CNN FPGA Implementation

Hardware Documentation

A Logic Design Project By:

Ahmed Abdulkader	1170472
Daniel Eskandar	1170524
Omar Essam	1162285
Omar Tarek	1170331
Ahmed Ezzat	1162033

CNN FPGA Implementation

This is a Hardware Documentation for the Logic Design Project aiming to implement a convolutional neural network on an FPGA using Verilog. The project is designed through different independent modules, each executed by a different team member. The first part of the report is a general overview of the project and a guide to the attached files, with general procedures of testing and execution outlined. Then the individual reports prepared by each individual team member are attached, along with additional appendices to satisfy all required project deliverables.

CONTENTS

Overview
Reporting
Section

This section is a summary and outline of the project as a whole, providing a good outline as a starting point, explaining several key points

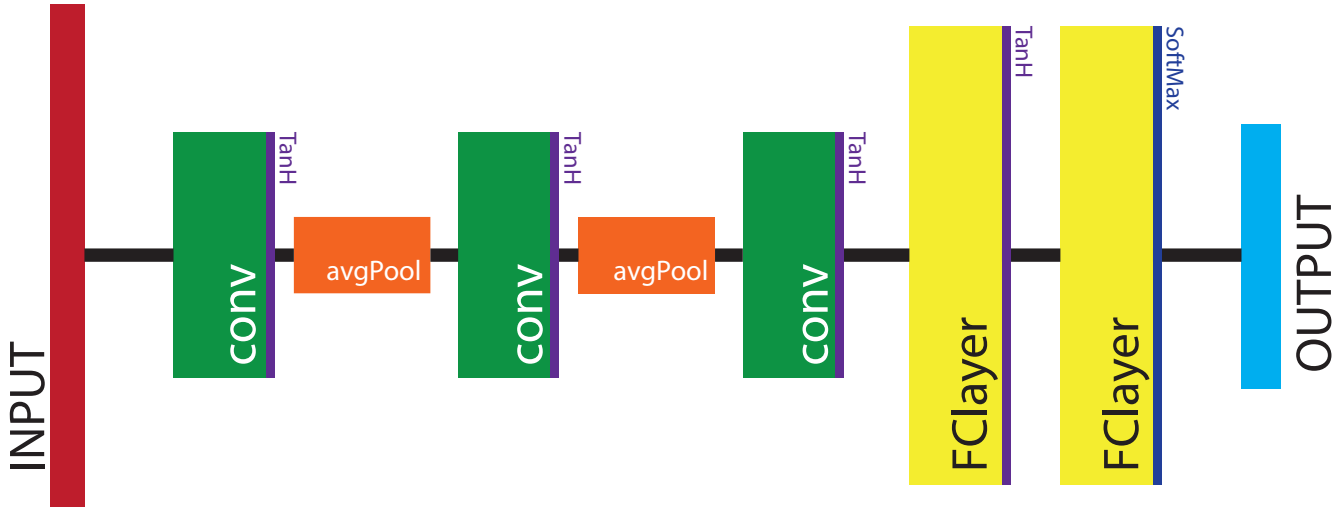
- **Project Overview**
- **Member Distribution**
- **Testing Overview**
- **Attachments Overview**
- **General Team Comments**

Project Overview

The Project is an implementation of the leNet-5 CNN architecture. As such our team consists of 5 members each executing a different part of the Network. The distribution is as follows:

Daniel Eskandar	Part 1- Convolution
Ahmed Ezzat	Part 2- Tanh Activation
Omar Essam	Part 3- SoftMax Activation
Ahmed Abdulkader	Part 4- Average Pooling
Omar Tarek	Part 5- Integration

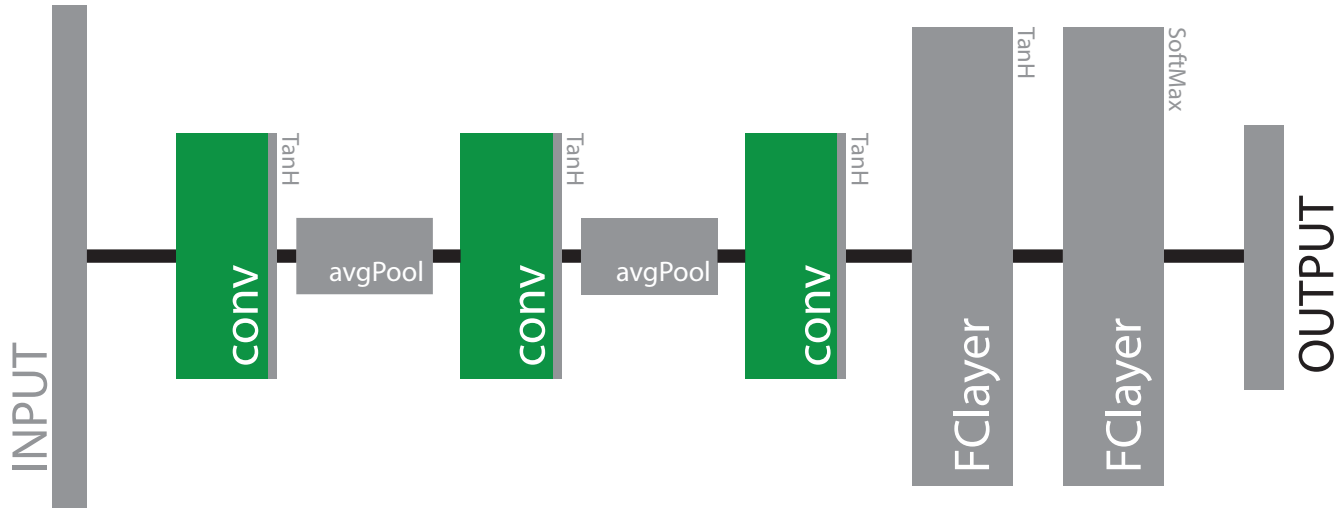
All modules are verfied to be functional and synthesizable. The testing procedures and verification will be outlined over the next pages explaining the attached folder layouts and additional information on the test cases used. Disscussions and relevant comments would also be attached to every special case handled.



The next few pages will cover a summary on the different parts and their implementations.

Part 1 - Convolution

Done By: Daniel Eskandar



The network has 3 convolution layers, each with different filters, all of Kernel Size 5x5. Details of implementation would be found at the individual report by Daniel. Several architectures were attempted and implemented by Daniel, in an attempt to reach a compromise between Parrallelism, Speed and Utilization. The trade off here is more parrallelism requires higher power and is more hardware extensive. Less parrallelism causes a decrease in speed. This balance is very tricky in convolution due to the huge sizes of Data required for this complicated function. A compromise was reached and a comparison between architectures is outlined in Daniel's individual report.

Code Folder:

\Final Code Files\Part 1- Convolution

Testing Scripts:

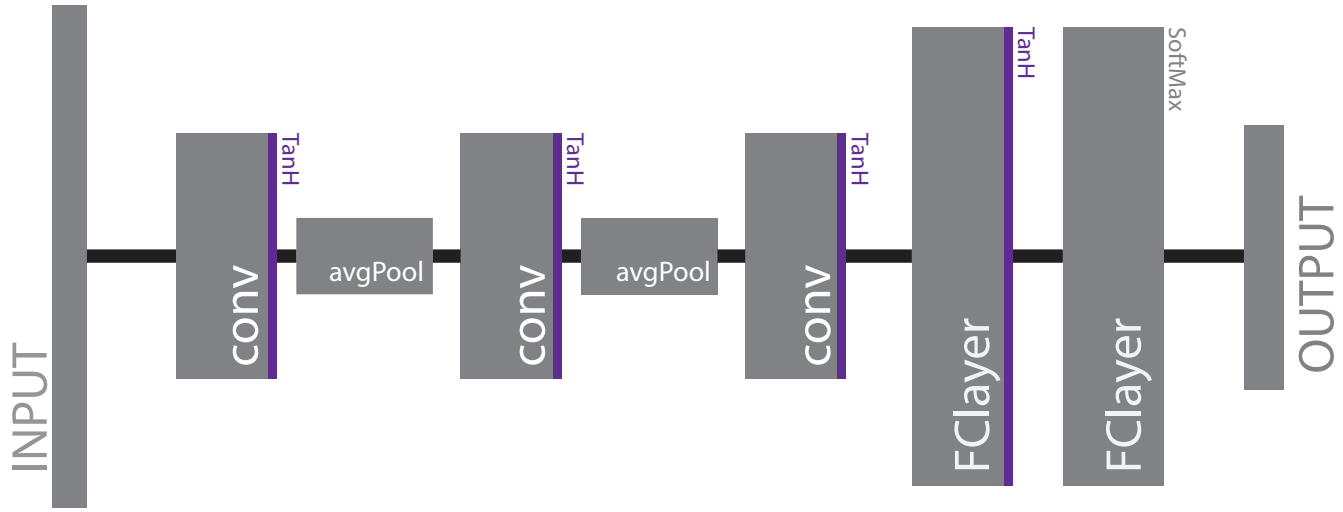
\Testing Scripts\Convolution Python Codes

Excel Sheets:

Conv1Test and Conv2Test

Part 2 - TanH Activation

Done By: Ahmed Ezzat



The activation function selected between the layers and each another is the Hyperbolic Tangent Function. More details on structure and testing are attached in the individual report prepared by Ahmed Ezzat. The function deals with a large number of inputs, same issue as convolution. During the convolution part of the integration, the TanH function is of size 16, and at the fully connected parts it's at size 32. The convergance condition and the limiting range of 16 bit half precision floating point numbers, and also the fact that this is implemented using the taylor approximation of the function, make this a bottleneck of accuracy. The accuracy is within the accepted range though.

Code Folder:

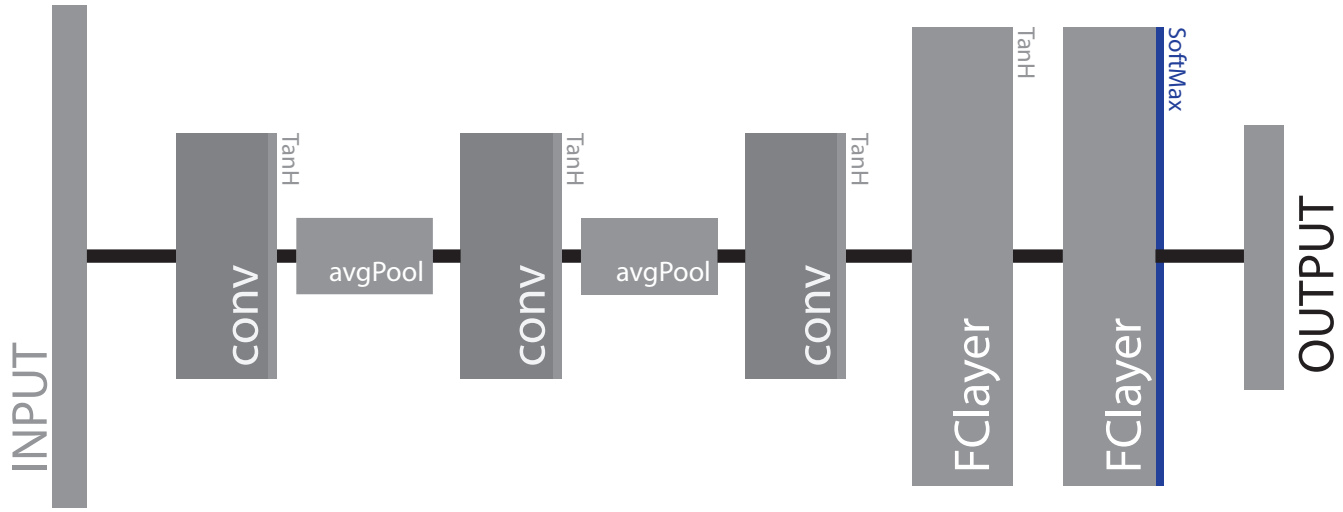
\Final Code Files\Part 2- TanH activation

Testing Scripts:

\Testing Scripts\Tanh testing Codes

Part 3 - SoftMax Activation

Done By: Omar Essam



The last layer in the leNet-5 architecture is a SoftMax activation layer. This special activation layer takes the 10 produced values and through a series of calculations produces the final classification of our neural network. The series of calculations mentioned, is an approximation of a function involving division of exponents. Implementations of hardware division of several floating point numbers in Verliog is exceptionally difficult, which led Omar Essam to develop multiple architectures, and choosing the best compromise. Difference between the architectures and the merits of each are discussed in detail in Omar Essam's individual report.

