1-D Time Domain Convolution

On Zynq Zedboard

Saad Afzal, Vyas Kovakkat
Electrical & Computer Engineering
University of Florida
Gainesville, FL
saadafzal@ufl.edu, vyas.kovakkat@ufl.edu

***Abstract*—Convolution is one of the most important techniques that is heavily used in the Digital Signal Processing domain. This report talks about the implementation of convolution of two signals that has been implemented on the zedboard and the issues that have been encountered, few of which could be rectified.**

Keywords—FPGA, clock domain crossing, styling, Zed Board, Zynq, time convolution, pipeline, buffer, DRAM, DMA.

INTRODUCTION

The concept of convolution involves two signals, one of which is the unit impulse i.e. impulse function at the sample time t =0. The output response of any system can be found out by simply convoluting the input signal x(t) with the impulse signal h(t) to generate the output of the linear system. In digital linear systems, convolution is a bitwise operation on the discrete form of the signals. This technique is one of the most power tools of digital signal processing.

The aim of the project was to implement 1-D time convolution for a signal of arbitrary length defined by the software. The input signals are defined by the software out of which the kernel data is passed directly from the software. The other input signal resides in the DRAM which is connected to the FPGA. However, in this case, we are using a simulated version of the DRAM which has been implemented on the block RAM of the FPGA board.

The project tasks have been divided into four parts which are as follows:

1. DRAM-DMA interface
2. Datapath pipeline
3. Signal/Smart buffer
4. Kernel Buffer

These designs that could be implemented and the issues we faced with each implementation and fixes along with pending outliers have been discussed below.

Also, we enlisted all possible roadblocks that we had encountered and the steps that were taken to fix the same. Some of the problems are still open ended, though we knew the cause of it, but could not rectify in the specified time frame.

Briefly describing, we could implement the DMA interface to read data from DRAM_RD_RAM0 and ran it on the board but the output being generated was different every time clearly indicating metastability. However, our simulation of the entity works perfectly on the simulator and generates a perfect score.

For the other half of the project, we designed a smart buffer and kernel buffer which was implemented like a FIFO with empty and full flag logic. It has been thoroughly tested on the simulator to test the functionality and correct working. We also integrated it with the datapath and provided DRAM_RD_RAM0 interface and ran simulations with our own test bench, as well as the provided wrapper test bench and simulated the correct working. However, there are differences in the output of HW and SW when tested on the board.

Note: Few parts of code snippets like handshake, user_app, delay, address generator have been reused from previous lab solutions and provided code.

I. DMA AND DRAM INTERFACE

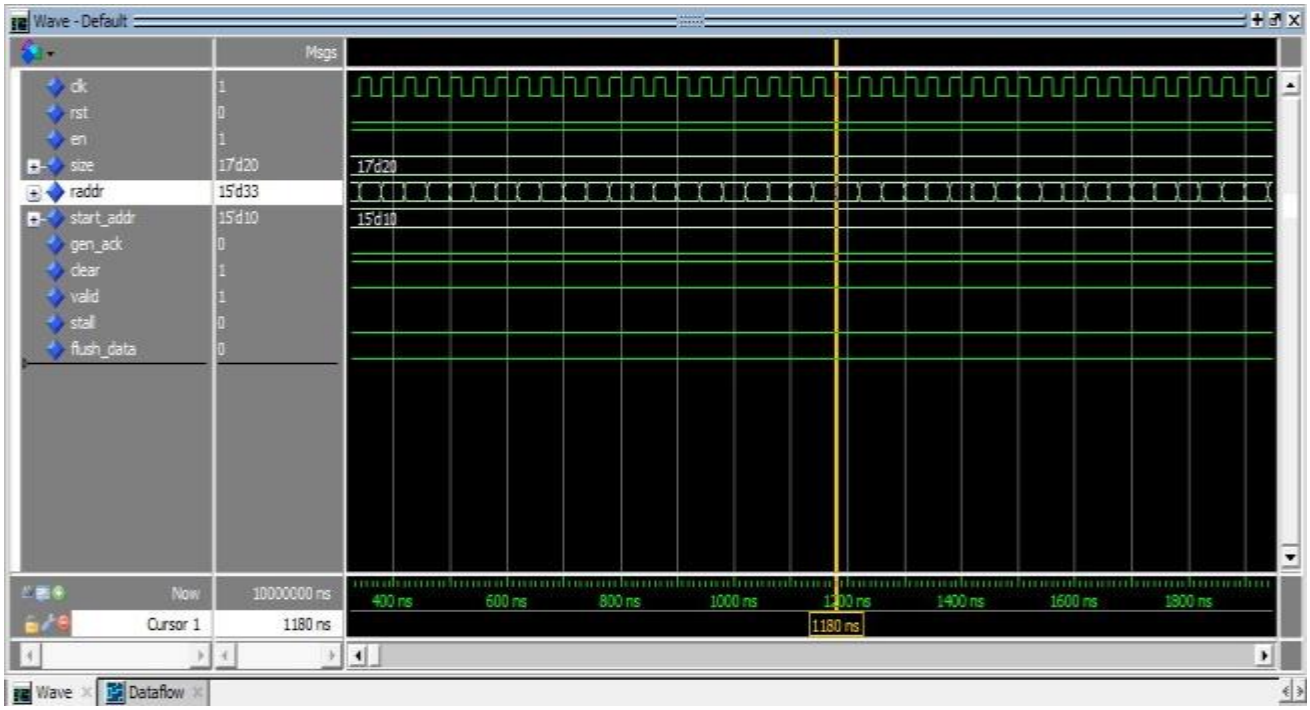
As the first part of the project, we implemented direct memory access entity of the DRAM_RD_RAM0 interface which reads data from the input DRAM and provides it to the signal buffer. Since the application works on two different clocks, the first important factor we needed to consider the clock domain crossing.

