Convolve User App

Summary of Implementation:

The implementation of the Convolve User App was broken into two main parts – the sliding window buffer and the user app.

Sliding Window Buffer:

- The sliding window buffer was implemented similarly to a FIFO, as recommended in the lectures. It accepts C_KERNEL_SIZE (128) elements before outputting a full signal. When the buffer is not full, it asserts an empty signal.
- The buffer keeps track of how many elements are stored using a counter. The counter increments whenever it is written to and decrements when it is read from. However, if it is both being read from and written to at the same time, then the count remains the same.
- As recommended in the lecture, the full signal is AND'ed with the inverted rd_en signal, allowing it to not be "full" even with C_KERNEL_SIZE elements, as long as it is being written from.
- The values of the buffer are held in shift registers that are the width of the signal. They are enabled when a write is asserted.

User App:

- Two sliding window buffers were instantiated, one for the signal and one for the kernel. The kernel buffer's output data is passed through a for-generate to reverse its bits to match the definition of convolution. Both arrays were passed through a provided VECTORIZE function to allow them to be passed into the pipeline.

- A provided multiply-add tree was instantiated as the pipeline. The input widths were set to C_KERNEL_WIDTH and C_SIGNAL_WIDTH. The number of inputs was set to be C_KERNEL_SIZE. The output was sent to a combinational logic block to handle clipping.
- The clipping was implemented by checking if the pipeline's output exceeded the maximum allowed by the SIGNAL/KERNEL widths, calculated as a constant by 2**C_KERNEL_WIDTH -1.
    - For a kernel/signal width of 16, this gave us a maximum value of 65,535.
    - If the pipeline output was greater than or equal to this value, then the output would be set to all 1's.
    - If the pipeline output was less than the max value, then the output would be truncated to the lower C_SIGNAL_WIDTH bits.
    - The clipped pipeline data is then sent to ram 1 to be written.
- A provided memory map was instantiated. New connections were made for it to be connected to the kernel buffer, including its write data, write enable/load, and full/ready. The connections that were no longer relevant from dram_test were removed.
- Many control signals were implemented for the entire user app.
    - Ram 0 rd_en: asserted when buffer is not full and the data from ram 0 is valid.
    - Signal Buffer wr_en: equivalent to Ram 0 rd_en.
    - Signal Buffer rd_en: asserted when pipeline is enabled (not stalled) and signal buffer is not empty.
    - Pipeline enable: Tied to the ram1 wr_rdy signal. Allows pipeline to continue when ram 1 is ready for new data. Stalls if ram 1 is not ready.
    - Pipeline valid in signal: equivalent to signal buffer rd_en.
    - Pipeline valid out signal: Sent to ram 1 wr_valid to enable writes of valid data.

- Ram 0 and Ram 1 rd/wr sizes: set to 2*(C_KERNEL_SIZE-1) + unpadded_size and C_KERNEL_SIZE – 1 + unpadded_size, respectively, as covered in the lecture.
- Ram 0/1 Starting addresses: set both to 0.

