EECS150 Fall 2013 Checkpoint 2: SRAM Arbiter

Prof. Ronald Fearing
GSIs: Austin Buchan, Stephen Twigg
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Due week of Nov. 7.
Revision 1.1, October 25, 2013

1 Introduction

In this checkpoint you will design, implement, and test an SRAM arbiter that enables 2 write and 2 read ports to share the SRAM. This arbiter will have clock-domain-crossing FIFOs at each port to allow each source and sink of data to operate in independent clock domains. This behavior will allow continuous streaming of pixel data to the DVI output while filling, reading, and overwriting portions of an input buffer from the VGA interface.

2 System Overview

Several components have been added to the top-level design to help integrate and test the SRAM arbiter. Make sure you get all of the new files by running a `git pull template master`. These components simulate the asynchronous behavior of VGA input and DVI output. Fig. 1 shows how these compents are connected. First we will discuss the components that have been implemented for you, and then move on to the specification of the arbiter and overlay modules that you will implement.

2.1 SRAM Controller

The SRAM controller handles the physical interface to the off-chip SRAM. It accepts one read or write operation per clock cycle, which is initiated by asserting the `addr_valid` input. A read occurs if `write_mask` is `0b0000`. The data stored in the SRAM at `addr` is available at `data_out` in three clock cycles, indicated by `data_out_valid`. Read operations cannot be stalled. `data_in` is ignored in a read operation.

If `write_mask` has any ones when `addr_valid` is asserted, the selected bytes of `data_in` are written to the SRAM at `addr`. Bytes not indicated by the write mask keep their value in the SRAM. For example, if an address contained the data `0xF1F0BEAD`, a write of `data_in = 0xC0DEAAEF` with `write_mask = 0b1101` would result in `0xC0DEBEEF` being stored at the address. Write operations take two cycles to be submitted to the SRAM. Because the SRAM is synchronous, you can assume that the data will be available at `data_out` after the two cycles.

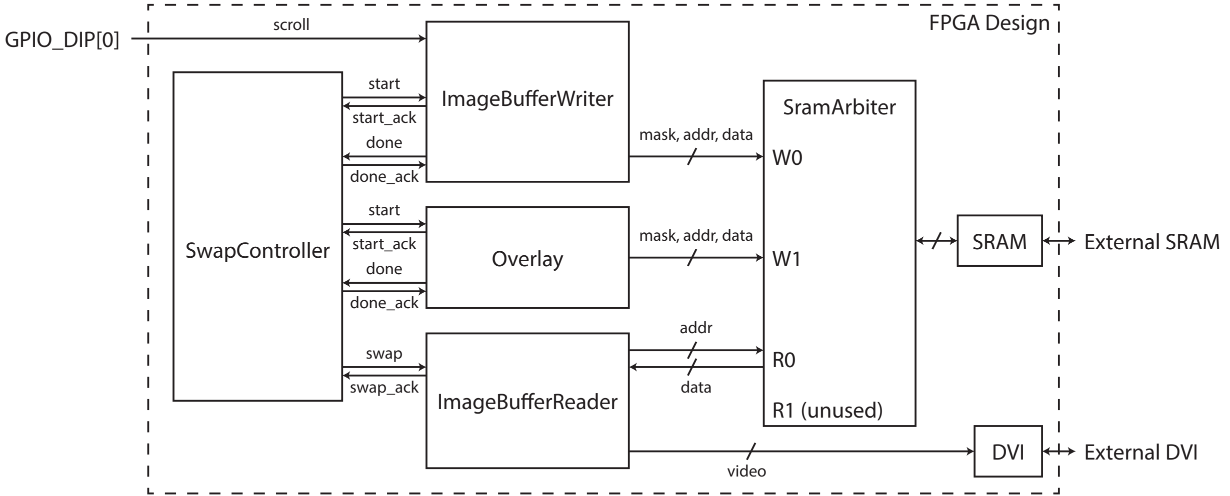

Figure 1: System Overview Diagram

The SRAM has been tested up to 100MHz. This is currently limited by a conservative estimate of off-chip clock skew. Stay tuned for a template repository update that will allow faster speeds via a clock-deskewing module.

2.2 Image Buffer Reader

The image buffer reader performs four main functions:

- Submit continuous address requests to one of the arbiter read ports, starting at the address that specifies the output image frame.
- Convert each pixel byte to an RGB triplet via colorspace conversion.
- Send one RGB value to the DVI per 50MHz clock cycle (when DVI ready).
- Swap to the alternate frame buffer on asychronous input, respecting frame boundaries and acknowledging when the swap has occured.

