"1-D Time-Domain Convolution"

EEL5721 – Reconfigurable Computing

Graduate Team

Team Members :

Yashwanth Katta – (3451 7972)

Avinash Ayalasomayajula - (0699 6946)

Under the Guidance of :

Dr. Greg Stitt (Associate Professor)

<u>NOTE</u> : We have completed the project successfully. The report shows various details about how we did it and the results. We used Vivado 2017.4 version to overcome the black box errors.

<u>OBJECTIVE</u> :

Convolution is one of the most widely used concepts in the field of signal processing, may it be linear or circular. This project is a demonstration of how it can be implemented on an FPGA, thus giving us a faster implementation.

The main objective is to develop a Dram read interface, and send data to the user app. Now in the user app we have to design a signal buffer and kernel buffer.

Now we shall explain in brief about the results of each phase and finally show the completed project results after integrating the created user app and the dma interface.

<u>IMPLEMENTATION</u> :

The implementation is done in two stages. Firstly we implement DMA interface and then move to implementing User App.

A. <u>DMA INTERFACE</u>:

Initially the dram_rd_0 file has been provided to test run the pipeline. Our aim is to be design a dram interface that sends data into the pipeline. To implement this, we need to design an address generator, a FIFO and a controller that controls the address generator.

i. **Controller:** This part controls the address generator and the registers that store size and start address. When the GO signal goes HIGH, it enables the size and the start address register. This stores the value of size and start address. Now this go signal is sent through a handshake which is sent enables the registers in the address generator and then it starts counting. We wouldn’t need a handshake for the registers because they don’t change once they are initialized. This also has a clear input which makes it halt. It is like a local reset.

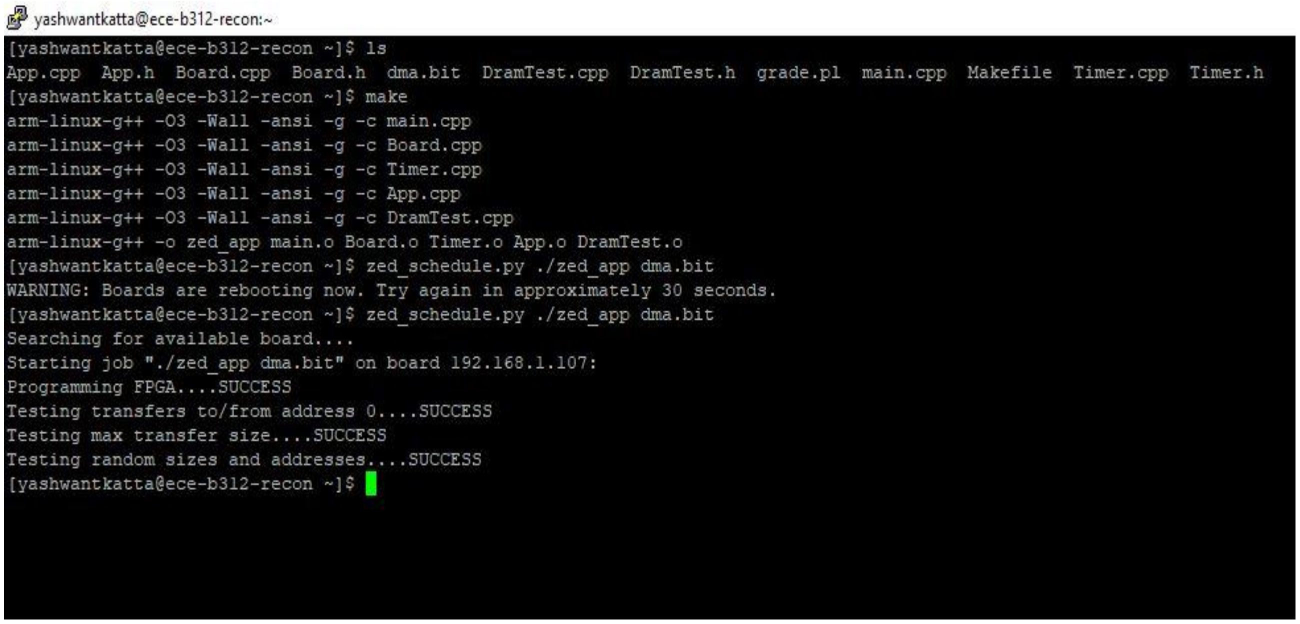
ii. **Address Generator:** The function of the address generator is to generate addresses once it receives the go signal from the handshake. Before it counts, once go = ‘1’, it flushes the data if it has any. The inputs to the address generator are the size and start address which are taken from their respective registers which are enabled by the go signal. Once this is done, addresses are generated and sent to the dram_rd_addr signal once the dram-rd-en signal goes HIGH. But this is only done if we get a HIGH dram-ready signal. Now there is a small catch here. If the FIFO is full and the address generator already generates some addresses which cannot be accommodated in the FIFO, then they are a waste and cannot be regenerated. So, we give a prog-full input from the FIFO which is enabled a few cycles before the FIFO reaches its full capacity, so that the generator stops generating addresses thus making up for the memory latency. This prog-full is set when generating the FIFO. Here we set it to 21. The clear is like a local reset to this component.

