA-based image processing techniques are already available [6–10]. Johnson et al. [9] detail several techniques for creating image processing designs, the majority of which used pipeline approaches. Hiraiwa et al. [8] demonstrate a real-time computer vision architecture which uses an FPGA-based pipeline to implement complex vision algorithms. Greisen et al. [10] demonstrate a high definition stereo video processing and correction pipeline whose front end is implemented using an FPGA. Battle et al. [6] present an image processing technique which uses a parallel pipeline architecture consisting of reprogrammable processors to implement image processing algorithms. Unzun et al. [7] present a fast Fourier transform algorithm implemented in an FPGA using a pipeline approach. The common theme between these image processing techniques and many others is that they used a pipeline approach in which data is processed on a continuous basis. This makes these algorithms excellent candidates for integration with our frame buffer. These algorithms could be used as a replacement for the VGA controller in our reference design, allowing them to take full advantage of the reduced latency and frame loss. The process of integrating algorithms similar to these into our design will be covered further in 6.3. Some examples of open source FPGA-based buffers exist [11,12], however these are tailored for a specific purpose such as HDMI video buffering or power saving. Many FPGA projects such as [13,14] include various buffer stages, but fail to provide any insights into their design.

Some of the source files for this project were originally obtained from [15]. Their implementation makes use of a dual-access memory buffer implemented using FPGA resources. While this method is faster and simpler than our triple buffer approach, it requires vast amounts of device resources to construct the memory, and thus is not feasible for smaller devices or very large buffers which would be better implemented as SRAM or SDRAM buffers for cost saving purposes. While many modern microcontrollers include features such as DMA which allow a frame buffer to be implemented [16], these implementations suffer from delay caused by the inherent overhead imposed by the microcontroller along with any overhead incurred from the DMA subsystem.

## 3. Hardware description

This implementation consists of a proprietary buffer controller module, three custom multiplexer (mux) modules, and several peripheral modules used for capturing and transmitting data. All modules from the authors are written in Verilog and are completely customizable to enable users to change bus width, frame eviction policies, and interface protocols to fit their project needs. Some modules are sourced from other projects [15]. Compared to previous methods such as ASIC or software implementations, this implementation has two advantages. While ASIC implementations are immutable and cannot be modified after they are put into silicon, our Verilog implementation allows for hardware specifics to be easily modified to suit the user's needs. While software implementations are arguably more customizable than HDL implementations, their speed and performance will be far less than that of an HDL-based implementation due to the execution overhead. This implementation can also serve as a starting point for more complex implementations which may include error checking, flow control, or other capabilities.

### *3.1. Possible usage cases*

- **Local frame buffer capable of a direct interface with any CCD or camera.** With a few modifications, this frame buffer can easily be made to work with proprietary CCD modules under development. The provided example design showcases this usage with the OV7670 camera module.

- **Educational Tool.** This frame buffer can be used to teach students about the concept of frame buffers, how to effectively implement them, and how to modify already-existing Verilog code to fit the needs of a given project. For example, the microprocessor projects course at Northwestern University revolves around the development of a small IP camera using an OV2640 JPEG camera module. A part of this project could revolve around modifying and using this buffer design to buffer incoming camera frames instead of using the built-in frame buffer in the microcontroller.
- **Generalized Data Buffer.** By implementing flow control, this frame buffer can be used to losslessly buffer any arbitrary data between a sender and a receiver. Typical usage cases would include buffering packets between two different types of connections (e.g. fiber to copper), or buffering incoming data streams and packetizing them in preparation for transmission via a network. An example implementation is described at the end of this article and is included in the source file repository.
- **Add-on to Real-Time Image Processing Hardware** As discussed in the literature review, this hardware can be integrated with existing image processing hardware in order to minimize latency and frame loss. The latency added by this hardware would be minimal as the critical paths of this design traverse the large MUX (assuming buffer handover time need not be shorter than 1 clock cycle) which is relatively fast compared to other processing blocks or buffers present in a user's design.

