1-D Time-Domain Convolution

EEL 4720/5721 – Reconfigurable Computing

Team members:

1) Raghul Shivakumar

2) Johnny Klarenbeek

We have done both the DRAM DMA Controller(read) and user_app entities and both are fully working.

DRAM DMA CONTROLLER (READ)

Block Diagram / Description:

As shown in fig. 1, the DMA controller consists of four main sub-blocks:

- **dma_ctrl:** Handles synchronization between clock domains using a handshake synchronizer. When 'go' is asserted on the input of this block, the signal is synchronized and passed to the address generator (as 'go_s'), starting it and indicating that the size and start_addr values have settled. Additionally, the block contains some logic to stall the address generator while the DRAM is not ready or when the 'prog_full' is asserted.
- **dma_addr_gen:** Generates the read addresses for the DRAM starting at the given 'start_addr' and ending at 'start_addr' + 'size'. The counter is stalled when 'prog_full' is high or when 'dram_rd_ready' is low
- **fifo_32_16**: This Xilinx library component (generated with 'fifo generator') buffers the 32-bit DRAM data in the dram_clk domain and outputs it as 16-bit data in the user_clk domain. Note that for proper output addressing it is necessary to swap the upper and lower 16 bits in the source data. This is performed using glue logic outside of the FIFO.
- **data_counter:** This module is responsible for keeping count of the amount of data that has been passed to the user_clk domain. When the full amount of requested data has been passed on, it asserts 'done'.