Primarily, the buffer reader submits read requests to the SRAM to extract the current output image pixel data. This starts at the address of the current frame (either `0x00000` or `0x20000`) and counts up until all pixels have been read. This submission is dependent on the read port being ready to accept a request, and will stall appropriately if ready is not asserted. This module is designed to keep the address and data FIFOs of the read port full since the DVI cannot be stalled for pixel information.

Colorspace conversion allows displaying select colors on a mostly grayscale palette using only one byte per pixel. The conversion outputs an RGB triplet based on a 3-byte by 256-entry lookup table. This table is specified by an input hex file, which can be passed as a module parameter `COLOR_MAP`. By default, it will use the input file `sim/data/colormap.hex`. You can either modify this file manually, or generate it with the `colormap.py` script, also in the `data` directory. As is, the colormap assigns:

- 0x00 → 0x000000 (black)
- 0x01 → 0xFF0000 (red)
- 0x02 → 0x00FF00 (green)
- 0x03 → 0x0000FF (blue)
- For all others, replicate the value 0xGR → 0xGRGRGR (gray)

This conversion is done and sent to the DVI port one pixel at a time. When the fourth byte from the SRAM read is sent to the DVI, the buffer reader will signal for the next read to be available at the data FIFO.

Reads from the current buffer will continue until the `swap` signal is asserted. When this occurs the buffer reader will wait until the last request of the frame is asserted, and then start submitting requests at the other frame address. Once the first request of the new frame is submitted, the module will assert the `swap_ack` output. Following the semantics of 4-cycle signaling, `swap` should be held high until `swap_ack` is asserted. Then `swap_ack` will be deasserted following the deassertion of `swap`.

2.3 Image Buffer Writer

The image buffer writer simulates VGA input by performing the opposite function of the image buffer reader. A test pattern of pixel information is generated and written to the SRAM in four-byte chunks. Unlike the buffer reader, the buffer writer will halt write submissions after a single frame is written. When the write is complete, the buffer writer alerts the swap controller, which in turn initiates a frame swap in the buffer reader. Writing continues (at the alternate buffer) when the `start` input is asserted on the buffer writer.

Currently the image buffer writer produces a repeated horizontal gradient from `0x00` to `0xFF`. Optionally this can be set to move to the left by one pixel per frame by flipping GPIO_DIP[0] on. There is currently an oddity in the test image production pipeline in that there are about 16 off-gray vertical strip artifacts appearing in the output image even though the pixel output has been verified to be a continuous gradient. Don't worry if you notice this. Hopefully we will track this issue down in the near future.

2.4 Overlay

You will implement a component that draws an overlay pattern on top of the image drawn by the buffer writer. This design should be something simple like a small (no more than 16x16 pixels) cross, box, or circle. Eventually this module will help you mark identified features on an input VGA image before sending to the DVI for display. You should also practice using the write mask bits with this module. If you have extra time you could have this module take input from the scroll wheel to move the cursor on screen.

This module will operate basically like the image buffer writer, except that it will only write to a few locations. Decide what your pattern will be, including what writes will be masked, and simply stream those masks, addresses, and data to the W1 port of the arbiter after the buffer writer has finished. If your pattern cannot be simply calculated from a row and column count, arrays initialized with a .hex file using `$readmemh` are a quick way to store arbitrary data that can be

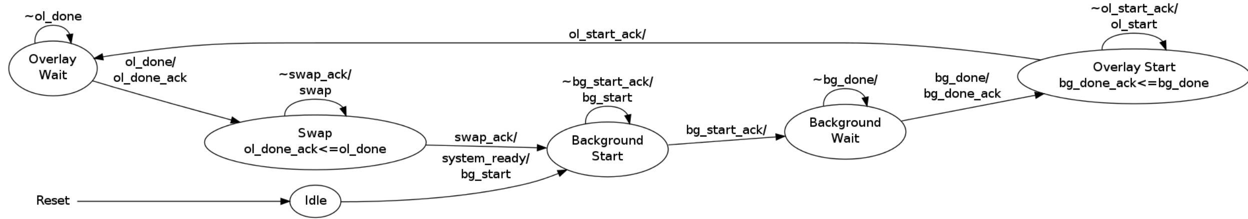

Figure 2: Swap Controller State Diagram

read with an index. See the color mapping code in ImageBufferReader.v for an example of this technique.

2.5 Swap Controller

As referenced above, the swap controller coordinates the image buffer reader and writer. It is designed to be agnostic to the clock frequency of two units; as such it uses 4-cycle asynchronous signaling on all signals. Fig. 2 shows a state transition diagram summarizing this behavior. As given to you, the controller only coordinates the background write. You will need to add the stage for coordinating the overlay write.

