Fall 2013 CS150 Final Project Report: Difference-of-Gaussians Filter

James Dunn and Amit Patankar

12 December 2013

Professor: Ronald Fearing

GSIs: Austin Buchan, Stephen Twigg

Abstract

For our final project, we have implemented a difference-of-Gaussians image filter using a Xilinx FPGA platform. Difference-of-Gaussians is an algorithm that enhances the features of a given data input. In the context of our project, which receives an image as input, the difference-of-Gaussians filter highlights the image's key points—that is, the locations which best define the image. Selecting these points is necessary for many types of image processing. The motivation for our Gaussian filtration is the SIFT (Scale-Invariant Feature Tracking) system, in which key points are defined and maintained given a moving image. Though the SIFT implementation itself proved too difficult to include in the final checkpoint, we still learned a great deal about its concepts and mechanics. This report includes a both a technical documentation of our Gaussian filtration system, and a comprehensive account of the ideas and work that went into its construction.

2. Overview

a. Design

The following block diagram represents a higher-level organization of the project as a whole. We designed this structure as part of checkpoint 3, and—other than removing the overlay module's connections—it has remained mostly unchanged.

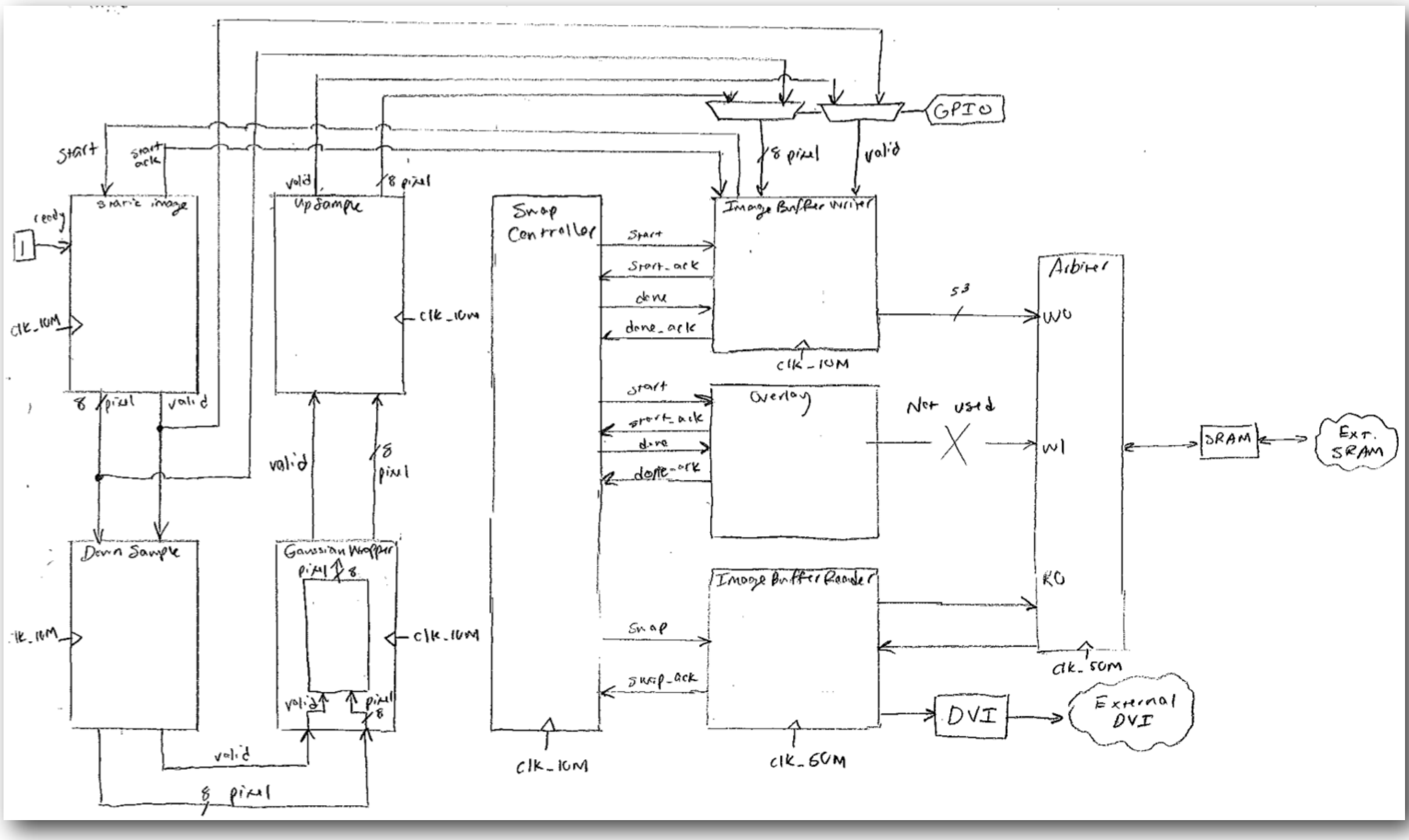

Fig. 1: Block Diagram of TOP file.

b. Brief Description of Major Sub-Modules

DownSample.v

The DownSample module takes as input a stream of 8-bit pixels, and a valid signal. The output is also a stream of pixels, and a valid signal. The input stream consists of the pixels comprising a full 800x600-pixel image (480,000 pixels). The down-sampler modulates its output valid signal such that only 120,000 pixels on the output stream are valid for a frame. This yields a 400x300-pixel output image.

UpSample.v

UpSample is essentially the dual of DownSample; it takes as input a valid signal, and a 120,000-pixel stream (400x300-pixel image) for a frame. It then outputs a scaled-up version of this image, yielding a 480,000-pixel output stream, and a corresponding output valid signal.

GaussianWrapper.v

Our Gaussian wrapper houses the Gaussian filter module. On the input side, the wrapper's job is to set up the filter with a properly-padded input pixel stream. On the output side, the wrapper separates output of a single filter from the output of a difference-of-Gaussians. It also asserts and de-asserts a valid out signal as necessary.

GaussianFilter.v

The heart of the project. Performs the difference-of-Gaussians algorithm as described in lecture on a properly-padded input stream of pixels. Outputs an 8-bit pixel stream of that will comprise the Gaussian-filtered image. Pixels coming out of GaussianFilter are always assumed to be valid; the wrapper handles the valid out assertion.

SramArbiter.v

This module is tasked with arbitrating read and writes to the external SRAM. This arbitration is necessary because there are two modules—ImageBufferReader and ImageBufferWriter—that both utilize the SRAM. This module utilizes a round-robin arbitration scheme in order to achieve maximum fairness and throughput.