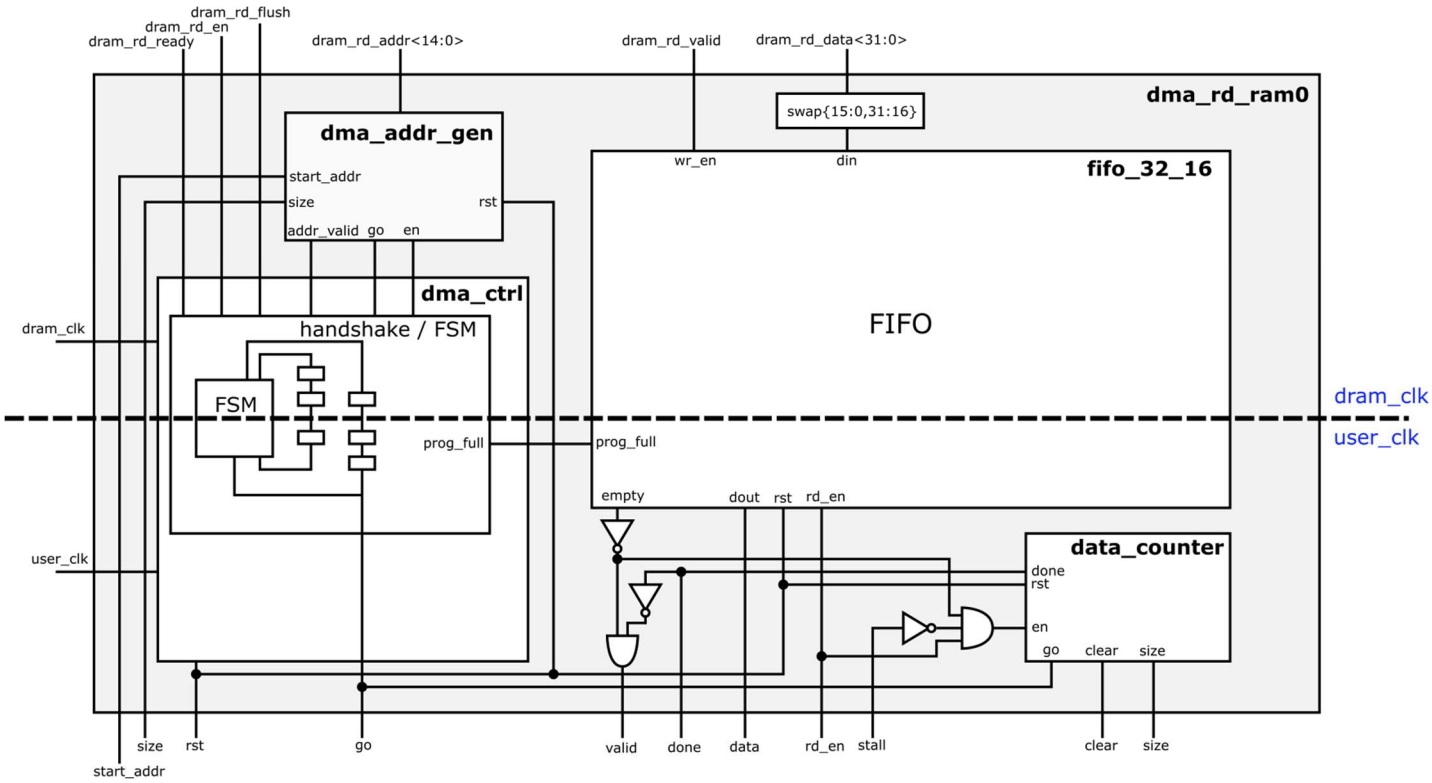


Figure 1: Basic structure of the DRAM0 (read) DMA controller. Note not all clock connections to sub-blocks are shown

Simulation Result:

The testbench was simulated with several different test sizes throughout the development of the DMA controller. Initially, a small test size of 4 was chosen as it greatly simplified reviewing the waveforms and debugging. After resolving some basic issues with signal timing, implementing necessary glue logic, and verifying proper behavior with a test size of 4, it was increased to a size of 1000 and verified. Once simulation passed the test size of 1000, we moved on to testing the actual implementation on the FPGA. A simulation waveform demonstrating the DMA controller behavior (start of DRAM read and data output) is presented in fig. 2.

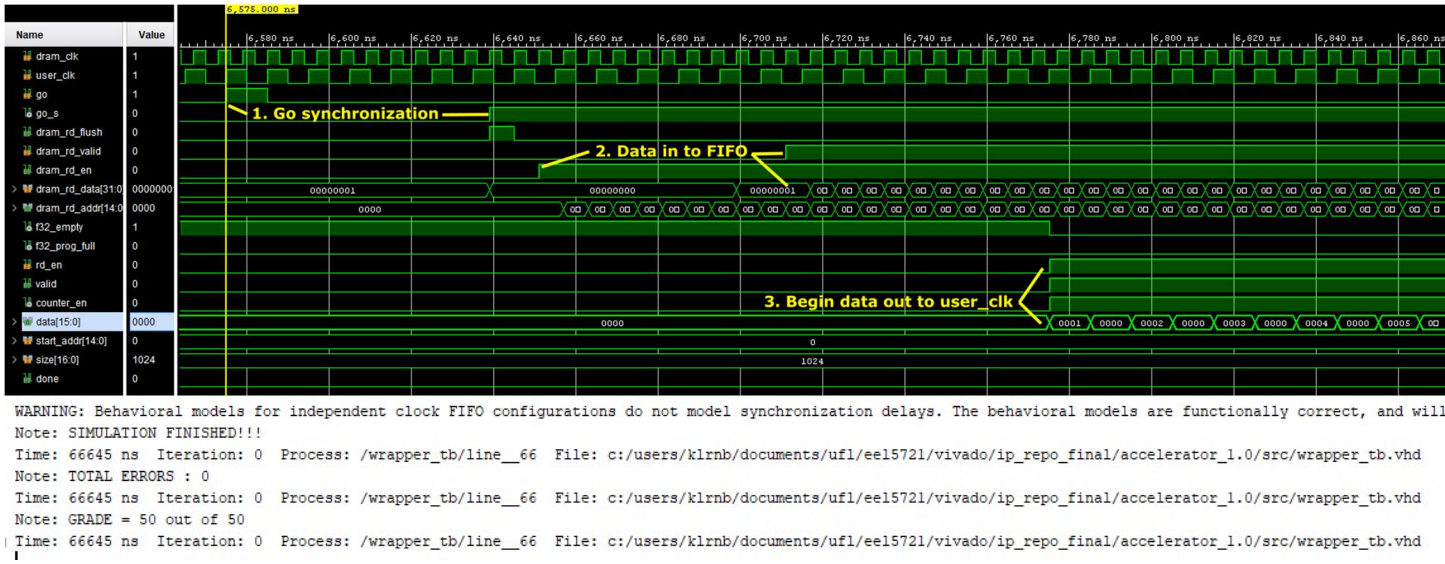


Figure 2: Simulation with DMA controller demonstrating transfer of 1000 32-bit sequential values (presented as 16-bit values on the output). The simulation passes the output check with a grade of 50/50