Steps taken to design DMA interface:

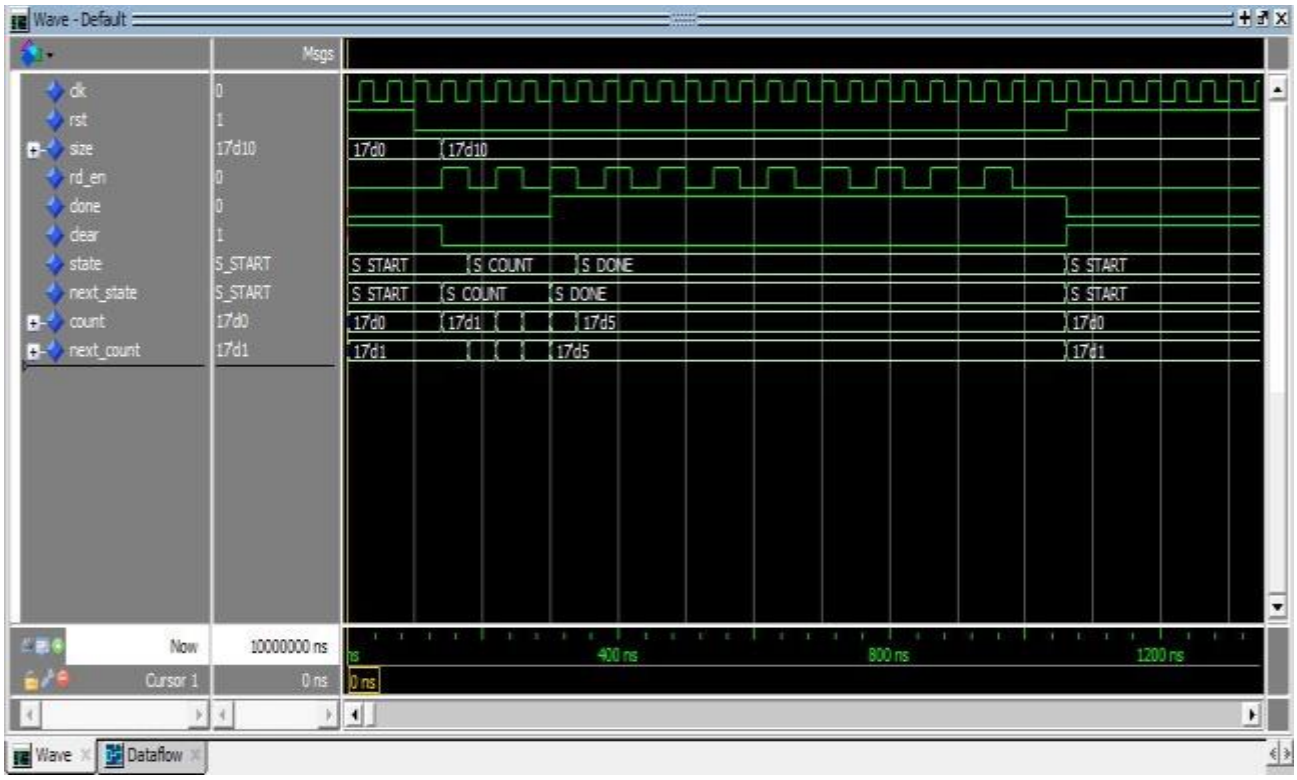
1. Created an address generator and set the clock frequencies of the two clocks to be 133Mhz and 100Mhz.
2. Created and used a handshake entity to handle signals crossing the clock domain namely ‘go’, ‘start_addr’ & ‘size’. This was done to synchronize the signal and reduces the errors due to metastability.
3. The above mentioned signals were provided through the controller in clock domain-2 to drive the address generator which in turn after synchronization drove the DRAM_RD_RAM0 interface.
4. Also, once the data was read from DRAM_RD_RAM0, we needed to provide this data to the signal buffer, again crossing a clock domain. Hence a FIFO was implemented and data was passed to it, since throughput was important in this case.