SwapController.v

This module coordinates the asynchronously-timed ImageBufferReader and ImageBufferWriter. Specifically, it coordinates when each of these modules write to a frame. It is based upon an FSM given to us in the project spec.

3. Detailed Description

a. Datapath

DownSample.v

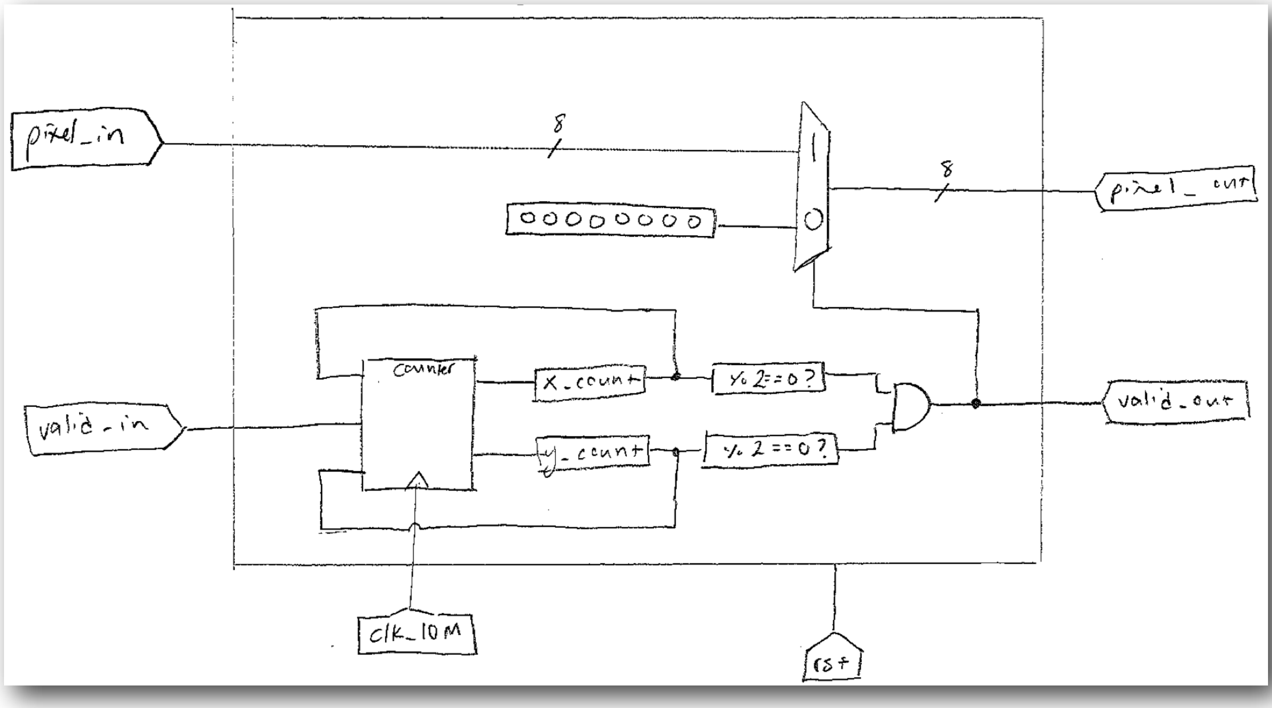

Fig. 2: Block Diagram of DownSample.v.

The datapath for our down-sampler is relatively simple. We are using the valid_out signal to "throw out" pixels in order to create a scaled-down image. Thus, we have a direct path from input to output pixel. The mux is actually not completely necessary; it simply helped for debugging the down-sampler before connecting to the up-sampler. When connected to the up-sampler, it could be a direct connection. Since the ImageBufferWriter expects a full 480,000 pixels, we outputted a dummy pixel even when valid_out was low. Otherwise, there was a strange effect where the down-scaled image was doubled horizontally but not vertically. The only datapath feature of interest here is the counter, where we increment for x and y values on a valid input, and a mod operation and AND gate where we use these counters to determine if the output should be valid. These pixel_out and valid_out signals will be connected directly to the Gaussian wrapper.