Software Result:

The software tests transfers from address 0, from random addresses, and with random sizes up to the maximum size. As shown in fig. 3, our implementation passes the DRAM test software without error.

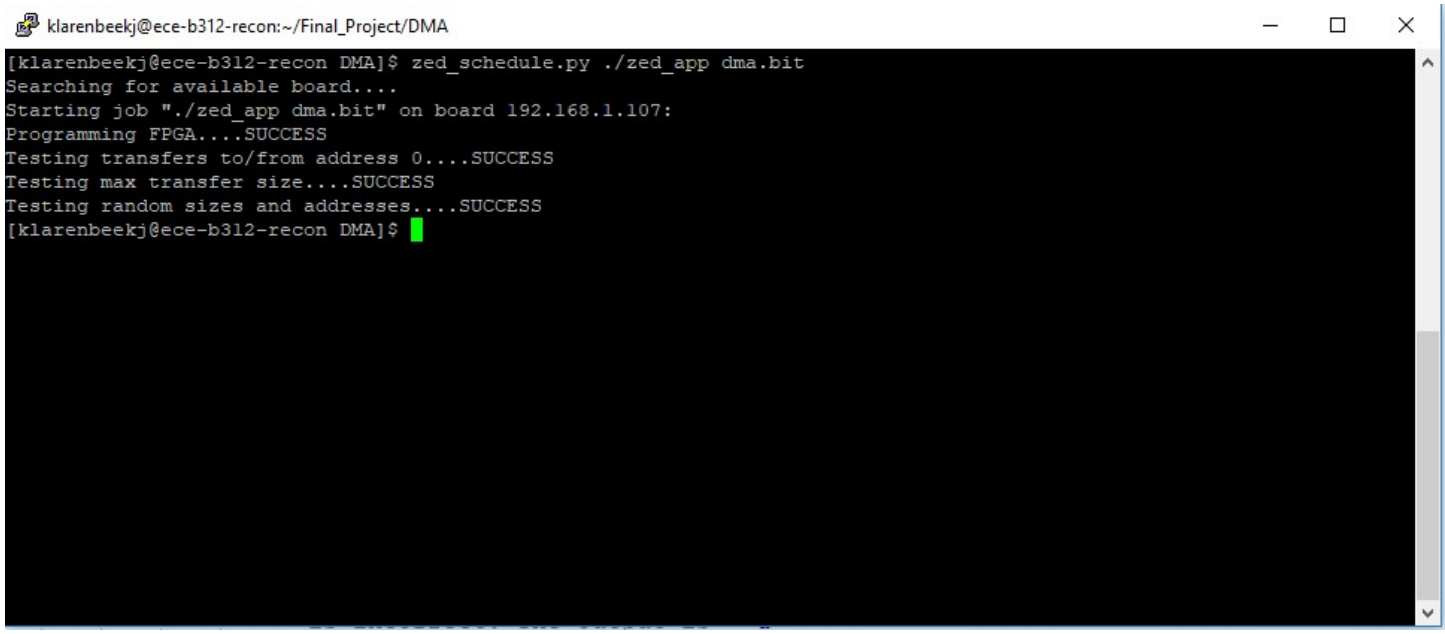


Figure 3: DRAM test software run indicating the implementation completes without error

Implementation Difficulties:

Implementation of the DMA controller was relatively smooth because the top-level behavior (signal sequencing and timing) of the provided sample controller was carefully studied. However, some problems were encountered during the software test which were not seen in the implementation:

- Boards were initially freezing indicating the ‘done’ signal did not get asserted. This was debugged by modifying the software to proceed after some delay instead of waiting for ‘done’. After realizing that no DRAM data was transferred, we tried to debug the go signal synchronization.
- The ‘done’ signal was rewired so that it was forced to go high whenever the address generator starts. Here we found out that the address generator never starts. We suspected that it was because we are only forcing the synchronized go signal high for one cycle (though we are not sure why). After forcing the go signal high for more than one cycle (until it is acknowledged by the address generator), the implementation passed all software tests

USER_APP

The user_app entity has the following components as shown in Fig1.

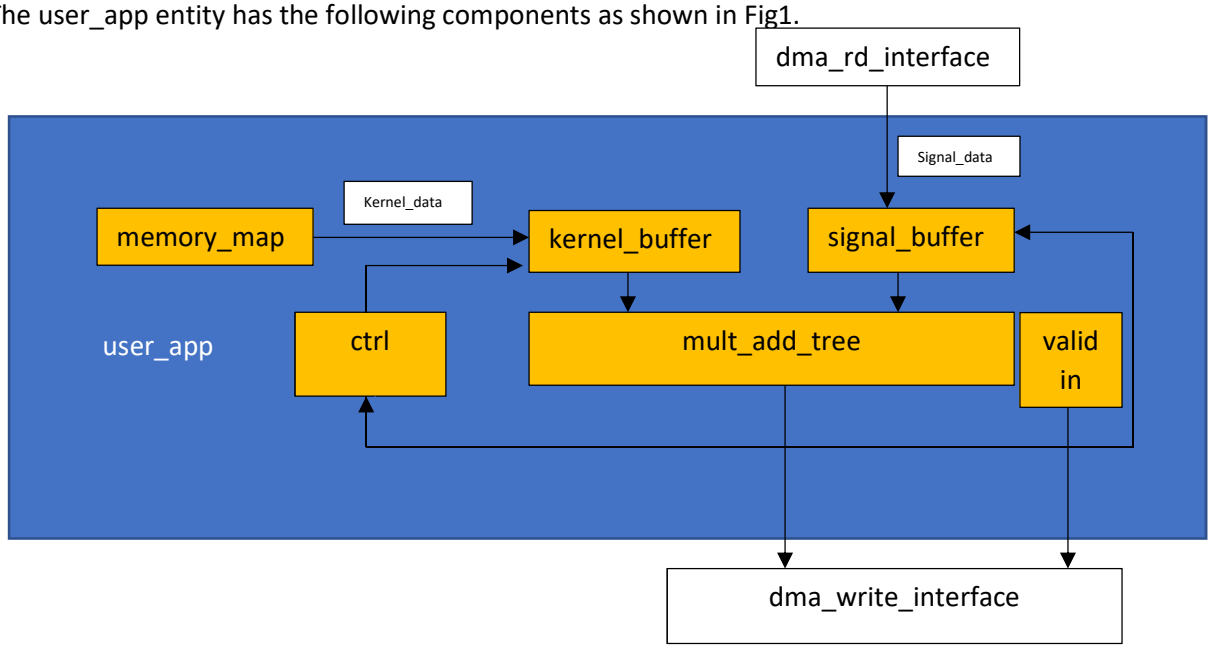


Fig1.

1) memory_map: This entity establishes connections with user-defined addresses and internal FPGA components (e.g. registers and blockRAMs).

2) window_buffer and kernel_buffer:

This is a sliding window buffer where the window slides by one value every iteration. Both window_buffer and kernel_buffer have the same functionality with one difference which is the order in which output signal comes out. For Convolution we need the signal buffer output to be reverted. A separate test bench was used to test signal and kernel buffer. The signal buffer is a sliding window buffer where the window size and buffer size are equal. The stride length is 1. The values for output in signal buffer are reversed. The rd_en signal is asserted when the buffer is not empty and there ram1_wr_ready is 1. The wr_en signal is asserted when the buffer is not full and there is data that can be read from ram0. The full signal is also controlled by the rd_en so that there is full throughput. As it

can be seen in the figure the values rd_data and kernel_output which are the outputs of signal and kernel buffer respectively are in the reverse order. A window of size 3 is chosen for clarity.