Problems Encountered

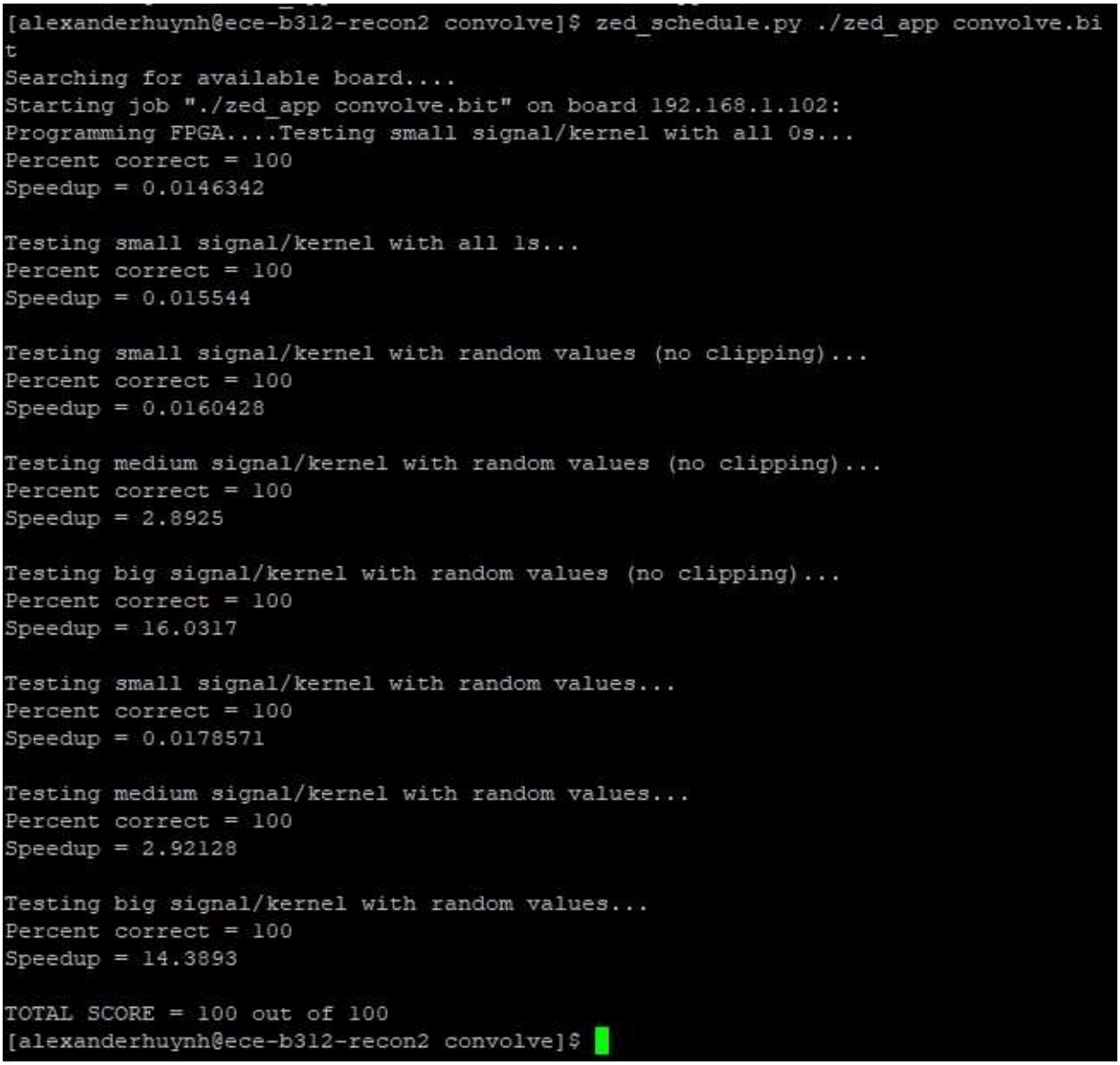
Non-thorough testbench of the sliding window buffer: The testbench that I wrote for the sliding window buffer did not expose some of the wrong behavior that would later cause problems for me in user_app. This was because I did not check what happens when the input write data changes after an empty. I was unaware that I had accidentally made the input of the shift registers the bottom-most element of my array, meaning that rd_data(0) was not latched in, causing it to change.

- I solved this by using the provided wrapper_tb, which exposed this error.

Vectorized rd_data of sliding window buffer before outputting: This is was not an issue per-se but I think that I made a mistake because of the additional DEVECTORIZE step I had to do in user_app.

- Dr. Stitt recommended that I leave rd_data as type window. Changing this might have solved some of my errors.