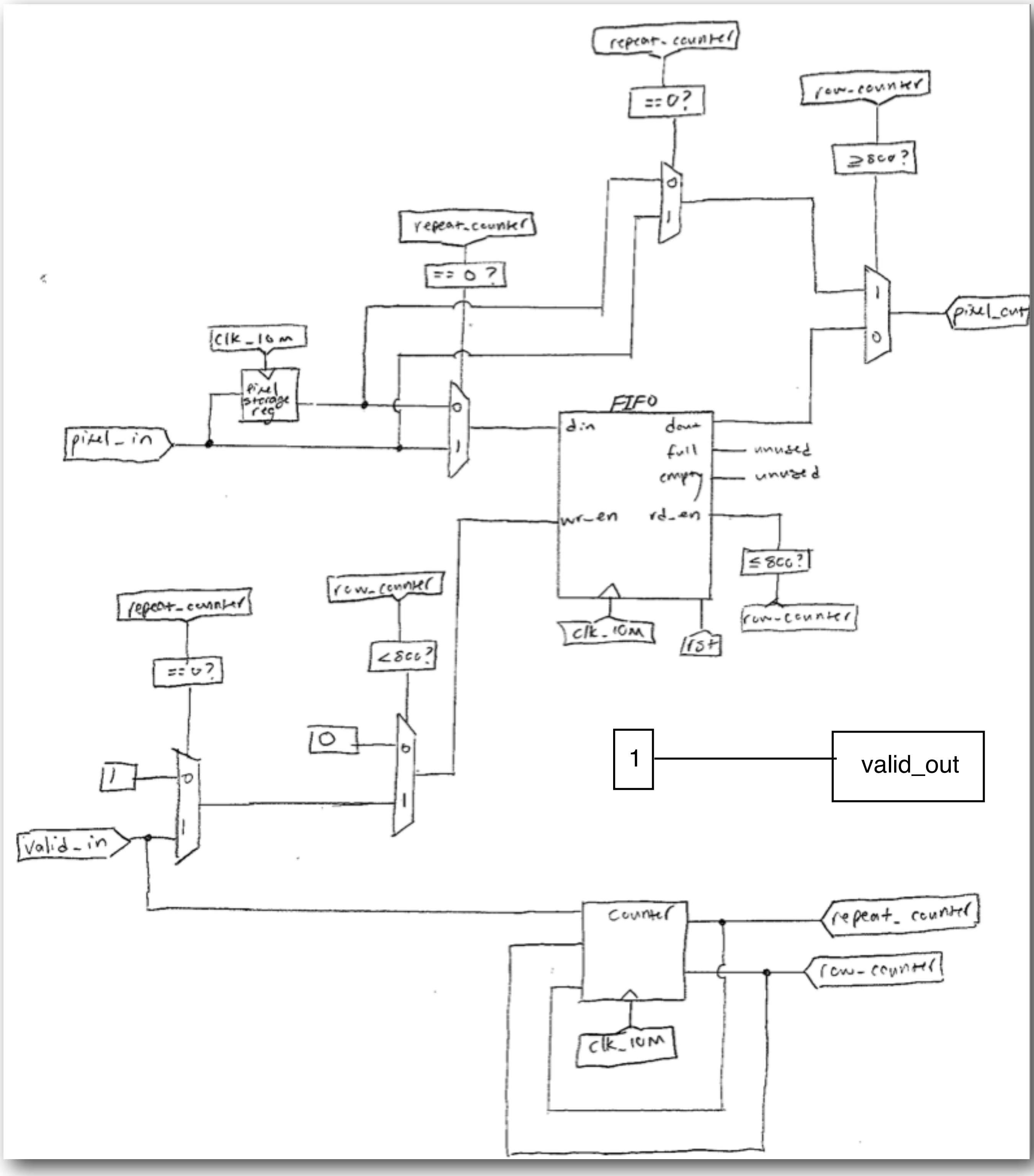

Fig. 3: Block Diagram of UpSample.v.

The block diagram for our up-sampler is a bit more complicated. This is mainly due to the fact that—unlike down-sampling, in which you can simply throw out pixels—up-sampling requires storage of pixels. This necessitates wiring up a FIFO and pixel-storage register. Since we utilized a counter instead of an FSM (more on this in the control section), a counting module also needs to be wired. The counting module and pixel-storage register are both wired into a series of muxes. These muxes represent ternary assign statements for wires, which we cascaded like so:

```
assign pixel_out = (row_counter >= 800) ? fifo_out : normal_out;
assign normal_out = (repeat_counter == 0) ? pixel_in : pixel_storage_reg;
```

The UpSampler's pixel_out and valid_out are connected via a switched GPIO mux to the ImageBufferWriter, where they determine if and how a pixel is written to the SRAM.

GaussianWrapper.v

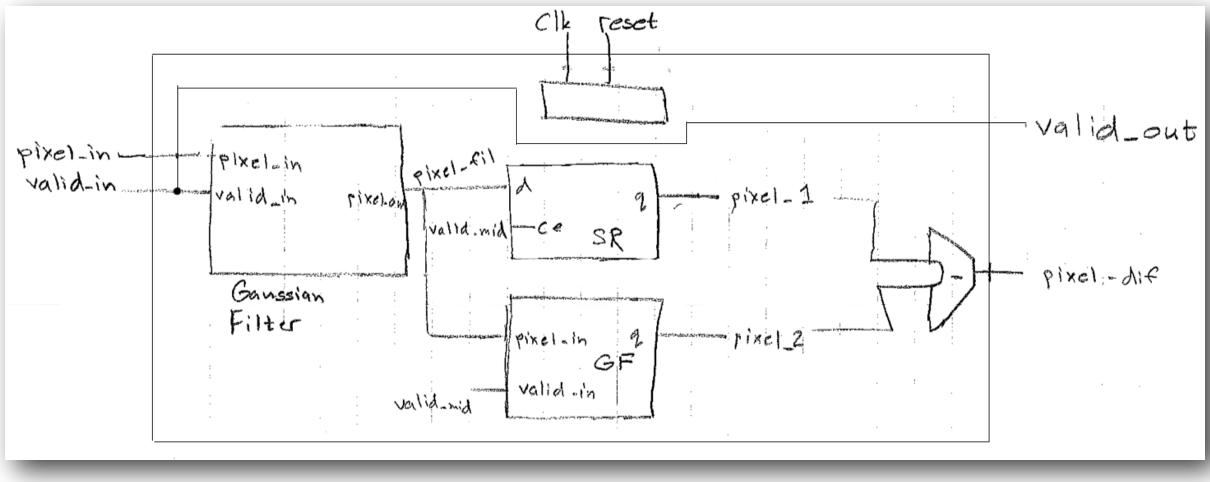

Fig. 4: Block Diagram of GaussianWrapper.v.

This module is given a valid signal and 8-bit pixel signal. There are two first-word fall-through FIFOs on both the exiting and entering ends of the module. The input valid signal determines the valid signals for the shift register and first filter, as well as later filters. The shift register has a depth of 802, also the delay of the second Gaussian filter. Because of this, both of the output pixels of the filter and the shift register are the same pixel. They are both subtracted and put into a FIFO. That FIFO is the output interface for the module. This approach of being liberal with the valid signal is the reason we have a slight shift of two pixels to the right and one pixel down for the image filtering (an 802 cycle delay).

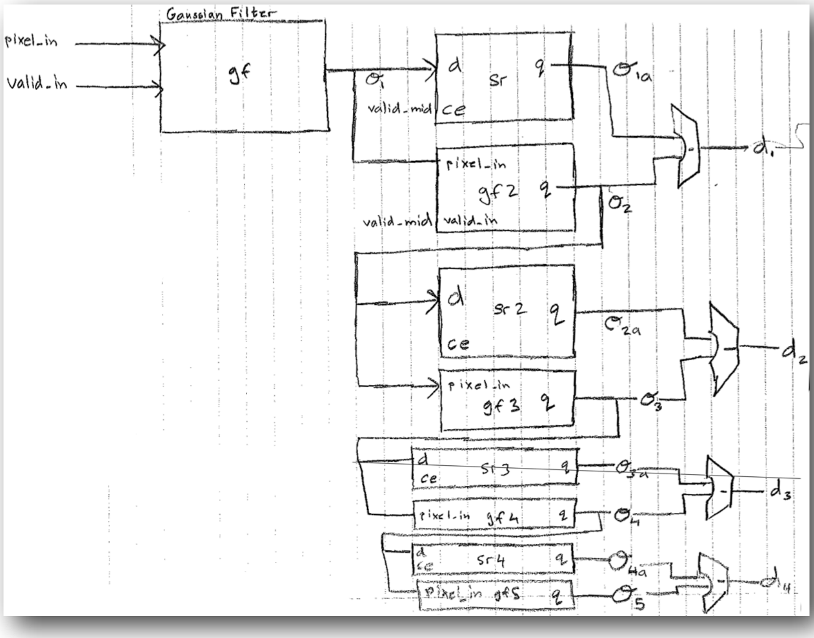

Fig. 5: Block Diagram of extra-credit GaussianWrapper.v, with multiple DoG blocks.