iii. **Handshake:** The handshake code was what we used in LAB 5. We used two handshakes in this project. One is for the go signal because it changes frequently, and one is for the clear signal. This made it easy for these signals to cross clock domains.

iv. **FIFO:** The FIFO is needed to take the inputs from the memory map (data stored in the generated addresses by the address generator) and process it to the user app clock domain. The FIFO takes 32 width input and gives 16-bit output. It has full ,empty, rd-en, wr-en, rd-data, wr-data and prog-full (which is set to 1 when the number of inputs in FIFO are 21). There isn’t much to it. We just needed to be careful while generating it in the accelerator. The rest was easy. The only thing we had to take care of is that the memory latency when the FIFO is full, is overcome by the prog-full flag. This goes HIGH when we have the FIFO is filled till 21 elements. The rd-en signal of the FIFO is given to a counter to set the done signal.

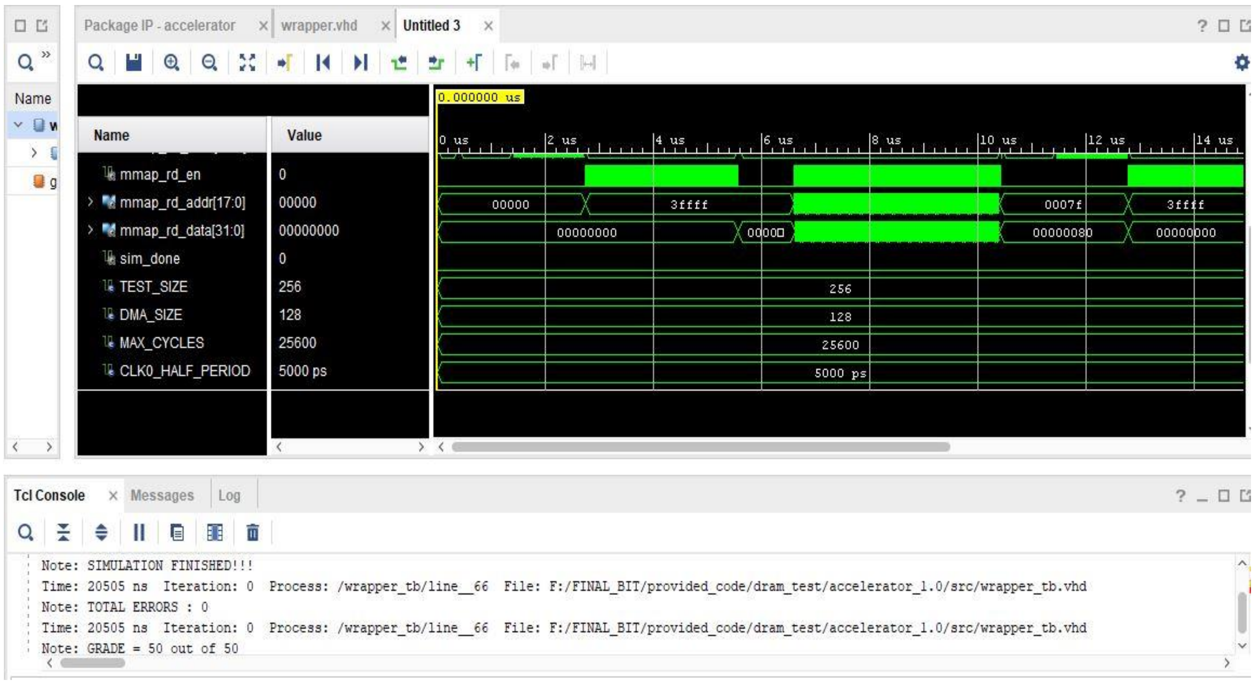
v. **Counter:** The counter in this case is used to make sure that the done is asserted. This is placed in the slower clock domain. The counter counts till size, because we get that many outputs, and starts counting when rd-en = 1 and the FIFO is empty. If not it would imply that all the outputs have been computed and we get the done asserted to ‘1’.