Screenshot of ZedBoard Output



DMA Read

Summary of implementation

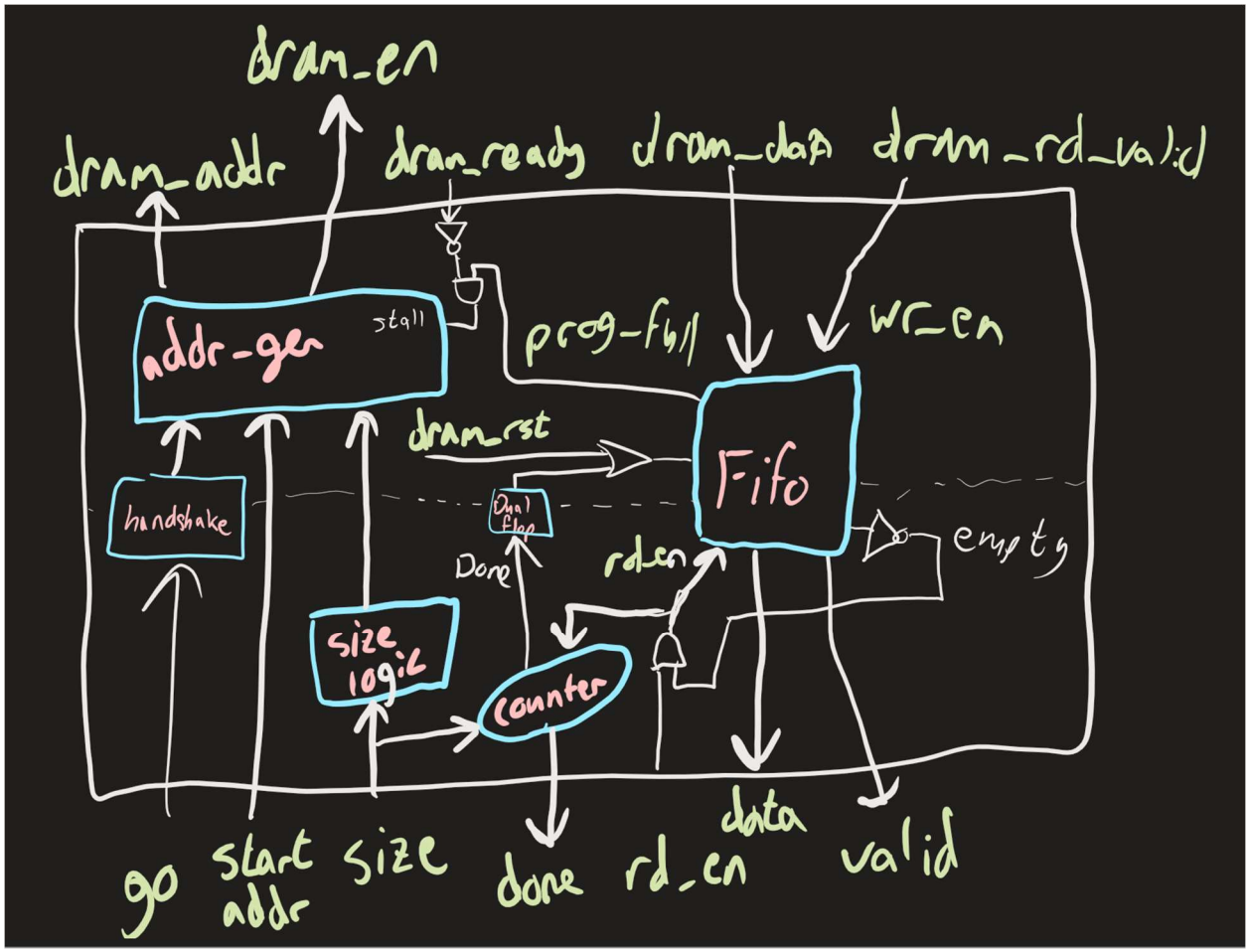


Figure X: Diagram of the implementation of dram_rd.vhd.

Dram_rd.vhd has 5 main parts—the address generator, the size logic, the done counter, the fifo, and the synchronization blocks (handshake, dual flops). A general description

Each test is triggered by “go” going high. It must pass through the handshake before getting to the address generator because it’s crossing clock domains. Since the start address and size are held at the same value all throughout each test, they are fed straight into the address generator and their values are read once go passes through the handshake. Before the size can make it to the address generator, it goes through a size logic block. If the size is odd, 1 is added, and then the size is shifted right once (divided by 2) and fed into the address generator, since the size specifies the number of 16 bit words and the block ram is 32 bits.