GaussianFilter.v

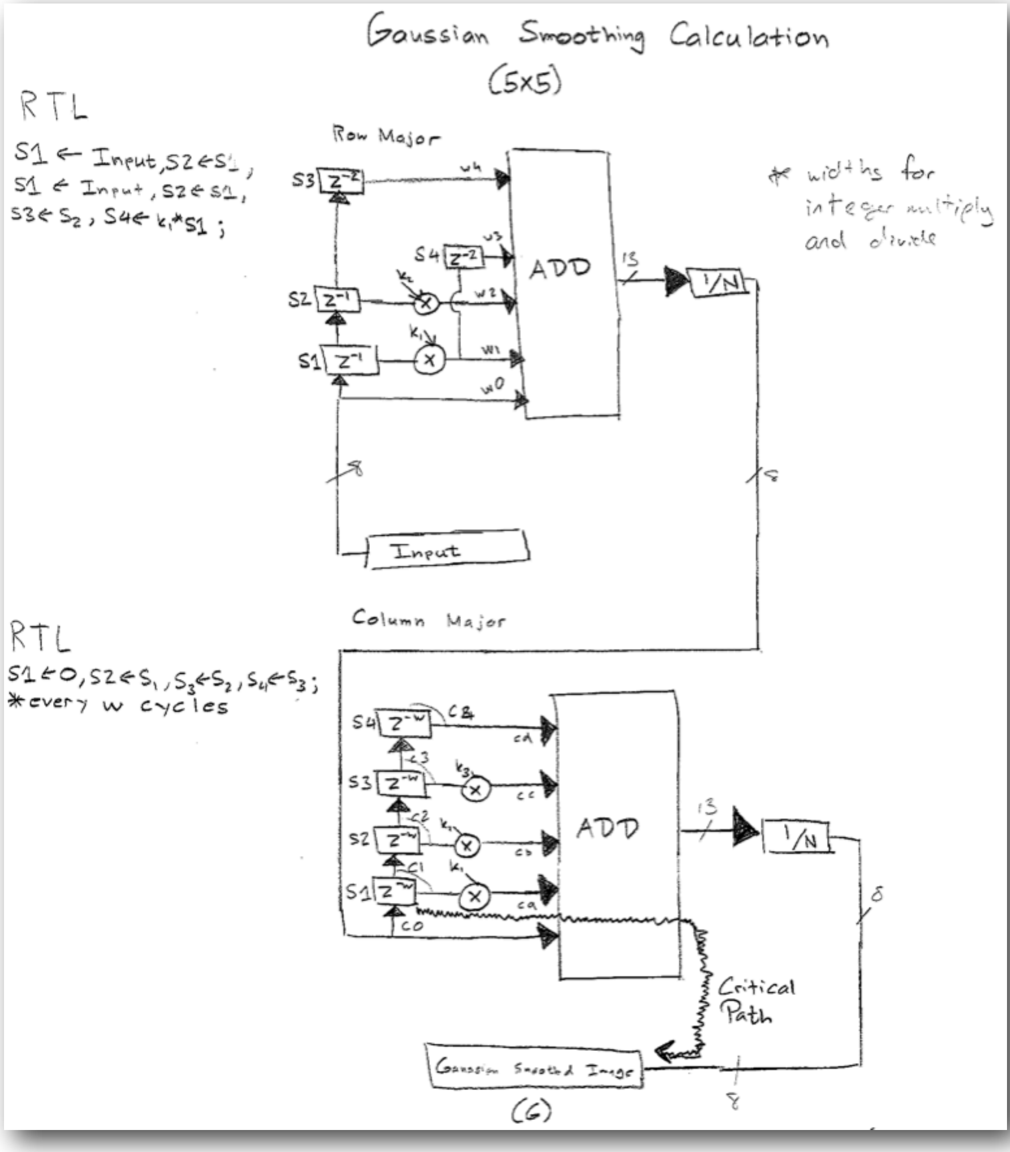

Fig. 6: Block Diagram of GaussianFilter.v.

This is the standard 5x5 block used to compute a filtered value for one pixel, based upon the 25-pixel window. The first section is row major. It stores the past 5 pixels in a pipeline and scales them based on specified constants. It sums them, then divides by 256. Each input is an 8-bit pixel; when they are summed and scaled, they are converted to a 16-bit width, of which the top 8 are sampled. This output is then fed into a column major pipeline, which stores 8 bits from the past 5 filtered pixels, each 800 pixels previous. These are once again scaled to 16 bits, sampled and outputted. The delay is 2w+2 = 2(400) + 2 = 802.

b. Control

i. Addressing Issues

We set up our modules such that there were really no addressing issues. Our down-sampler would invalidate every other row and column handed to the Gaussian wrapper. Because the gaussian filter ran at the same clock speed, it too would invalidate every other row and column handed to the up-sampler. Finally, since our up-sampler also ran at the same clock speed, it could rely on the fact that every other row and column had an invalid pixel to do its doubling during that cycle, effectively replacing the "invalid-pixel cycles" with doubled pixels. This way, nothing got out of sync. We did have a few addressing issues at first, because we were using a FIFO in the up-sampler that was not first-word fall-through. This introduced a one-cycle delay that manifested itself in every other row being skewed to the right by one pixel. Once we re-did the FIFO, the issue went away.

ii. Control Design

<u>DownSample.v</u>
We utilized counters for control in this module. Our down-sampler has a single input control, valid_in, and a single output control, valid_out. Valid out is controlled by the counters, which are incremented as follows: upon assertion of valid_in, an x counter is incremented. When this x counter reaches 800, a y counter is incremented and the x counter is reset to zero. When the y counter reaches 600, it is reset to zero. A mod operation allows only even pixels to trigger a valid_out. Thus, modulating the valid_out in this way scales the image down by a factor of two in each dimension.