## 4. Design files

### 4.1. Design files summary

| Design filename | File type | Open source license | Location of the file |
|---|---|---|---|
| triple_controller.v | Verilog Source File | GNU GPL V3 | [17] |
| triple_muxes.v | Verilog Source File | GNU GPL V3 | [17] |
| sram_controller.v *# | Verilog Source File | GNU GPL V3 | [17] |
| capture_buffer.v * | Verilog Source File | GNU GPL V3 | [17] |
| transmission_buffer.v * | Verilog Source File | GNU GPL V3 | [17] |
| TripleBuffer_top.bdf * | Block Diagram File | GNU GPL V3 | [17] |
| vga.vhd *# | VHDL Source File | Unknown | [15] or [17] |
| rgb.vhd *# | VHDL Source File | Unknown | [15] or [17] |
| vga_addr_gen.vhd *# | VHDL Source File | Unknown | [15] or [17] |
| ov7670_capture.vhd *# | VHDL Source File | Unknown | [15] or [17] |
| ov7670_controller.vhd *# | VHDL Source File | Unknown | [15] or [17] |
| clock_dividers.v * | Verilog Source File | GNU GPL V3 | [17] |
| test_bench.v * | Verilog Source File | GNU GPL V3 | [17] |
| * = not required in user design, # = from other source (see references). | | | |

### 4.2. triple_controller.v and triple_muxes.v

"triple_controller.v" contains the buffer controller "tri_control.v" referred to as "Data Controller" in Fig. 1. This module is responsible for controlling the buffer switching and deciding which SRAM controller a module should be switched into once it has finished with its previous buffer. A logic diagram of the controller can be found in Fig. 2. Definitions for the active state of the trigger signals (low or high) can be changed by providing parameters when declaring the module.

When a trigger signal is detected from either the capture or transmission buffers (i.e. buffers), the buffer controller first performs low-pass filtering on the edge to determine whether the edge is due to noise or the start of a frame ("State Machine Trigger Filter" in Fig. 2). If the edge is valid, the buffer switching logic decides which SRAM controller (i.e. controller) to switch the relevant buffer into based on which controller the other buffer is currently occupying. These trigger signals are typically the "vsync" signals for either buffer as this always denotes the end of a frame. For example, assume that controllers X, Y, and Z are available, and that the capture buffer is using X and the transmission buffer is using Y. If the capture module sends a trigger signal to the buffer controller, it will switch the capture buffer into Z, the only available SRAM controller.

The buffer controller also implements a counter to avoid problems with "ghost frames". In certain cases, the capture and transmission buffers can switch in a pattern such that one controller is never accessed by the capture buffer for long periods of time, resulting in an old frame staying in the system and appearing in the video feed as a "ghost frame" or a shadow on top of new frames. The counter works by counting the number of times the transmission module has accessed a buffer. If the transmission module sends a trigger signal and the only available buffer has been accessed more than two times, the buffer controller keeps the transmission module on its current buffer and does not switch it into the stale buf-