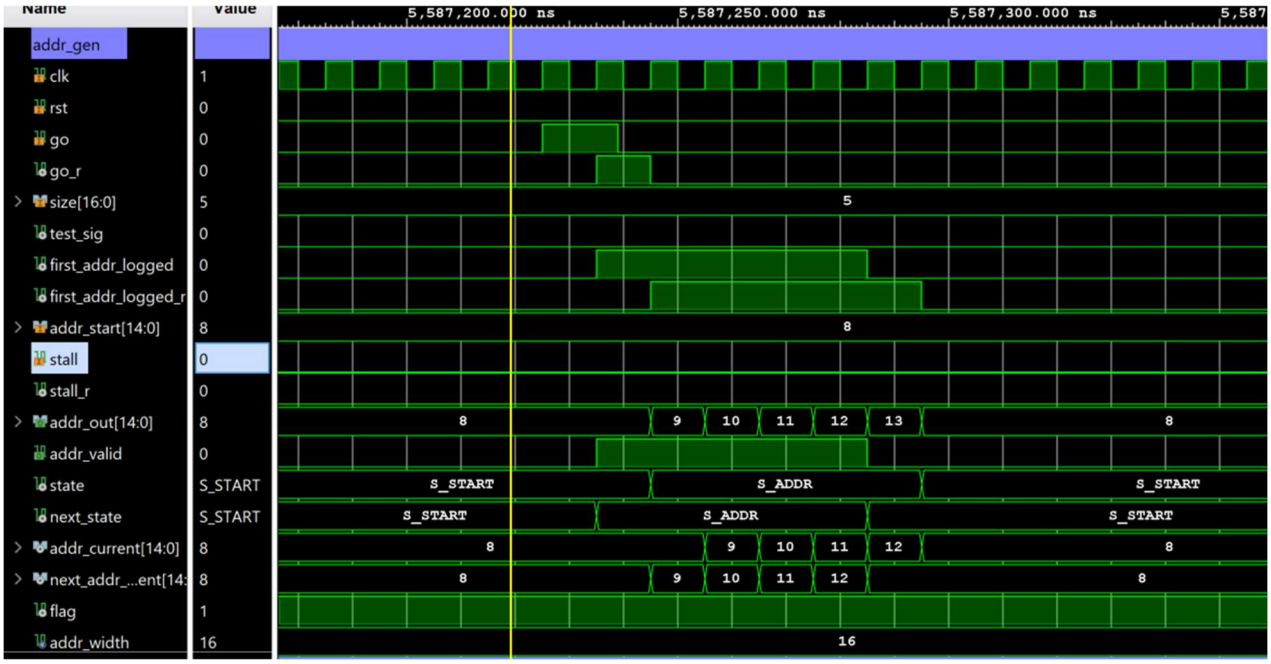
Address Generator

At this point, the address generator starts creating the addresses. It's built as a 2 process FSM. The default values before the combinational process have the output address equal to the starting address, the next address equal to the current address, next state equals state, address valid is 0, and there's a signal tracking when the first address is valid.

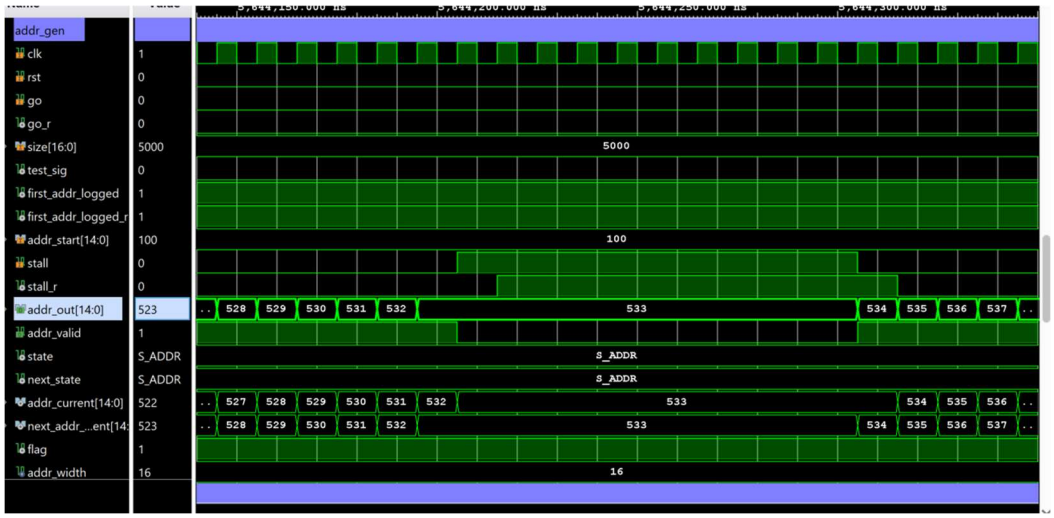
When go goes high, the state changes from START to ADDR and the valid address signal goes high, unless stall is high. The signal that tracks when the first address is valid follows the same rules here. Since the starting address is the default address output, that's the first valid output. In the ADDR state, the address is updated to the next address only if stall is low. If stall had been high in the previous cycle but is low again, the address stays the same so that it gets a chance to be valid. If this is the last address, the address is valid, and the state is reset back to START.