➢ **<u>RESULTS FOR DMA</u> :**



```
yashwantkatta@ece-b312-recon:~
[yashwantkatta@ece-b312-recon ~]$ ls
App.cpp  App.h  Board.cpp  Board.h  dma.bit  DramTest.cpp  DramTest.h  grade.pl  main.cpp  Makefile  Timer.cpp  Timer.h
[yashwantkatta@ece-b312-recon ~]$ make
arm-linux-g++ -O3 -Wall -ansi -g -c main.cpp
arm-linux-g++ -O3 -Wall -ansi -g -c Board.cpp
arm-linux-g++ -O3 -Wall -ansi -g -c Timer.cpp
arm-linux-g++ -O3 -Wall -ansi -g -c App.cpp
arm-linux-g++ -O3 -Wall -ansi -g -c DramTest.cpp
arm-linux-g++ -o zed_app main.o Board.o Timer.o App.o DramTest.o
[yashwantkatta@ece-b312-recon ~]$ zed_schedule.py ./zed_app dma.bit
WARNING: Boards are rebooting now. Try again in approximately 30 seconds.
[yashwantkatta@ece-b312-recon ~]$ zed_schedule.py ./zed_app dma.bit
Searching for available board....
Starting job "./zed_app dma.bit" on board 192.168.1.107:
Programming FPGA....SUCCESS
Testing transfers to/from address 0....SUCCESS
Testing max transfer size....SUCCESS
Testing random sizes and addresses....SUCCESS
[yashwantkatta@ece-b312-recon ~]$
```

<u>Figure 1</u>: DMA Interface test on FPGA Board. (as you see all tests are a success).



Name	Value
mmap_rd_en	0
mmap_rd_addr[17:0]	00000
mmap_rd_data[31:0]	00000000
sim_done	0
TEST_SIZE	256
DMA_SIZE	128
MAX_CYCLES	25600
CLK0_HALF_PERIOD	5000 ps

```
Note: SIMULATION FINISHED!!!
Time: 20505 ns  Iteration: 0  Process: /wrapper_tb/line__66  File: F:/FINAL_BIT/provided_code/dram_test/accelerator_1.0/src/wrapper_tb.vhd
Note: TOTAL ERRORS : 0
Time: 20505 ns  Iteration: 0  Process: /wrapper_tb/line__66  File: F:/FINAL_BIT/provided_code/dram_test/accelerator_1.0/src/wrapper_tb.vhd
Note: GRADE = 50 out of 50
```

<u>Figure 2</u>: DMA Interface Simulation Graph showing a grade of 50. (Thus it simulated correctly)

B. <u>USER APP</u> :

The user app interface mainly consists of 3 individual components namely

1) Convolution pipeline
2) Signal Buffer
3) Kernel Buffer

1. **Convolution Pipeline:**

The convolution pipeline is a multiplier-adder tree that takes two arrays of inputs multiplies the corresponding elements and adds up all the multiplier outputs through an adder tree. It essentially consists of 128 multipliers and 127 adders.

