**1-D Time-Domain Convolution**

**Final Project**

**Course:** EEL5721-Reconfigurable Computing (Fall2020)

**Date of Submission:** 12/14/2020

**Submitted By:**

Nathan Jessurun

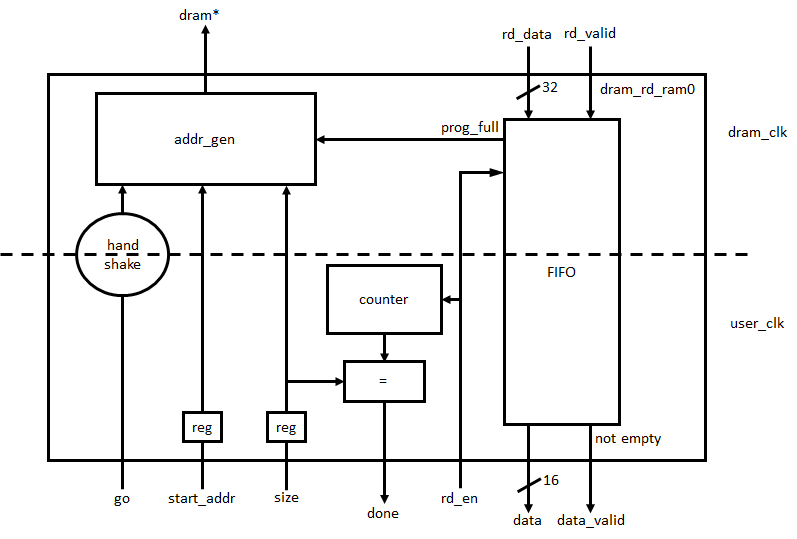
Mir Tanjidur Rahman

# Objective:

In this project, we developed a hardware implementation of 1-D time domain convolution. We have implemented the convolution operation in a ZEDBOARD FPGA. The kernel is an array of 128 element and 16 bit.

# Part 1: DRAM DMA Interface

Our DRAM implementation is largely based off the lecture diagrams provided by Dr. Stitt. The architecture overview is shown below:



Almost all computation is performed in the user clock domain to avoid clock rate slowdowns.

## Handshake + signal registers

To facilitate sending signals across clock domains, first, the input data must be stored in registers to avoid propagating changes after a “go” signal is received. Note that “go” does not need its own register because this is handled internally by the handshake mechanism.

The handshake works as defined in previous labs. The user clock receives this “go” signal and forwards a “send” signal to the DRAM clock. This is in turn forwarded to the address generator.

## Address generator

Once “go” is received, the address generator continuously creates sequential addresses until “size” addresses have been created. However, since the DRAM input is 32 bits and the DMA output is 16 bits, only half the requested addresses must be generated. As explained later by the counter entity, halving the number of generated addresses for this reason is unnecessary.

Note that when the FIFO is full, no addresses are generated for that cycle. This prevents received data from the DRAM from being lost.

## FIFO

The FIFO use is quite straightforward and directly generated from Vivado IP cores. As previously stated, it has a 32 bit input and 16 bit output meeting I/O requirements. Moreover, it has a depth of 65536 with programmable full flag at 65000 to allow a wide margin of delays during memory access. Since circuit area is not a concern during our schematic planning, this large FIFO has a minimal impact on overall performance.