Code Folder:

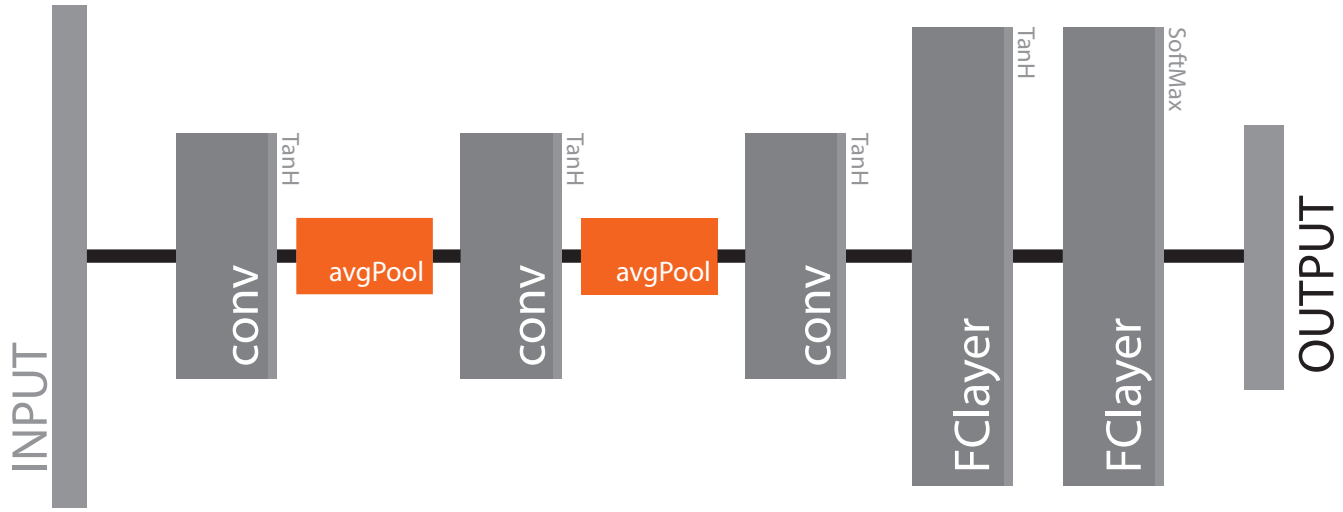
\Final Code Files\Part 3- SoftMax activation

Testing Scripts:

\Testing Scripts\SoftMax Python Codes

Part 4 - Average Pooling

Done By: Ahmed Abdulkader



The network has 2 average pooling layers. These decrease the size of the output data from the first 2 convolution layers. While fairly easy to implement, they suffer from the same Data Size issue found in convolution layers (part 1). These issues cause synthesis and utilization problems, leading us to use compromises that may slow down the process but improve hardware utilization significantly. The other drawback of the large data size, is the inability of producing proper post synthesis timing simulations. This is an issue that was agreed on with faculty staff and we reached a compromise of simulating the smallest processing element of each matrix.

Code Folder:

\Final Code Files\Part 4- Average Pooling

Testing Scripts:

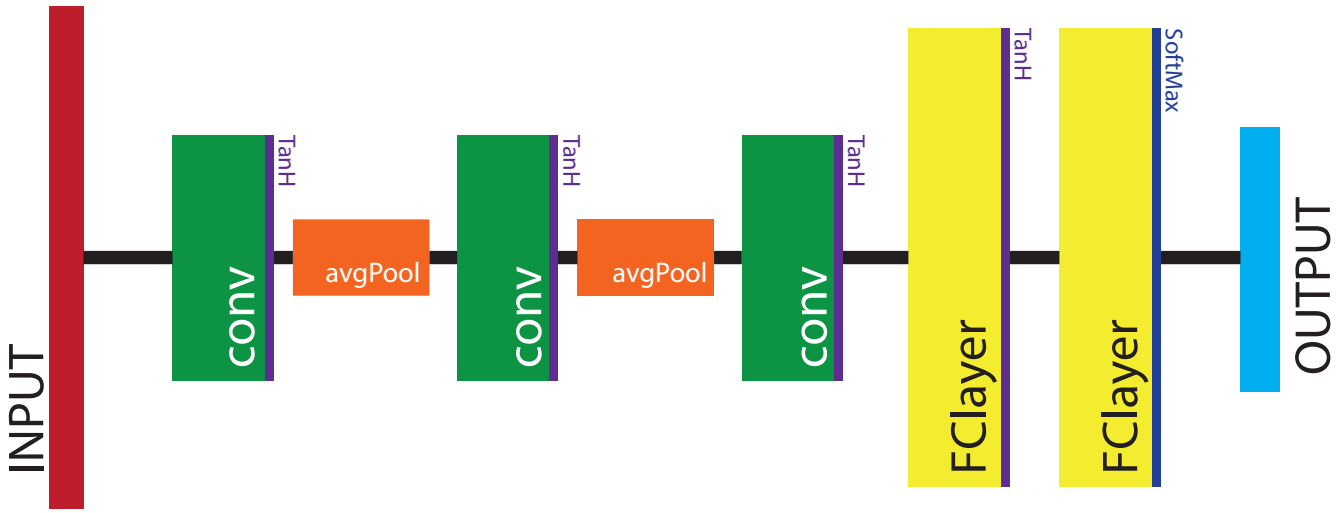
\Testing Scripts\Average Pooling Python Codes

Excel Sheets:

AvgPool1Test and AvgPool2Test

Part 5 - Integrating Network

Done By: Omar Tarek



Arguably the hardest, least synthesizable and largest part of this project is making sure that the network integrates properly. To make this happen collabaration between the member responsible for integration, Omar Tarek, and the rest of the team was essential. To also make the testing easier, testing on the convolution units and the average pool units was done consecutively with one input number image and plugging the output data of the previous layer into the next layer. More on this could be found on the testing section of the group report. Due to the sheer size of this network synthesis was not attempted due to hardware limits. More details are in the individual report.

Code Folder:

\Final Code Files\Part 5- Integration

Testing Scripts:

\Testing Scripts\ANN testing code +***previously mentioned***

Excel Sheets:

Testing Overview

Testing is done through the following procedure:

1) A code of similar function is run on an external script written in a different software design language (Python and C++)
2) The corresponding testbench is run on ModelSim and Data Results are extracted
3) Both Sets of data are compared against one another, with random samples taken for conversion to check if results are correct
4) If errors are found, the code is corrected and we start over

Special testing procedure for Convolution and AvgPool:

*For average pooling and convolution the same procedure is carried out, however it is done consecutively. The results of each prior stage are passed on as inputs of the next. This method ensures that **all values** are accumulated into a single data set that if correct by the end would mean the whole process was probably correctly functioning.*

All test scripts used are attached in the \Testing Scripts\ folder. In order to save time and effort the data sets used to verify the function are attached on a separate excel spreadsheet. This has data extracted from the test codes and the actual codes stacked against one another as a proof of function. This is only for layers that function on large data set level, like convolution, average pooling, and fully connected layers. Layers that do not change dimension (TanH and SoftMax) don't need large amounts of Data Comparison, a proof of function of the smallest unit was found to be enough to proof the function of the full unit.

Attachments Overview

This report comes attached with the hardware and supplementary files attached. This is a guide to the file folders:

CNN-FPGA

- Vivado Project *(Vivado Project with full architecture)*
- Final Code Files *(Individual Parts Verilog Files)*
- Screenshots and Reports *(Full Screenshots and Synthesis)*
- Testing Scripts *(All supplementary testing codes)*
- Weight Files *(All used weight files, with IEEE variants)*
- Extra Convolution *(Additional Convlution Architectures)*

The code files are also attached at the end of this report and linked through each memeber's individual report. In addition to the above mentioned folders there are also several files. The excel file mentioned in the testing section along with a read me file for work load distribution.

General Team Comments

This Report has attached files embedded into it. Next to every needed attachable deliverable you will find a paper clip icon that when opened will automatically open the required file.

Synthesis was attempted for all files. And the maximum achievable utilization for each code was attempted trying to compromise with speed and parallelism.

In the next part each document was written entirely by the team member whose name is mentioned before it. The overview part was written by Ahmed Abdulkader.

Post Synthesis simulation was agreed upon to simply cover the smallest processing unit in codes where sizes are massive.

All original screenshots, reports, and test files are attached with the folder supplied with this report.

Individual **Reporting** Section

The next section covers individual reports, that contain each members independent deliverables

Part 1- Daniel Eskandar
Part 2- Ahmed Ezzat
Part 3- Omar Essam
Part 4- Ahmed Abdulkader
Part 5- Omar Tarek