The kernel buffer and the signal buffer act as inputs to the convolution pipeline with the signal buffer being a sliding window buffer. So, the convolution pipeline takes corresponding elements from both the kernel and signal buffers and then the adder tree adds up the corresponding values. The kernel buffer remains the same through the process, whereas the inputs coming in from the signal buffer change by one value via a sliding window. We have slightly modified the convolution pipeline by introducing two new perks.

i. **Clipped Output:** During the process of convolution there is a change that the final output might be greater than 16 bits. So, for such cases we clip the result i.e. makes all bits of the output as '1'. For this we introduce an extra signal 'mult_add_tree_clip'. We check if the result of the multiplier-adder tree exceeds 16 bits and if it does, we instantiate 'mult_add_tree_clip' to all '1's. Else we copy the data of the result into 'mult_add_tree_clip' and we connect this to the DRAM_1 interface.

ii. **Valid In and Valid Out:** The output of the multiplier-adder tree is to be sent to a DRAM which reads the output and then sends it to the memory map. But we need to inform the reading DRAM_1 whether the output from the multiplier-adder tree is valid of not. For this we introduce two signals 'Valid In' and Valid Out'. We instantiate 'Valid In' to '1' when the kernel buffer is full, the DRAM_1 interface is ready to read, and the read-enable is a high for the signal buffer. Then using a delay entity and waiting for a 135 number of cycles (depth of pipeline) we assert the ‘Valid_Out’ signal informing the DRAM _1 interface that the data being sent is valid.

The multiplier-adder tree is enabled via a enable signal(en) which is asserted when the DRAM_1 interface is ready(ram1_wr_ready).