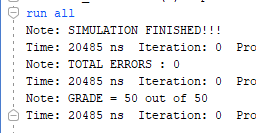
## Counter + comparator

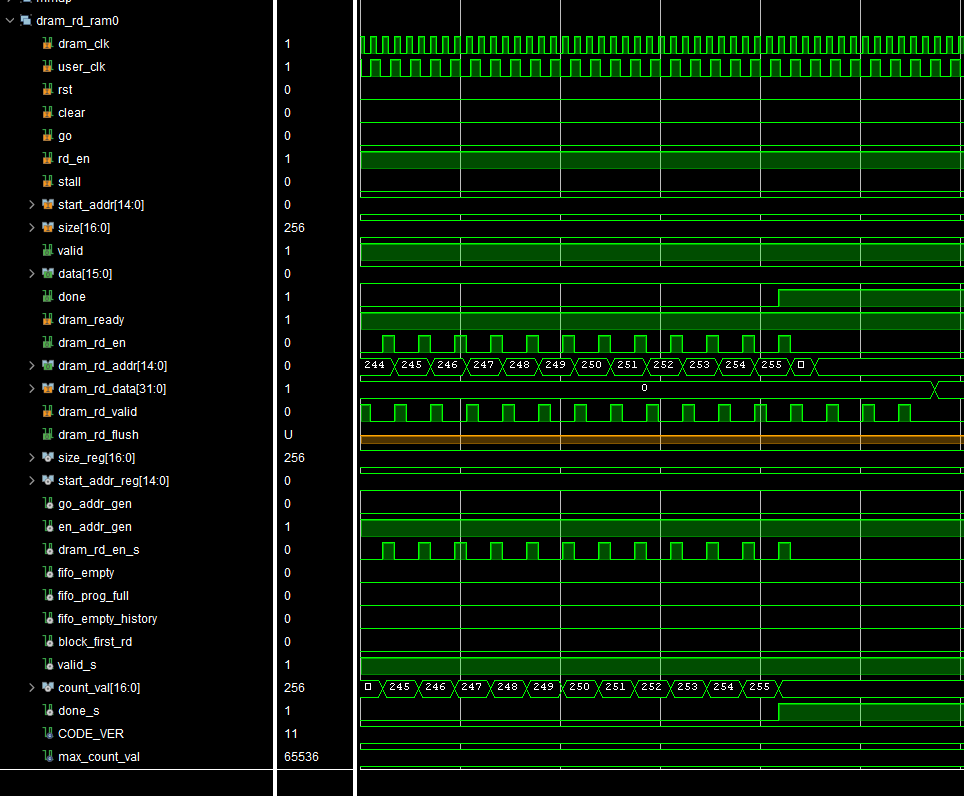
The DMA reading process is considered complete when “size” addresses have been read from memory. A counter entity keeps track of these addresses in the user clock domain so no conversion is needed between user-space address sizes and DRAM-space addresses.

Addresses are counted using the dram\_rd\_en flag, which pulses every time the user domain requests an address from the DMA. Hence, when “size” reads have been performed, “done” should be set high. This action is performed simply by comparing the current “count” value to the “size” register.

## Correctness

Unfortunately, **while the DMA described above successfully simulates, it times out without sending any address data when run on the server**. We are unsure what is causing this discrepancy, considering all logic works as predicted in the DRAM testbench. Some screenshots confirming this are shown below, where size addresses are correctly retrieved from memory and the “done” signal is emitted. It could be that “go” is not received in hardware, but this is unlikely considering the handshake is listening for this signal within the user clock domain. Since all other items are taken from other labs or from the class website, it is unlikely that functional errors are a result of their implementation as well.





# Part 2: Convolution Pipeline

To implement convolution pipeline we have used the provided code in the course website. We have used the recursive implementation of pipeline (mult\_add\_tree.vhdl). The pipeline takes 128 bits of kernel and signal input and generate the convolution outcome.

## Challenges:

Valid\_input bit logic selection: We have implemented a delay chain to define the exact timing of valid outcome. The valid chain takes valid input when the signal buffer and kernel buffer is full, and corresponding read enable is activated.

# Part 3: Signal Buffer

As signal buffer a large shift register is implemented of array for 128 X 16 bits. The signal buffer is output can generate the abovementioned size of output data for each clock cycle. It takes the input form the DMA interface and stores the data in the output registers, depending on the availability of the shift registers, the buffer provides full or empty signal to the DMA interface.

## Challenges:

There are few challenges we have faced which we will discuss through the testbench we have developed. We used different size of signal buffer. The result is presented for 6 element, 8 bit signal.

Signal buffer read enable activation: The signal buffer was losing data once it the full\_flag = 1. The reason behind it is, if empty\_flag relies only on full\_flag to store information, it may store input data before the data reading is complete. We solved this issue by using the full\_flag and rd\_en flag to initiate empty\_flag.

Signal\_buffer empty buffer switching: Once the data write to signal buffer is complete the empty\_flag must increase its value to show that empty space in buffer in rising. We solved this issue by connecting a counter with the wr\_en able of the signal buffer.

Defining empty and full flag: for defining the empty and full flag we have used a counter which is activated based on the read\_enable and write\_enable flag of the signal buffer.

Inversing the order of signal buffer: We were having a HW = 0 when the bitstream is first implemented in the vivado. Later based on our understanding of convolution we reversed the signal buffer input sequence. Later our finding is confirmed by course instructor.

Once we have address all the issue we have evaluated our signal buffer with the test bench for different conrner values. And the final result is shown in the following fig. 2.