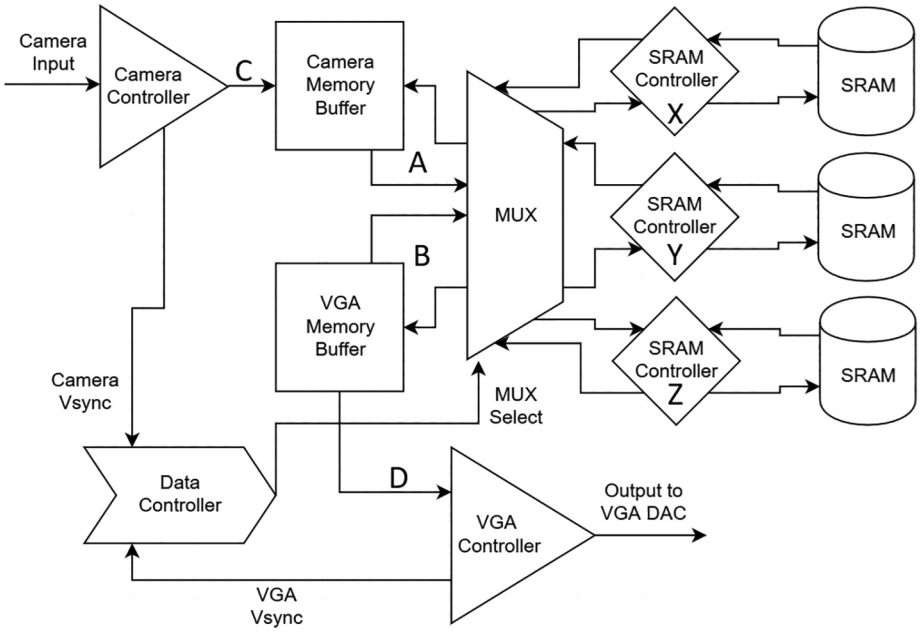


**Fig. 1.** A logic diagram of a typical implementation of the buffer.

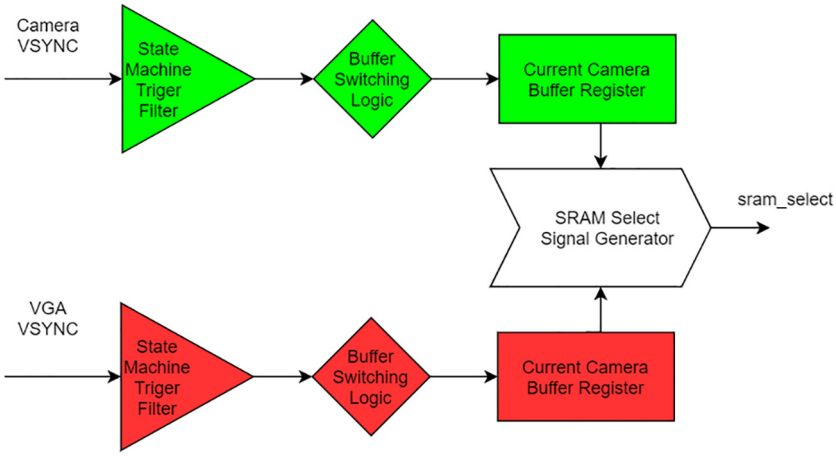


**Fig. 2.** A logic diagram of a the buffer controller in Section 4.2.

fer. This ensures that the next module to access the stale buffer will be the capture module so that the old frame can be updated.

"triple_muxes.v" contains three multiplexers: tri_mem_mux, tri_data_mux, and tri_ready_mux. tri_mem_mux is responsible for switching the output signals of the capture and transmission buffers to the input signals for the SRAM controllers (sram_controller.v). tri_data_mux is responsible for switching the output data of the SRAM controllers back to the transmission buffer. tri_ready_mux is responsible for switching the ready outputs from the SRAM drivers to the capture and transmission buffers. All three multiplexers are implemented using ternary operators. A definition of the buffer switching mnemonics can be found in the code for each multiplexer as well as each buffer.

All address and bus sizes of the modules created by the authors can be modified by changing the "addr_bus_size" and "data_bus_size" parameters in the module declaration with the use of #(addr_bus_size, data_bus_size) before the port declaration and after the module name. Some bus sizes will need to be changed manually within the module definition as they are specific to third party modules found within the design.

### 4.3. sram_controller.v

The file "sram_controller.v" contains an implementation of an SRAM controller suitable for working with most generic asynchronous SRAM chips. Our implementation uses [18]. The controller is originally the work of Shawn Gerber and can be found at [19].

### 4.4. capture_buffer.v and transmission_buffer.v

The file "capture_buffer.v" contains the module "capture_buff". This module is responsible for reading incoming pixel data from the OV7670 capture module and writing it to the SRAM. It works by simply latching data from the OV7670 module when "WE (write enable)" is active, and then pushing the data to the SRAM's input lines and initiating a write by pulling "start" low. The address to which data is written must also be provided via the "addr_in" input.

This module can serve as a good starting point for a user implementation. Its address and data bus sizes can be changed by modifying the parameters at the top of the module declaration. More specific implementation details are included in the code comments, which should enable users to modify the buffer to suit their exact needs. Please note that by default, the capture module only stores the lower 12 bits of incoming data and sets the upper 4 bits to 0 as per our implementation. This behavior can be changed by editing the "assign" statements at the end of "capture_buffer.v".

The file "transmission_buffer.v" contains the module "tr_buff". This module is responsible for reading data from SRAM and transmitting it to an external module. It does this by looking for changes on its "addr_in" input. When a change is detected, the module tells the SRAM to read the memory location at the new address value, and then pushes the result to its "data_out" output. In our implementation, this external module is the VGA driver (see 4.6).

As with the previous module, this module can also serve as a good starting point for a user implementation. It is customizable in an identical fashion to the capture_buff, and includes extensive code comments detailing operating specifics as well. Further implementation details for both of these modules will be covered in 6.2.

### 4.5. TripleBuffer_top.bdf and TripleBuffer_top.v

The files "TripleBuffer_top.bdf" and "TripleBuffer_top.v" contain the top level design of our sample implementation. They should be used as a reference for the user top level implementation as all connections to and from the buffer controller and the muxes should be very similar if not identical.

### 4.6. VGA components

The files "vga.vhd", "rgb.vhd", and "vga_addr_gen.vhd" contain all modules responsible for driving the VGA output signal of our sample implementation. They were originally sourced from [15].