<u>UpSample.v</u>
We also utilized counters for control in this module. Since our up-sampler contains a FIFO, more control lines are necessary than in the down-sampler. Two counters are utilized: a "repeat counter" and a "row counter". The repeat counter pertains to the horizontal doubling of pixels; it goes back and forth between 0 and 1 to determine whether the input pixel or a pixel storage register should be fed as the output pixel, and stored in the FIFO. The "row counter" pertains to the vertical doubling of pixels. It determines whether an entire line should be written into the FIFO, or read from the FIFO. The row counter allows the FIFO to be written for 800 pixels, then halts the writing and enables the reading of the FIFO for another 800 pixels.

<u>GaussianWrapper.v</u>
All control for this module is based on the valid_in signal.

SramArbiter.v

The SRAM Arbiter is a fairly simple five-state Moore machine that allocates the arbitration of the SRAM amongst two read ports and two write ports. Typically, arbiters are done using Mealy machines to keep the arbitration from being preferential to any one port. We took a chance that our state machine would be preferential to the W0 port and based its output control signals on state alone. There is little risk we run by doing so, because we assume the input valid signals are dependent upon the arbiter servicing the port and not some external application. So, if the machine's current state is at a port, and no ports are ready, it goes back to the IDLE state. There is the case when sequential valid signals will not be fair. For example, if W0 and W1 are ready, and in 10 cycles, W0, W1, and R0 are ready at the same time, R0 will not be serviced first. Considering the arbiter only had 5 states, it gets the job done, and is efficient.

SwapController.v

For the swap controller, we designed our control based exactly on the FSM diagram provided in the checkpoint 2 spec. This diagram can be found in the following section.

iii. State Diagrams, Timing Diagrams

DownSample.v

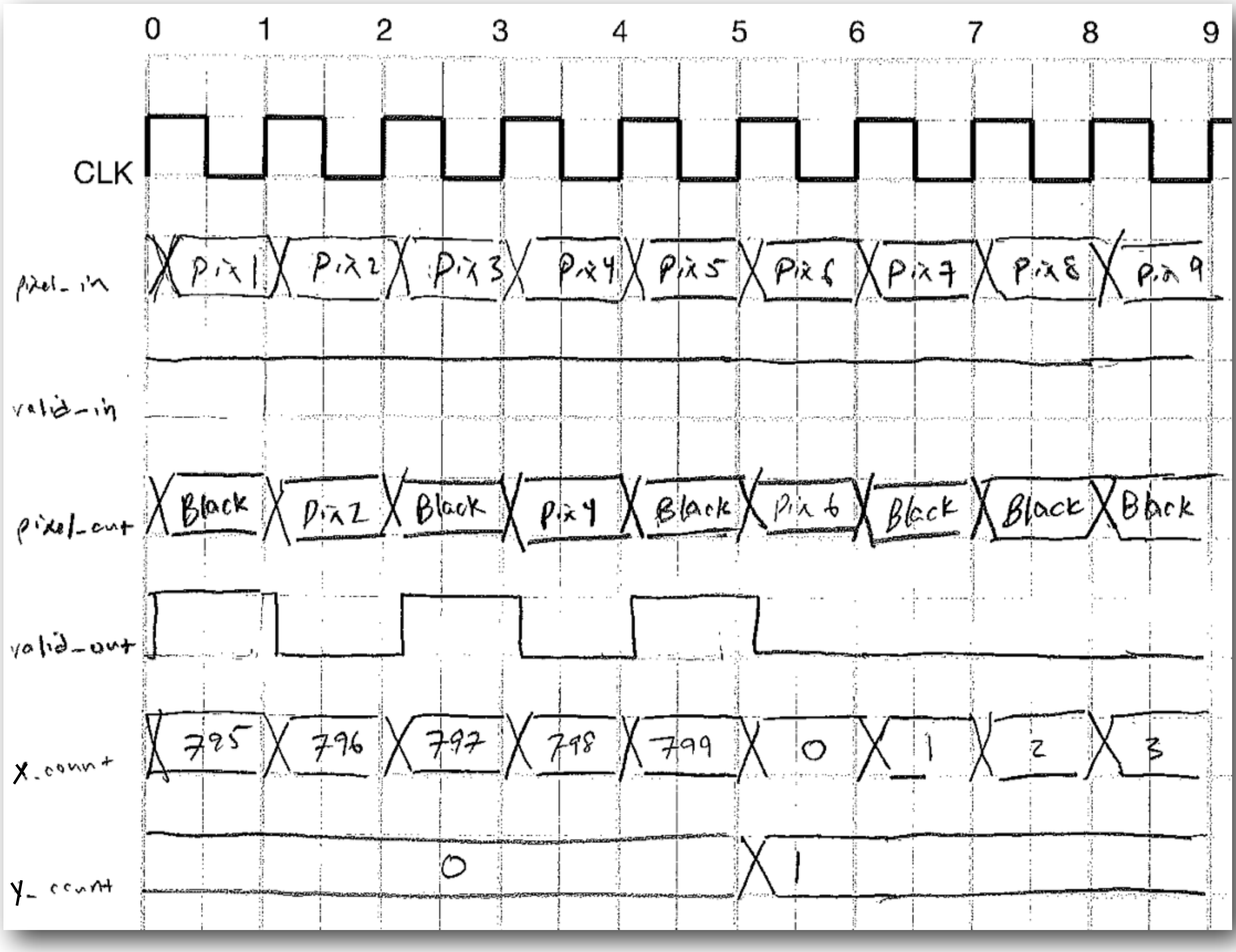

Fig. 7: Timing Diagram of DownSample.v.