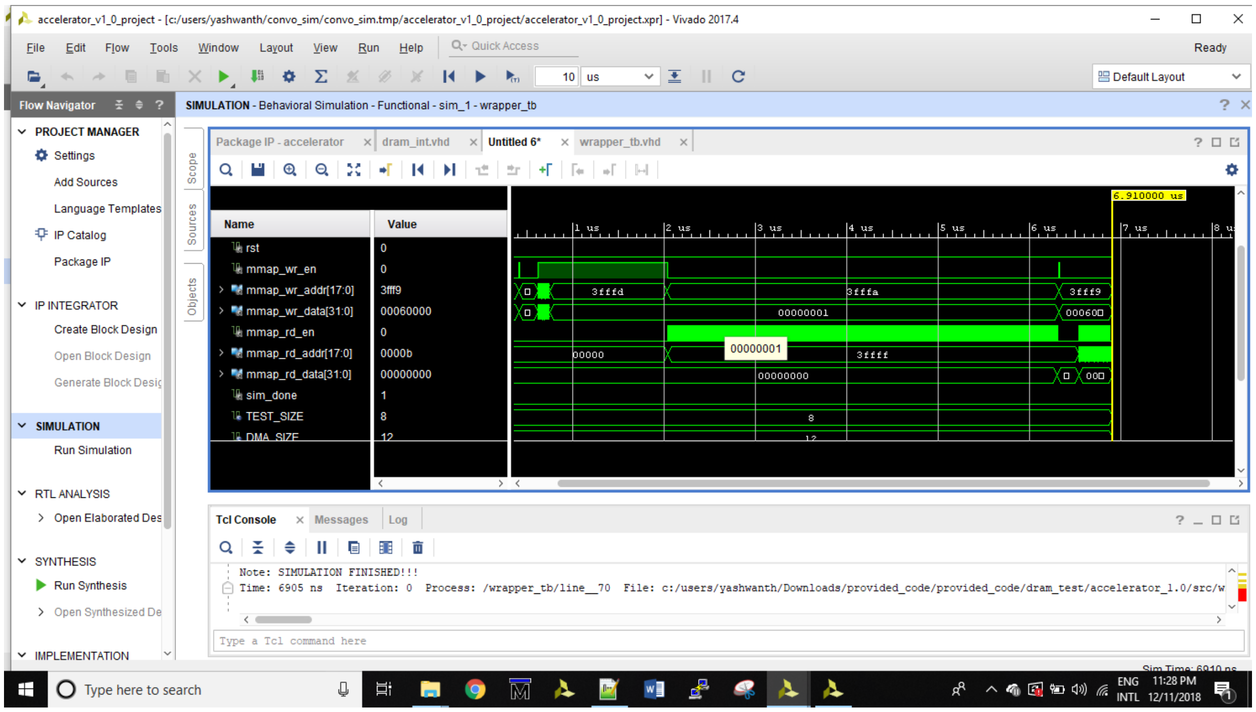
2. **Signal Buffer:** Signal buffer acts as one the inputs to the convolution pipeline. In convolution the sequential multiplications have a large overlap of elements. Thus, we can use a signal buffer that uses the sliding window application to our advantage. The designed signal buffer consists of 128 registers each capable of holding 16 bits. Each window differs only by one element, hence every cycle the elements in the registers get shifted to the next and one new data input is taken. We have designed a signal buffer that takes one input data value i.e. 16 bits when the write enable(wr_en) is high and shifts it along the registers. We make sure that write enable is asserted only when the DRAM_1 interface is ready(ram1_wr_ready). We also have a full flag(sb_full) that goes high once all the 128 registers are filled. For this application the window size is equal to the buffer size hence we must read the whole buffer. For this we have a read enable signal(sb_empty), which when high transfers the values stored inside the registers to the convolution pipeline. Once read-enable has been asserted thus the signal buffer is empty (not literally) and an empty flag(sb_empty) is asserted, while the full flag is de-asserted. The read enable signal is asserted only when the signal buffer isn’t empty (not sb_empty) and the DRAM_1 interface is ready(ram1_wr_ready). On The next clock cycle the inputs get shifted by a register thus knocking out one input and making way for the new one. We maintain a **counter** (count_user) to check the number of times write and read enables have been asserted. The full and empty flags are asserted based on the count i.e. full if count= buffer size and empty if count =0. We have implemented using three process, one for writing values, one for reading and one to check count. This was required as values could be both being written to and read from the signal buffer.

3. **Kernel Buffer:** The kernel buffer acts as another input to the convolution. It is designed very similar to the signal buffer, except that it takes inputs only once from the memory map. It doesn’t get new inputs every cycle and thus no sliding window application. It also consists of 128 registers each of 16

bits. The kernel buffer is similar to the signal buffer also has a full and empty flag. These are used for communication to the memory map. The kernel buffer full flag(kb_full) is connected to the memory map via 'kernel_loaded' signal which tells that the kernel buffer has been loaded with the data. The write enable signal (kb_wr_en) is connected to the 'kernel_load' i.e. if kernel is to be loaded the 'kernel_load' goes high, so does 'wr_en' inputting data. The data is read is from kernel buffer via read enable signal(kb_rd_en) which is asserted only when DRAM_1 interface is ready(ram1_wr_ready) and the kernel buffer is full(kb_full).

The user app also needs to communicate to the DRAM_0 interface to tell when the data should be sent. For this we have a 'ram0_rd_rd_en' which is asserted only when the signal buffer isn't full(sb_full).