The user may wish to replace this output module with a SPI or UART interface. The user replacement for this module should request data from the transmission module by pushing the desired memory location to the "addr_in" input of the transmission module. A response can then be expected from the "data_out" line of the transmission module once its corresponding "ready" signal is active. More implementation and integration specifics will be covered in 6.2.

### 4.7. OV7670 components

The files "ov7670_capture.vhd", "ov7670_controller.vhd" and "ov7670_registers.vhd" contain the modules necessary for configuring the OV7670 camera over I2C and reading out the pixel data. They were originally sourced from [15].

The user may wish to replace these modules with modules which are specific to their camera or their incoming data. Only the capture module is necessary for the frame buffer. This module must capture incoming data and provide the "capture_buffer" (Section 4.4) module with the data, an address at which to store the data, and a write enable line to latch the data. More implementation and integration specifics will be covered in 6.2.

### 4.8. Reference design

The reference Quartus project included with this paper is intended to be demonstrated using the DE2-115 development board [20]. The pin planning has been set up such that the camera should be connected to the 40 pin GPIO expansion header, the two external SRAM chips should be connected to the HSMC expansion header, and the VGA signal should originate from the on board VGA DAC. Please see the pin planning window within the Quartus project along with Section 7 for further pin planning details. The top level design contains PLLs and clock dividers for generating a 150 MHz clock and a 50 kHz clock. These are only for use with Signal Tap, and will be optimized out of the design if Signal Tap is not enabled.

## 5. Bill of materials

| Device | Cost | Description |
|---|---|---|
| DE2-115 Development Board [20] | $595 USD | An FPGA development board containing the Cyclone EP4CE115. The provided reference implementation is configured for use with this board. |
| OV7670 Camera | $7.08 USD | A simple bitmap camera capable of being configured over I2C and transmitting data over a simple parallel interface. |
| 16-bit SRAM chip [18] | 2 units at $22.82 USD = $45.64 | Standard SRAM chips functioning as the three buffers. Can also be implemented in the FPGA. |
| VGA Monitor | $24.97 (refurbished) | A standard VGA monitor capable of viewing images outputted by the VGA DAC on the DE2-115 development board. |
| Assorted Wires | $7 USD | Wire connections for connecting the camera and SRAM chips to the board. |
| HSMC Breakout Board [21] | $55 USD | Breakout board for connecting SRAM to HSMC header. |

## 6. Build instructions

### 6.1. Initial module setup

This section will cover how to set up the buffer controller (Section 4.2), the multiplexers (Section 4.2), and the SRAM controllers (Section 4.3). All clock and reset lines of these modules should be connected to the same global clock and reset lines.

We begin by instantiating the main multiplexer "tri_mem_mux" found in the file "triple_muxes.v" (MUX in Fig. 4). This multiplexer is responsible for connecting the control, address input, and data input lines of the capture and transmission buffers to the SRAM controllers. The output side of the mux has three groups of inputs labeled "X","Y", and "Z". Three SRAM controllers should be instantiated, and their inputs should be connected to the outputs of the mux with respect to the groupings (see SRAM Controllers in Fig. 4). All grouping labels are identical in the code base.

Next, the data and ready multiplexers should be instantiated. Note that while the design itself contains 3 muxes, the memory mux, the data mux and the ready mux, these have been simplified to a single mux in all figures. The three inputs of the data mux should be connected to the data outputs of the three SRAM controllers with respect to the "X", "Y", and "Z" groupings found on the inputs of the data mux. The output of the data mux should be connected to the data input of the transmission buffer. The ready mux should then be instantiated, and its inputs should be connected to the ready outputs of the SRAM controllers with respect to the "X", "Y", and "Z" groupings. Its outputs should then be connected to the ready inputs of the capture and transmission buffers with respect to the "A" and "B" groupings.

Finally, the buffer controller should be instantiated. The capture_trigger input should be connected to the vsync signal of the camera, or any equivalent signal which denotes that the capture buffer has finished writing data, and is ready for a new buffer. The transmission_trigger input should be connected to the vsync signal of the VGA controller, or any equivalent signal which denotes that the transmission buffer has finished reading data, and is ready for a new buffer. The "sram_select" output should then be connected to the select inputs of the memory, data, and ready multiplexers.

### 6.2. User-defined module setup

Once the setup of the initial modules has been completed (Section 6.1), the user modules for defining the data input and output protocols can be added to the design. The first module to be added should be a module capable of accepting incoming data and writing it to the buffer. In our implementation, the capture_buffer module performs this task (see Section 4.4 and "Camera Memory Buffer" in Fig. 3). It should be connected to the memory, data, and ready muxes through group "A" (see Fig. 3).