The stall signal is controlled by dram_ready, which means the dram is ready to accept addresses, and prog_full, which signals the FIFO is almost full.

Here's an example of the address generator at work:



Go goes high, and then the registered go follows, and since the size is 9, 5 address are read from (when addr_valid is high). You can see the states and address counter changing as well. In this next example, you can see the address generator handling a stall:

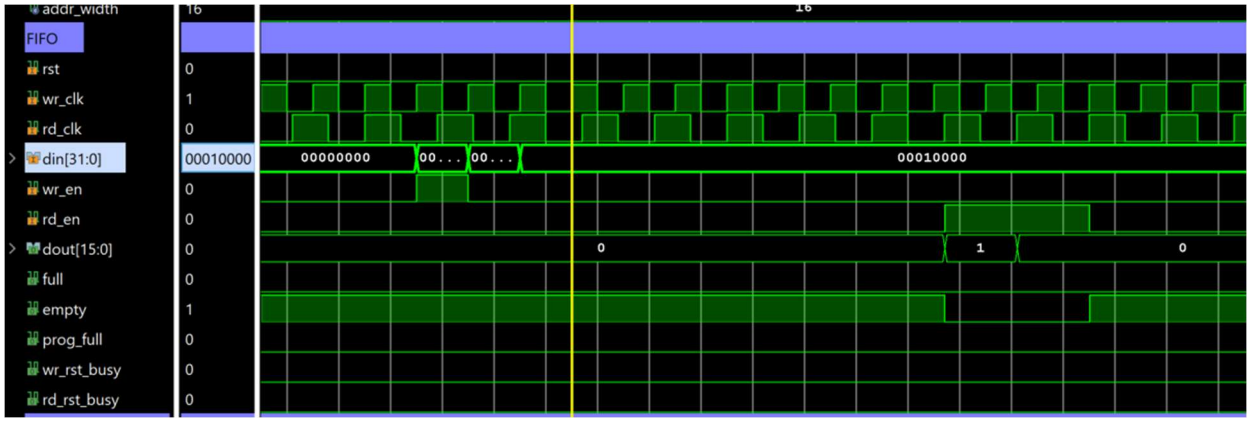


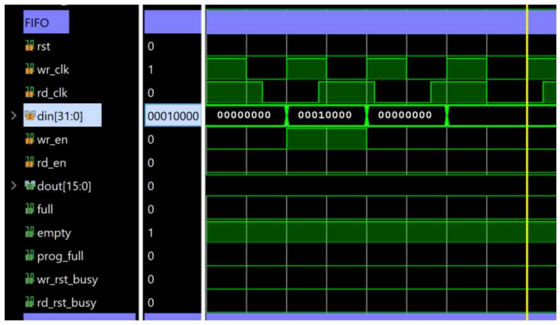
533 is read from once, and then 534, no duplicates.

FIFO

Once the data is read from its respective address from the block ram, it's fed into the FIFO. The FIFO's write enable is controlled by the dram_rd_valid signal, which tells when the block ram has valid data, the data comes from the block ram itself, and the reset is controlled by dram_rst and the done signal, so that the FIFO resets after each test. Before going to the FIFO, the done signal goes through a dual flop, but that's explained in the done counter section. The dram_rst and done signals are ORed and sent into a 6 cycle delay, where the OR of each output is combined into the reset of the FIFO. This is because documentation online revealed the FIFO reset has to be high for 6 cycles for the FIFO to be reset. Prog_full feeds into the address generator to delay it.

The rd_en for the FIFO is the and of the not of the empty signal and the rd_en input to the dram_rd.vhd. Data from the FIFO outputs to the user_app as does the valid signal.