3 Arbiter Detail

You will be designing an arbiter that shares the interface to the SRAM controller between two write and two read ports. When implemented correctly, each port should be able to submit requests regardless of the behavior of the other ports. Aditionally, the arbiter should maximize throughput and fairness to each of the ports using a round-robin arbitration scheme.

A skeleton of SRAM_Arbiter has been included in the template code for this checkoff with input and output signals. You will need to implement any additional datapath and state machine elements to accomplish the functionality described here. You should not need to add any additional input/output signals to achieve this, but you may do so for testing purposes.

To accomplish complete isolation across clock domains for each port, all data in and out of the arbiter will pass through asynchronous FIFOs. This way all of the logic for arbitration can run at the SRAM clock rate. Section 3.4 details how to generate efficient asynchronous FIFOs. The FIFO for the write ports (SRAM_WRITE_FIFO) has been designed and instantiated for you, but you must wire it.

3.1 Round-Robin Scheduling

As discussed in lecture, discussion, and homeworks, this arbiter will use round-robin scheduling to service read and write requests. For now we can assume that the SRAM and arbiter logic will be running fast enough to service all incoming requests. This means that each type of request has equal priority. As such, after the logic should services a request, it should select the next available request in the order: W0, W1, R0, R1. For our application it is not critical that the arbiter have a particular behavior after a cycle of no active requests. However, it is good to think about the

implications of different policies if we couldn't make our assumption of having slack in our request versus service clock rates.

3.2 Write Ports

The two write ports of the arbiter accept a mask, address, and data (bundled as `wN_din`) with associated ready/valid signals. Each port also has a clock input, which all signals for that port should be synchronous to. This interface basically exposes the input interface of a FIFO, with some signals renamed. Although we are assuming that these write FIFOs will never be full, the `wN_ready` output should still be connected and behave appropriately.

3.3 Read Ports

Each read port will have an incoming address FIFO, and outgoing data FIFO. These two buses each have associated ready and valid signals. Since there is no write mask associated with a read request, the arbiter logic will have to generate the correct mask for performing a read.

A critical aspect of the read port abitration is putting the SRAM data in the correct port's output FIFO. In all cases the requested data must return to the requesting port in the order it was requested. This means you will need to design logic that maintains the history of read requests that have been sent to the SRAM, and asserts the appropriate `rd_en` when the data is available.

In order to function correctly with the image buffer reader, the read ports must also be able to impose "back pressure" on the address port in the case that the data FIFO becomes full. This function has to be carefully considered so that no read request loses data. The Xilinx FIFOs have an optional `prog_full` output that can be programmed to assert when there are a specified number of slots remaining in the FIFO. You should use this signal to avoid servicing read requests from the port if it cannot be guaranteed that all resulting data can be stored in the data FIFO.

3.4 FIFO Generation

We will be using the Xilinx Coregen tool to generate FIFOs with the desired width, depth, and behavior. A starter Coregen project has been placed in `src/sram/fifo`. You can open this project by running the following in the fifo directory:

```
coregen -p coregen.cgp
```

Fig. 3 shows what Coregen should look like when you open it, with the IP list and Project CORE list highlighted. It also shows where to find the FIFO Generator v9.3 IP.

You should see the SRAM_WRITE_FIFO unit in the project CORE list. Double-click the unit's name to bring up the configuration dialog. Pay careful attention to the configuration used and make sure you understand what each of the design parameters correlate to. If you are unsure of the behavior, check the FIFO Generator documentation on the class website. When done you should click the "Generate" button to compile the unit.

Add units for the SRAM_ADDR_FIFO and SRAM_DATA_FIFO to the project by double clicking the FIFO Generator v9.3 option in the IP list. Set the the configuration of the FIFO based on that for the SRAM_WRITE_FIFO, changing values as necessary for correct functioning of the different FIFOs. When done you should generate each of the units, and close the project, saving if necessary.