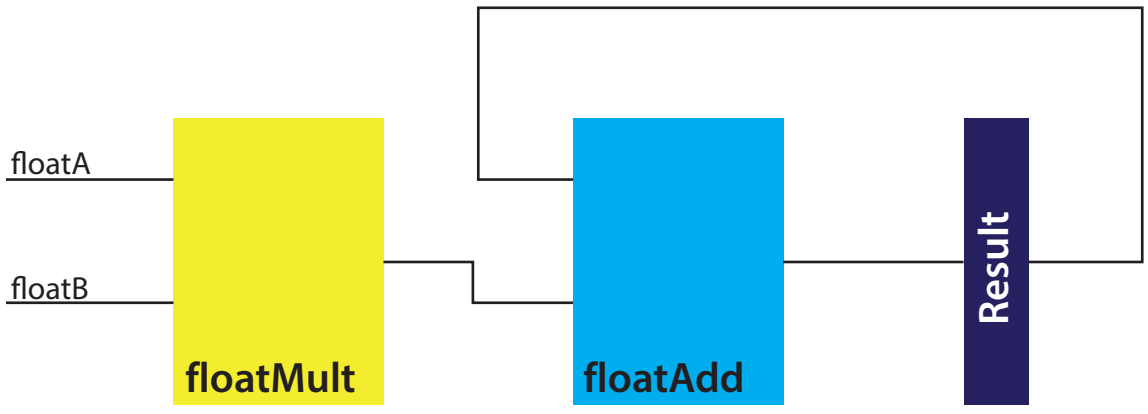
Part 1
Convolution

Daniel Eskandar

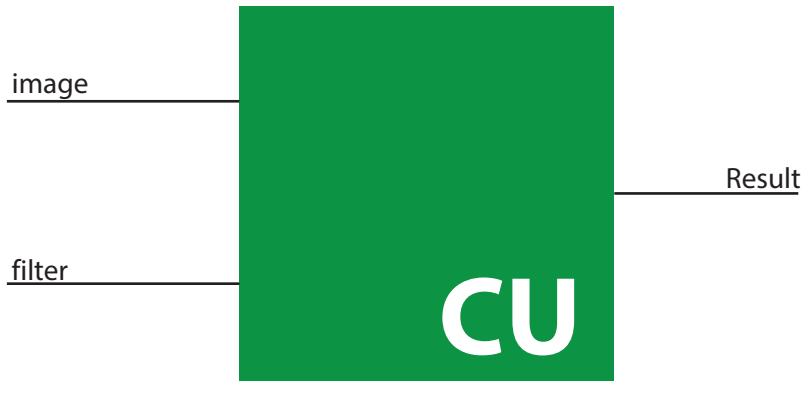
Part 1-Convolution

1) Block Diagrams

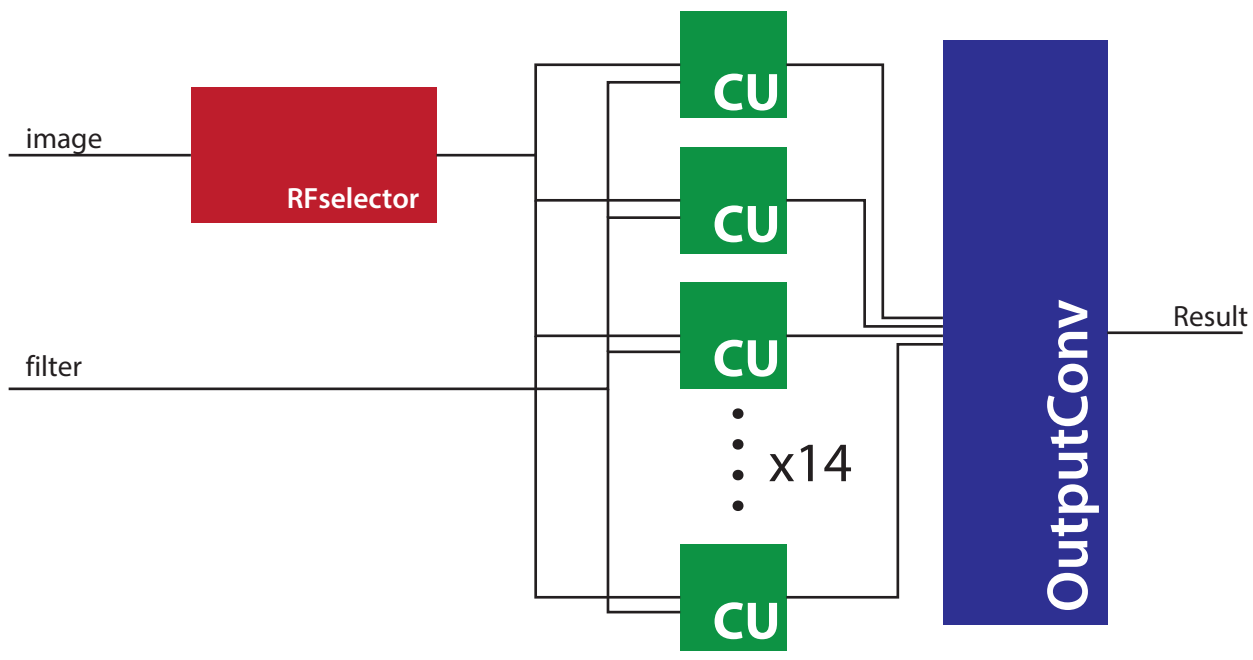
Processing Element



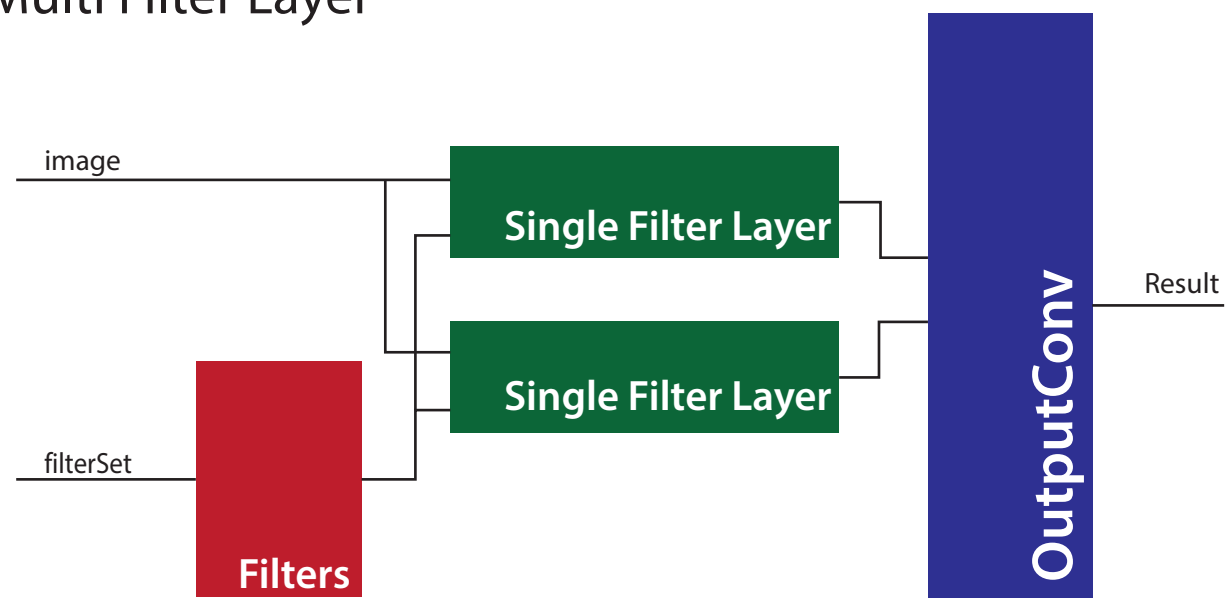
Convolution Unit



Single Filter Layer



Multi Filter Layer



2) Verilog Codes

convUnit.v

convLayerSingle.v

convLayerMulti.v

RFselector.v

3) Verilog Tesbench

convUnit_TB.v

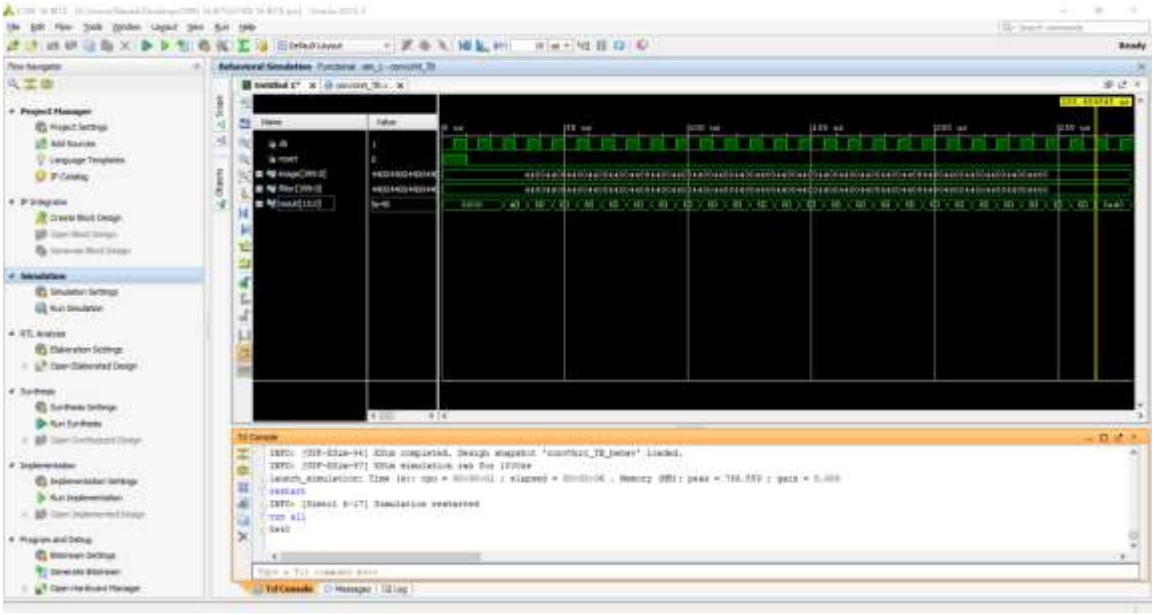
convLayerSingle_TB.v

convLayerMulti_TB.v

4) Screenshots:

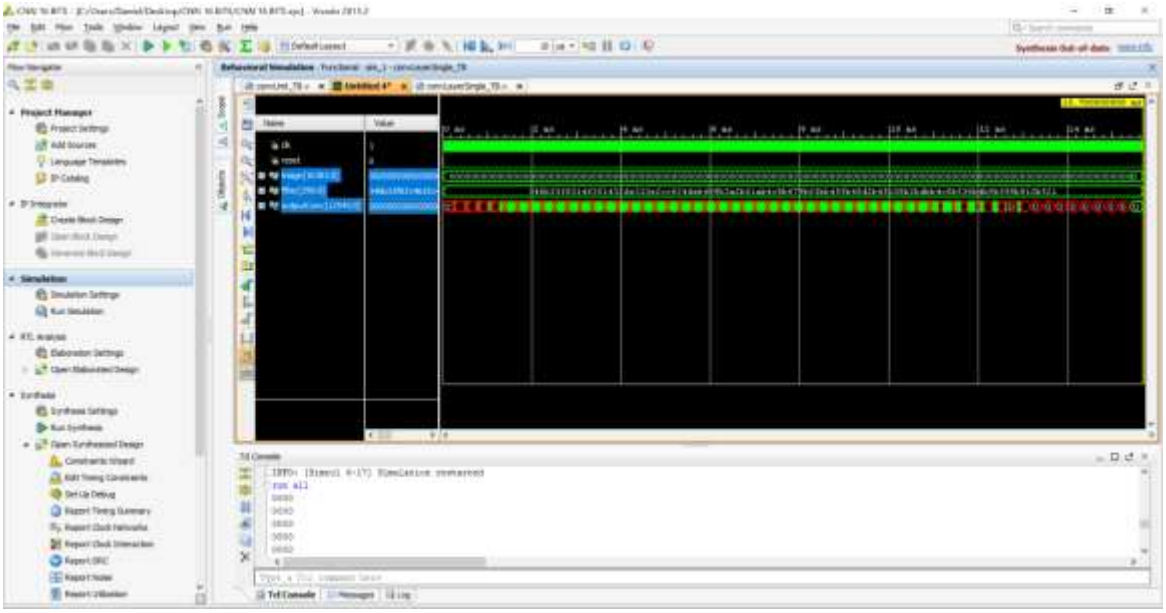
a. Simulation (Waveform/TCL console):

i. ConvUnit

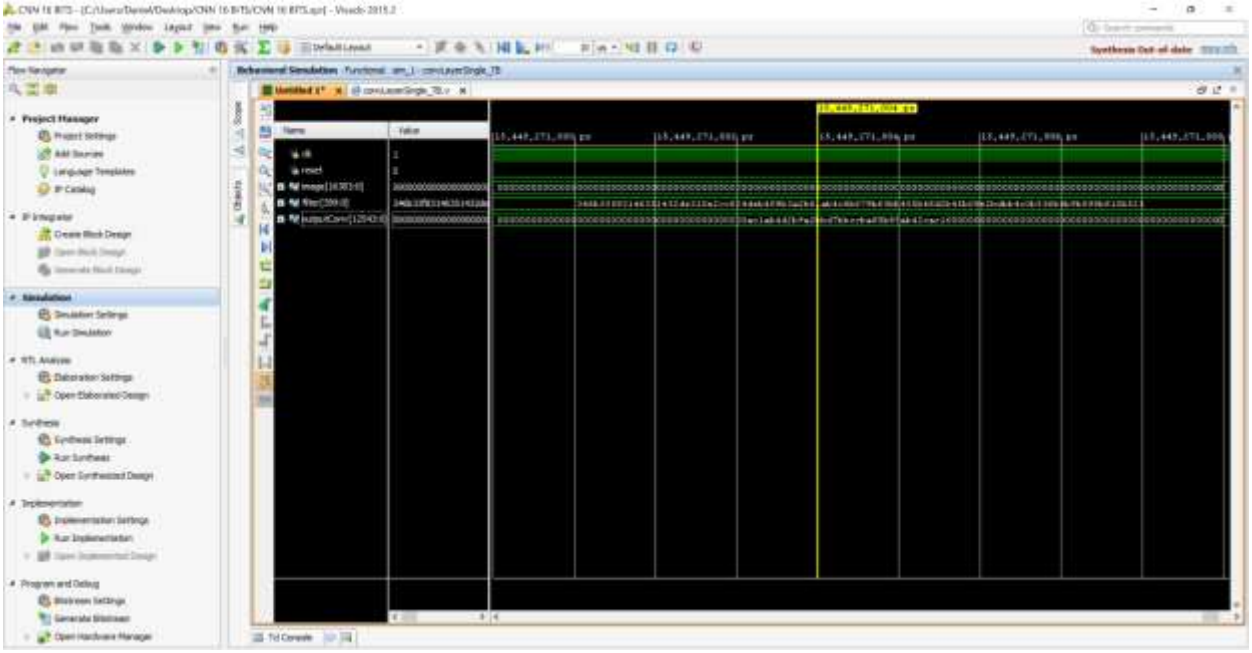


Testing the conv unit was done using one number repeatedly for checking if the process is done correctly quickly. The number used here is 4.