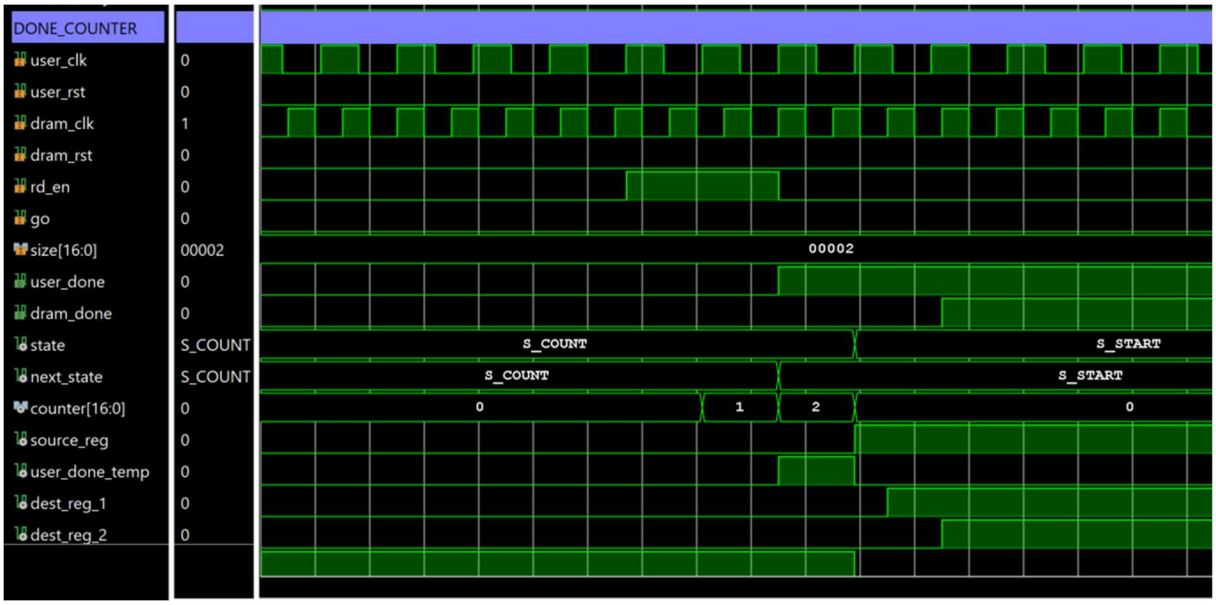


FIFO simulation example. Reads 0001 0000 and then outputs it shortly after, this is for the start addr 3 size 1 test. Rd_en controls the read.

Done Counter

The done counter counts the cycles and is enabled by the same rd_en the FIFO uses to signal when all the data has been read. This is also a 2 process FSM. In the start state, all that happens is the next state is COUNT. In the COUNT state, the counter is incremented as long as rd_en is high. If the counter has reached the size, the done signal goes high. There's additional logic in place so that the done signal stays high until the moment go is asserted again. There are actually 2 done outputs, one for user_app and one for the FIFO. The FIFO done signal must go through a dual flop because it is crossing clock domains.

Simulation of done counter:



Once read enable goes low, the FIFO is empty, and the test is done. The dram_done goes high shortly after once through the dual flop.

Below are screenshots of the design on the board. It does not work each time, but it does most the time:

```
[ngazda@ece-b312-recon2 sw]$ zed_schedule.py ./zed_app design_1_wrapper.bit
Searching for available board....
Starting job "./zed_app design_1_wrapper.bit" on board 192.168.1.100:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....SUCCESS
[ngazda@ece-b312-recon2 sw]$ zed_schedule.py ./zed_app design_1_wrapper.bit
Searching for available board....
Starting job "./zed_app design_1_wrapper.bit" on board 192.168.1.102:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....SUCCESS
[ngazda@ece-b312-recon2 sw]$ zed_schedule.py ./zed_app design_1_wrapper.bit
Searching for available board....
Starting job "./zed_app design_1_wrapper.bit" on board 192.168.1.173:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....SUCCESS
[ngazda@ece-b312-recon2 sw]$
```