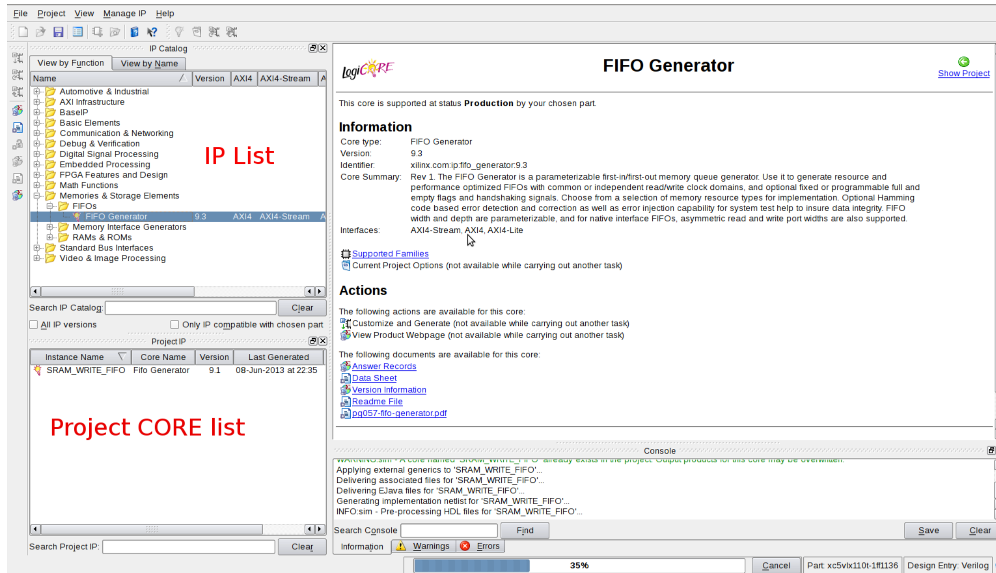

Figure 3: Coregen Project Window

The repository is set to ignore all compile outputs of the Coregen project, since they are large files. All of the information needed to create your FIFOs is in the project files (coregen.cgp and coregen.cgc) and CORE unit files (*.xco). You should maintain these 5 files in version control, and generate the output designs when necessary. Running `make` from the `fifo` directory will generate these designs from the files without the GUI. Alternatively, if you are positive that the design files are in sync, you can copy the contents of the `fifo` directory from one repository to another to avoid rebuilding.

4 Testing and Simulation

Designing effective unit tests will be an important part of accomplishing this checkpoint in a timely manner. While it is possible to simulate the entire FPGA_TOP_ML505 module, it is rarely a good idea to do so. Many signals, such as the DVI I2C initialization bus, SRAM inputs, and reset button are not driven unless you do so explicitly.

A much better practice is to write several smaller unit tests that verify specific, important interactions in your design. As mentioned in Checkpoint 1, the file structure is set up so that you can define as many testbenches as you want, and only run the desired tests using symbolic links to .do files in the `sim/tests` directory. To complete this checkpoint, you must show at least two test benches with the behavior listed in below in the goals. Maintaining earlier testbenches is a great way to ensure your design does not break previously implemented features when moving on to new ones.

Since you will be dealing with memories in this checkpoint, the `$readmemh` function will be your friend. This can be used to initialize the contents of large memories (such as a simulated SRAM), or to have a list of test vectors (say masks, address, and data) that you want to step through in simulation. The `colormap.py` script is an an example of how you can make simple but patterns for hex files. Note that if you want your initialized memories to synthesise, XST is very finicky about

hex files. Include only the hex values (no comments, or separating underscores, which is legal for simulation).

You will not need to simulate the full behavior of the SRAM (like correct data storage) to test correct functioning of the arbiter. Consider simulating data that makes it possible to tell which address a read or write came from when placing data in the read FIFOs. This can be done by only using a subset of the address space for each port, and setting the simulated SRAM data_in to the address that was used two cycles ago.

With the first point of not simulating the whole design in mind, it is possible to change the behavior of a design between simulation and synthesis. The `MODELSIM` macro will only be defined when simulating. Using `` `ifdef MODELSIM ... `else ... `endif `` blocks, you can add code to your design that will exercise unconnected inputs, or reduce the number of memory operations done to shorten a test. Be careful that you don't inadvertently change the behavior of the synthesized module.

5 Checkpoint Goals

All code should be pushed to GitHub. By the end of this checkpoint, you should be able to show:

- Completed SramArbiter.v, Overlay.v, and modified SwapController.v.
- Testbenches for SramArbiter that exercise the following behaviors. You must show at least two different testbench files that can be selectively run with symbolic links in the `sim/test` directory. These testbenches should print out which port is being served in each clock cycle (or "NONE").
  - Different rate clocks for the arbiter logic and port logic.
  - At least two complete sequences of all 4 ports being served in order.
  - Stop requesting to some of the ports, and show that the arbiter service order changes appropriately.
  - Stop reading from the data FIFO of a read port while read requests continue, and show that requests stop being serviced and the address FIFO is stalled when the data FIFO becomes full.
  - Allow all request FIFOs to empty, showing that the arbiter submits no writes, and inserts no read data in either data FIFO.
- The FPGA programmed with a design that demonstrates drawing a background from one write port, overlay from another port, and displaying the image over the DVI interface after a frame pointer swap. This display should change over time to show working repeated pointer swaps.