The read and write clocks for FIFO were operating at different frequencies. Also, the input and output width to FIFO was 32 and 16 bits respectively. The lower and higher 16 bits of each data element from DRAM were swapped to obtain the output of FIFO in desired order.

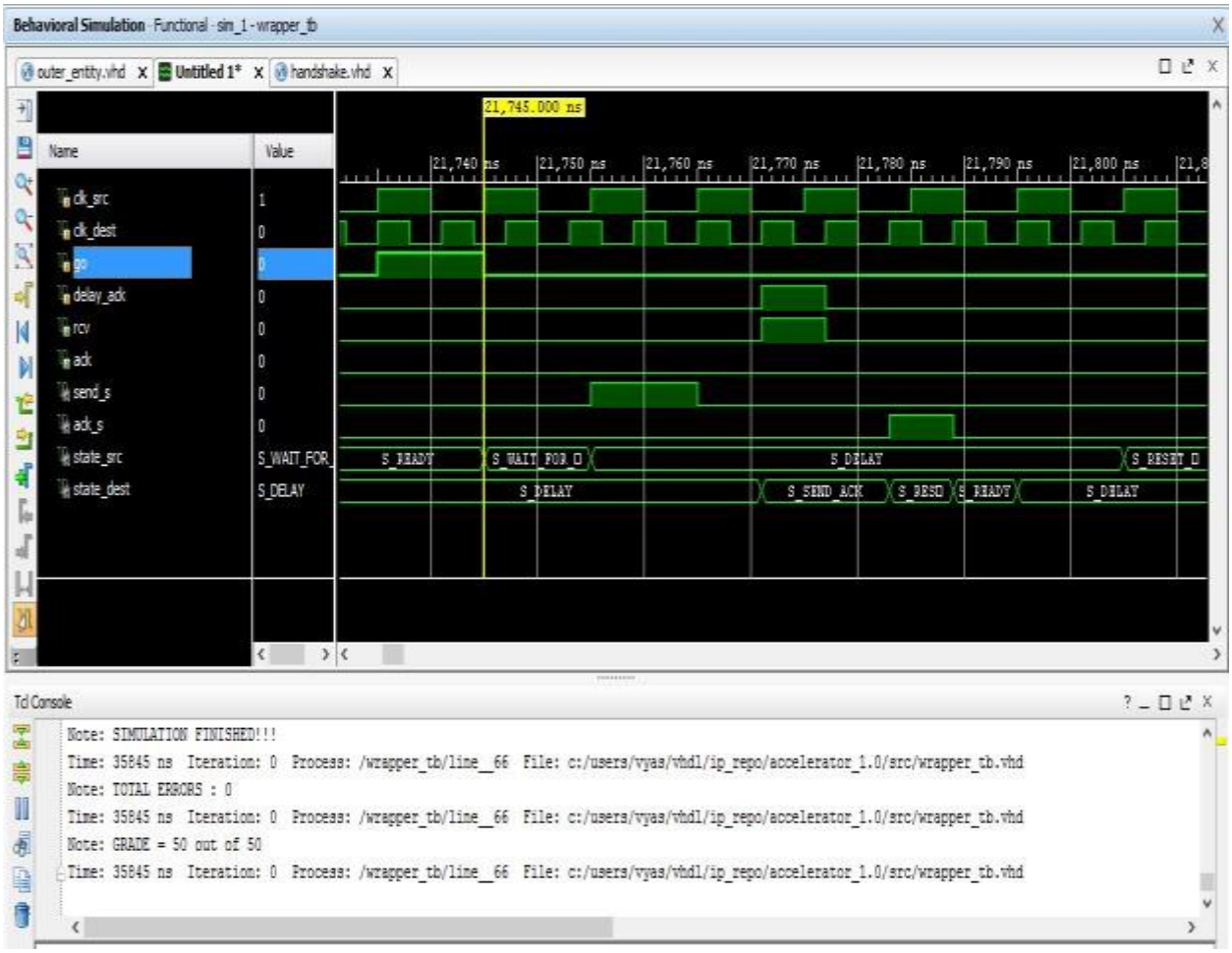
Screenshots:



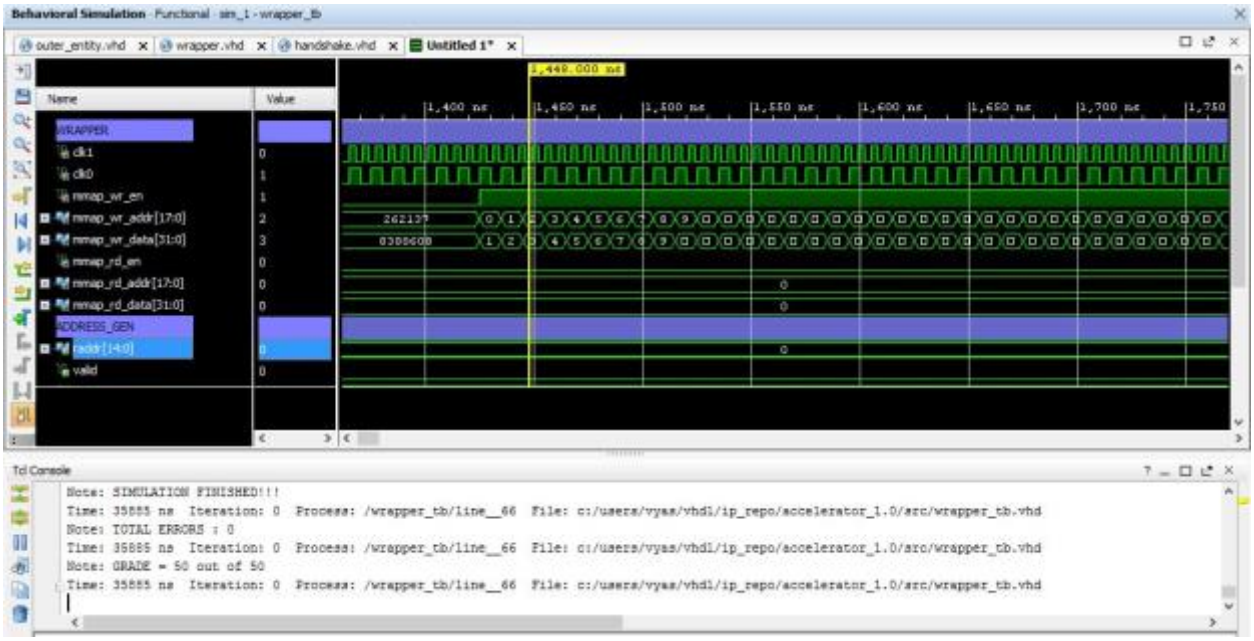
Address generator Simulation



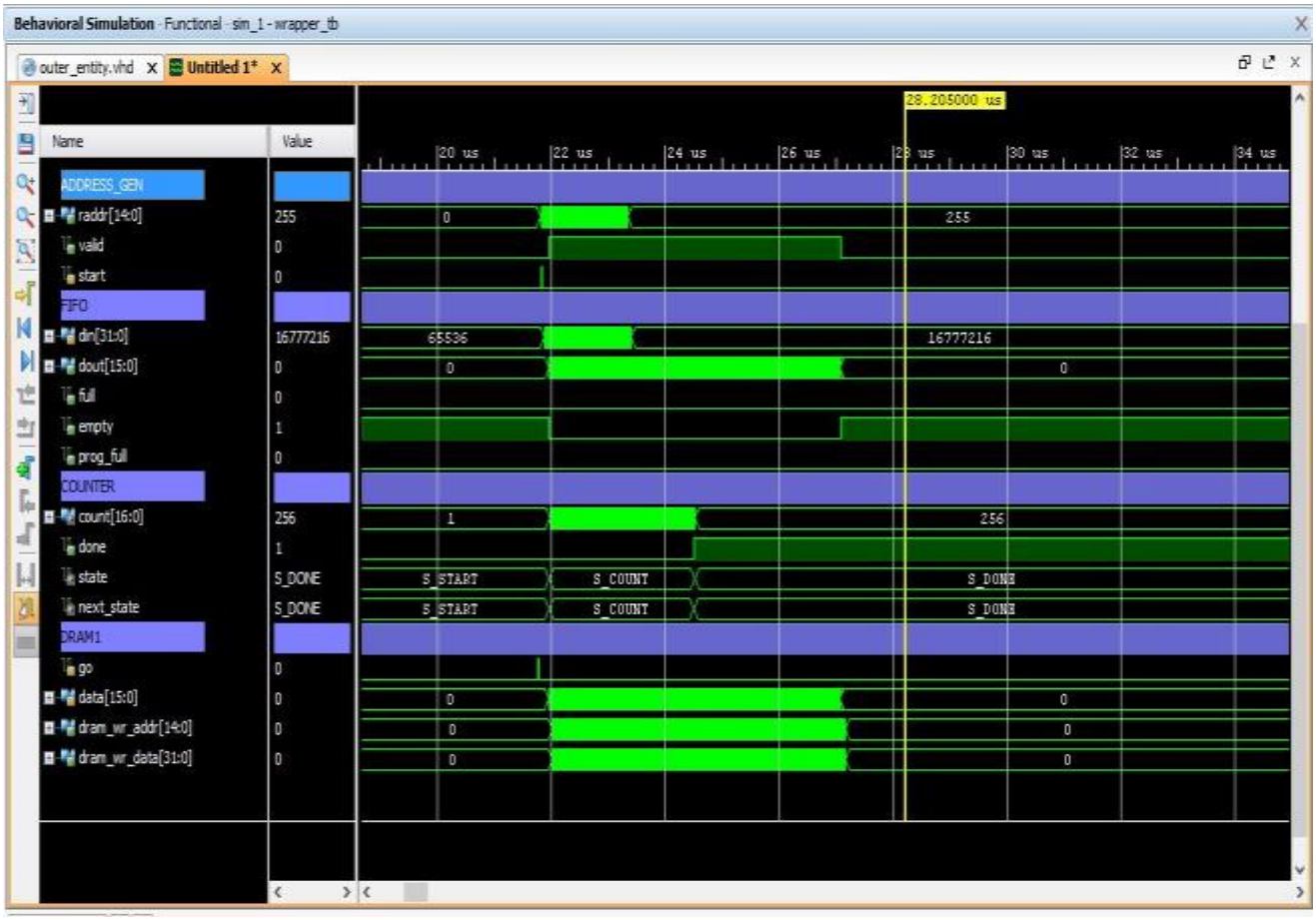
Done Assertion



Handshake synchronizer for DRAM_RD_RAM0



Part-1 Wrapper TB simulation



DRAM0_RD waveform

```
[rcfall2016-033@reconfig FinalProject]$ 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....SUCCESS
Testing max transfer size....SUCCESS
^CTesting random sizes and addresses....(Size = 17403, addr = 6509)[rcfall2016-033@reconfig FinalProject]$
```

Test on Zedboard for Part-1

II. CONVOLUTION PIPELINE

The datapath consists of a pipeline that taken in data of width ***C_KERNEL_WIDTH*** and the number of elements as ***C_KERNEL_SIZE***. The inputs are then devectorized into elements of 16 bits each. The datapath was picked up from the provided code used.

The data path is enabled using the ***DRAM1_READY*** 'ANDED' with the ***NOT EMPTY*** flag signal from the smart buffer. This implies that the datapath is generating new values only when there is a complete window available and the output DRAM is ready to write data.

Steps taken to design and implement datapath:

1. A delay entity was used to drive the valid out signal of the datapath which is then used to drive the DRAM_WR_RAM1.
2. The number of delay cycles are equal to the depth of the adder tree i.e. 7 cycles.
3. The output width of the datapath is 39 of STD_LOGIC_VECTOR type, whereas the output FIFO expects an input of 16 bits.
4. To handle this issue, we check if out of the higher 16 bits, if any of the bits is '1', then we make all the output bits to be 1's, else the output remains as computed.
5. This was done by checking bit '38 downto 16' bits of the output of datapath for any '1s'.

III. SIGNAL BUFFER

In continuation to the second part of the project, we designed a signal buffer also referred as smart buffer. This was done with the aim of reducing the memory access to the DRAM and providing a new window to the pipeline every cycle after some initial overhead.

The signal buffer was implemented as a FIFO with similar control signals. The enable to the buffer was given by the FIFO at the output of the DRAM_RD_RAM0. There is an internal register array of size '***C_KERNEL_SIZE***' registers. The received input is then taken in the last indexed array and then shifted to left by 1. Also, we are maintaining a count of registers that not empty. This count is incremented by 1 every time a data element is being read, and decremented every time a window is read out.

The empty flag is set whenever the number of valid registers are less than 128 and full only when the valid registers are 128 and data path is not reading from the buffer. The output of the buffer is ***C_KERNEL_SIZE * C_KERNEL_WIDTH*** bits (eg. 128 * 16 = 2048 bits).

The output of signal buffer is then passed to the datapath which then devectorizes this combined chunk of input into 16-bit values.

Control Signal for synchronization:

The following signals/ signal combinations were used to control and synchronize the entire datapath along with buffers, FIFOs and DRAMs:

DRAM_RD_RAM0
go ← RAM0_RD_GO (memory map)
done ← gets asserted when DRAM_RD_RAM0 finishes generating 'size' amount of data from DRAM
valid ← gets asserted when the address generator of DRAM_RD_RAM0 generates 'size' amount memory address

Smart_buffer
wr_en ← RAM0_RD_VALID (from DMA)
rd_en ← (NOT signal_empty_flag) AND RAM1_WR_READY

Kernel_Buffer
Kernel_load ← mmap_kernel_load AND (NOT kernel_full_flag)