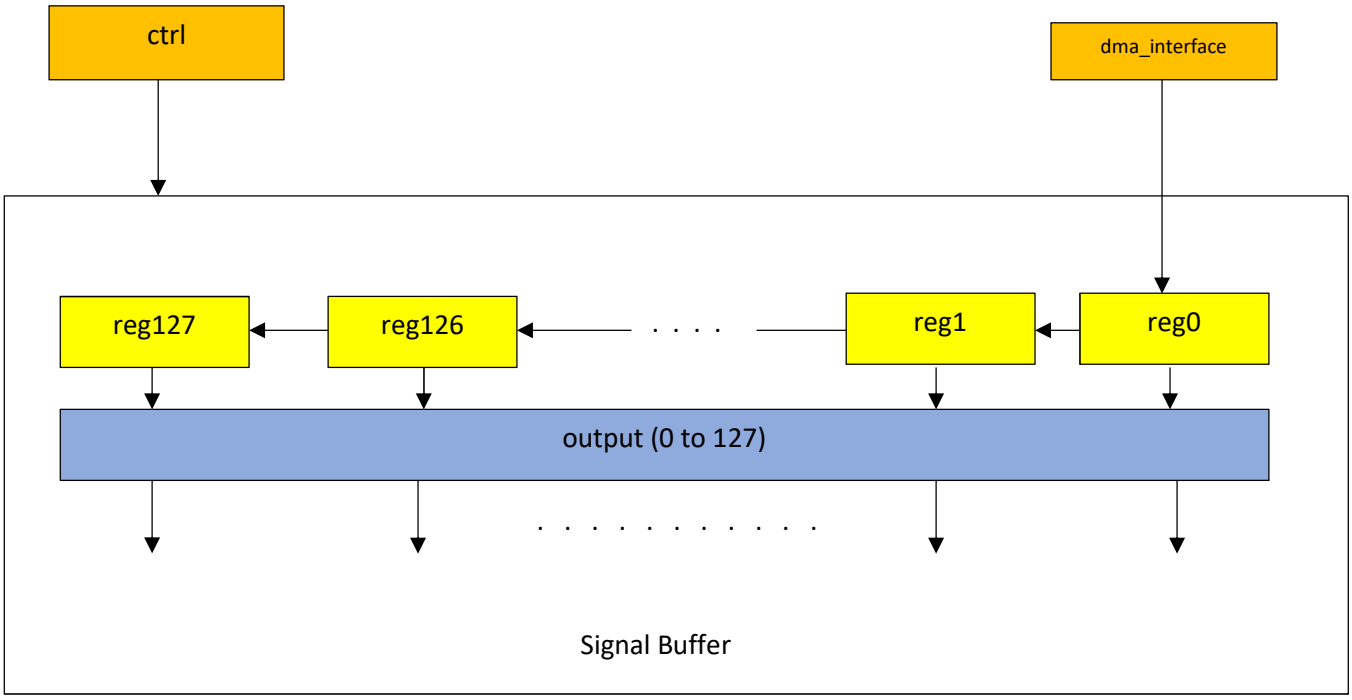
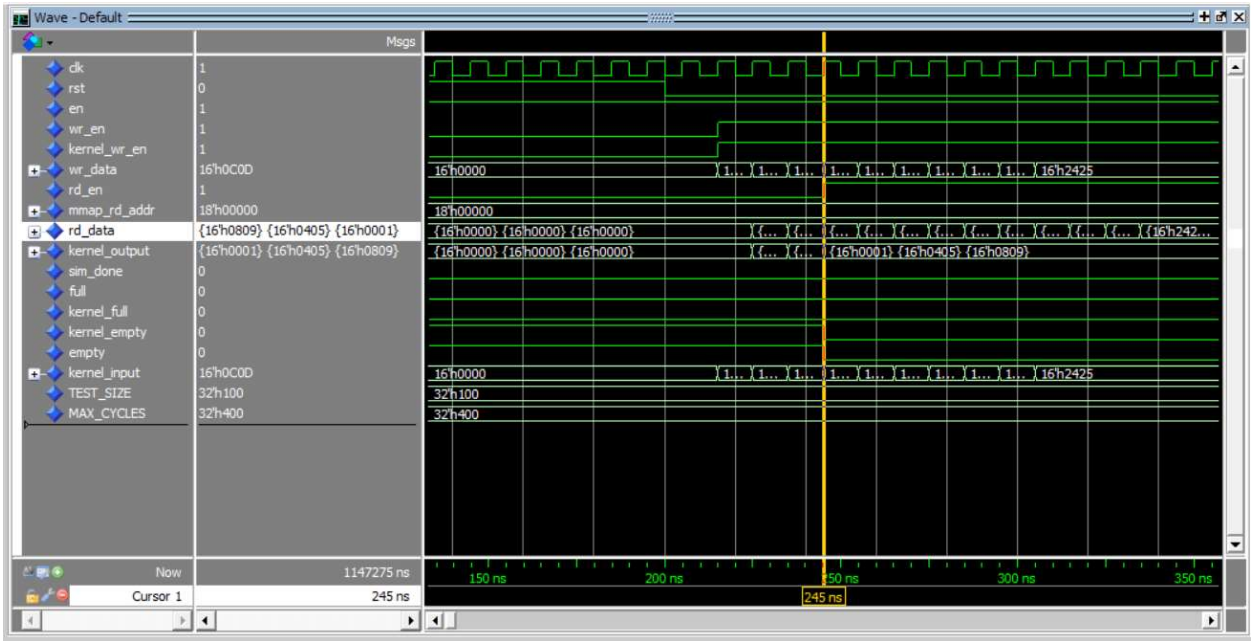