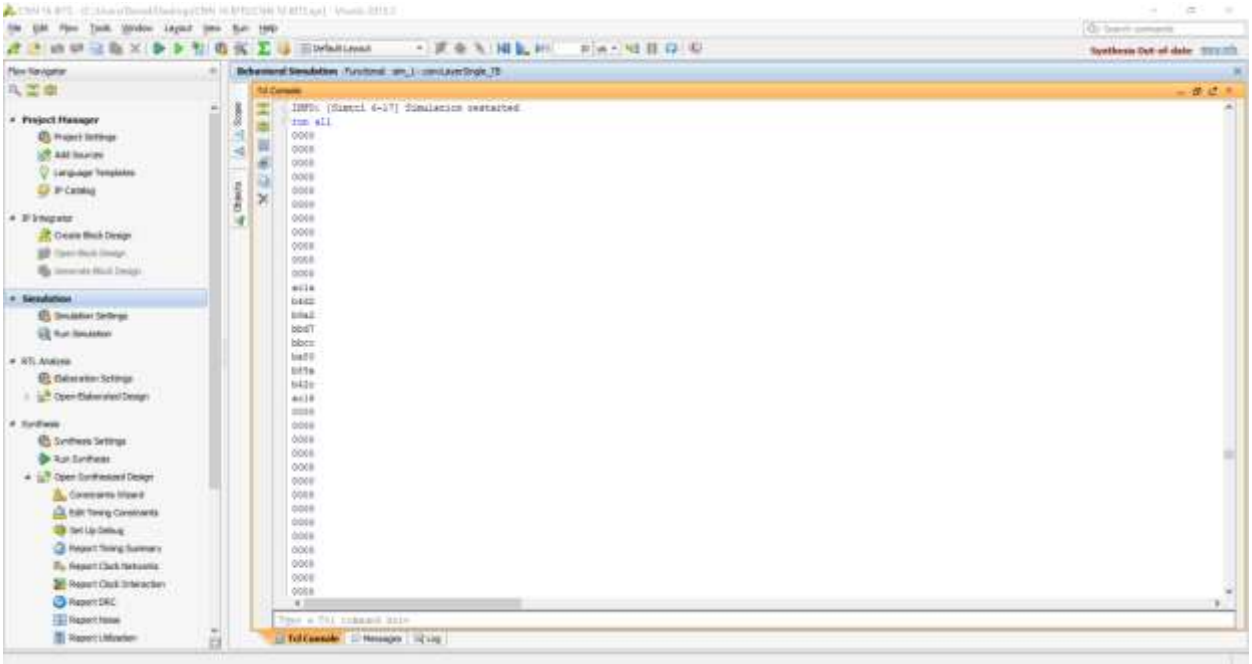
ii. Single Filter Layer



This screenshot is on a zoom full view. The red signals signify that the chip is still mid processing and filling out part by part of the complete signal, for the final result check the next screenshot.

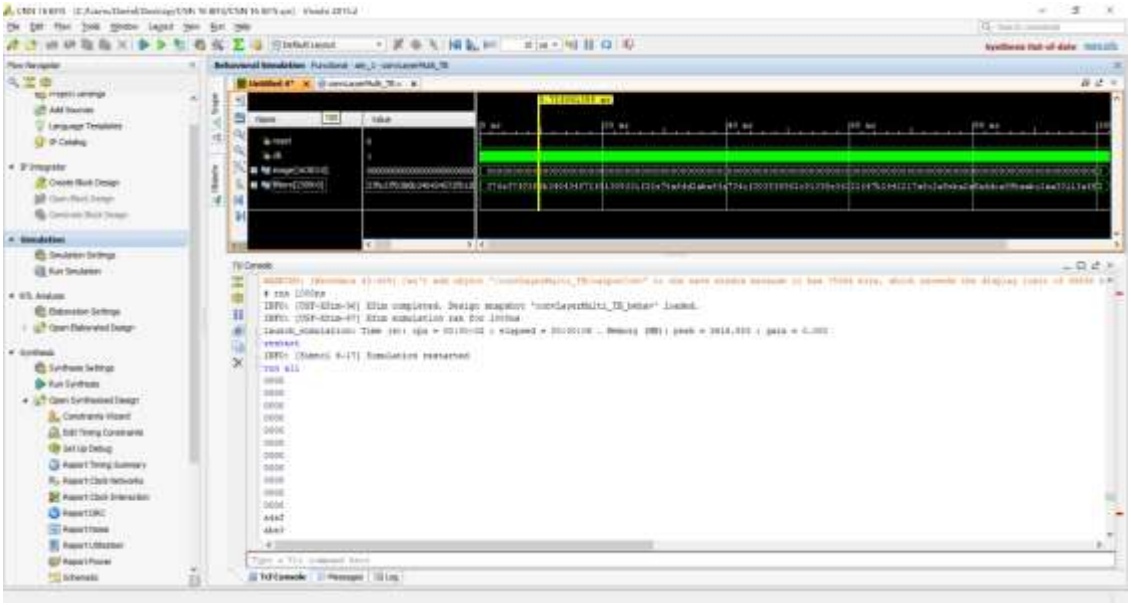


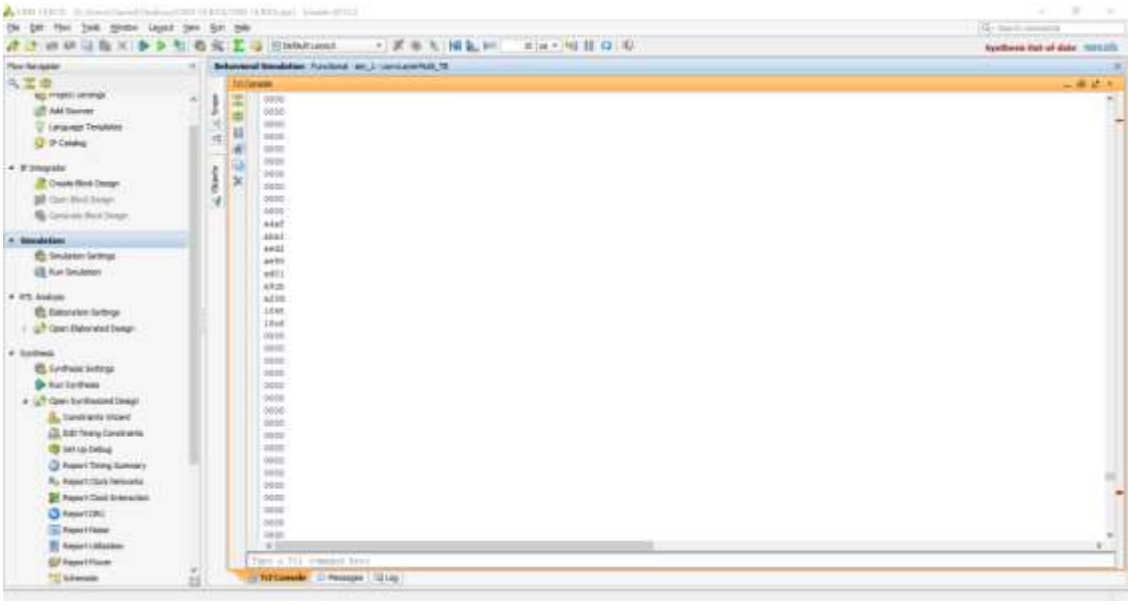
Here we zoomed on the end of the code run, the processing has stopped and the result is ready!



TCL script and waveform of the final output

iii. Multi Filter Layer





All results were verified using the simulated Convolution Layer run on python. You can find the python script in the folder Python Tests called ‘cLayer.py’. All result comparisons are also listed out on the Convolution1Test sheet on the attached CNNtest excel file (CNNtest.xlsx)

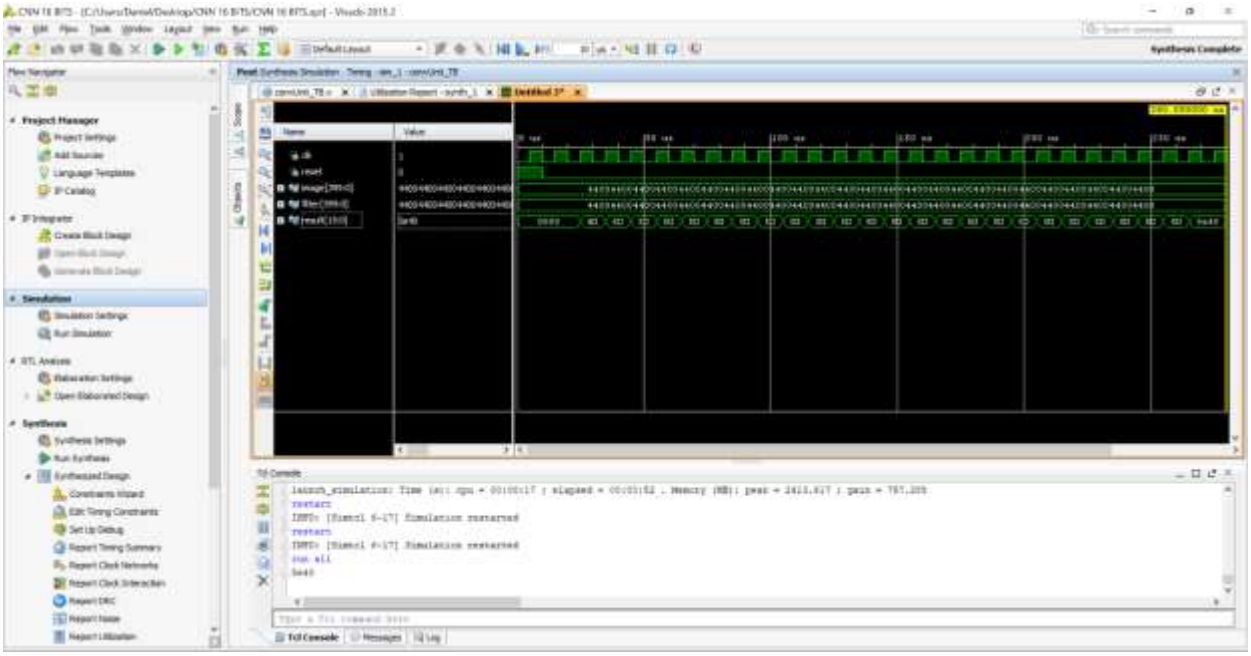
Excel Proof of function:

(compared to output from simulations)