The data is shifting one element per clk. This creates a sliding window along the signal array provided from RAM.



Fig 2: Signal Buffer simulation waveforms

# Part 4: Kernel Buffer

The difference between signal buffer and kernel buffer is that once the kernel buffer is full it does not update its value unless it is explicitly requested to do so, i.e., the wr\_enable flag is activated.

Similar to signal buffer we have tested the kernel buffer using 6 element of 8 bit array. The test bench shows that the kernel maintain its output memory irrespective to kernel\_input signal. However, we have allowed the kernel to change its output once the wr\_enable flag is activated to avoid an additional kernel\_buffer\_enable signal to activate the module. It reduced a lot of complexity in the control signal for our design.

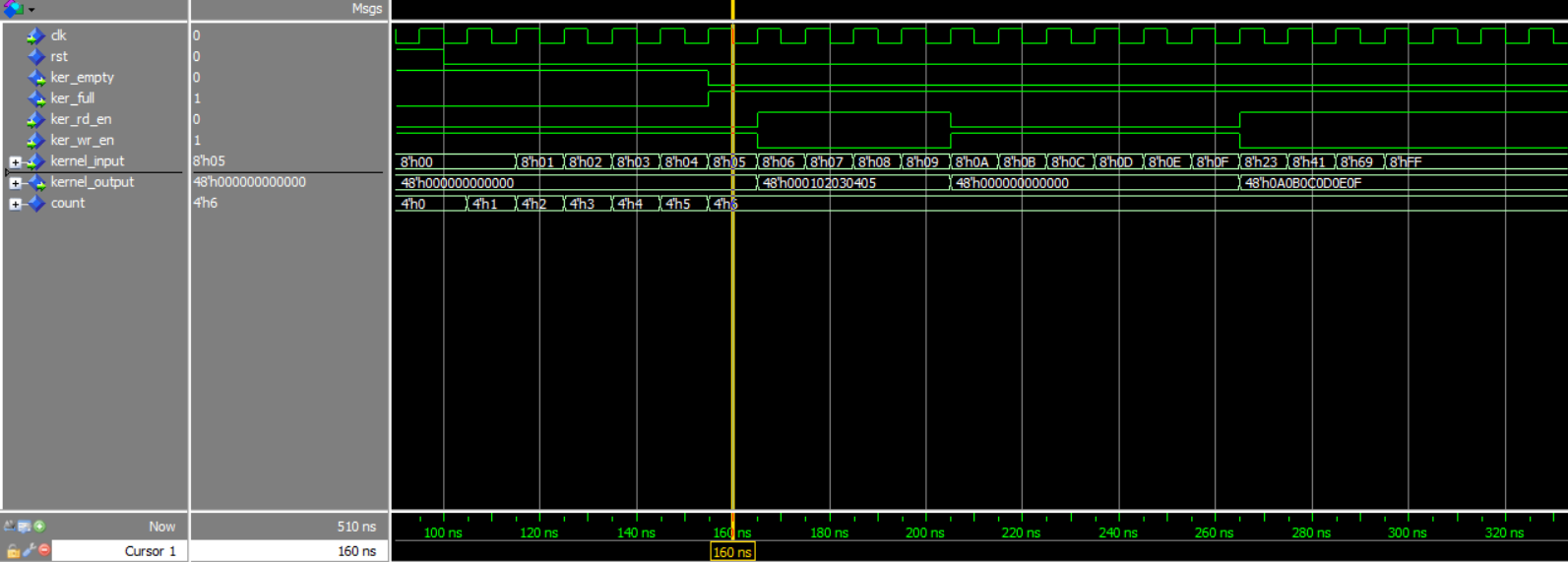


Fig 3: Kernel Buffer simulation waveforms

As shown in Fig 3, We read a sequence of data from ram (ram\_data) and once we are done with reading 6 elements (00h, 01h, 02h, 03h, 04h, 05h) we loaded the data to the output (buf\_data). From this point, even if more data is available in the ram for the buffer (06h, 07h, ….. etc.), data will not be read by the buffer to make it available to the output which reflects the exact functionality of the kernel buffer.

# Part 5: Others

## Counter:

In order to maintain the count of signal and kernel buffer, we have designed a counter. The counter is connected to the read\_enable and write\_enable of the buffers. Therefore, the counter is not dependent on buffer activity rather the information received form the user\_app.

# Part 6: Working Hardware Implementation

The screenshot below shows the resulting bitfile running successfully on the Zedboard server:

