1-D Time-Domain Convolution using FPGA

Priyanka Abhijit Tamhankar
UFID - 40893970
Department of Electrical and Computer Engineering
University of Florida
Gainesville FL, USA
ptamhankar@ufl.edu

Meghana Koduru
UFID - 64766702
Department of Electrical and Computer Engineering
University of Florida
Gainesville FL, USA
meghanakoduru@ufl.edu

***Abstract* – In this project a FPGA is programmed to perform 1-D time domain convolution using VHDL. It consists of two major parts to be completed which were the DMA interface between memory(DRAM) and the user_app and the design oof the signal and kernel buffers to perform convolution.**

I. INTRODUCTION:

Time domain convolution is a very common operation in image and digital signal processing applications. Convolution function takes in two signals, one of which is the input signal and the other is the filter function or kernel function. Convolution represents how the input varies as it's passed through or slide through the filter function. As this function has a series of repetitive multiplications and additions, it has a scope for huge amount of parallelism and hence, can be implemented on FPGA to achieve good performance.

II. DESIGN AND IMPLEMENTATION:

A. DMA INTERFACE :

The DMA controller serves the purpose of passing blocks of data between input/output device ports without minimal intervention of processor. In this project, A DMA controller is designed to read data from the DRAM (RAM0) and pass it to the USER_APP interface for computation. It has to serve various purposes like generating addresses to indicate which memory address the data has to be read from and synchronization of signals as the data has to passed across different clock domains. The DRAM is running at a frequency of 133Mhz and the USER_APP is running at a frequency of 100 Mhz.

1. FIFO BUFFER

The data read from the DRAM has to be passed through the DMA entity and it's read by USER_APP for computation. The FIFO acts as a buffer to synchronize data that is read from and memory and written into the pipeline at different clock frequencies. It serves the purpose of a buffer and also avoids metastability to occur. FIFO is used here as this is a case where high throughput is a requirement. The FIFO is designed to read a 32 bit word every time there is a valid address generated from the address generator and the read enable of the DRAM is enabled, and output a 16 bit word to USER_APP every time read enable is asserted from USER_APP.

2. ADDRESS GENERATOR

When go signal of DMA interface is asserted, it triggers the address generator to produce addresses starting from the input START_ADDRESS until SIZE addresses are generated. The address generation needs to be stalled whenever the capacity of FIFO is full or whenever The DRAM_READY signal is zero which signals that there is no valid data in DRAM to be read. One important thing that needs to be considered is that the size from the USER_APP corresponds to 16 bit words where as the one address in DRAM corresponds to a 32 bit word. Hence only size/2 addresses need to be generated for reading 'size' number of 16-bit data words.

3. HANDSHAKE SYNCHRONIZER

The signal that triggers the address generator to start generation of addresses is in a different clock domain than the USER_APP, hence it should be synchronized or else metastability can occur. A handshake synchronizer is used between the address generator and the USER_APP interface of the DMA. The size and start address values are stored in registers and passed to the address generator inputs only when the synchronized go triggers address generation.

4. COUNTER
The counter indicates that size data bits have been read from FIFO and DMA operation is completed for the particular case. The counter will count up to size every time valid data is being read from FIFO and the FIFO is not empty. After the count is incremented till size, done is asserted indicating that all the expected data has being read from FIFO.

5. REGISTERS, AND and NOT gates
The register entities are instantiated to the store the values of size and start addresses until the go signal from the USER_APP interfaced in the DRAM clock domain. The AND and NOT gates are used to add additional logic for the flush signal of the DRAM and empty signal of FIFO

B. USERAPP

It consists of three components:
1. convolution pipeline
2. signal buffer
3. kernel buffer

1. CONVOLUTION PIPELINE:
Convolution pipeline is a multiplier-adder tree which takes two arrays of inputs. One input to the pipeline is the output from the signal buffer whereas the other is the output from the kernel buffer. The output of the pipeline is obtained by sum of products of multiplying kernel and signal elements. It consists of 128 multipliers and 127 adders in the adder tree. Inputs from kernel buffer stay the same throughout the process whereas from signal buffer, they shift by one as it acts as a sliding window.
We made sure to enable the pipeline only when RAM1 is ready.

MultAddTree_En 🡐 ram1_wr_ready

Convolution code is obtained from the provided code in which we made a few modifications:
a. DELAY LOGIC
The output of the datapath goes to the RAM1 entity and hence there should be a way to know when the output of the pipeline is valid. For this reason, valid_in and valid_out are added. valid_in is asserted whenever signal buffer rd_en is '1'. When the data is valid and RAM1_wr is ready, then the output will be written to RAM1. Valid_out when asserted informs RAM1 that the data being sent is valid.

MultAddTree_Valin_In 🡐 Signalbuffer_rd_en

ram1_wr_valid 🡐 Datapath_Valid_Out and ram1_wr_ready

Furthermore, Valid_in is delayed by cycles equal to the depth or latency of the datapath.

Latency = 9 cycles(for multipliers + adder tree ----- gets reduced by half every cycle)

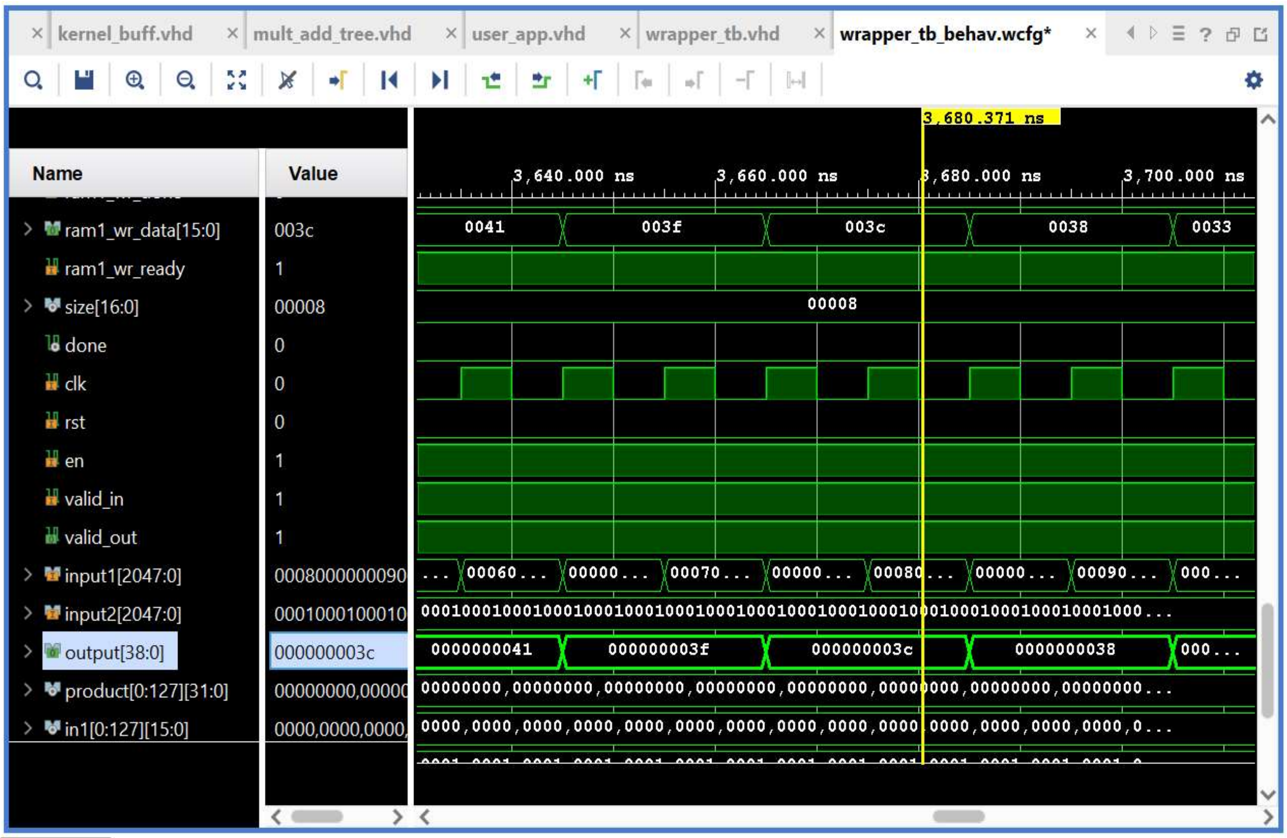
b. CLIPPING LOGIC

The output of the datapath is greater than 16 bits but we need to send 16 bit output to RAM1. For this purpose, clipping logic is created to clip the output whenever it is greater than 16 bits. The output of this logic is then connected the DRAM1 interface. The logic is such that if the overflow bits are 0's then, the output of clipping logic will be the first 16 bits of datapath otherwise the output is all 1's.

The following screenshot shows the clipping logic:

→As indicated by the yellow marker, the output from the datapath is 000000003c. This output is later later clipped to 003c. (ram1_wr_data).

→As seen below, we also made sure that datapath enable is asserted only when ram1_wr_ready is asserted.



Name	Value
ram1_wr_data[15:0]	003c
ram1_wr_ready	1
size[16:0]	00008
done	0
clk	0
rst	0
en	1
valid_in	1
valid_out	1
input1[2047:0]	0008000000090
input2[2047:0]	0001000100010
output[38:0]	000000003c
product[0:127][31:0]	00000000,00000
in1[0:127][15:0]	0000,0000,0000,

Fig 1: Datapath clipping logic

2. <u>SIGNAL BUFFER:</u>

The input to this entity comes from RAM0. This consists of 128 16-bit registers. In every cycle, the elements will be shifted and one new input arrives and thus has the sliding window ability which is advantageous for convolution as it has huge overlaps. Count signal is added to keep a count of number of elements. We have given Full and Empty flags such that the signal buffer is full whenever there are 128 elements in the registers and it is empty when count < 128.
We made sure to assert the signal buffer wr_en only when RAM1 rd_en is asserted that is when there is valid output from RAM0 and signal buffer is not full. Also, rd_en is asserted only when ram1 is ready, signal buffer is not empty and kernel is loaded.
Signal_rd_en 🡐 (not Signal_Empty) and ram1_wr_ready and kernel_loaded

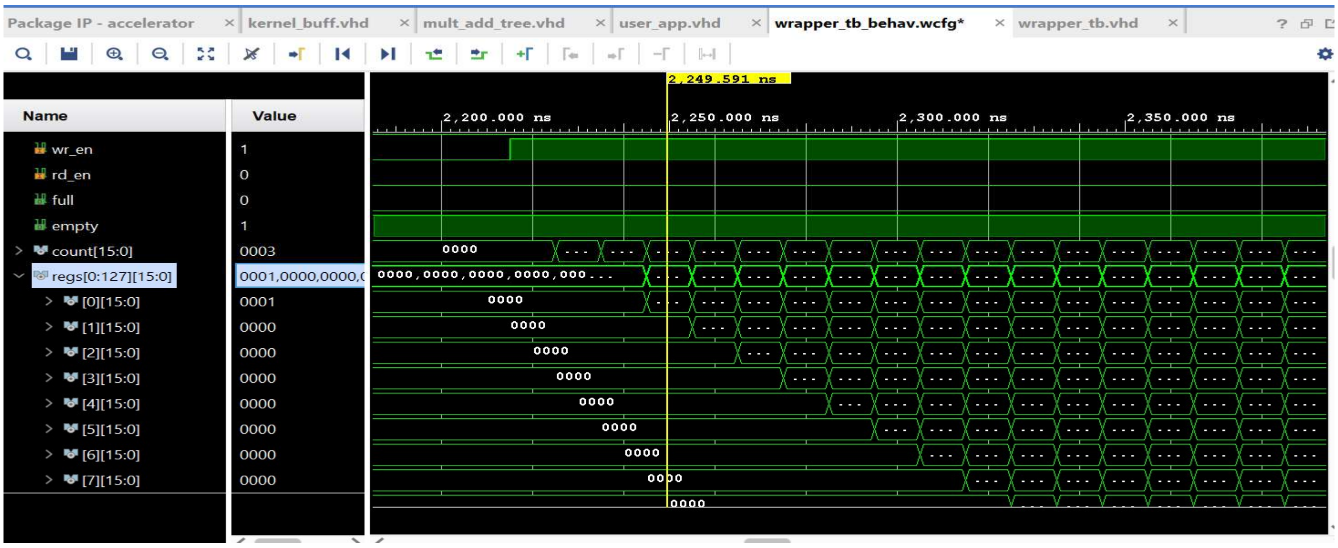
ram0_rd_en_signal 🡐 ram0_rd_valid and (not Signal_Full)

The output of the signal buffer then serves as one of the inputs to convolution pipeline.

For test size : 8 ;

The following screenshots below show the shifting of inputs in the registers :

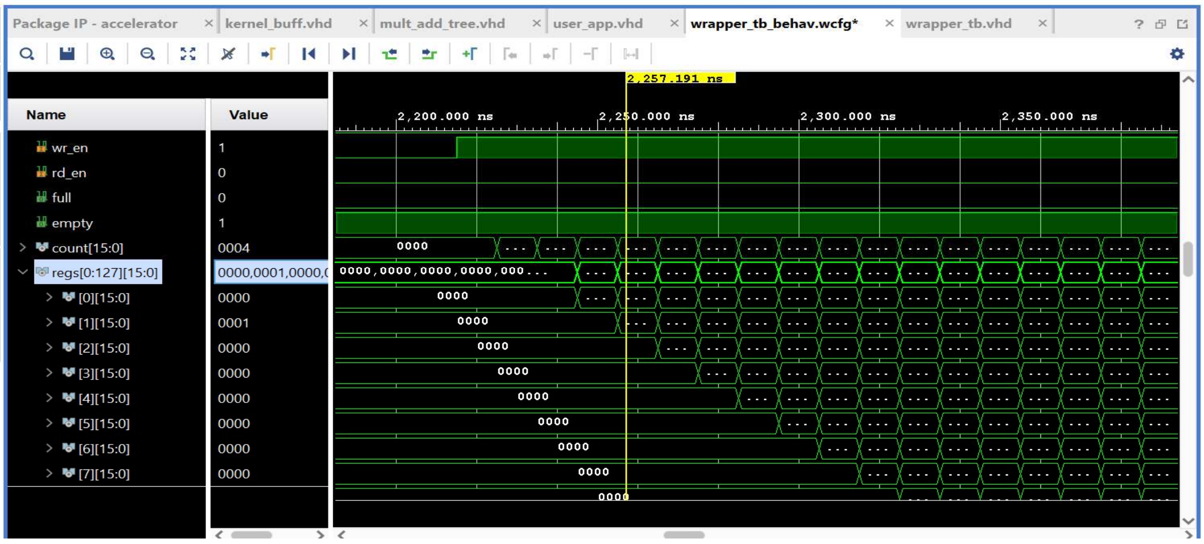
(i) Input 0001 is stored in register 0.



Name	Value
wr_en	1
rd_en	0
full	0
empty	1
count[15:0]	0003
regs[0:127][15:0]	0001,0000,0000,(
[0][15:0]	0001
[1][15:0]	0000
[2][15:0]	0000
[3][15:0]	0000
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Fig 2: Signal Buffer functioning

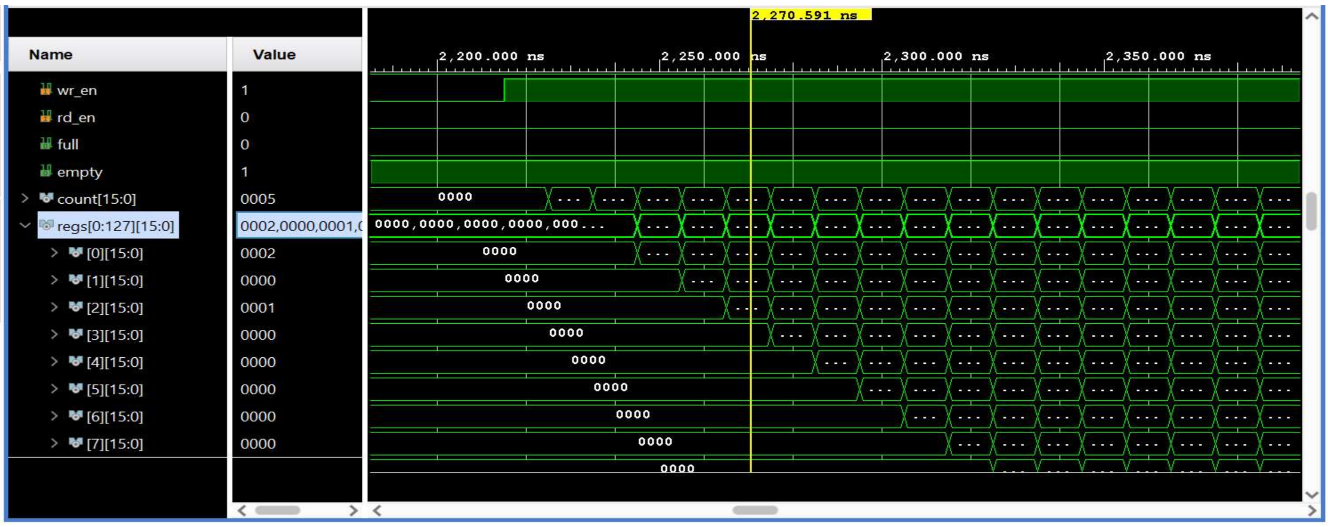
(ii) Input 0001 from previous register is shifted to next register and a new input arrives



Name	Value
wr_en	1
rd_en	0
full	0
empty	1
count[15:0]	0004
regs[0:127][15:0]	0000,0001,0000,(
[0][15:0]	0000
[1][15:0]	0001
[2][15:0]	0000
[3][15:0]	0000
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Fig 3: Signal Buffer functioning

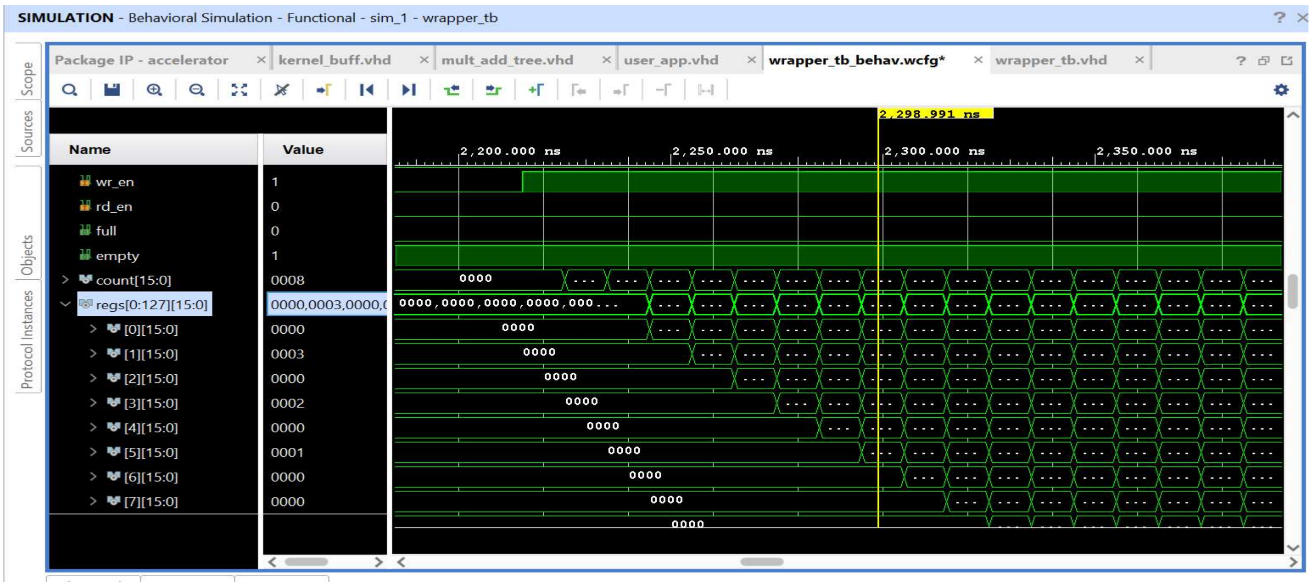
(iii) The same way shifting happens and 0002 is stored



Name	Value
wr_en	1
rd_en	0
full	0
empty	1
count[15:0]	0005
regs[0:127][15:0]	0002,0000,0001,(
[0][15:0]	0002
[1][15:0]	0000
[2][15:0]	0001
[3][15:0]	0000
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Fig 4: Signal Buffer functioning

(iv) Previous inputs are shifted and new element 0003 is stored. This goes on till 000b (12) and then padding occurs.

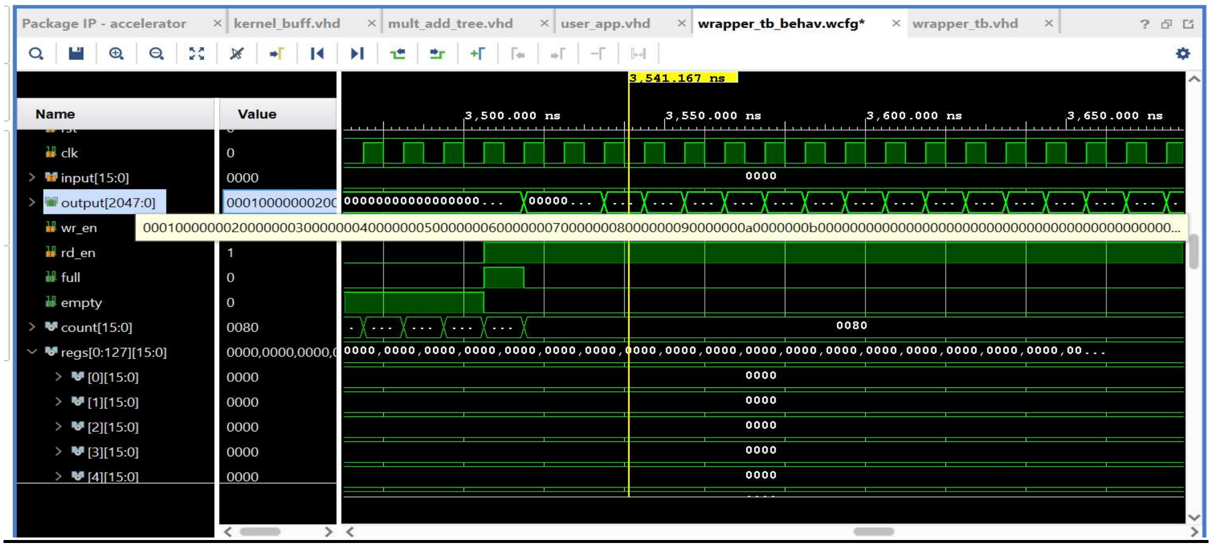


Name	Value
wr_en	1
rd_en	0
full	0
empty	1
count[15:0]	0008
regs[0:127][15:0]	0000,0003,0000,(
[0][15:0]	0000
[1][15:0]	0003
[2][15:0]	0000
[3][15:0]	0002
[4][15:0]	0000
[5][15:0]	0001
[6][15:0]	0000
[7][15:0]	0000

Fig 5: Signal Buffer functioning

The following screenshots show the output of the signal buffer :

(i) The outputs will be read as shown below from 0001 to 000b along with 0's

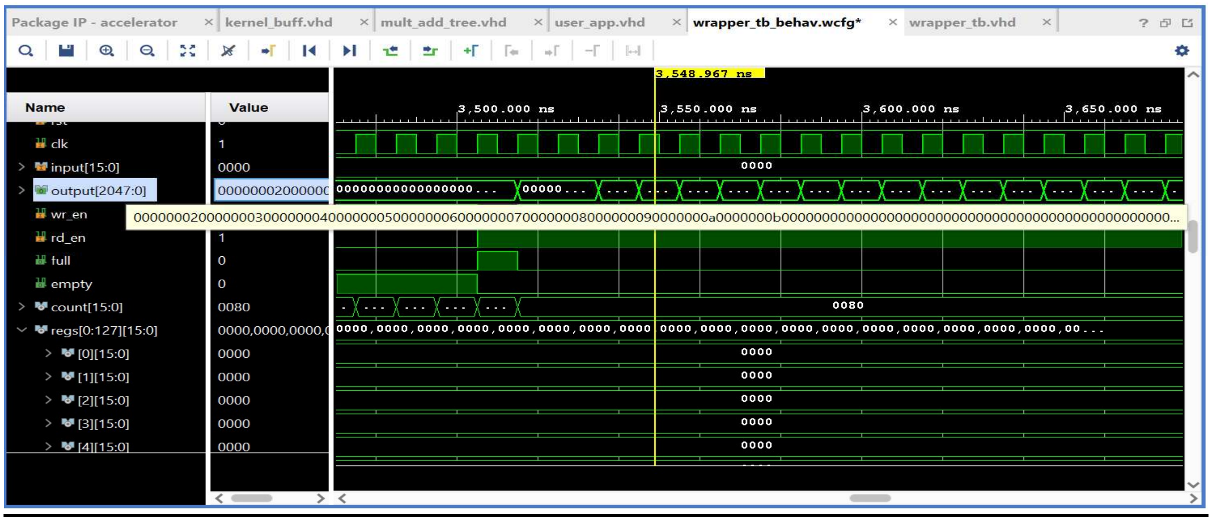


```
Name                Value
clk                 0
input[15:0]         0000
output[2047:0]      00010000000200
wr_en
rd_en               1
full                0
empty               0
count[15:0]         0080
regs[0:127][15:0]   0000,0000,0000,
  [0][15:0]         0000
  [1][15:0]         0000
  [2][15:0]         0000
  [3][15:0]         0000
  [4][15:0]         0000
```

```
000100000002000000030000000400000005000000060000000700000008000000090000000a0000000b000000000000000000000000000000000000000000000...
```

Fig 6: Signal Buffer Output

(ii) Then in the next step 0001 is discarded. In the later step, 0002 will be discarded and so on.



```
Name                Value
clk                 1
input[15:0]         0000
output[2047:0]      00000002000000
wr_en
rd_en               1
full                0
empty               0
count[15:0]         0080
regs[0:127][15:0]   0000,0000,0000,
  [0][15:0]         0000
  [1][15:0]         0000
  [2][15:0]         0000
  [3][15:0]         0000
  [4][15:0]         0000
```

```
00000002000000030000000400000005000000060000000700000008000000090000000a0000000b0000000000000000000000000000000000000000000000000...
```

Fig 7: Signal Buffer Output

3. <u>KERNEL BUFFER</u>

The input to this entity comes from memory map. This also consists of 128 16-bit registers. The elements are shifted and new inputs arrive till the count becomes 128. The difference between signal buffer and kernel is that once the inputs are stored in 128 registers, it won't accept or need new elements. In other words, kernel doesn't have sliding window functionality. We designed in such a way that kernel full flag is asserted when count =128, i.e., there are 128 elements stored in registers otherwise it is not full.

The output of the kernel buffer further serves as another input to convolution pipeline.

The following screenshot shows the input and output of kernel buffer :

Inputs are stored and gets shifted one by one till all the 128 registers are filled.

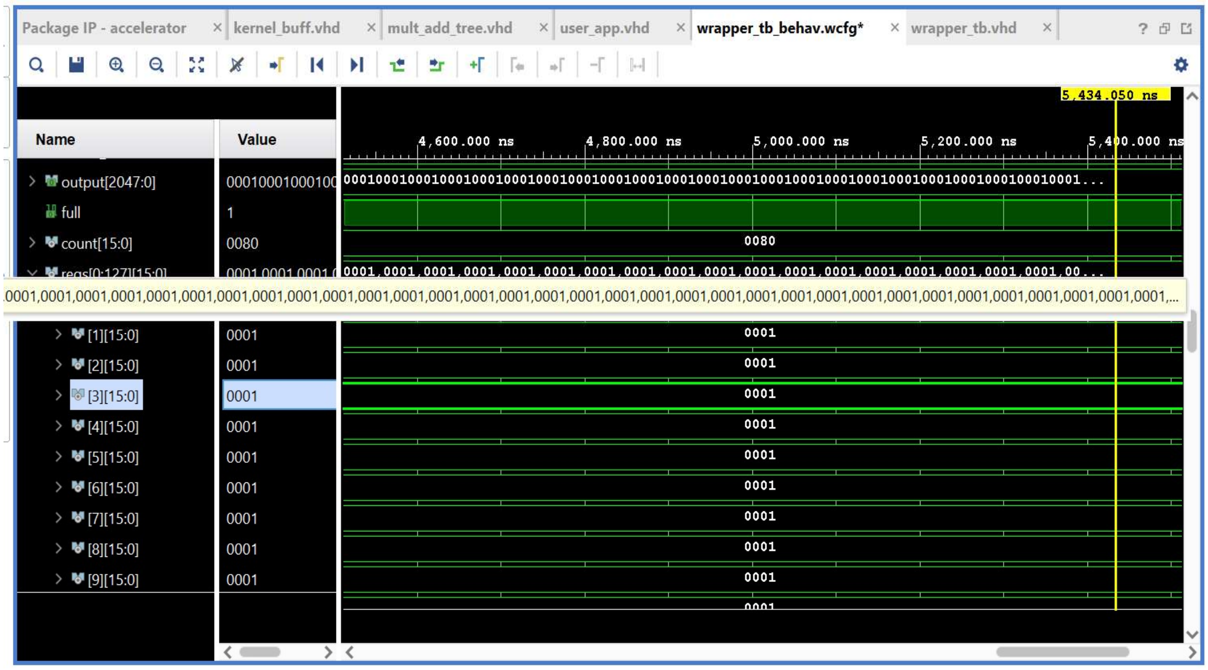


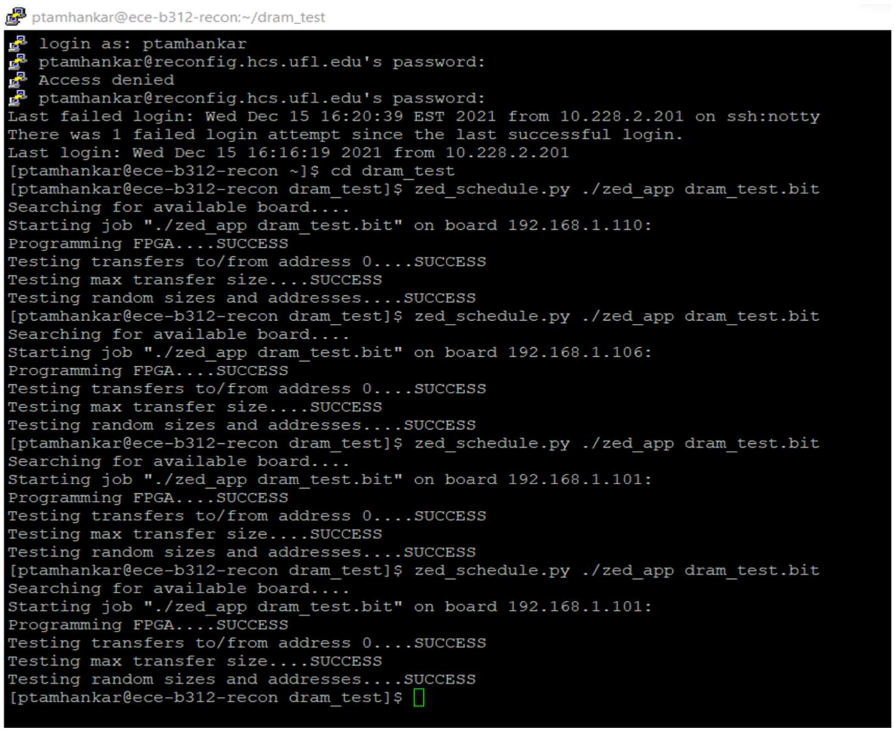
Fig 8: Kernel Buffer

III. RESULTS:

A. DMA INTERFACE

The DMA interface is fully functional on both simulation and FPGA for all test cases.

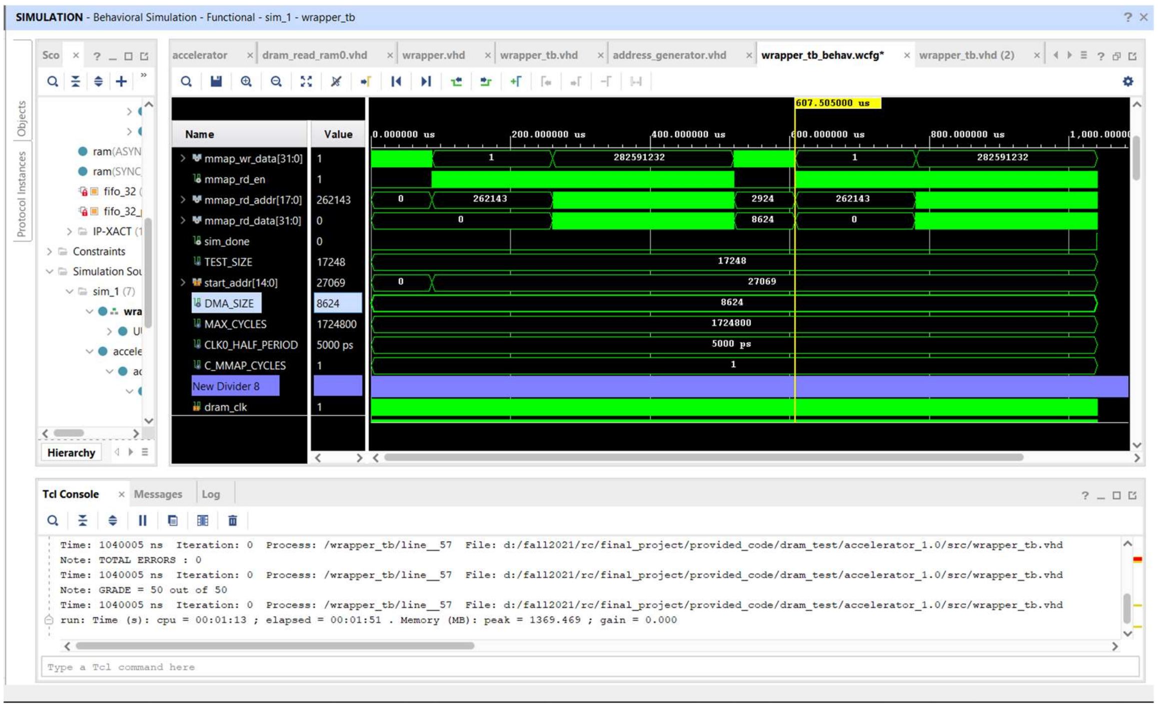
Hardware output:



```
ptamhankar@ece-b312-recon:~/dram_test
login as: ptamhankar
ptamhankar@reconfig.hcs.ufl.edu's password:
Access denied
ptamhankar@reconfig.hcs.ufl.edu's password:
Last failed login: Wed Dec 15 16:20:39 EST 2021 from 10.228.2.201 on ssh:notty
There was 1 failed login attempt since the last successful login.
Last login: Wed Dec 15 16:16:19 2021 from 10.228.2.201
[ptamhankar@ece-b312-recon ~]$ cd dram_test
[ptamhankar@ece-b312-recon dram_test]$ zed_schedule.py ./zed_app dram_test.bit
Searching for available board....
Starting job "./zed_app dram_test.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....SUCCESS
[ptamhankar@ece-b312-recon dram_test]$ zed_schedule.py ./zed_app dram_test.bit
Searching for available board....
Starting job "./zed_app dram_test.bit" on board 192.168.1.106:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....SUCCESS
[ptamhankar@ece-b312-recon dram_test]$ zed_schedule.py ./zed_app dram_test.bit
Searching for available board....
Starting job "./zed_app dram_test.bit" on board 192.168.1.101:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....SUCCESS
[ptamhankar@ece-b312-recon dram_test]$ zed_schedule.py ./zed_app dram_test.bit
Searching for available board....
Starting job "./zed_app dram_test.bit" on board 192.168.1.101:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....SUCCESS
[ptamhankar@ece-b312-recon dram_test]$
```

Fig 9: Hardware output for DMA interface

<u>Simulation results for different test sizes and start addresses</u> :

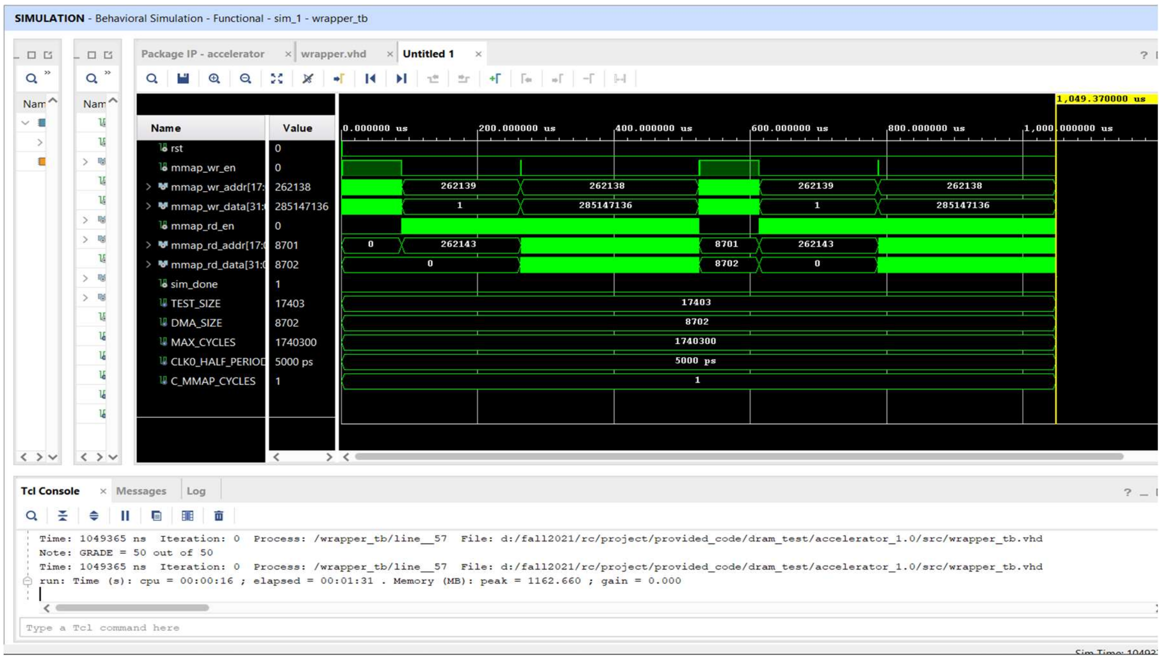


SIMULATION - Behavioral Simulation - Functional - sim_1 - wrapper_tb

Name	Value
mmap_wr_data[31:0]	1
mmap_rd_en	1
mmap_rd_addr[17:0]	262143
mmap_rd_data[31:0]	0
sim_done	0
TEST_SIZE	17248
start_addr[14:0]	27069
DMA_SIZE	8624
MAX_CYCLES	1724800
CLK0_HALF_PERIOD	5000 ps
C_MMAP_CYCLES	1
New Divider 8	
dram_clk	1

```
Time: 1040005 ns  Iteration: 0  Process: /wrapper_tb/line__57  File: d:/fall2021/rc/final_project/provided_code/dram_test/accelerator_1.0/src/wrapper_tb.vhd
Note: TOTAL ERRORS : 0
Time: 1040005 ns  Iteration: 0  Process: /wrapper_tb/line__57  File: d:/fall2021/rc/final_project/provided_code/dram_test/accelerator_1.0/src/wrapper_tb.vhd
Note: GRADE = 50 out of 50
Time: 1040005 ns  Iteration: 0  Process: /wrapper_tb/line__57  File: d:/fall2021/rc/final_project/provided_code/dram_test/accelerator_1.0/src/wrapper_tb.vhd
run: Time (s): cpu = 00:01:13 ; elapsed = 00:01:51 . Memory (MB): peak = 1369.469 ; gain = 0.000
```

Fig 10: Simulation output for DMA interface with test size 17248 and address 27069



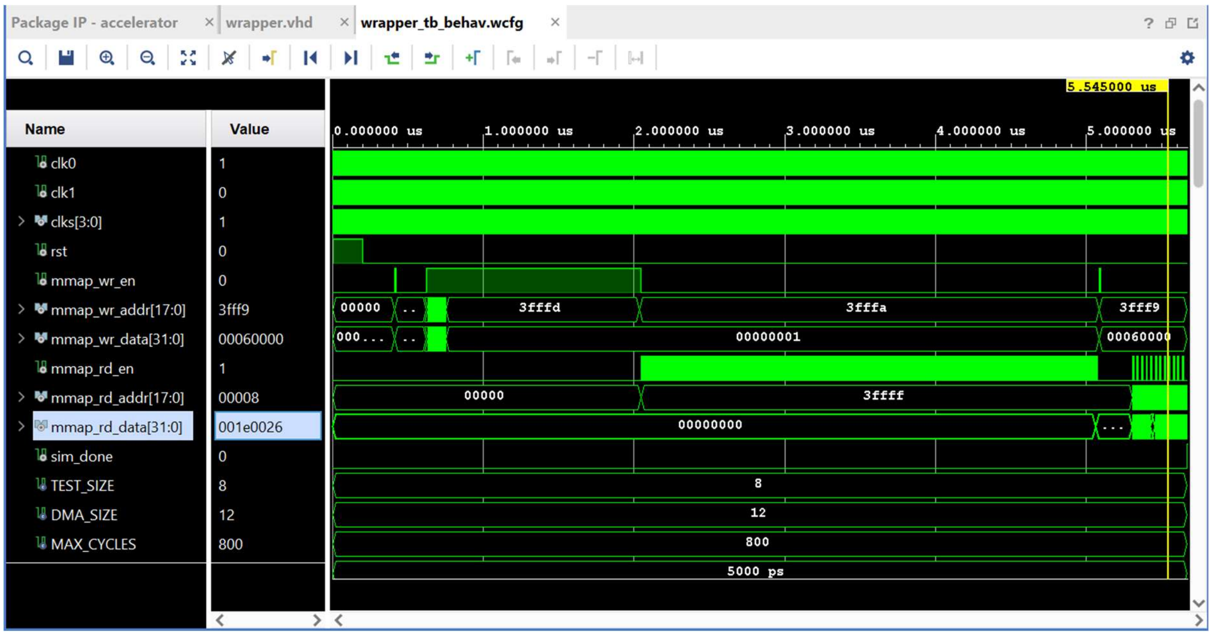
SIMULATION - Behavioral Simulation - Functional - sim_1 - wrapper_tb

Name	Value
rst	0
mmap_wr_en	0
mmap_wr_addr[17:	262138
mmap_wr_data[31:	285147136
mmap_rd_en	0
mmap_rd_addr[17:	8701
mmap_rd_data[31:	8702
sim_done	1
TEST_SIZE	17403
DMA_SIZE	8702
MAX_CYCLES	1740300
CLK0_HALF_PERIOD	5000 ps
C_MMAP_CYCLES	1

```
Time: 1049365 ns  Iteration: 0  Process: /wrapper_tb/line__57  File: d:/fall2021/rc/project/provided_code/dram_test/accelerator_1.0/src/wrapper_tb.vhd
Note: GRADE = 50 out of 50
Time: 1049365 ns  Iteration: 0  Process: /wrapper_tb/line__57  File: d:/fall2021/rc/project/provided_code/dram_test/accelerator_1.0/src/wrapper_tb.vhd
run: Time (s): cpu = 00:00:16 ; elapsed = 00:01:31 . Memory (MB): peak = 1162.660 ; gain = 0.000
```

Fig 11: Simulation output for DMA interface with test size 17403 and address 6509

B. USERAPP



Name	Value
clk0	1
clk1	0
clks[3:0]	1
rst	0
mmap_wr_en	0
mmap_wr_addr[17:0]	3fff9
mmap_wr_data[31:0]	00060000
mmap_rd_en	1
mmap_rd_addr[17:0]	00008
mmap_rd_data[31:0]	001e0026
sim_done	0
TEST_SIZE	8
DMA_SIZE	12
MAX_CYCLES	800

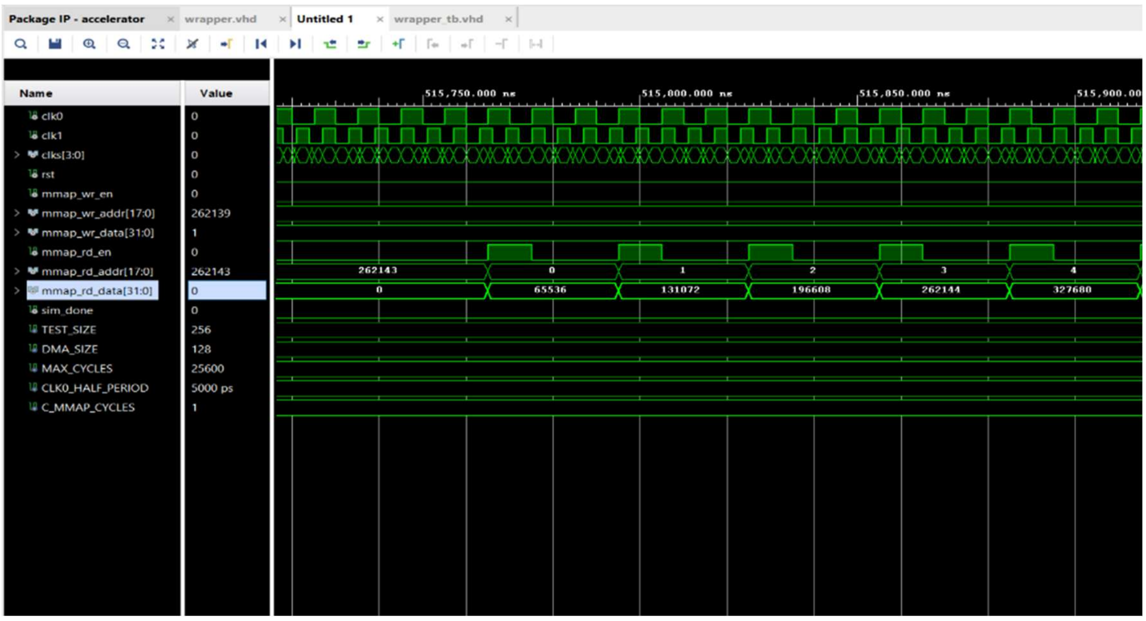
Fig 123: Simulation output for USER_APP for TEST SIZE 8

IV. CHALLENGES:

A. DMA INTERFACE

1. FIFO asserting EMPTY, FULL AND PROGRAMMABLE FULL together
   Solution: ANDing the done and go signals solved the reset behaviour of the FIFO IP from the IP catalogue provided by Vivado.

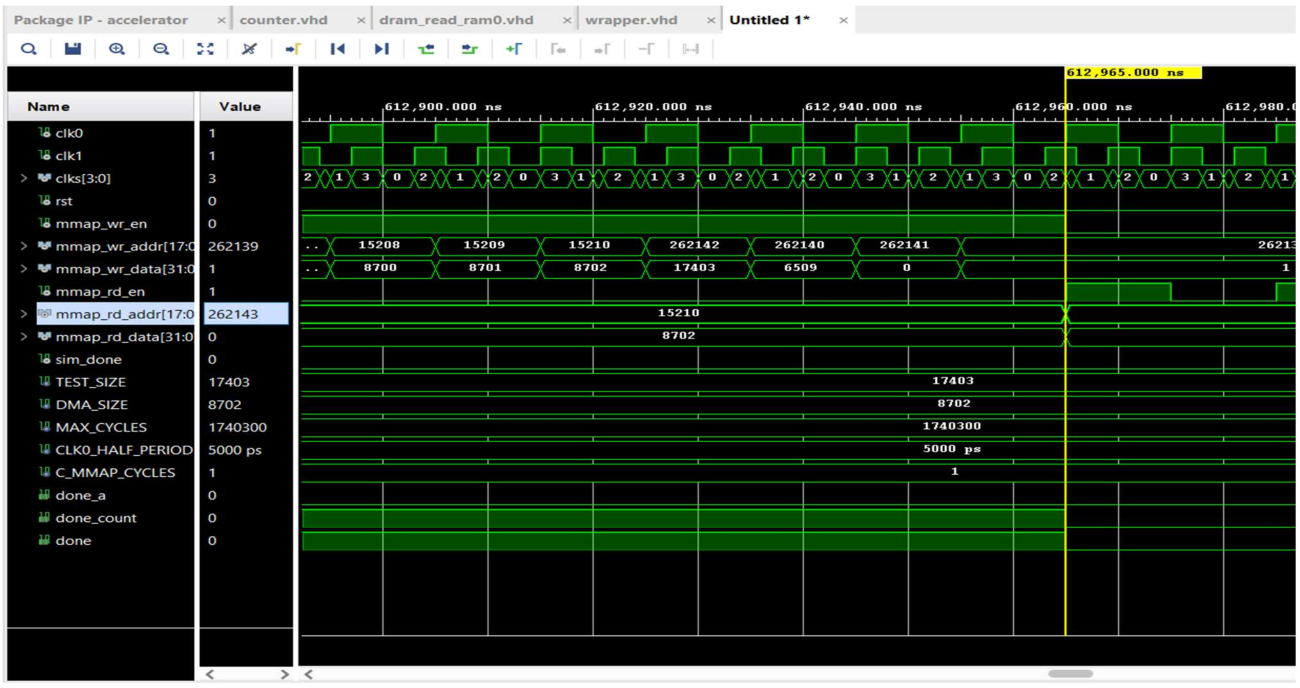
2. FIFO reading inverted bits
   Solution: The FIFO was inverting the two halves of the 32 bit word while reading from external memory. By flipping the two halves of the 32 bit word, correct data was obtained in the buffer



Name	Value
clk0	0
clk1	0
clks[3:0]	0
rst	0
mmap_wr_en	0
mmap_wr_addr[17:0]	262139
mmap_wr_data[31:0]	1
mmap_rd_en	0
mmap_rd_addr[17:0]	262143
mmap_rd_data[31:0]	0
sim_done	0
TEST_SIZE	256
DMA_SIZE	128
MAX_CYCLES	25600
CLK0_HALF_PERIOD	5000 ps
C_MMAP_CYCLES	1

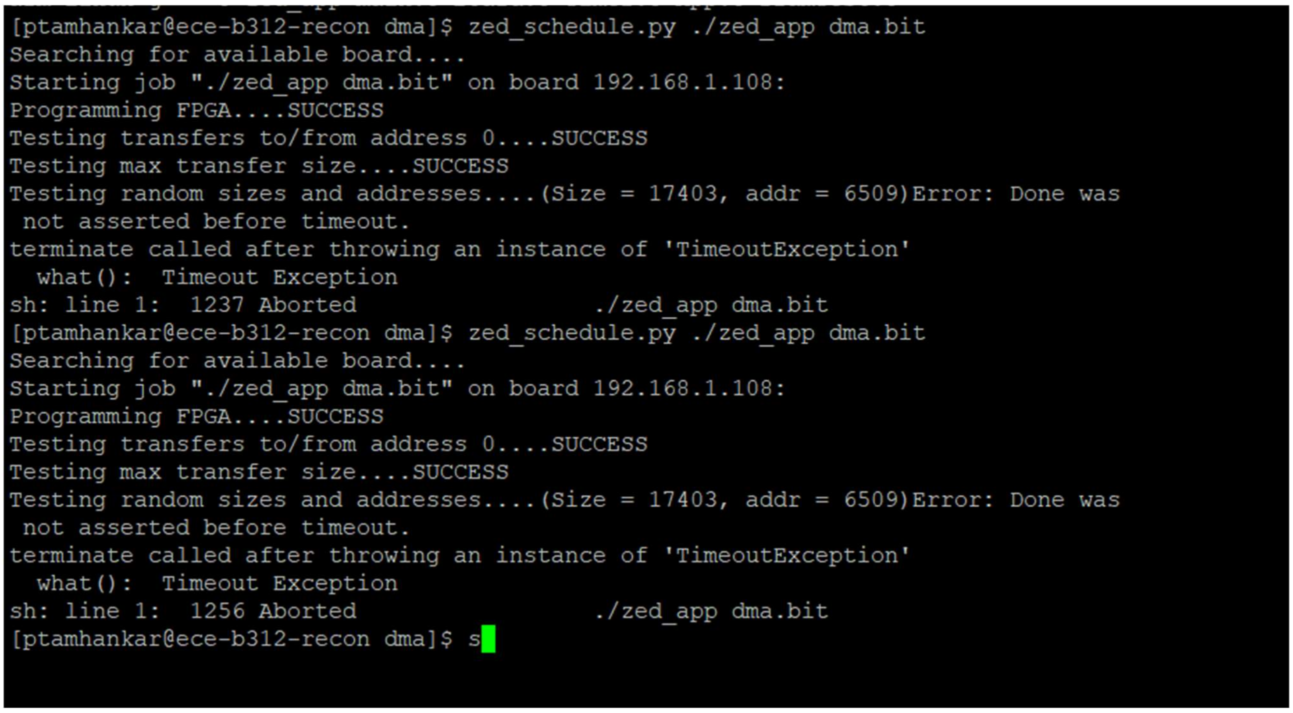
Fig 13: Garbage values in DIN from FIFO due to inverted bits

3. Metastability
   Solution: The mistake that was done was adding 3 registers in the source and destination domain which did not solve metastability. After adding one register in source domain and two registers in destination domain for send_s signal and one register in destination domain and two registers in source domain for ack_s signal, metastability was fixed and done was asserted on time



Name	Value
clk0	1
clk1	1
clks[3:0]	3
rst	0
mmap_wr_en	0
mmap_wr_addr[17:0	262139
mmap_wr_data[31:0	1
mmap_rd_en	1
mmap_rd_addr[17:0	262143
mmap_rd_data[31:0	0
sim_done	0
TEST_SIZE	17403
DMA_SIZE	8702
MAX_CYCLES	1740300
CLK0_HALF_PERIOD	5000 ps
C_MMAP_CYCLES	1
done_a	0
done_count	0
done	0

Fig 14: DONE perfectly asserted on simulation for failed test case on board



```
[ptamhankar@ece-b312-recon dma]$ zed_schedule.py ./zed_app dma.bit
Searching for available board....
Starting job "./zed_app dma.bit" on board 192.168.1.108:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....(Size = 17403, addr = 6509)Error: Done was
 not asserted before timeout.
terminate called after throwing an instance of 'TimeoutException'
  what():  Timeout Exception
sh: line 1:  1237 Aborted                 ./zed_app dma.bit
[ptamhankar@ece-b312-recon dma]$ zed_schedule.py ./zed_app dma.bit
Searching for available board....
Starting job "./zed_app dma.bit" on board 192.168.1.108:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....(Size = 17403, addr = 6509)Error: Done was
 not asserted before timeout.
terminate called after throwing an instance of 'TimeoutException'
  what():  Timeout Exception
sh: line 1:  1256 Aborted                 ./zed_app dma.bit
[ptamhankar@ece-b312-recon dma]$ s
```

Fig 15: Done not being asserted on board hinting towards metastability

4. Address Generator giving timing issues
Solution: Converting it into a FSMD solved most of the timing and logical issues that the previous design (behavioural) was causing.

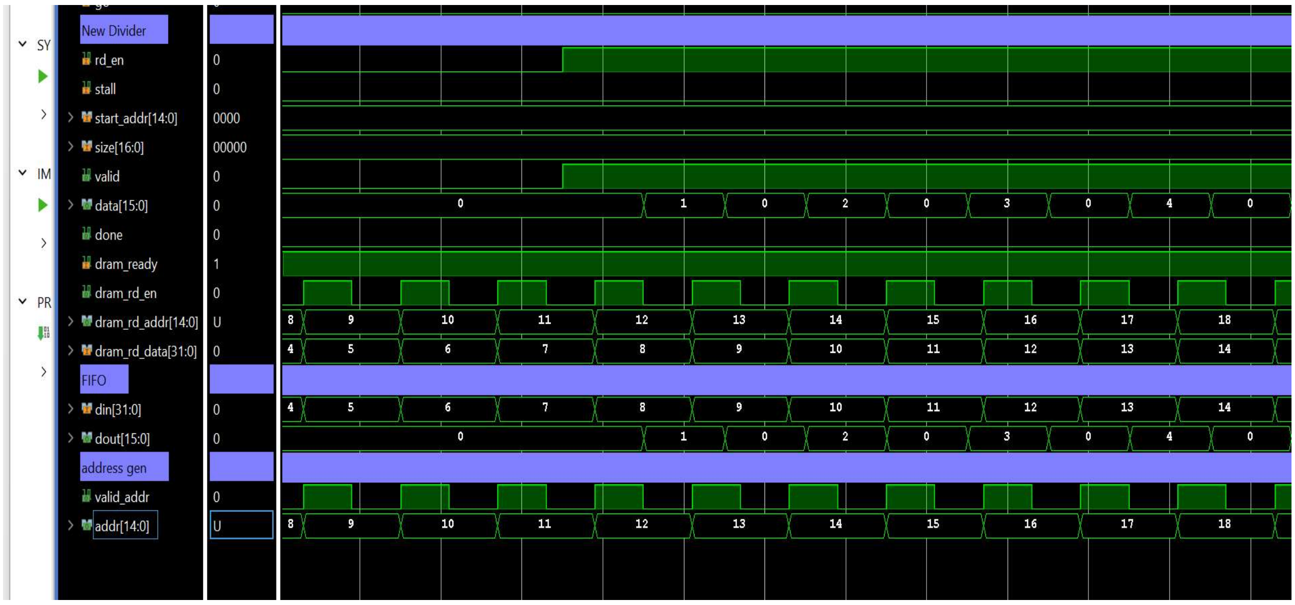
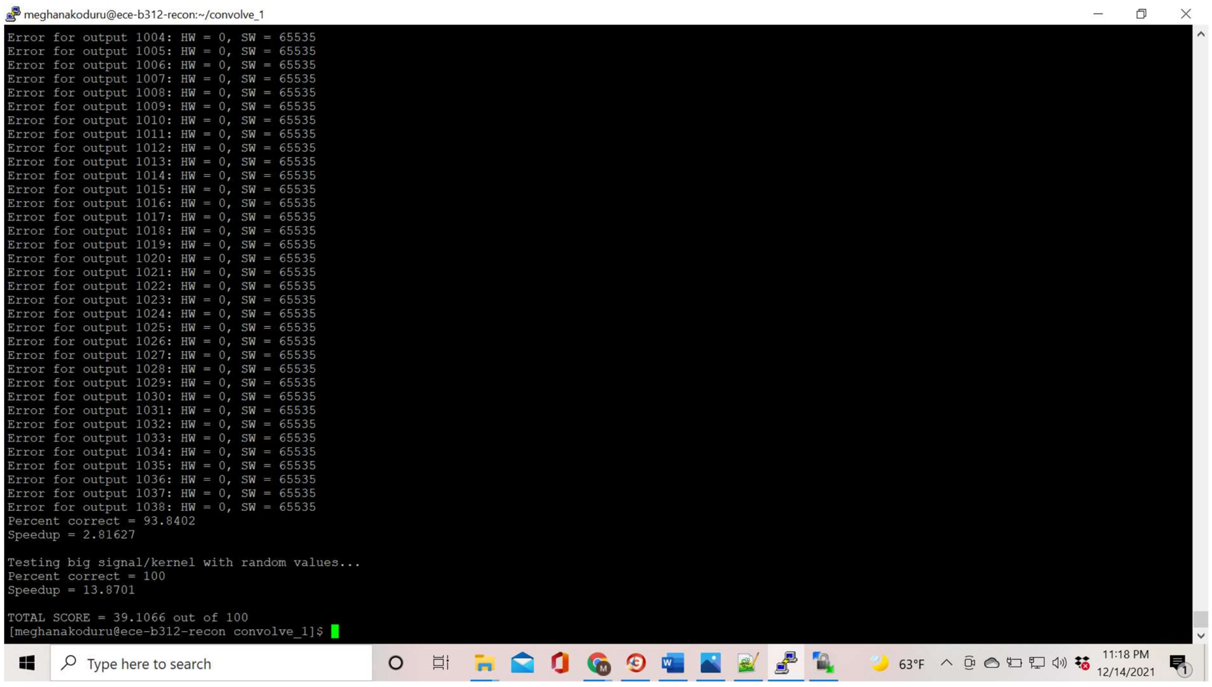


Fig 16: Correct address generator and FIFO functionality

B. USERAPP

(1) When we designed the signal and kernel buffer, we put them in same directions which caused a problem. We later rectified it by reverting signal buffer to make sure they are in opposite directions while performing convolution.

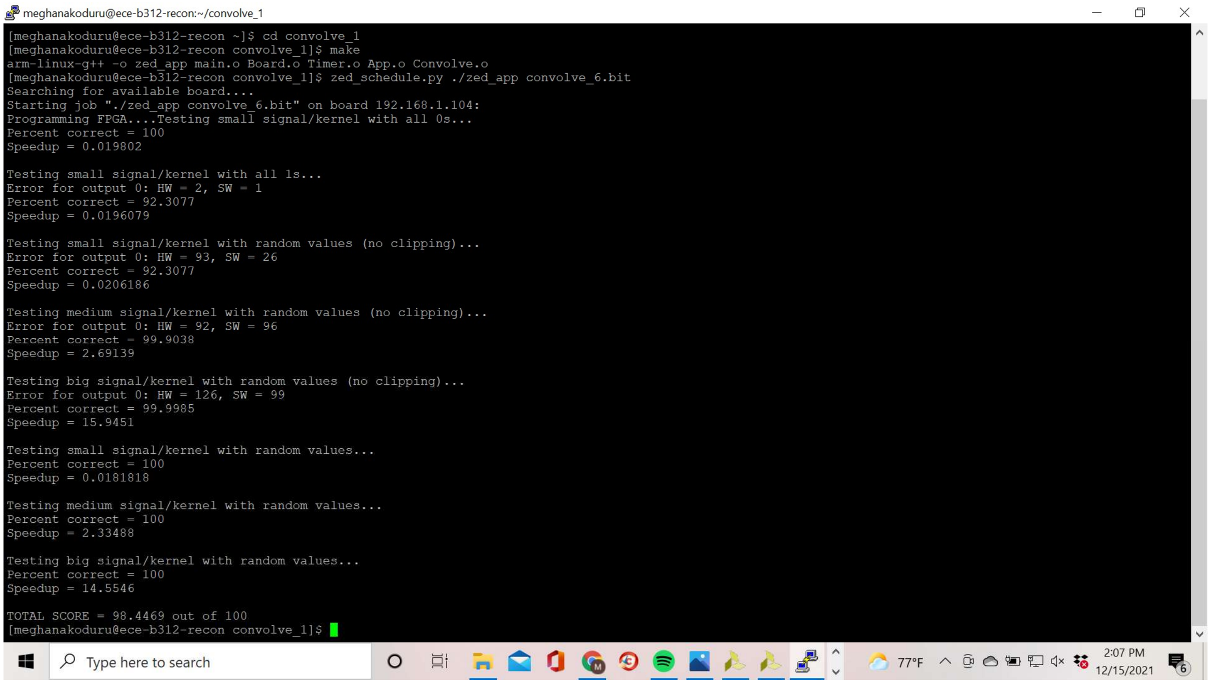
When signal buffer and kernel are in same directions, we got the performance was very low (39.1066)



```
Error for output 1004: HW = 0, SW = 65535
Error for output 1005: HW = 0, SW = 65535
Error for output 1006: HW = 0, SW = 65535
Error for output 1007: HW = 0, SW = 65535
Error for output 1008: HW = 0, SW = 65535
Error for output 1009: HW = 0, SW = 65535
Error for output 1010: HW = 0, SW = 65535
Error for output 1011: HW = 0, SW = 65535
Error for output 1012: HW = 0, SW = 65535
Error for output 1013: HW = 0, SW = 65535
Error for output 1014: HW = 0, SW = 65535
Error for output 1015: HW = 0, SW = 65535
Error for output 1016: HW = 0, SW = 65535
Error for output 1017: HW = 0, SW = 65535
Error for output 1018: HW = 0, SW = 65535
Error for output 1019: HW = 0, SW = 65535
Error for output 1020: HW = 0, SW = 65535
Error for output 1021: HW = 0, SW = 65535
Error for output 1022: HW = 0, SW = 65535
Error for output 1023: HW = 0, SW = 65535
Error for output 1024: HW = 0, SW = 65535
Error for output 1025: HW = 0, SW = 65535
Error for output 1026: HW = 0, SW = 65535
Error for output 1027: HW = 0, SW = 65535
Error for output 1028: HW = 0, SW = 65535
Error for output 1029: HW = 0, SW = 65535
Error for output 1030: HW = 0, SW = 65535
Error for output 1031: HW = 0, SW = 65535
Error for output 1032: HW = 0, SW = 65535
Error for output 1033: HW = 0, SW = 65535
Error for output 1034: HW = 0, SW = 65535
Error for output 1035: HW = 0, SW = 65535
Error for output 1036: HW = 0, SW = 65535
Error for output 1037: HW = 0, SW = 65535
Error for output 1038: HW = 0, SW = 65535
Percent correct = 93.8402
Speedup = 2.81627

Testing big signal/kernel with random values...
Percent correct = 100
Speedup = 13.8701

TOTAL SCORE = 39.1066 out of 100
[meghanakoduru@ece-b312-recon convolve_1]$
```