UpSample.v

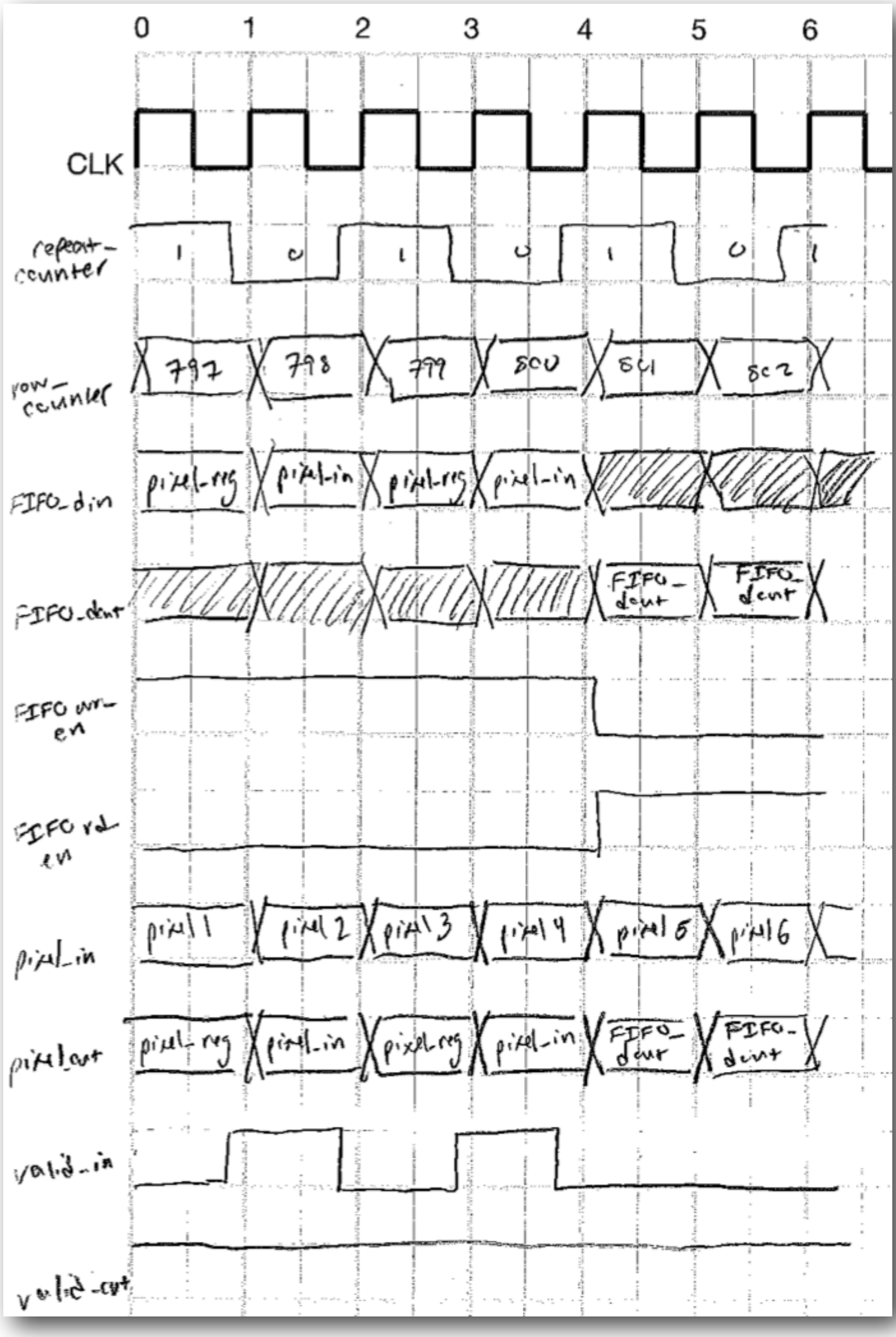

Fig. 8: Timing Diagram of UpSample.v.

GaussianWrapper.v

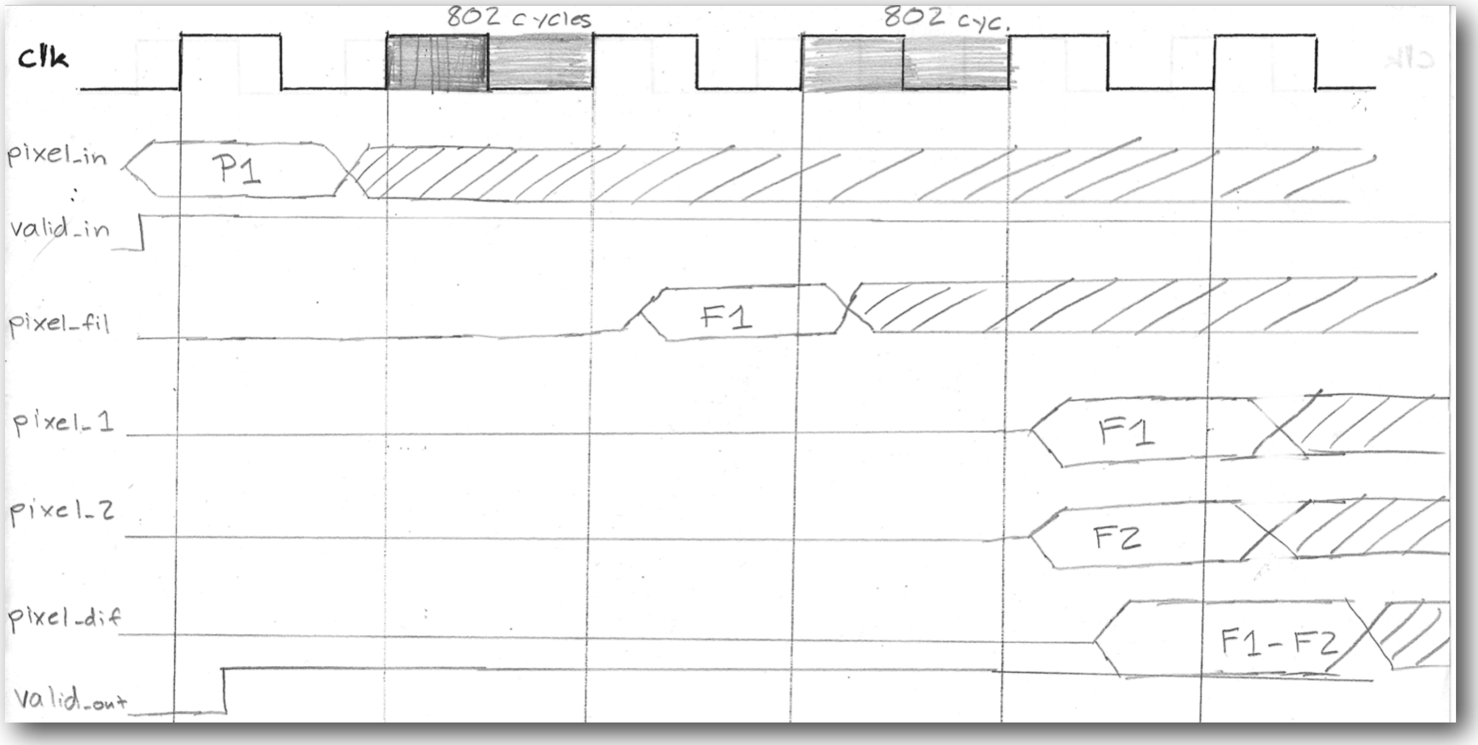

Fig. 9: Timing Diagram of GaussianWrapper.v

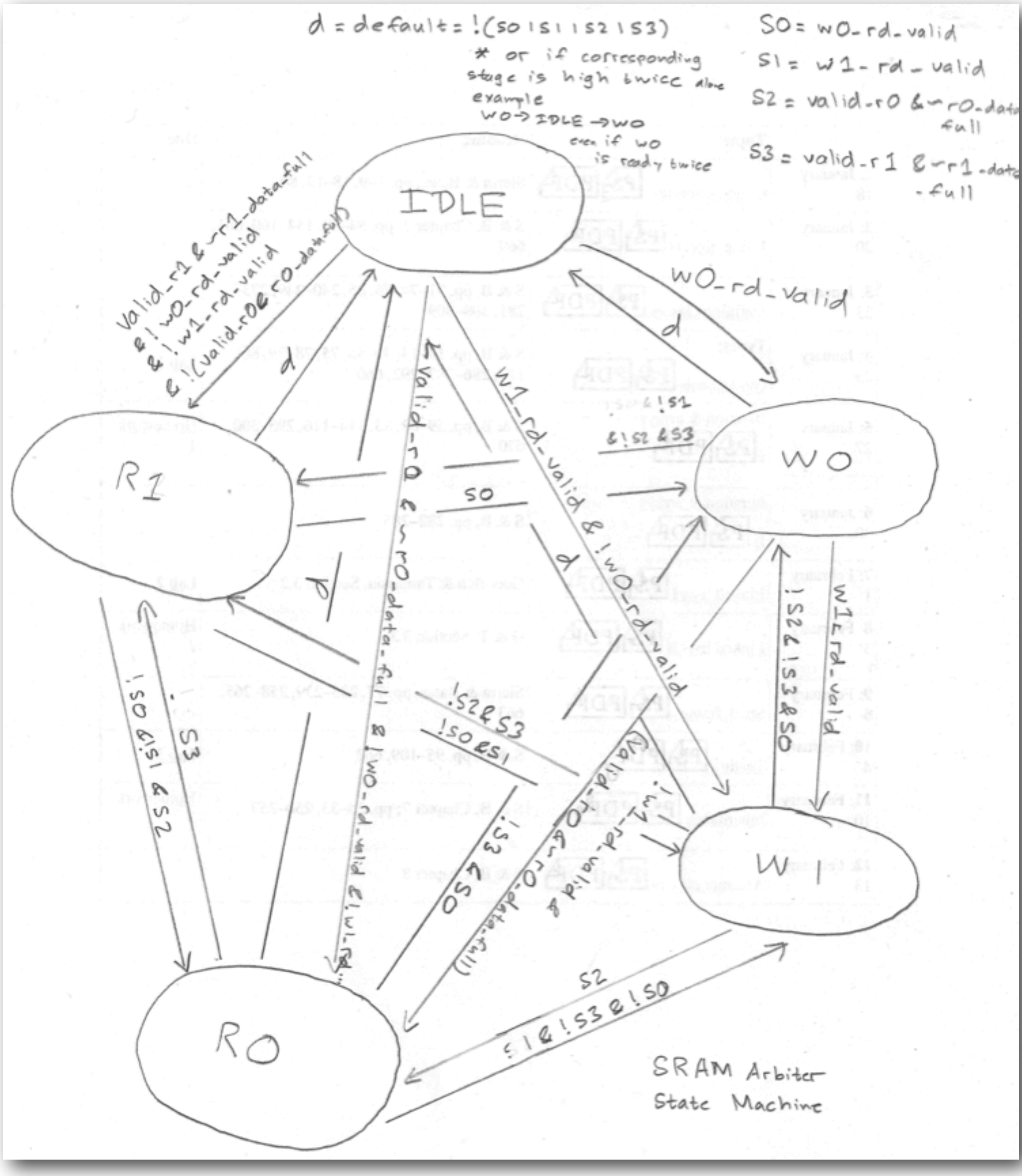
d = default = !(S0 | S1 | S2 | S3)
* or if corresponding stage is high twice alone
example
W0 → IDLE → W0
even if W0 is ready twice
S0 = w0_rd_valid
S1 = w1_rd_valid
S2 = valid_r0 & ~r0_data full
S3 = valid_r1 & ~r1_data_full
IDLE
R1
W0
W1
R0
valid_r1 & ~r1_data_full & !w0_rd_valid & !w1_rd_valid & !(valid_r0 & ~r0_data_full)
d
w0_rd_valid
d
valid_r0 & ~r0_data_full & !w0_rd_valid & !w1_rd...
w1_rd_valid & !w0_rd_valid
& !S1 & !S2 & S3
S0
d
d
!S2 & S3 & S0
w1_rd_valid
!S2 & S3
!S0 & S1
!S3 & S0
(valid_r0 & ~r0_data_full) & !w1_rd_valid
!S0 & S1 & S2
S3
S2
S1 & !S3 & !S0
SRAM Arbiter State Machine

Fig. 10: Finite State Machine of SramArbiter.v

SwapController.v

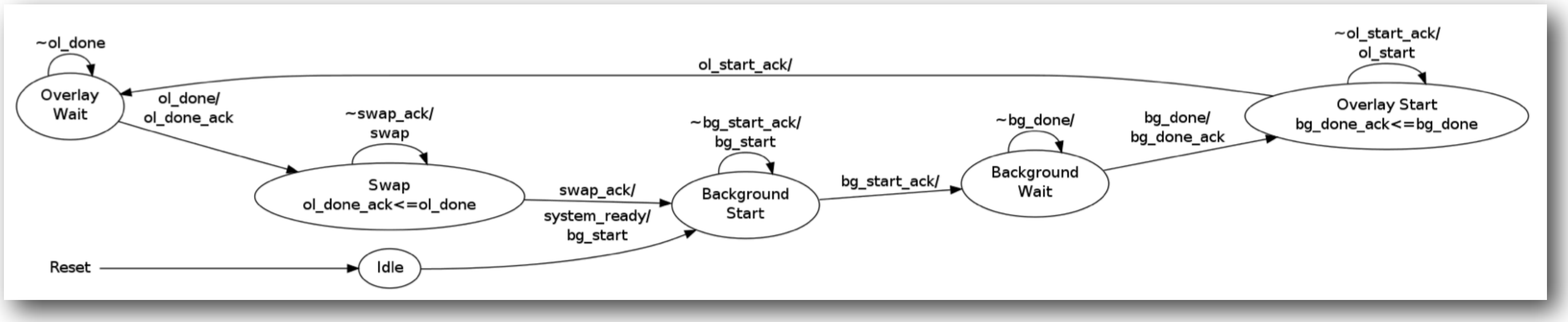

Fig. 11: Finite State Machine of SwapController.v

c. Design Decisions and Tradeoffs

i. As described in the control section, we used the same clock domain for all the filtering stages, including down-sampler, Gaussian filter, and up-sampler. We decided not to do any kind of clock doubling or halving to down or up-sample; we thought this might introduce more headaches with things like clock skew. Instead, we decided to rely on the invalid-pixel cycles to do our doubling, and it worked out.