	A	B	C	D	E	F	G	H	I	J	K	L
1	6		5		4		3		2		1	
2	Sim	Python	Sim	Python	Sim	Python	Sim	Python	Sim	Python	Sim	Python
3	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
4	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
5	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
6	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
7	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
8	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
9	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
10	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
11	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
12	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
13	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
14	a4af	-0.01831	a5ed	-0.02315	98c2	-0.00232	242a	0.01627	a8ce	-0.03757	ac1a	-0.06409
15	aba3	-0.05969	b048	-0.13389	a4d9	-0.01895	2ba5	0.05976	b131	-0.16235	b4d2	-0.30135
16	aed2	-0.10662	b611	-0.37938	aacf	-0.05326	30b4	0.14706	b5e9	-0.36962	b9a2	-0.70447
17	ae59	-0.09922	b8fa	-0.62251	acea	-0.07686	322b	0.19278	b741	-0.45377	bbd7	-0.98068
18	ad01	-0.07826	b967	-0.67566	9efd	-0.00682	3107	0.15716	b5de	-0.36699	bbcc	-0.97562
19	a9cb	-0.04528	b81b	-0.51371	3029	0.1301	9e90	-0.00647	b28c	-0.20477	ba80	-0.81358
20	a23d	-0.01219	b576	-0.34181	3324	0.22326	aca0	-0.07236	b1c2	-0.18002	b89a	-0.57575
21	1898	0.00224	b0d7	-0.15147	3070	0.13885	aa19	-0.04769	aeca	-0.10614	b42c	-0.26094
22	18c6	0.00233	a955	-0.04168	2965	0.04218	a03d	-0.00828	a909	-0.03938	ac16	-0.06385
23	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0
24	0000	0	0000	0	0000	0	0000	0	0000	0	0000	0

The data was copied from the TCL console and pasted against the results from our test code as shown clearly. The numbers in the first row are of the 6 depths of the first convolution layer.

b. Post Synthesis Timing for the basic convUnit:



(as agreed upon post timing for larger layers of hardware are not achievable with our hardware capabilities. Our attempts reached up to **14 hours**, all while stalling the progress bar. It's not achievable on consumer hardware)

5) Clock Cycle Count:

a. Conv Unit

(Depth*FilterW*FilterH)+1

(+1 for the reset cycle)

Example:

*For a filter of size 5x5 and a depth of 1 would take (1*5*5)+1 = 26 CC*

b. Single Filter Layer

[(ImgH-FilterH+1)(ImgW-FilterW+1)/(ImgH-FilterH+1/2)]*[(Depth*FilterW*FilterH)+1]

(Number of ConvUnits in the single layer multiplied by time taken for each unit)

Example:

For a layer of size 32x32 and a filter 5x5, it would take
*[28*28/(28/2)]*[**26**] = 1456 CC*

c. Multi Filter Layer

(fCnt/2)*[(ImgH-FilterH+1)(ImgW-FilterW+1)/(ImgH-FilterH+1/2)]*[(Depth*FilterW*FilterH)+1]

(Half Number of Filters in the multi-layer multiplied by time taken for each single layer)

Example:

For a layer of size 32x32 and 6 filters of 5x5, it would take
3[28*28/(28/2)]*[**26**] = 4368 CC*

6) Synthesis Utilization Report

Conv Unit

Site Type	Used	Fixed	Available	Util%
CLB LUTs*	654	0	242400	0.27
LUT as Logic	654	0	242400	0.27
LUT as Memory	0	0	112800	0.00
CLB Registers	82	0	484800	0.02
Register as Flip Flop	82	0	484800	0.02
Register as Latch	0	0	484800	0.00
CARRY8	12	0	30300	0.04
F7 Muxes	64	0	121200	0.05
F8 Muxes	32	0	60600	0.05
F9 Muxes	0	0	30300	0.00

Conv Single-Layer

Site Type	Used	Fixed	Available	Util%
CLB LUTs*	48422	0	242400	19.98
LUT as Logic	48422	0	242400	19.98
LUT as Memory	0	0	112800	0.00
CLB Registers	13753	0	484800	2.84
Register as Flip Flop	1209	0	484800	0.25
Register as Latch	12544	0	484800	2.59
CARRY8	181	0	30300	0.60
F7 Muxes	0	0	121200	0.00
F8 Muxes	0	0	60600	0.00
F9 Muxes	0	0	30300	0.00

Conv Multi-Layer

Site Type	Used	Fixed	Available	Util%
CLB LUTs*	455605	0	242400	187.96
LUT as Logic	455605	0	242400	187.96
LUT as Memory	0	0	112800	0.00
CLB Registers	103048	0	484800	21.26
Register as Flip Flop	2696	0	484800	0.56
Register as Latch	100352	0	484800	20.70
CARRY8	381	0	30300	1.26
F7 Muxes	1599	0	121200	1.32
F8 Muxes	799	0	60600	1.32
F9 Muxes	0	0	30300	0.00

7) Synthesis Schematic and Comments:

The Synthesis Schematics are attached on the next page.

The number of LUTs generated is 654 for the conv unit, 48422 for the conv layer single filter and 455605 for the conv layer multiple filters. The number of LUTs is small for the conv unit because it contains only one processing element which contains one floating point adder and one floating point multiplier. The number of LUTs increases exponentially in the single and multiple filter modules because we make a number of instances of the conv unit equal to half the number of pixels in a single row in the output image (14 conv units for one filter for the first convolution layer of LeNet). Also, the logic used of slicing and indexing the input and output arrays to send parts of the image to these conv units and connect their output to different parts of the output of the whole module is very complicated in synthesis which increases the number of LUTs in the layer with single filter but especially in the layer with multiple filters.

8) Answers to extra questions:

I have implemented the convolution part with 5 different architectures. They are all fully functional but the changes in the design of each architecture reduced the utilization dramatically. The differences between the 5 architectures are in the design of the convolution layer that uses a single filter.

Architecture 1: Full Parallelism

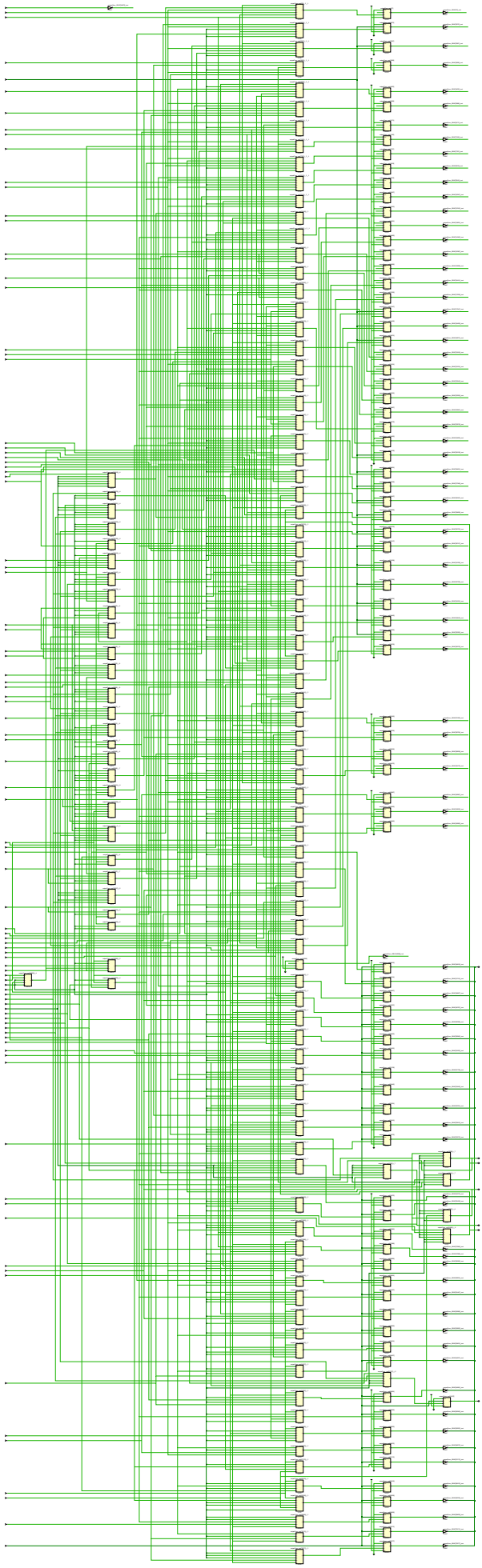
In the conv layer single filter, the number of instances of conv units is equal to the number of pixels in the output image (784 conv units for the first layer). The results appear of only 26 clock cycles (number of clock cycles need for a conv unit to finish the elementwise multiplication). This is the fastest design but the biggest design in terms of LUTs and utilization.

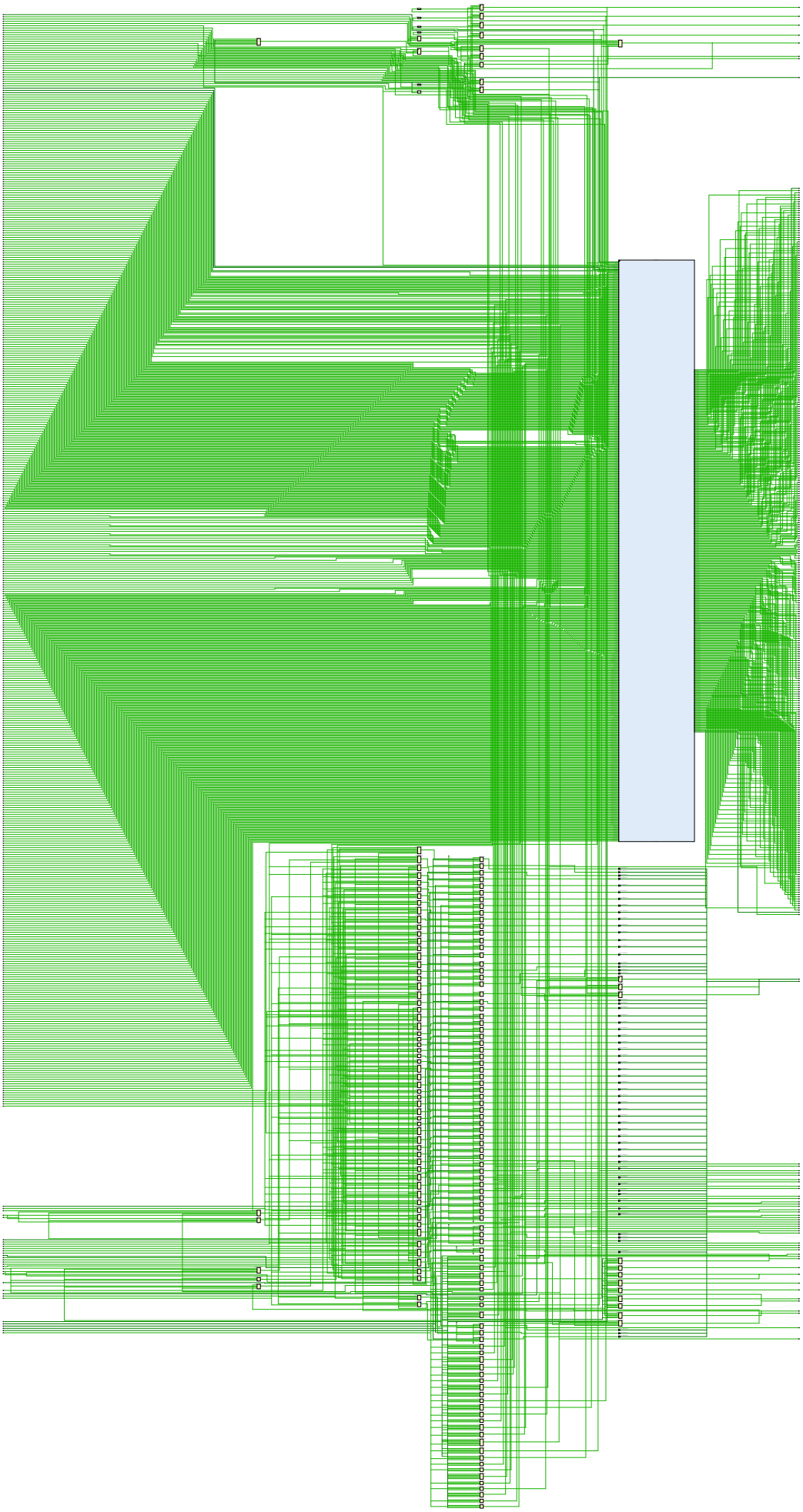
Architecture 2: Sequential Convolution with 28 conv units (for one filter of the first convolution layer LeNet)

In the conv layer single filter, the number of instances of conv units is equal to the number of pixels of one row of the output image (28 conv units for the first layer). The results appear after 728 clock cycles. This design is slower but has improved the utilization a lot.

Architecture 3: Optimized design of Architecture 2 (28 conv units also for the first layer with smaller receptive field arrays)

The parts of the image sent to the conv units were stored in a large array called the receptive field. In the previous two architectures, this array was calculated with a combinational logic for the whole input image generating the matrices sent to all the conv units at once. This huge array contained 196000 (=28*28*25) values (each one represented by 32 bits). The addition in this architecture is that we added a module RF selector that receives sequentially the row and column of the selected pixels and returns the matrices of those pixels only shrinking the size of the array to 700 (=28*25) values to improve the utilization of architecture 3.





Architecture 4: Sequential Convolution with 14 conv units (for one filter of the first convolution layer LeNet)

This architecture contains the improvements of all the previous architectures (receptive field array is calculated for the selected pixels only). The number of conv units is equal to half the number of pixels of a single row of the output image: 14 units (for the first convolution layer of LeNet). The result appears after 1456 clock cycles.

Architecture 5: Sequential Convolution with 14 conv units with half precision floating point (16 bits)

Finally, I changed the floating-point precision from 32 bits to lower the utilization in part 5 and reach the lowest utilization possible for the con layer with multiple filters.

This comparison between architectures is done on module of the conv layer single filter with the sizes and number of the first convolution layer.

	Arch 1	**Arch 2**	**Arch 3**	**Arch 4**	**Arch 5**
Floating Point Precision	32 bits	32 bits	32 bits	32 bits	16 bits
Number of clock cycles need to output the result	26	728	728	1456	1456
Number of LUTS	1712256	177300	135176	117521	48422
Utilization on the KCU	707.54%	73.26%	55.86%	48.48%	19.98%
Utilization on the ZCU	738.04%	77.09%	58.77%	51.10%	21.05%

Note: the reports and codes for each architecture can be found in \Extra Convolution\ folder

Some reports show the synthesis on the PYNQ board and I used the number of LUTS to calculate the utilization for the KCU and ZCU boards.

Part 2

TanH Activation

Ahmed Ezzat

Name: Ahmed Mohammed Ezzat ID: 1162033

Logic-2 tanh Function

Section1

Design

Mainly there is a whole layer which is in the module " UsingTheTanh.v" , it controls the smaller processing unit "HyperBolicTangent.v".

How It Works

Mainly to save some hardware but in the opposite side clock cycles will be extended.

In the Hyperbolic Tangent:

I divide the operations into 3 **Multipliers and 1 Adder.**

First, the "MSquaring" is for getting the incrementing term of the x term which is x^2.

Second, the "MGeneratingXterm" is for multiplying the present x term with its next one.

Third, the "MTheCoefficientTerm" is for multiplying each corresponding final X term with its coefficient which here I chose 4 terms for good taylor expansion coeff.

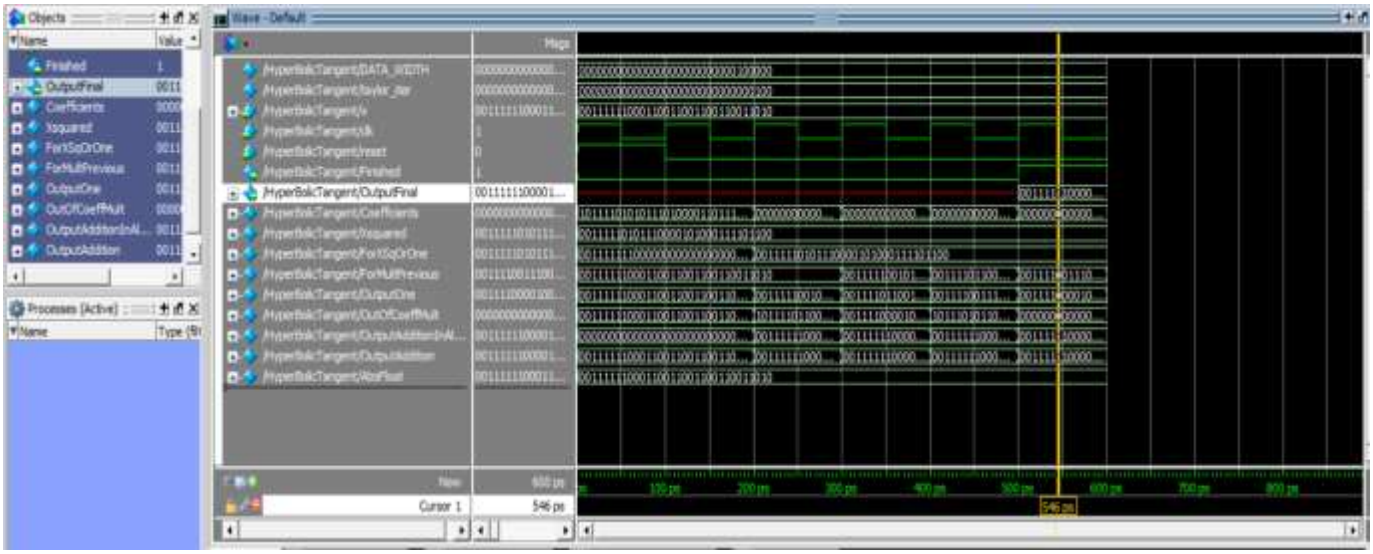
Fourth, the last step of our operation is to add each resulting term to its previous one.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	1700	0	53200	3.20
LUT as Logic	1700	0	53200	3.20
LUT as Memory	0	0	17400	0.00
Slice Registers	139	0	106400	0.13
Register as Flip Flop	139	0	106400	0.13
Register as Latch	0	0	106400	0.00
F7 Muxes	0	0	26600	0.00
F8 Muxes	0	0	13300	0.00

Name: Ahmed Mohammed Ezzat ID: 1162033

Section 4&5

This is the simulation of an inrange floating number: it needs 5 clock cycles



The input here is 0.600000023842 in floating numbers is 00111111000110011001100110011010

The output here:

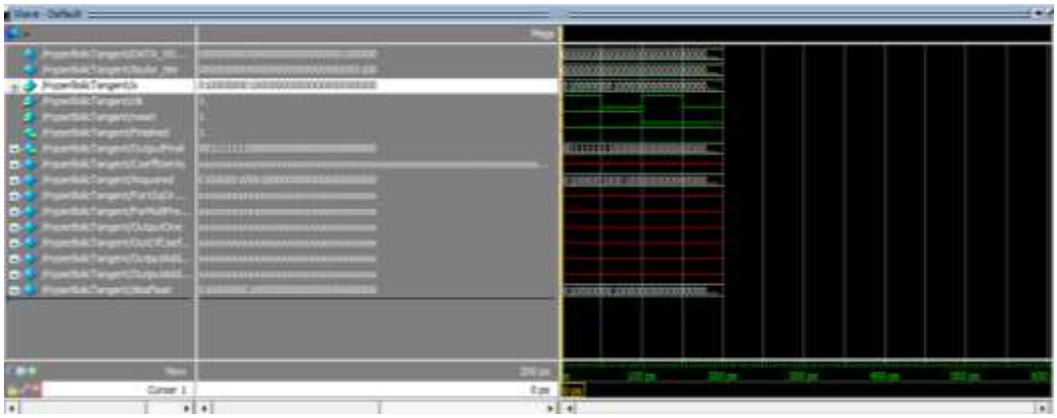
1. Actual is 0.53704 in floating is 00111111000010010111101101110100
2. Experimental is 0.53685 in float 00111111000010010110111101111011

Which is a very slight difference regarding Taylor approx.

Now tanh converges at |x|>pi/2 so in this case I made a constant assigned value to numbers in convergence region which is 1 or -1

Here is an example

This is the simulation of an out of range floating number: it needs only 1 clock cycle



Here the input is 3 in floating 01000000010000000000000000000000:

1- The Actual output is 0.995

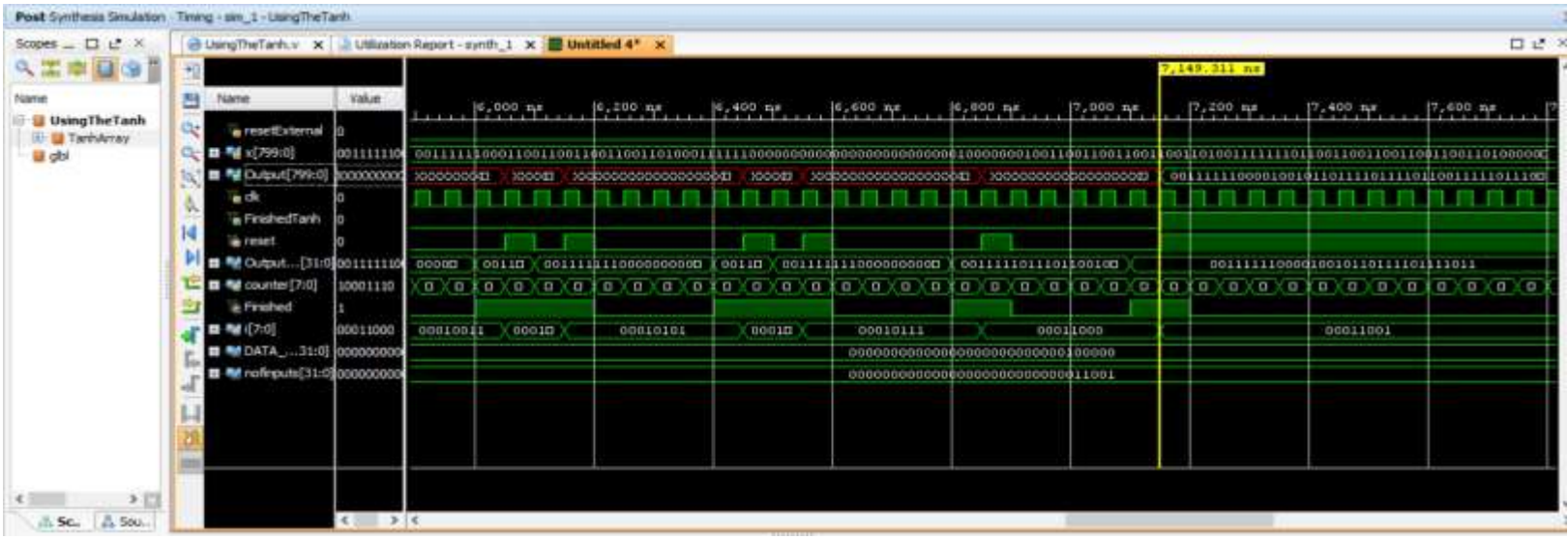
Name: Ahmed Mohammed Ezzat ID: 1162033

2- The simulation output is 1 because it lies in the convergence region

Now here comes the tanh layer:

Regarding the layer, the input is in kernels in the size of 25 inputs:

Post Synthesis-Timing Simulation



Here my input is:

```
InputNumbers = [0.6,0.5,3.2,1.4,2,1,0,0.5,-0.4,-0.5,-0.57,0.57,0.3,0.2,0.112,
0.2,-0.2,-0.5,-0.9,-0.567,-0.43,0.5,0.7,0.86543,0.4345,]
```

In IEEE Form:

00111111000110011001100110011010_00111111100000000000000000000000_01000000010011001
100110011001101_00111111101100110011001100110011_01000000000000000000000000000000_0
01111111000000000000000000000000_00000000000000000000000000000000_00111111000000000
00000000000000_10111110110011001100110011001101_10111111000000000000000000000000_10
11111100010001111010111000010 1_00111111000100011110101110000101_00111110100110011001
100110011010_00111110010011001100110011001101_00111101111001010110000001000010_001
11110010011001100110011001101_10111110010011001100110011001101_10111111000000000000
000000000000_10111111011001100110011001100110_10111111000100010010011011101001_1011
11101101110000101000111110110_00111111100000000000000000000000_00111111001100110011
01100110011_00111111010111011000110011010010_00111110110111100111011011001001