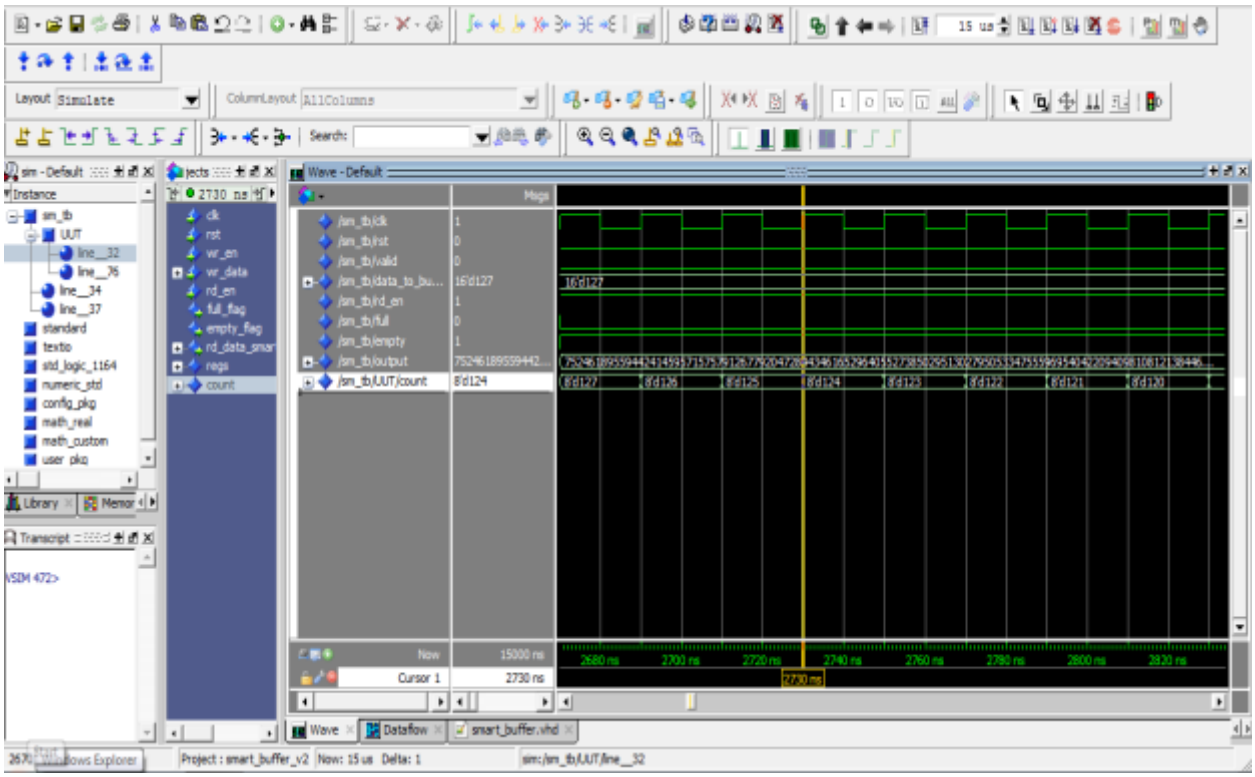
DRAM_WR_RAM1
go ← RAM0_RD_GO (memory map)
done ← gets asserted when DRAM_WR_RAM1 finishes writing 'size' amount of data to DRAM1