We also decided that wrapping the Gaussian filter in a module that took care of the padding and valid_out assertion was better organizationally, since these tasks are beyond the duties of the filter itself and should be abstracted away.

We did not have to worry about Mealy vs. Moore beyond what was described in the SRAM arbiter section. Our modules functioned properly with a Moore machine, even though there was some risk with this implementation.

ii. We encountered a few problems over the course of the project. The most significant were in the checkpoint 4 phase. It took us a while to get the down and up-sampler working properly together; as explained in the control section, the fact that we used a counter produced some tricky off-by-one errors that caused rows to be skewed by a single pixel. At this point, we weighed the option of re-writing the modules with an FSM as a controller; such a design probably would have made de-bugging easier. However, since we already had a mostly functioning down and up-sampler, we decided to continue with the current design. Our strategy to debug this error was to first consider what components of the module could possibly introduce an off-by-one error. We thoroughly stepped through our counters, and determined that they were functioning. After doing this, we focused on the FIFO, which also contributes to delay when not configured properly. Sure enough, the FIFO was not created as a fall-through, and this was the culprit.

In completing the final checkpoint, we also utilized a testbench to verify the functionality of our design. The module DoGTest.v test simulated a full 400x300-pixel image that entered the GaussianWrapper module. The idea was to test if the corresponding pixel delay of 802 functions correctly with the delay from the Gaussian filter. It also measured the behavior of our valid signals—specifically, if they were toggled at the end of each row and frame if the filtered signal was reset.

4. Design Metrics

a. Critical Timing Path and Maximum Clock Rate

We have marked the critical path on Fig. 6. It is located in the Gaussian filter block, as this location has the most operations between registers. The path travels through a multiplication block, addition block, and division block. There are also clock-to-q and setup times to factor in. So the maximum clock rate becomes:

$T_{mul} + T_{add} + T_{div} + T_{clk\text{-}to\text{-}q} + T_{setup} = T_{total}$
$\rightarrow\ f_{max} = 1 \,/\, (T_{mul} + T_{add} + T_{div} + T_{clk\text{-}to\text{-}q} + T_{setup}\ )$

b. Usage Statistics

```
--------GaussianWrapper.v-------
Flip-flops:               448
Slice LUTs:               1226
BlockRAM & FIFO:          4
BlockRAM Only:            0
FIFO Only:                4
DSP48Es:                  12

--------GaussianFilter.v--------
Flip-flops:               168
Slice LUTs:               498
BlockRAM & FIFO:          0
BlockRAM Only:            0
FIFO Only:                0
DSP48Es:                  6

--------FPGA_TOP_ML505.v--------
Flip-flops:               1708
Slice LUTs:               4277
BlockRAM & FIFO:          21
BlockRAM Only:            15
FIFO Only:                6
DSP48Es:                  12

-----------UpSample.v-----------
Flip-flops:               36
Slice LUTs:               45
BlockRAM & FIFO:          2
BlockRAM Only:            0
FIFO Only:                2
DSP48Es:                  0

----------DownSample.v----------
Flip-flops:               21
Slice LUTs:               52
BlockRAM & FIFO:          0
BlockRAM Only:            0
FIFO Only:                0
DSP48Es:                  0
```

i. The limiting resources in our GaussianFilter.v module were the Slice LUTs. We utilized a total of 498 for two, and 69,120 were available. Thus, we could add a total of 277 additional Gaussian filter blocks.

c. Division of Labor

We split up the project as follows:

For checkpoint 2, Amit worked on the Arbiter and Swap Controller, and James did the overlay. For checkpoint 3, James drew up the top-level block diagram, and Amit did those for the sub-modules. For checkpoint 4, Amit did the Gaussian filter and wrapper, and James did the down-sampler, up-sampler, and top-level wiring. We made sure to explain our work to one another, so that we both had a holistic understanding of the project.

In all, we estimate to have spent between us: 10 hours designing (that is, discussing, planning, drawing out control and data path designs); 20 hours constructing (coding); and 30 hours de-bugging (re-coding, re-making, pushing to the FPGA). As professor Fearing warned, the de-bugging process took the longest.

5. Conclusion
a. Summary of Main Features

- FPGA is not dependent on external VGA clock, and can function independently.

- Has the ability to take an input image and apply the difference-of-Gaussians filter as described in lecture. This image should be placed in the `hardware/src/img_mem/src` directory.

- DIP switches may be utilized to switch between levels of filtering as follows:
  - Switch 0: Alternate between VGA and static image inputs.
  - Switch 1: Shift the VGA input.
  - Switch 2: Display the static image processed with a single Gaussian filter.
  - Switch 3: Display the static image processed with two Gaussian filters.
  - Switch 4: Display the difference of the two Gaussian-filtered images.

b. What We Would Have Done Differently

I don't think we anticipated that the de-bug process would take as long as it did. Even though we were able to get everything working by the final checkpoint deadline, leaving more time for de-bugging would have made things less stressful.

In terms of the modules, there are a few things we could have changed that might have made the project easier. The arbiter and swap controller left less room for design freedom; there was a pretty clear method to implement those, so we wouldn't have done anything differently there. Similarly, our Gaussian filtration system with wrapper implementation worked well and was easy to debug, so we would not have changed it. Our down and up-sampler, however, could have been made easier to debug if we had used an explicit finite state machine for control, rather than alternating counters. Though our off-by-one error ended up being due to a non-fall-through FIFO, it would have been easier to rule out counter errors had we gone with an FSM.

c. What We Learned

We think that having learned to implement a Gaussian filtration system will prove very useful in our future; it is a fundamental algorithm in signal processing, and nearly any work in the EECS field involves some kind of signal processing. Additionally, seeing a digital design project through from plan to implementation gave us a great amount of engineering experience: learning to debug; learning to leave time to debug; learning to take a break and come back to a problem with a fresh mind. As Professor Fearing said, this kind of experience is universally applicable to any type of problem solving, and we think it will be immensely valuable.