➢ <u>RESULTS FOR USER APP</u> :

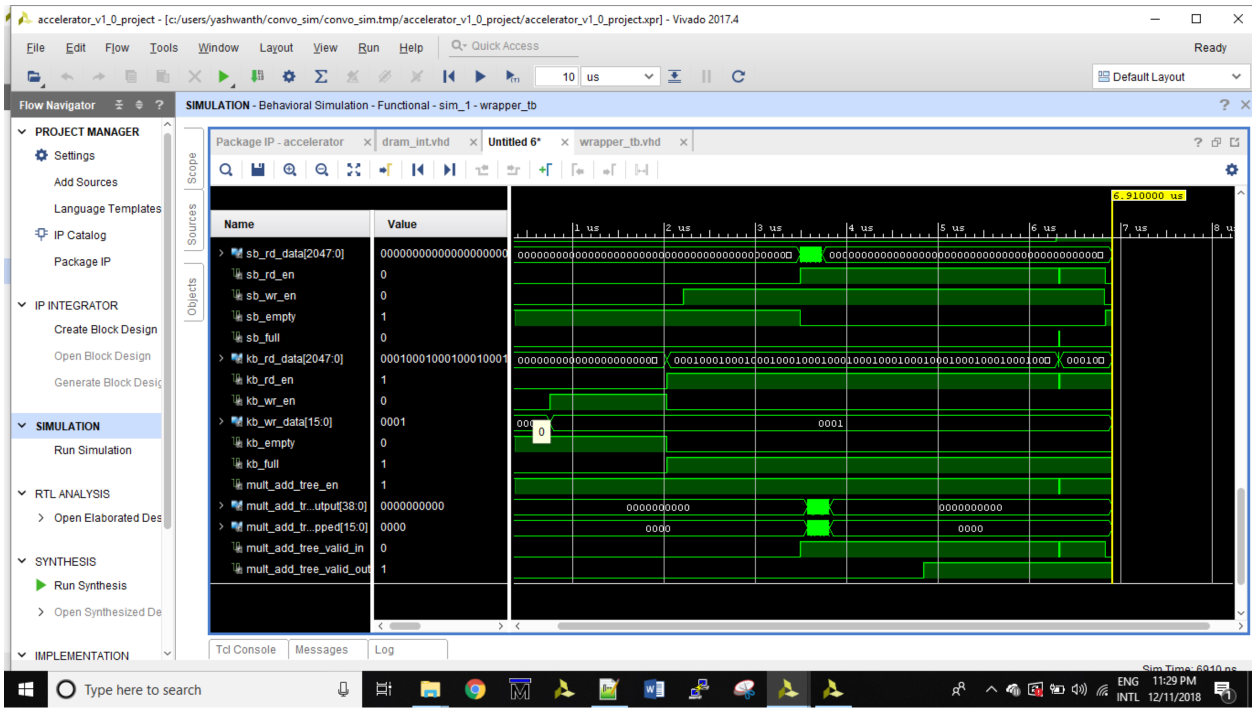


Name	Value
rst	0
mmap_wr_en	0
mmap_wr_addr[17:0]	3fff9
mmap_wr_data[31:0]	00060000
mmap_rd_en	0
mmap_rd_addr[17:0]	0000b
mmap_rd_data[31:0]	00000000
sim_done	1
TEST_SIZE	8
DMA_SIZE	12

```
Note: SIMULATION FINISHED!!!
Time: 6905 ns  Iteration: 0  Process: /wrapper_tb/line__70  File: c:/users/yashwanth/Downloads/provided_code/provided_code/dram_test/accelerator_1.0/src/w
```