As said before,

The number of clock cycles is 1 clock for each converged input and 5 cycles for each non-converged input so it varies according to the input array

Name: Ahmed Mohammed Ezzat ID: 1162033

And here are the Outputs:

Simulation Output is =
00111111000010010110111101111011_00111110111011001001010110001011_00111111100000000
000000000000000_00111111001000100010111101000110_00111111100000000000000000000000_0
01111110011111011111011111110001_00000000000000000000000000000000_00111110111011001
01010110001011_10111110110000101000011111111000_10111110111011001001010110001011_10
11111100000011111001101000101 1_00111111000000111110011010001011_00111110100101010
0011011100000_00111110010010100001110011000010_00111101111001000110101111110101_001
11110010010100001110011000010_10111110010010100001110011000010_10111110111011001001
010110001011_10111111001101011011110101000011_10111111000000110101011001001011_1011
11101100111110000100111110000_00111110111011001001010110001011_00111111000110101000
11110011001_00111111001100011100100100101110_00111110110100010111000011010101

And in decimals:
Outputs=0.536857306957,0.462078422308,1.0,0.633533835411,1.0,0.746031820774,0,0.462078422308,-0.379943609238,-0.462078422308
,-0.51523655653,0.51523655653,0.291312217712,0.197375327349,0.111534036696,0.197375327349
,-0.197375327349,-0.462078422308,-0.709919154644,-0.513035476208
,-0.405311107635,0.462078422308,0.603631556034,0.694476008415,0.409063965082

And the Actual Outputs are:

Actual Output is =
00111111000010010111110000010101_00111110111011001001101010011111_00111111011111110
010011010010100_00111111011000101010011001101000_00111111011101101100101010000011_0
01111110100001011110111111010110_00000000000000000000000000000000_00111110111011001
01101010011111_10111111011000010100010001010 1100_10111110111011001001101010011111_10
11111100000011111011010010110_00111111000000111110111010010110_00111110100101010 01
0011011101101_00111110010010100001110011000010_00111101111001000110101111110100_001
11110010010100001110011000010_10111110010010100001110011000010_10111110111011001001
101010011111_10111111001101110101111101001100_10111111000000110101110111111001_1011
11101100111110000011001000110_00111110111011001001101010011111_00111111000110101011
01111011001_00111111001100101111010010011001_00111110110100010111001001001100

And in decimals:
Outputs = 0.53704957,0.46211716,0.9966824,0.88535165,0.96402758,0.76159416,0,0.46211716,-0.37994896,-0.46211716,-0.51535928,0.51535928,0.29131261,0.19737532,0.11153403,0.19737532,-0.19737532,-0.46211716,-0.71629787,-0.51315266,-0.40532131,0.46211716,0.60436778,0.69904478,0.40907515

Name: Ahmed Mohammed Ezzat ID: 1162033

Section 6

Now here comes the Utilization of One kernel:

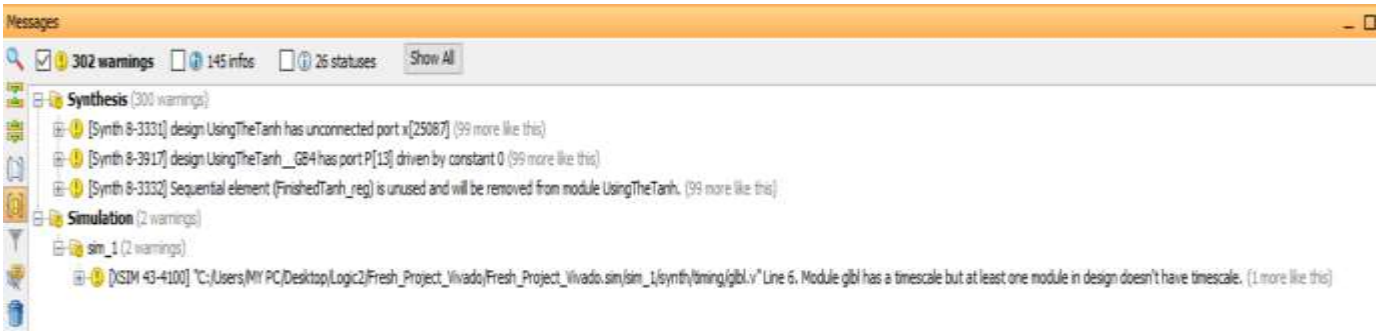
25 inputs kernel

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	2279	0	53200	4.28
LUT as Logic	2279	0	53200	4.28
LUT as Memory	0	0	17400	0.00
Slice Registers	964	0	106400	0.91
Register as Flip Flop	964	0	106400	0.91
Register as Latch	0	0	106400	0.00
F7 Muxes	96	0	26600	0.36
F8 Muxes	0	0	13300	0.00

And to be more real the utilization of a full 748 pixels image is:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	22066	0	53200	41.48
LUT as Logic	22066	0	53200	41.48
LUT as Memory	0	0	17400	0.00
Slice Registers	16712	0	106400	15.71
Register as Flip Flop	16712	0	106400	15.71
Register as Latch	0	0	106400	0.00
F7 Muxes	1088	0	26600	4.09
F8 Muxes	544	0	13300	4.09

TCL Console:



These warning mean nothing I think these are due to constant driven parameters.

Name: Ahmed Mohammed Ezzat ID: 1162033

File Description Table:

HyperBolicTangent.v	It is considered the processing unit of the layer which performs the tanh operation per input [32bits]
HyperBolicTangent_TB.v	It is the test bench for the tanh operation and verifying that it is true and working either in convergence or not
UsingTheTanh.v	It is the tanh layer which takes a whole array of inputs and produces a whole array of outputs
UsingTheTanh-TB.v	It is the test bench for the layer of the tanh and its verification of how it works on an array of inputs

Parallelism here will over utilize the module and will result in some complications, so I preferred not to implement it in the tanh layer. Also the utilization of my module for a whole input image is 22K LUTs which is small compared to other layers so it is enough I think.

Name: Ahmed Mohammed Ezzat ID: 1162033

Section 7

The shown schematic shows all modules, Using the tanh module and hyperbolic tangent module and how they are connected together within the controlling layer:

The Hyperbolic tangent uses 1700 LUT's, Multiplier uses 340 LUT's, Adder uses 540 LUTs and finally the Using the tanh uses at 784 input 22066 LUT's.

The final number of LUT's can be reduced if the precision of the tanh reduced to 16 bit without a big loss in the accuracy of the results.

I actually reduced the input bits from 32-bits to 16-bits and Part 5 now uses them

Part 3
SoftMax Activation

Omar Essam

SoftMax Report

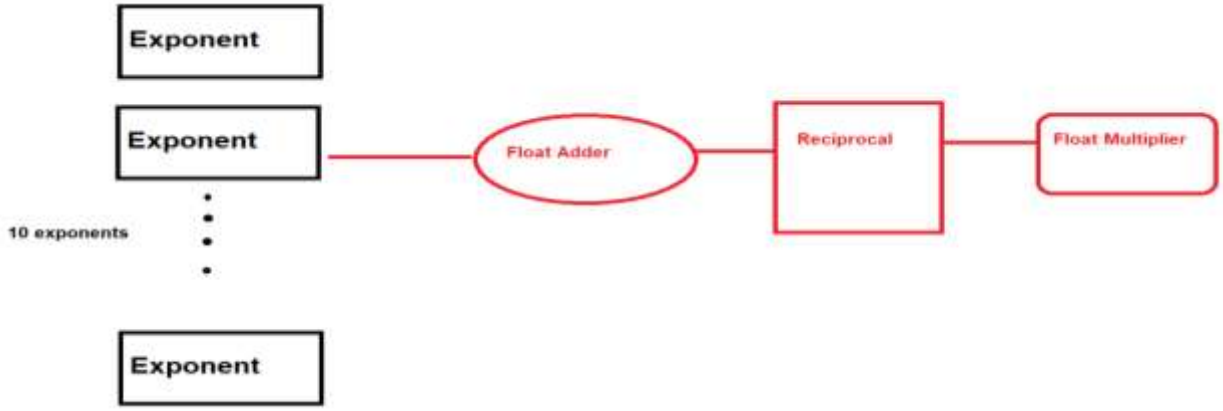
I will discuss three different implementations of softmax and make a comparison.

1. Design 1: softmax module
2. Design 2: softmax_1 module
3. Design 3: softmax_2 module.

All Answers were double checked with a python script and are correct.

Design 1

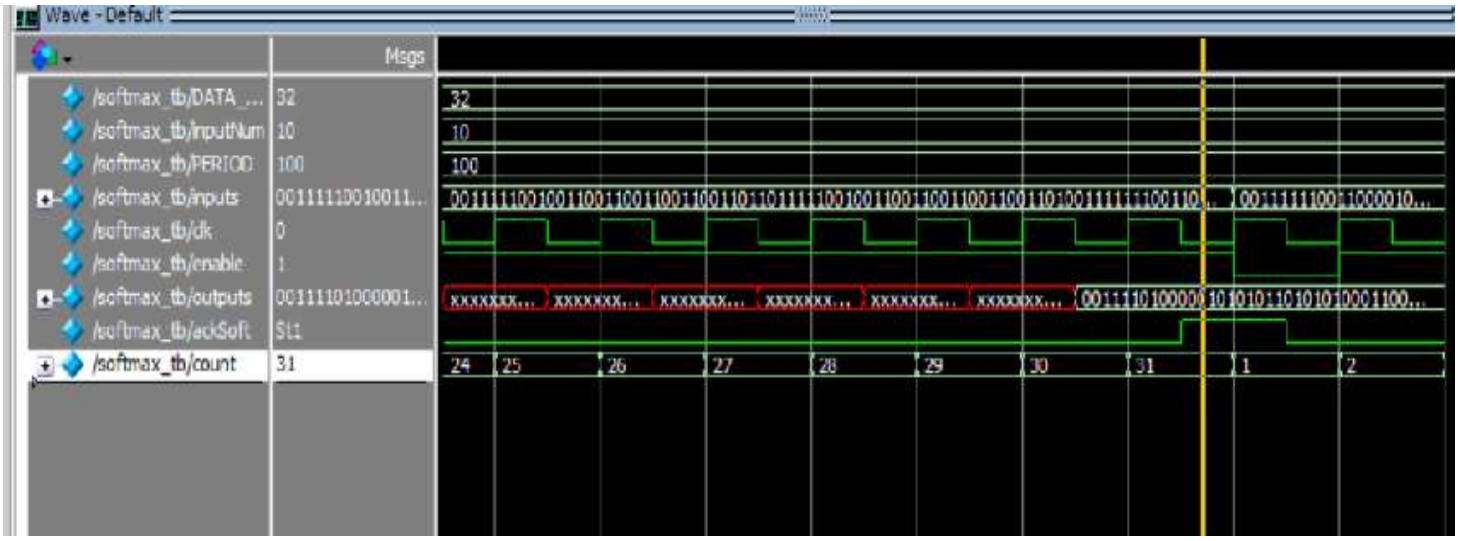
1- Block Diagram:



Design Description: 10 exponent units, 1 floating adder , 1 multiplier, 1 division unit.

Description : 10 parallel exponent units to calculate exponent of 10 inputs, then 1 floating adder will used with 10 clock cycles to calculate their sum and send an enable signal to Newton Raphson module which will calculate the Reciprocal after the Newton Raphson module sends the ackDiv signal , the 1 multiplier will be used for 10 clock cycles to multiply the 10 exponents with the 10 reciprocals.