Rarely, the design fails on the 0 address tests:

```
[ngazda@ece-b312-recon2 sw]$ zed_schedule.py ./zed_app design_1_wrapper.bit
Searching for available board....
Starting job "./zed_app design_1_wrapper.bit" on board 192.168.1.107:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....(Size = 1)Error: Done was not asserted be
fore timeout.
Count = 0
Start Addr = 0
DMA Addr = 0
DMA Size = 1
Prog full = 0
DMA Empty = 1
terminate called after throwing an instance of 'TimeoutException'
  what():  Timeout Exception
sh: line 1:  2590 Aborted                 ./zed_app design_1_wrapper.bit
[ngazda@ece-b312-recon2 sw]$
```

When this happens, it's a "Done not asserted" error, most likely because of the stall signal interacting with the address generator. While the address generator usually can handle the stall signal, it sometimes sees issues when stall goes high right as the address generator is beginning operation:

Also sometimes, the design will fail on one of the random address/size combinations that has a combined value of 32768, or the size of the buffer.

```
[ngazda@ece-b312-recon2 sw]$ zed_schedule.py ./zed_app design_1_wrapper.bit
Searching for available board....
Starting job "./zed_app design_1_wrapper.bit" on board 192.168.1.173:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....(Size = 51610, addr = 6963)ERROR: Input
20168 output = 24264
ERROR: Input = 24256 output = 20160
ERROR: Input = 24260 output = 20164
ERROR: Input = 24262 output = 20166
ERROR: Input = 24270 output = 20174
ERROR: Input = 24272 output = 20176
ERROR: Input = 24274 output = 20178
ERROR: Input = 24276 output = 20180
ERROR: Input = 24284 output = 20188

ERROR: Failed test for size 51610 and address 6963
[ngazda@ece-b312-recon2 sw]$
```

```
[ngazda@ece-b312-recon2 sw]$ zed_schedule.py ./zed_app design_1_wrapper.bit
Searching for available board....
Starting job "./zed_app design_1_wrapper.bit" on board 192.168.1.110:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....(Size = 56, addr = 32740)ERROR: Input =
746 output = 24554
ERROR: Input = 32752 output = 24560
ERROR: Input = 32758 output = 24566
ERROR: Input = 32764 output = 24572

ERROR: Failed test for size 56 and address 32740
[ngazda@ece-b312-recon2 sw]$
```

In both these examples, the size (after dividing by 2) and data add up to 32678. This only seems to happen at the addresses that touch the boundaries of the dram and we could never figure it out. We thought it might be the buffer size because the prog_full was toggling pretty often and long tests couldn't finish in time before another test started, and so the done signal never went high to reset the buffer, but after messing with that a lot it still kept giving issues, though less frequent.

The following screenshot shows the dram_rd working for each test case on the board but in simulation:

```
Note: Test132 completed. Total Errors: 0
Time: 5584655 ns  Iteration: 0  Process: /wrapper_tb/li
Note: finished address 0 tests
Time: 5584655 ns  Iteration: 0  Process: /wrapper_tb/li
Note: Test0 completed. Total Errors: 0
Time: 5585725 ns  Iteration: 0  Process: /wrapper_tb/li
Note: Test0 completed. Total Errors: 0
Time: 5586805 ns  Iteration: 0  Process: /wrapper_tb/li
Note: Test1 completed. Total Errors: 0
Time: 5588005 ns  Iteration: 0  Process: /wrapper_tb/li
Note: Test2 completed. Total Errors: 0
Time: 5589105 ns  Iteration: 0  Process: /wrapper_tb/li
Note: Test3 completed. Total Errors: 0
Time: 5880015 ns  Iteration: 0  Process: /wrapper_tb/li
Note: TestBig completed. Total Errors: 0
Time: 6211425 ns  Iteration: 0  Process: /wrapper_tb/li
Note: TestBig2 completed. Total Errors: 0
Time: 6542835 ns  Iteration: 0  Process: /wrapper_tb/li
Note: SIMULATION FINISHED.
Time: 6542835 ns  Iteration: 0  Process: /wrapper_tb/li
run: Time (s): cpu = 00:00:07 ; elapsed = 00:00:31 . Me
```

At the top you can see test132—this tests reads from address zero to size 132*132, and each one before that as well (1*1, 2*2…. 131*131, 132*132). The middle tests are some of the included tests in wrapper_tb. Testbig is run twice to show that it works in simulation—these tests test the component with a size+address of 32768, run one right after the other, to show that the FIFO is clearing and it's able to carry these tests one after the other.