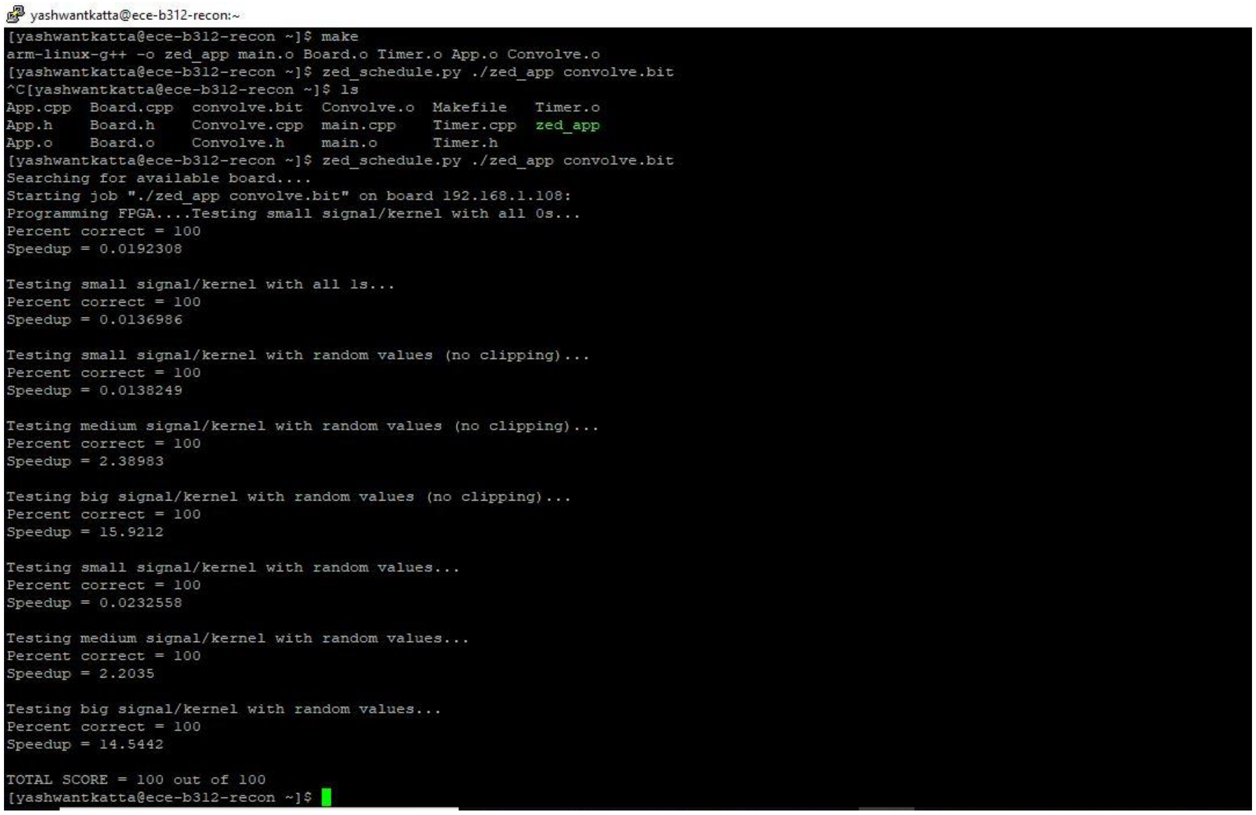
accelerator_v1_0_project - [c:/users/yashwanth/convo_sim/convo_sim.tmp/accelerator_v1_0_project/accelerator_v1_0_project.xpr] - Vivado 2017.4
SIMULATION - Behavioral Simulation - Functional - sim_1 - wrapper_tb
Package IP - accelerator
dram_int.vhd
Untitled 6*
wrapper_tb.vhd
6.910000 us
Tcl Console
Messages
Log
Sim Time: 6910 ns

<u>Figure 3</u>: Wrapper Simulation showing successful simulation.