IV. KERNEL BUFFER

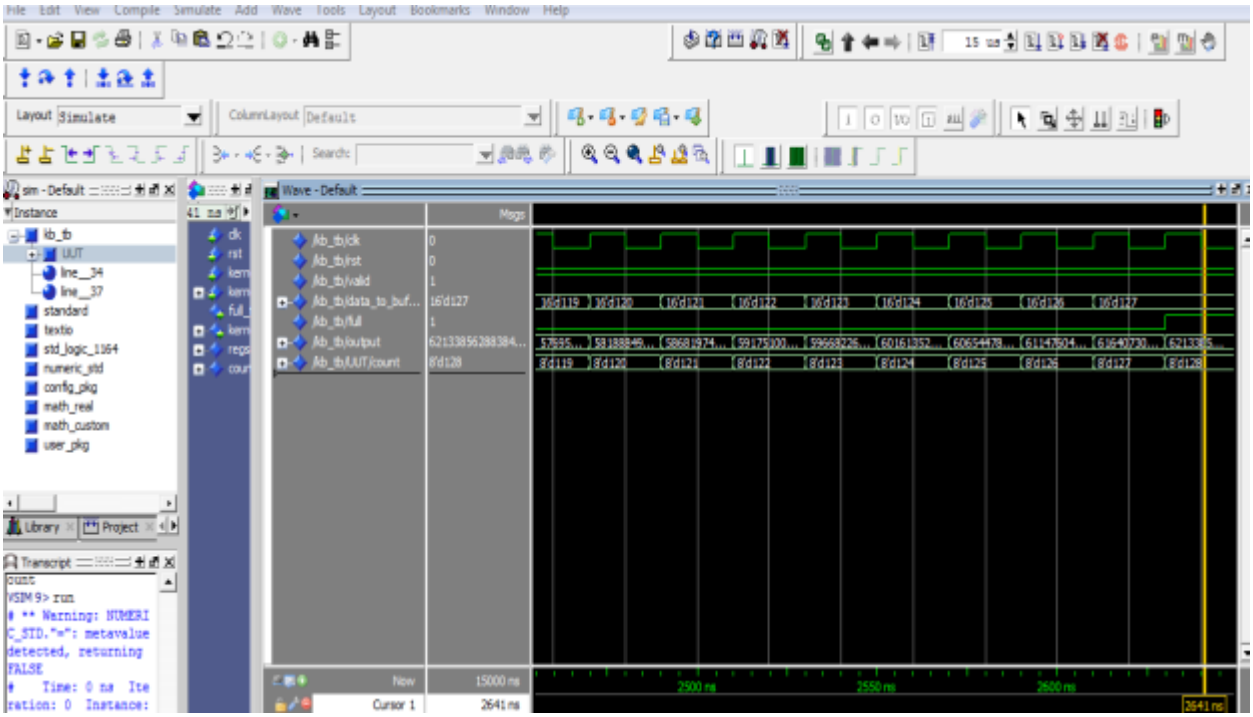
The kernel buffer implementation is similar to signal buffer. It has similar interface with all the control signals except the '***rd_en'*** and '***empty_flag***'. The '***full_flag***' is given to memory map signal named 'kernel loaded' which indicates to the software that all the elements are filled up in the kernel buffer.

The only difference between the signal buffer and kernel buffer is that kernel buffer does not implement a sliding window of data. The data does not change for the elements in the kernel buffer once it has been loaded.

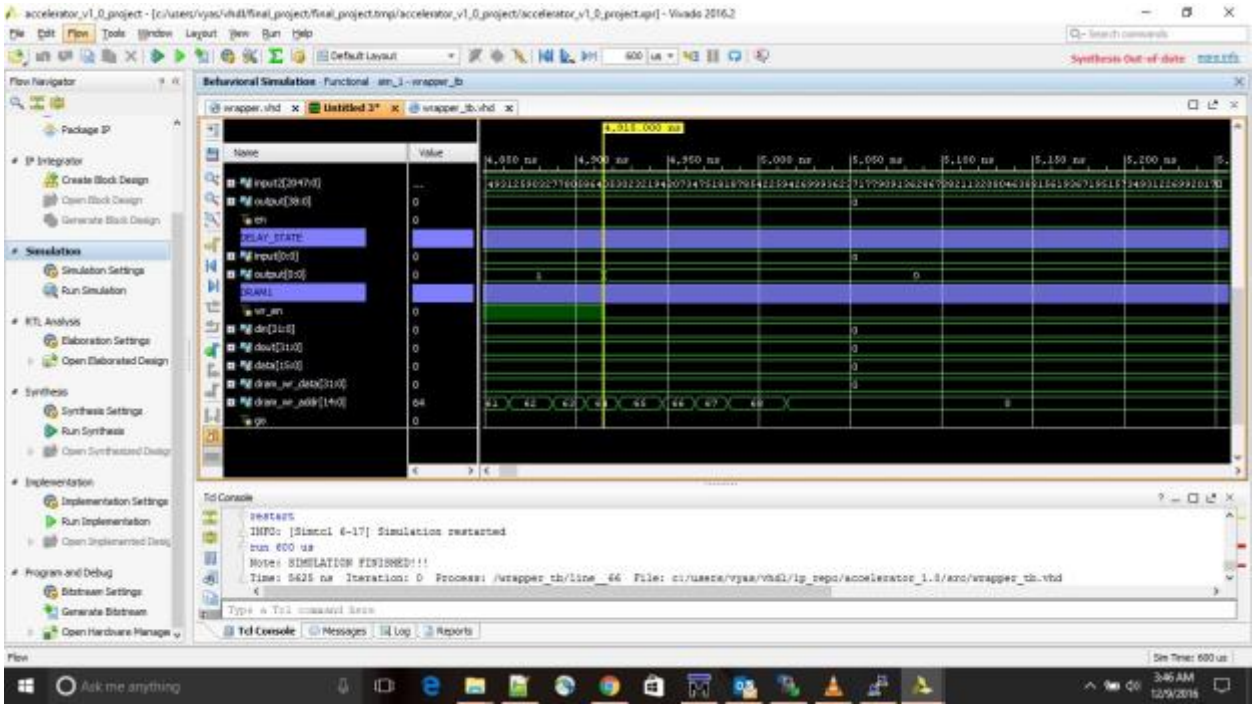
Screenshots:



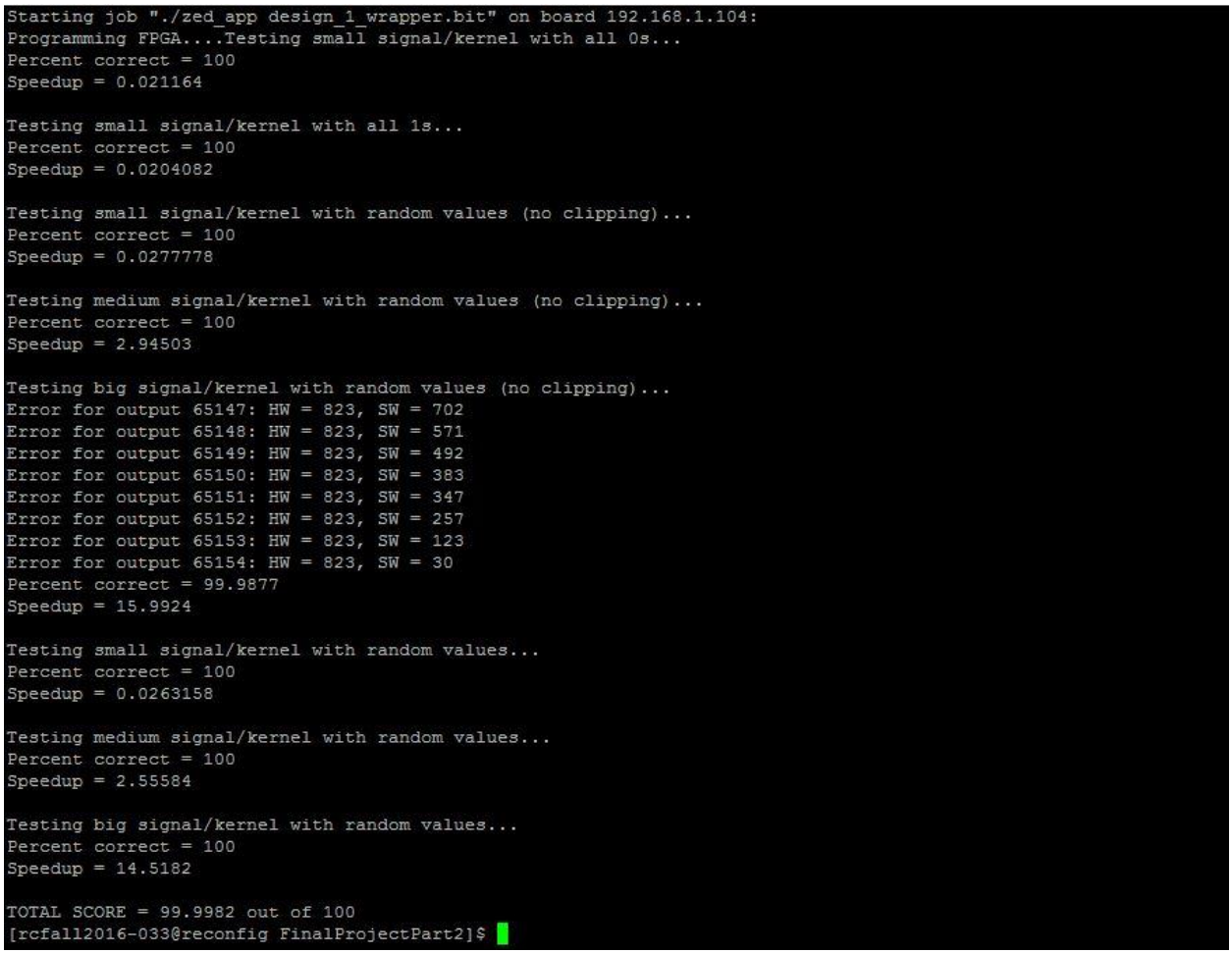
Signal Buffer standalone simulation



Kernel Buffer standalone simulation



Part-2 simulation



Final output of convolution on zed board

RESULTS

1. We have coded and tested the part-1 of the project and thoroughly tested the DMA interface on the simulator and generated a full score.
2. The DMA interface works on the board for all cases except the last one where it is tested for "*random sizes and addresses*" at address '6509' and size '17403'.
3. Data path code works perfectly after adding the functionality of delay entity.
4. Signal and kernel buffers have been thoroughly tested/ simulated on the simulator.
5. The 2nd, 3rd, and 4th part of the project have been integrated properly and tested on the board.
6. This integrated code works on the board except for last few cases for "Testing big signal/kernel with random values (no clipping)" generating a score of 99.9982 on 100.