Second, a module capable of reading data from the buffer and writing it to an output controller should be added to the design. In our implementation, the transmission_buffer module performs this task (see Section 4.4 and "VGA Memory Buffer" in Fig. 3). It should be connected to the memory, data, and ready muxes through group "B".

Next, a module capable of accepting incoming data from an external source and parallelizing it so that it can be stored in the memory buffer by the capture module should be added to the design. In our reference implementation, the "ov7670_capture" module performs this task (see Section 4.7 and "Camera Controller" in Fig. 3) Its data output should be connected to the capture buffer via connection "C" in (Fig. 3).

Finally, a module capable of accepting incoming data from the transmission buffer and outputting it from the FPGA should be added to the design. In our reference implementation, the VGA modules perform this task (see Section 4.6 and "VGA Controller" in Fig. 3). Their data inputs should be connected to the outputs of the transmission buffer (see Fig. 3 connection D).

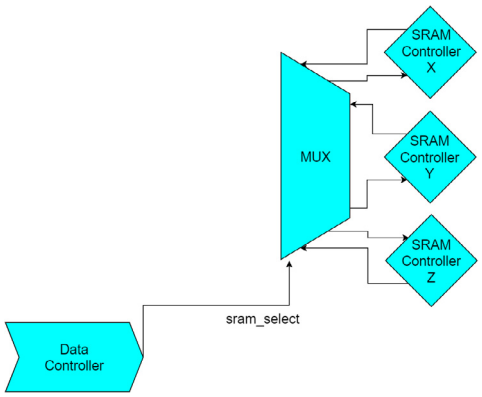


**Fig. 3.** A logic diagram of the necessary modules needed to implement the buffer (necessary modules shown in blue).

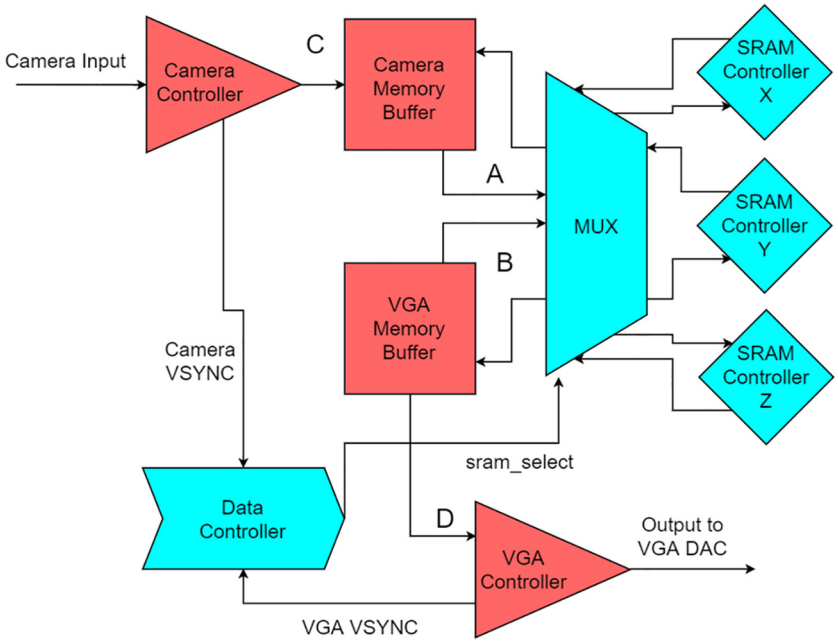


**Fig. 4.** A logic diagram of the necessary and user supplied modules (red modules are user supplied).

### *6.3. Image processing module integration*

The best method for integrating image processing in our design would to replace the "VGA Memory Buffer" in Fig. 3 with the initial stage of a pipeline capable of accepting packets of data whose size matches the width of the data bus (12 bits in our case). This initial pipeline stage would read out data from the SRAM controller and feed it to the rest of the pipeline. If later pipeline stages require larger pieces of data to work with, then this initial stage could also package data into larger blocks before sending it down the pipeline.

If the user wishes to process images before they are stored in memory, then the pipeline could be implemented within connection C in Fig. 3. This pipeline would continuously encode pixels from the camera into a byte stream to be stored in the buffer. The user could implement two separate pipelines, one to process images before they come into the buffer and another to process images as they leave the buffer. Depending on the architecture of the pipeline, it may be necessary to implement flow control as to not lose frames and avoid overrunning a pipeline process.

### *6.4. Compilation, debugging, and the test bench*

Typical problems with compilation arise from invalid fitter settings or reserved pin settings. As one of the reserved pins (P28 for the reference project) is used by one of the SRAM chips, it will need to be set as default IO via Assignments, Device,


**Fig. 5.** A screen shot of the SignalTap wave view window showing the hsync and vsync signals of the camera and VGA modules along with the buffer select signal.

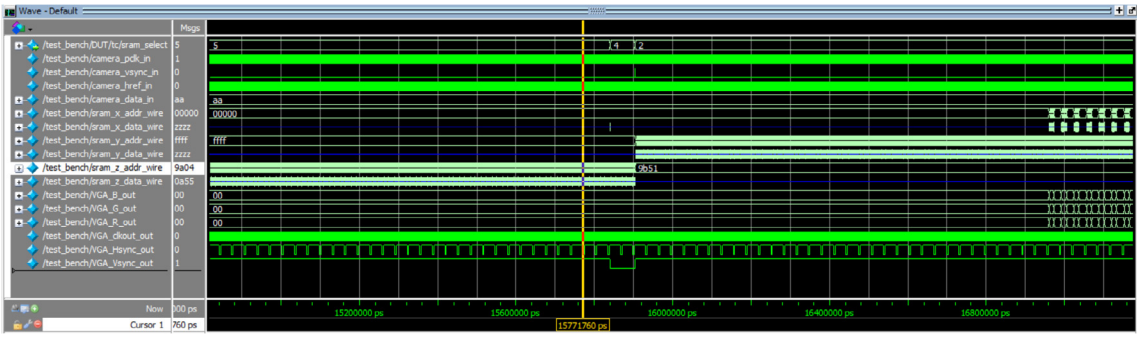


**Fig. 6.** A simulation of the buffer made using the provided test bench. This wave view shows the camera and VGA control signals as well as the SRAM address and data signals. Two frame buffer swaps can be seen immediately after the yellow cursor position due to the activation of the camera and VGA vsync signals.

Device and Pin Options, Dual Purpose Pins, and then changing the value of the appropriate pin to Use as regular I/O. Other fitter problems also include setting the IO standard settings to their appropriate values in the pin planning window. The clock input, button inputs, and LEDs should all be set to 2.5 V. Everything else should be set to 3.3 V CMOS.

Perhaps the most useful debugging tool is the Quartus Signal Tap logic analyzer. Signal Tap allows you to instantiate a logic analyzer within the FPGA to analyze and debug internal signals. Common debugging uses for Signal Tap for this design would include analyzing incoming and outgoing data to ensure that the data is free of corruption, and analyzing buffer timing to ensure that the buffer is switching properly (see Section 8 for more details). For example, if your design uses a camera capable of exporting JPEG images, you can check the incoming and outgoing data to ensure that each frame begins with 0xFFD8 and ends with 0xFFD9. A screen shot of a typical SignalTap wave view can be seen in Fig. 5.

The provided reference design also includes a test bench in "test_bench.v" which instantiates the top level diagram and provides it with simple test inputs which allow for easy verification of functionality. A full run of the test bench using Altera's ModelSim can be see in Fig. 6. The test bench works by providing the camera with a series of pixel inputs and SRAM inputs to ensure that the SRAM outputs and VGA outputs are working properly. Due to the inherently large amount of memory consumed by simulating SRAM, the included test bench does not connect the top level design to simulated SRAM. A simulated SRAM module is however included in the project for users who can simulate the design using Icarus Verilog on a 64-bit system to avoid memory issues.

## 7. Operation instructions

When operating the provided reference implementation, the OV7670 must first be connected to the DE2-115 development board via the GPIO interface. The exact pin connection specifics can be found in the pin planning window of the reference Quartus project, and the pinout of the GPIO header can be found on the DE2-115 website [20].

The next step is to connect the two external SRAM chips. Our reference implementation uses the chips found here: [18]. These two chips should be connected to the FPGA via the HMSC expansion header. The third required chip is already on the board. A standard 40-pin breakout for this header can be found here: [21]. A complete list of the pin connections can be found in the Quartus pin planning window for the project along with the data sheet for the breakout board. It should be noted that the "lb_a_n", "ub_a_n" and "ce_a_n" pins for both external SRAM chips should be connected to ground rather than the FPGA.

Next a VGA monitor must be connected to the VGA output on the top of the development board. Power can then be applied to the board, and the design can be uploaded via the onboard Quartus USB Blaster and the device programming window in the Quartus software. A few moments after the design has been uploaded, a video should be visible on the VGA monitor. A labeled photo of our test setup is available in Fig. 7.

Button #0 is configured to be the reset button for the frame buffer. This will reset all internal states within the frame buffer along with reconfiguring the camera via the I2C interface. In the top level design simply called "TripleBuffer", Button

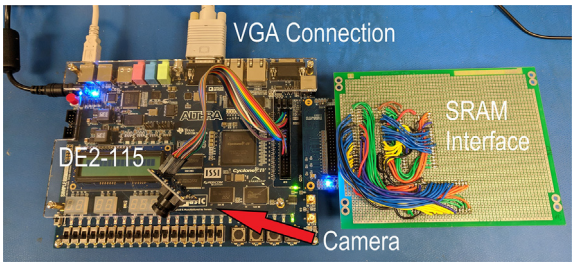


**Fig. 7.** The test setup used to validate our design.

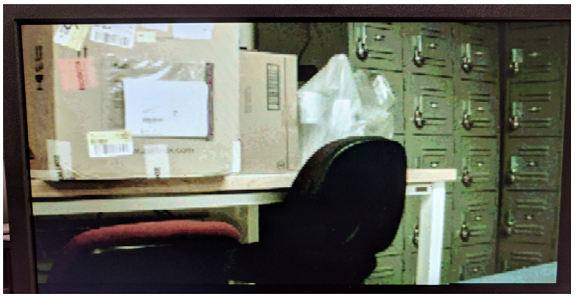

**Fig. 8.** A test image from the buffer.

#1 will reconfigure the camera without resetting the system. The green LED #0 is the "config_finished" LED which denotes when the FPGA has finished configuring the camera.

## 8. Validation and characterization

In order to characterize the operation of the frame buffer, several important internal signals can be analyzed in order to provide information such as frame rate, buffer handover timing, and the frequency of stale frames. A sample frame from our test setup is available in Fig. 8.

To ensure that the buffer handover is working properly, the "transmission_trigger", "capture_trigger", and "sram_select" signals should be probed via SignalTap along with the internal "x_cnt", "y_cnt", and "z_cnt" registers as well. The SignalTap file is already set up within the reference project, however it is disabled by default. After enabling Signal Tap and enabling the aforementioned signals, the buffer handover procedure can be validated by ensuring that every time one of the trigger signals becomes active and the counter associated with the available buffer is less than or equal to 2, the "sram_select" signal changes, indicating that one of the buffers has been switched into a new SRAM chip. For the case in which one of the counters is 2 or greater, it should be ensured that when the "transmission_trigger" goes active, the buffer controller does not change its "sram_select" signal and prevents the handover from occurring.

These signals can also be used to measure frame rate. The frame rate for this device assuming that each frame is a new frame can be taken to be the rate at which the "sram_select" signal changes state immediately after the transmission trigger becomes active. Simply measuring the rate at which the "sram_select" signal changes will provide an inaccurate measurement as the signal also changes when the camera has finished writing a frame. The actual frame rate for this device, or the frequency at which a frame is sent to the VGA controller, is defined by the rate at which the VGA vsync signal becomes active.

## 9. Scalability and timing closure

As mentioned in Section 4.2, the address and data bus sized can be modified by explicitly providing parameters for these values in the module declaration. This allows for the MUXes to be scaled to fit any design. While increasing the bus widths will change the sizes of the MUXes and thus change the number of logic elements needed, the size of the data controller will not change. The largest buffer that could be created with this hardware is only a function of the memory modules, not the control hardware. For example, on the DE-115 board, one could use the HSMC expansion header to add 2 additional 19-bit SRAM chips, bringing the total buffer storage to 1.5 MB (500 KB per buffer).

While the authors will provide relevant timing information regarding the example design, it is important to note that timing requirements can vary greatly across various platforms. For example, high-end FPGAs will be able to run this design at much higher speeds while still meeting timing requirements while cheap and widely available FPGAs may struggle to achieve timing closure at faster clock speeds. When operating larger designs, it is likely that the critical paths will be found

in modules added by the user. This is because the buffer logic does not contain data paths long enough to dominate the timing requirements in most situations. This is the case for the presented example design as the longest delay path is found in the VGA signal generation logic.

This design can be properly constrained with an input clock no faster than 167.5 MHz under the current operating conditions of the camera and VGA module (both running at 25 MHz) and the current hardware. It may be possible to still constrain the design at higher clock frequencies by increasing the clock frequencies of the camera and VGA signal generator.

## 10. Additional reference design

An additional reference design has been provided which showcases a usage case in which the incoming data stream is significantly faster than the outgoing data stream. In the design, the capture buffer has been replaced by a UART receiver while the transmission buffer has been replaced by a UART transmitter which operates 8 times as fast as the receiver. The buffer controller allows the two to exchange data while allowing both UART controllers to swap buffers independently.