Name	Value
sb_rd_data[2047:0]	0000000000000000000
sb_rd_en	0
sb_wr_en	0
sb_empty	1
sb_full	0
kb_rd_data[2047:0]	0001000100010001
kb_rd_en	1
kb_wr_en	0
kb_wr_data[15:0]	0001
kb_empty	0
kb_full	1
mult_add_tree_en	1
mult_add_tr...utput[38:0]	0000000000
mult_add_tr...pped[15:0]	0000
mult_add_tree_valid_in	0
mult_add_tree_valid_out	1

Figure 4: User_app Interface simulation graph.



```
[yashwantkatta@ece-b312-recon ~]$ make
arm-linux-g++ -o zed_app main.o Board.o Timer.o App.o Convolve.o
[yashwantkatta@ece-b312-recon ~]$ zed_schedule.py ./zed_app convolve.bit
^C[yashwantkatta@ece-b312-recon ~]$ ls
App.cpp   Board.cpp   convolve.bit   Convolve.o   Makefile   Timer.o
App.h     Board.h     Convolve.cpp   main.cpp     Timer.cpp  zed_app
App.o     Board.o     Convolve.h     main.o       Timer.h
[yashwantkatta@ece-b312-recon ~]$ zed_schedule.py ./zed_app convolve.bit
Searching for available board....
Starting job "./zed_app convolve.bit" on board 192.168.1.108:
Programming FPGA....Testing small signal/kernel with all 0s...
Percent correct = 100
Speedup = 0.0192308

Testing small signal/kernel with all 1s...
Percent correct = 100
Speedup = 0.0136986

Testing small signal/kernel with random values (no clipping)...
Percent correct = 100
Speedup = 0.0138249

Testing medium signal/kernel with random values (no clipping)...
Percent correct = 100
Speedup = 2.38983

Testing big signal/kernel with random values (no clipping)...
Percent correct = 100
Speedup = 15.9212

Testing small signal/kernel with random values...
Percent correct = 100
Speedup = 0.0232558

Testing medium signal/kernel with random values...
Percent correct = 100
Speedup = 2.2035

Testing big signal/kernel with random values...
Percent correct = 100
Speedup = 14.5442

TOTAL SCORE = 100 out of 100
[yashwantkatta@ece-b312-recon ~]$
```

Figure 5: User_app interface tests on FPGA

CHALLENGES FACED :

The major challenge was the black box error. So, we tried a lot of workarounds but using Vivado 2017.4 solved the issue.

A. DMA_INTERFACE :

1. When we tried to simulate the dma interface we created, there was a delay because of which the output was zero. So, we added an extra state. That fixed the problem. This was a major issue.
2. When we ran the dma bitfile on the board, we got stuck at the first step. Professor Stitt told it might be a metastability issue. We investigated it and it was a minor tweak that fixed it too.
   We changed the testbench a little to find this issue. It helped.
3. Then the random test failed. When we checked the simulation, the grade was 50/50. So, we checked and found that in one of the states we forgot to set the initial address. That fixed the issue and our DMA Interface was working perfectly.
4. The rest were minor issues which were easily fixed once we looked at the simulated graph. The major challenge we faced was when we ran the bitfile on the board. Thanks to the lectures, we were able to figure out a bit.
5. Had a problem with the FIFO. We had to create a new one and we messed it up most of the times in selecting the First Word Fall Through.

B. USER APP:

1. Had initial problem with ‘Valid In’ for convolution pipeline being asserted to early. Added ‘kernel_full’ signal to the control of ‘Valid in’.
2. We were getting wrong output as data was inserted in reverse for both buffers and was being taken that way. Rectified it by inverting the data of signal buffer again.
3. Faced problem with simulating the user_app as we forgot to add ‘_0’ to the dram funcsim files within wrapper.
4. Tried using state machines with each stage performing an operation but full data wasn’t being read within a lock cycle, hence changed to combinational loop methods using a counter as the Professor suggested.

CONCLUSION :

The project was a great challenge to our coding and conceptual understanding. We had a few difficulties while implementing it, but Professor Stitt did help a lot in clarifying the crucial parts which was greatly helpful. We had a lot of problem with the black box, but we fixed it at the end. The whole project was implemented with working results. The bit files for both DMA and User APP have succeeded in all the tests and the final score 100.

Thus, the project ended in positive results.