Fig2

Signal and kernel Buffer output.



3) ctrl: This entity forms the control unit of the user app. It is a state machine which asserts go signal, passes the control signals to buffers and then waits for done signal to asserted by the dram1_wr_interface.

4) mult_add_tree: This is the pipeline data path entity of the convolution entity.

5) delay: This entity is used to delay the write_en signal to dram1_write interface.

Software Output: Our Implementation passes all the tests performed by the software.

```
[shivakumarraghul@ece-b312-recon sw]$ zed_schedule.py ./zed_app convolve.bit
Searching for available board....
Starting job "./zed_app convolve.bit" on board 192.168.1.106:
Programming FPGA....Testing small signal/kernel with all 0s...
Percent correct = 100
Speedup = 0.0144231

Testing small signal/kernel with all 1s...
Percent correct = 100
Speedup = 0.0151515

Testing small signal/kernel with random values (no clipping)...
Percent correct = 100
Speedup = 0.0186916

Testing medium signal/kernel with random values (no clipping)...
Percent correct = 100
Speedup = 6.76869

Testing big signal/kernel with random values (no clipping)...
Percent correct = 100
Speedup = 15.9435

Testing small signal/kernel with random values...
Percent correct = 100
Speedup = 0.0236686

Testing medium signal/kernel with random values...
Percent correct = 100
Speedup = 5.35876

Testing big signal/kernel with random values...
Percent correct = 100
Speedup = 14.5302

TOTAL SCORE = 100 out of 100
[shivakumarraghul@ece-b312-recon sw]$
```

Issues faced while designing and the solutions for them:

1) Size for ram0_read and ram1_write:

The outputs were not proper, and the main reason was that the software sends the size of unpadded signal and we had directly passed it. The right value must be:

ram0_rd_size <= signal_size(unpadded) + 2*(kernel_size-1)

ram1_wr_addr <= signal_size(unpadded) + kernel_size-1

2) Combinatorial loops:

Initially we read the value of full and empty signals to assert the rd_en and wr_en signals. The full and empty signals depend on count value and the count value is in turn dependent upon the value of wr_en and rd_en. This causes a circular dependency which the synthesis tool

detected and stopped the bitstream generation. We overcame this problem by storing the full, empty and count signals in a register.