Fig 17: Previous results before invertin

After rectifying the problem by reverting signal buffer, the performance extensively increased to 98.4469

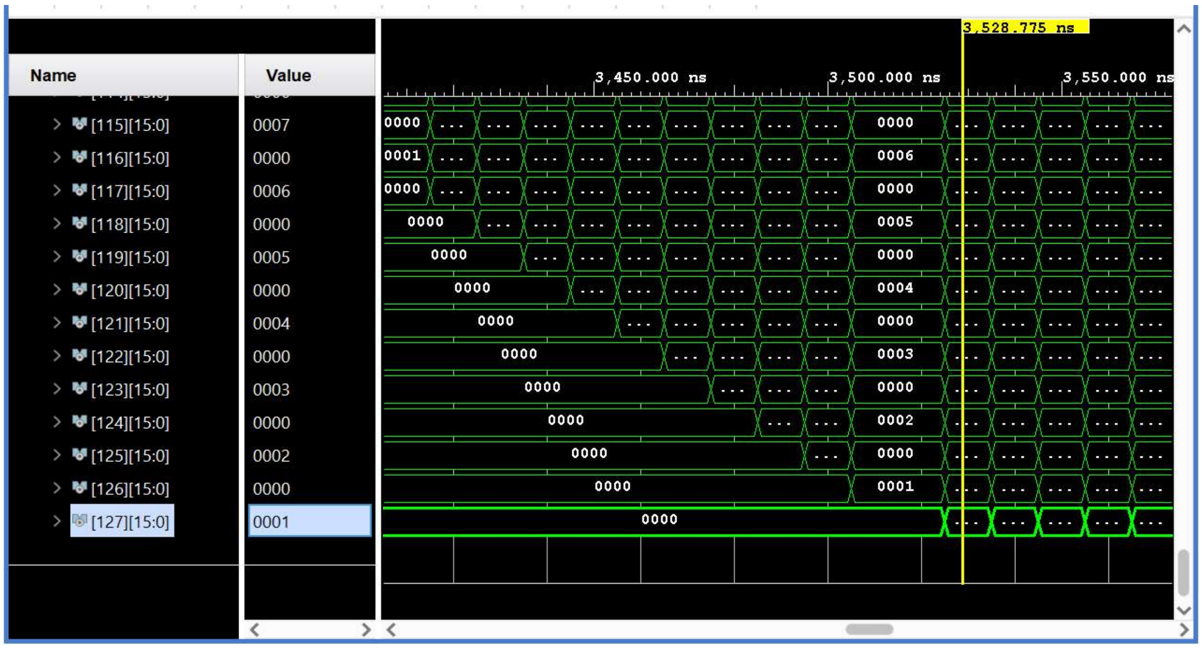


meghanakoduru@ece-b312-recon:~/convolve_1

```
[meghanakoduru@ece-b312-recon ~]$ cd convolve_1
[meghanakoduru@ece-b312-recon convolve_1]$ make
arm-linux-g++ -o zed_app main.o Board.o Timer.o App.o Convolve.o
[meghanakoduru@ece-b312-recon convolve_1]$ zed_schedule.py ./zed_app convolve_6.bit
Searching for available board....
Starting job "./zed_app convolve_6.bit" on board 192.168.1.104:
Programming FPGA....Testing small signal/kernel with all 0s...
Percent correct = 100
Speedup = 0.019802

Testing small signal/kernel with all 1s...
Error for output 0: HW = 2, SW = 1
Percent correct = 92.3077
Speedup = 0.0196079

Testing small signal/kernel with random values (no clipping)...
Error for output 0: HW = 93, SW = 26
Percent correct = 92.3077
Speedup = 0.0206186

Testing medium signal/kernel with random values (no clipping)...
Error for output 0: HW = 92, SW = 96
Percent correct = 99.9038
Speedup = 2.69139

Testing big signal/kernel with random values (no clipping)...
Error for output 0: HW = 126, SW = 99
Percent correct = 99.9985
Speedup = 15.9451

Testing small signal/kernel with random values...
Percent correct = 100
Speedup = 0.0181818

Testing medium signal/kernel with random values...
Percent correct = 100
Speedup = 2.33488

Testing big signal/kernel with random values...
Percent correct = 100
Speedup = 14.5546

TOTAL SCORE = 98.4469 out of 100
[meghanakoduru@ece-b312-recon convolve_1]$
```

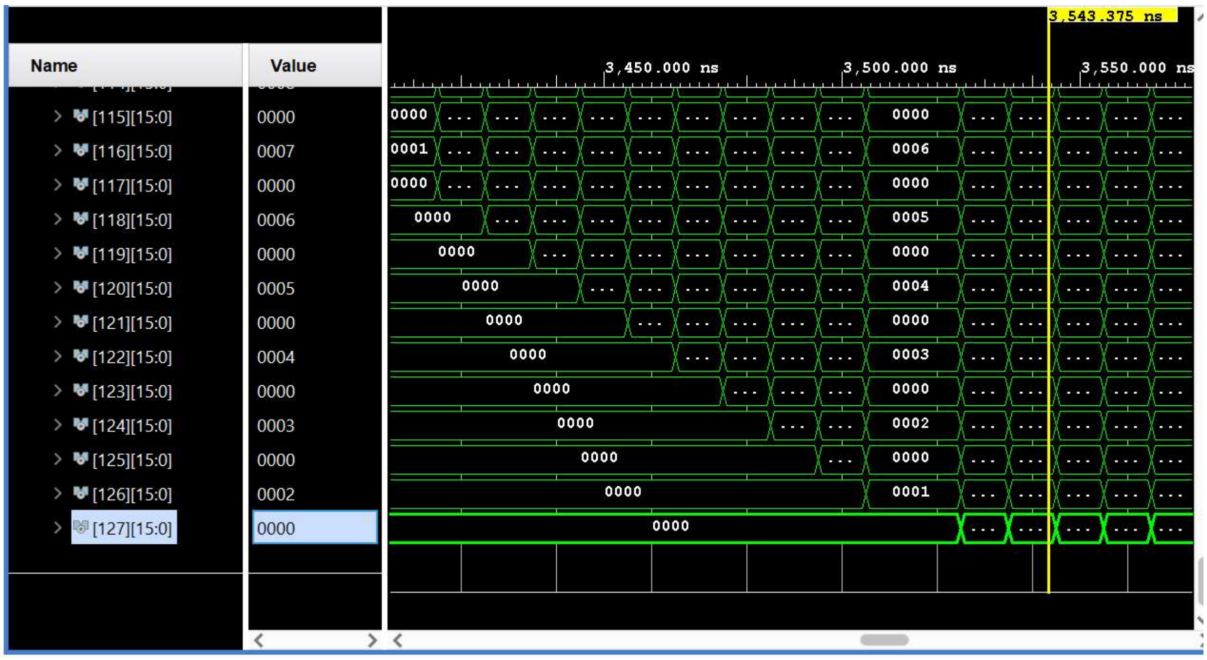
Fig 18: Results after inverting

The following screenshots below shows the reversal of signal buffer :



Name	Value
[115][15:0]	0007
[116][15:0]	0000
[117][15:0]	0006
[118][15:0]	0000
[119][15:0]	0005
[120][15:0]	0000
[121][15:0]	0004
[122][15:0]	0000
[123][15:0]	0003
[124][15:0]	0000
[125][15:0]	0002
[126][15:0]	0000
[127][15:0]	0001

Fig 19: Signal Buffer Inversion



Name	Value
[115][15:0]	0000
[116][15:0]	0007
[117][15:0]	0000
[118][15:0]	0006
[119][15:0]	0000
[120][15:0]	0005
[121][15:0]	0000
[122][15:0]	0004
[123][15:0]	0000
[124][15:0]	0003
[125][15:0]	0000
[126][15:0]	0002
[127][15:0]	0000

Fig 20: Signal Buffer inversion

(2) Had an issue while simulating user_app as we forgot to add '_0' to the dram_funcsim files as per instructions given.

V. CONCLUSION:

The project helped gain knowledge about different concepts of digital design and FPGA programming. The concept of design the circuit then write the code definitely helped simplify the design process and writing code to minimize design errors. The DMA interface dealt mainly with handling metastability and address generation. Resolving the timing issues was the main challenge when it came to making the DMA interface work perfectly. For the convolution part, although the score is 98.4469, the signal buffer, kernel buffer and pipeline entities are functioning correctly, when checked individually.