In the design, the UART transmitter is gated by a control signal to showcase the importance of the independent buffer exchange. If the UART transmitter did not have any restrictions on when and how fast it could transmit, then a triple buffer system would be unnecessary. However, because the transmitter can be delayed unexpectedly via the control signals, it is possible that the transmitter would not be finished with a buffer by the time the receiver had finished with its buffer, which would force the receiver to wait for the transmitter. With three buffers, the receiver would be free to switch to the third buffer while the transmitter finishes up its buffer and switches on its own time.

## 11. Declaration of interest and funding

The authors of this paper have no conflicts of interest involving any of the components or technologies used in this project. We have not been paid by anyone to publish this work. This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

## Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, athttps://doi.org/10.1016/j.ohx.2019.e00064.

## References

[1] A.M. Noll, Scanned-display computer graphics, Commun. ACM 14 (3) (1971) 143–150.
[2] Bounce buffers. [Online]. Available: http://www.chudov.com/tmp/LinuxVM/html/understand/node65.html.
[3] P.E. Sabella, N.P. Witt, Double-buffering of pixel data using copy-on-write semantics, Jun. 28 2005, uS Patent 6,911,983.
[4] M. Lavelle, A. Koltzoff, D. Kehlet, Fast frame buffer system architecture for video display system, Feb. 1 2000, uS Patent 6,020,901.
[5] Fast sync. [Online]. Available: https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/13.
[6] J. Batlle, J. Martí, P. Ridao, J. Amat, A new fpga/dsp-based parallel architecture for real-time image processing, Real-Time Imaging 8 (5) (2002) 345–356.
[7] I.S. Uzun, A. Amira, A. Bouridane, Fpga implementations of fast fourier transforms for real-time signal and image processing, IEE Proc.-Vision, Image Signal Processing 152 (3) (2005) 283–296.
[8] J. Hiraiwa, H. Amano, An fpga implementation of reconfigurable real-time vision architecture, 2013 27th International Conference on Advanced Information Networking and Applications Workshops, IEEE, 2013, pp. 150–155.
[9] C. Johnston, K. Gribbon, D. Bailey, Implementing image processing algorithms on fpgas, Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon-04, 2004, pp. 118–123.
[10] P. Greisen, S. Heinzle, M. Gross, A.P. Burg, An fpga-based processing pipeline for high-definition stereo video, EURASIP J. Image Video Processing 2011 (1) (2011) 18.
[11] J. Williams, Simple hdmi frame buffer. [Online]. Available: https://joelw.id.au/FPGA/SimpleHDMIFrameBuffer.
[12] H. Shim, N. Chang, M. Pedram, A compressed frame buffer to reduce display power consumption in mobile systems, Proceedings of the 2004 Asia and South Pacific Design Automation Conference, IEEE Press, 2004, pp. 818–823.
[13] J. Matai, A. Irturk, R. Kastner, Design and implementation of an fpga-based real-time face recognition system, 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, IEEE, 2011, pp. 97–100.
[14] G. Knittel, A pci-compatible fpga-coprocessor for 2d/3d image processing, FCCM, 1996, pp. 136–145.
[15] Digital Camera Project. [Online]. Available: http://www.dejazzer.com/eigenpi/digital_camera/digital_camera.html.
[16] digital camera interface (dcmi) for stm32 mcus application notes, 2017. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/group0/c0/ef/15/38/d1/d6/49/88/DM00373474/files/DM00373474.pdf/jcr:content/translations/en.DM00373474.pdf.
[17] J. Williams, Mendeley data repository for source files. [Online]. Available: https://data.mendeley.com/datasets/cwsbd5cwbf/draft?a=757c3e2c-9ed0-4ae5-a2ea-b9041e96c284.
[18] Is61wv102416 sram. [Online]. Available: http://www.issi.com/WW/pdf/61WV102416ALL.pdf.
[19] ShawnGerber, Sram controller in verilog for altera de1 board, Aug 2012. [Online]. Available: https://www.youtube.com/watch?v=UBSREEqE8hw.
[20] De2115 development board, Mar 2017. [Online]. Available: https://www.altera.com/solutions/partners/partner-profile/terasic-inc-/board/altera-de2-115-development-and-education-board.html.
[21] T. Technologies, Terasic - daughter cards - interface conversion - gpio-hstc card. [Online]. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=67&No=322&PartNo=2#section.