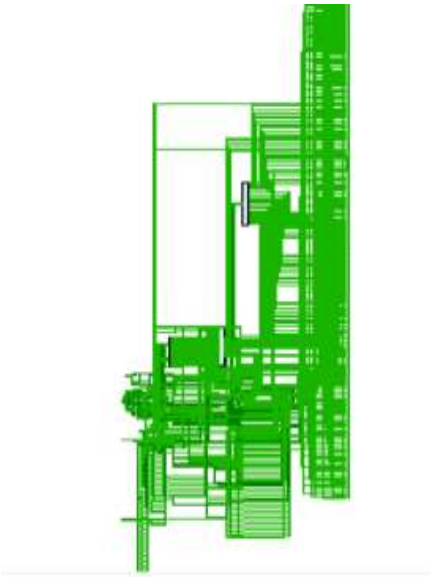
5- Clock Cycles: 31 clock cycles as shown below. ackSoft is triggered



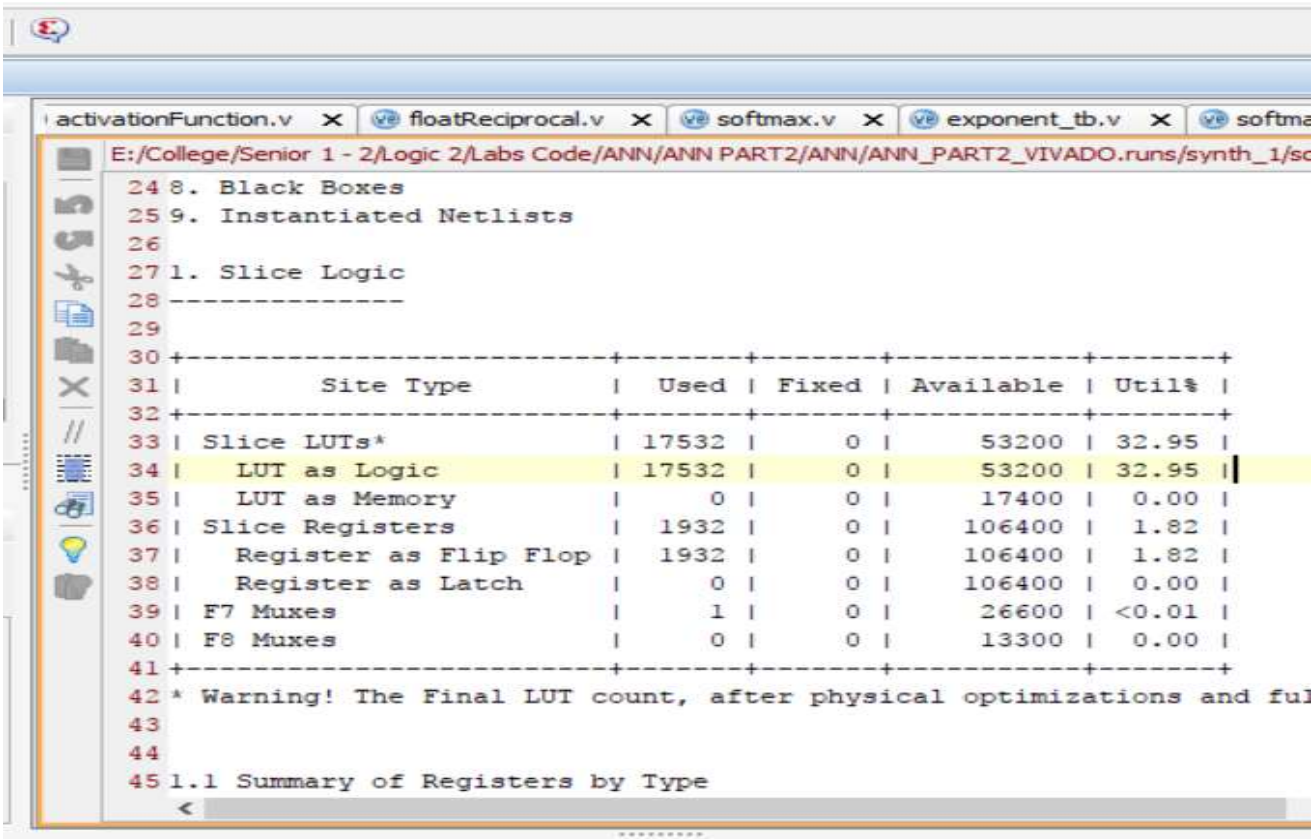
7 – Synthesis schematic

The shown schematic shows all modules , exponent and reciprocal and adder and multiplier and how they are connected together with extra sequential circuits. : The exponent uses 1356 LUTs , Reciprocal (Newton Raphson) is 2600 LUTs , Multiplication is 340, Addition is 540. Which is the reason the design uses 17k luts.

The overall design luts could be improved If the base unit , Multipliers and adders are smaller



6- Synthesis Report: 17532/53200 LUTs 32.95% utilization.



activationFunction.v | floatReciprocal.v | softmax.v | exponent_tb.v | softma

E:/College/Senior 1 - 2/Logic 2/Labs Code/ANN/ANN PART2/ANN/ANN_PART2_VIVADO.runs/synth_1/sc

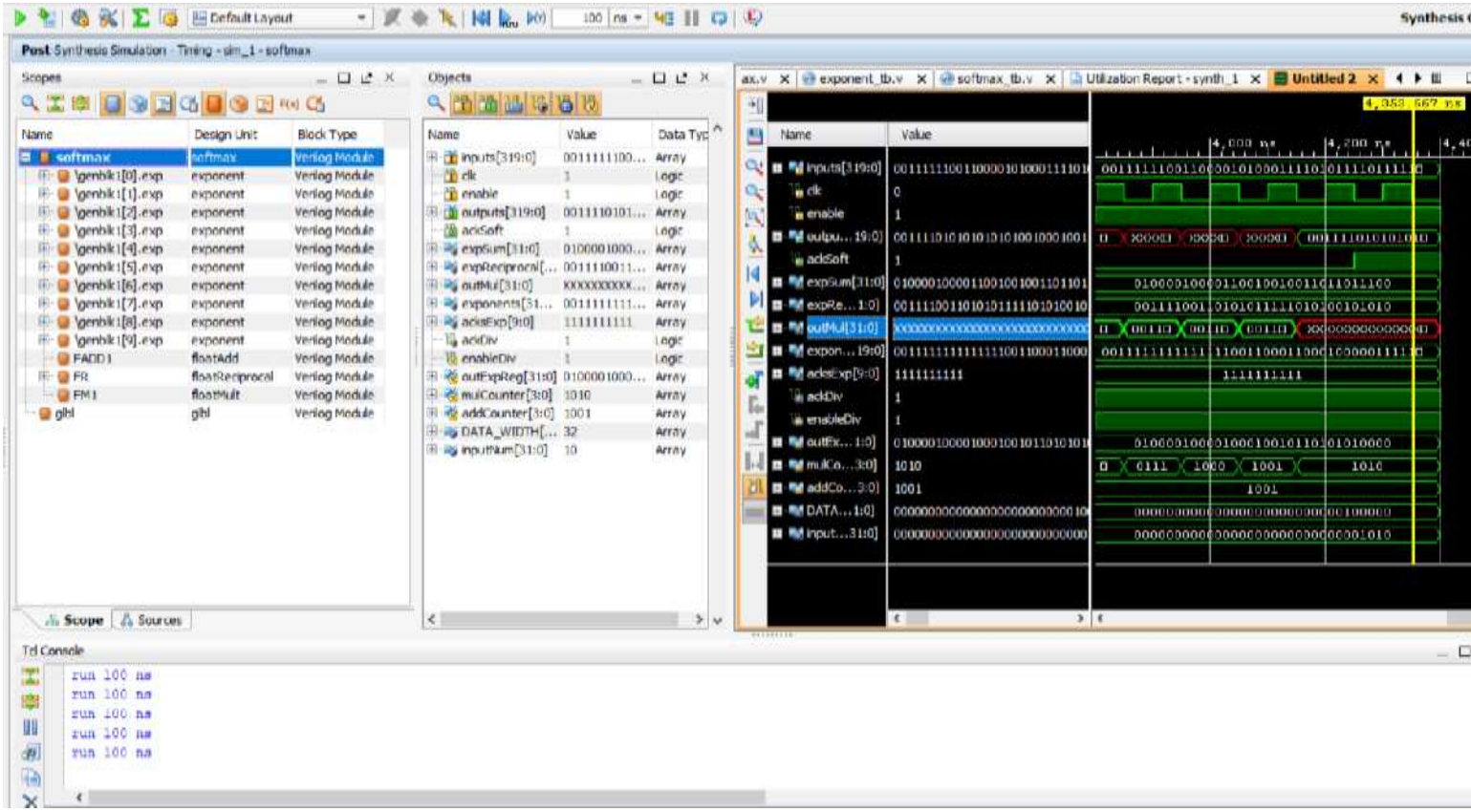
```
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+-------------------------+-------+-------+-----------+-------+
|        Site Type        |  Used | Fixed | Available | Util% |
+-------------------------+-------+-------+-----------+-------+
| Slice LUTs*             | 17532 |     0 |     53200 | 32.95 |
|   LUT as Logic          | 17532 |     0 |     53200 | 32.95 |
|   LUT as Memory         |     0 |     0 |     17400 |  0.00 |
| Slice Registers         |  1932 |     0 |    106400 |  1.82 |
|   Register as Flip Flop |  1932 |     0 |    106400 |  1.82 |
|   Register as Latch     |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                |     1 |     0 |     26600 | <0.01 |
| F8 Muxes                |     0 |     0 |     13300 |  0.00 |
+-------------------------+-------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and ful


1.1 Summary of Registers by Type
```

4 - Timing Diagram

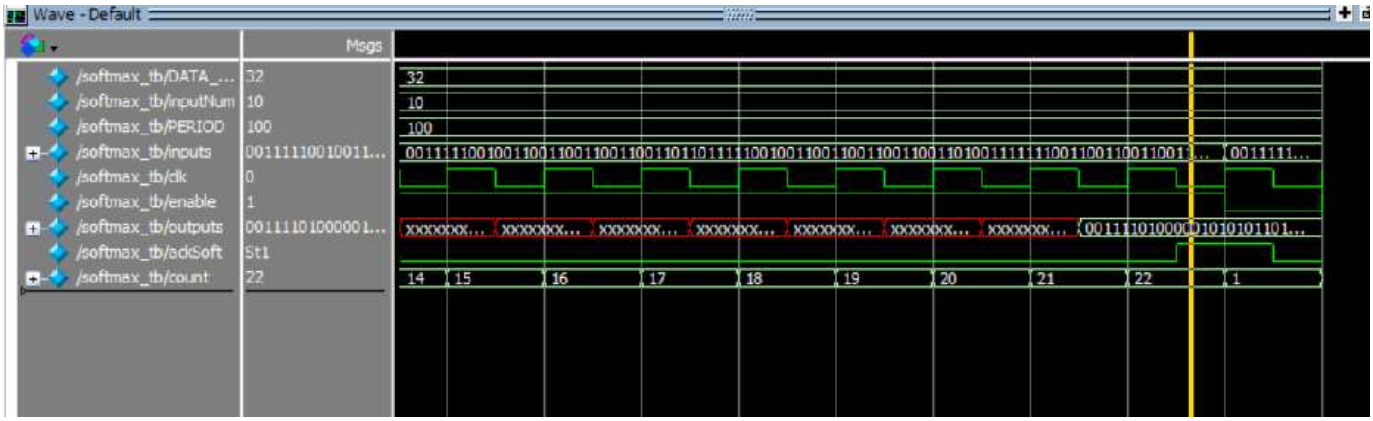


Design 2

Design Description: 10 exponent units, 10 floating adder , 1 multiplier, 1 division unit.

5- Clock Cycles: 22 clock cycles as shown below, ackSoft is triggered

Description: same as Design 1 the only difference is 10 adders are generated to calculate the sum in one cycle.



6 -Synthesis Report: 22417/53200 LUTs 42.14% utilization

activationFunction.v | floatReciprocal.v | softmax.v | exponent_tb.v | softmax_tb.v | Utilization Report -

E:/College/Senior 1 - 2/Logic 2/Labs Code/ANN/ANN PART2/ANN/ANN_PART2_VIVADO.runs/synth_1/softmax1_utilization_synth.rpt

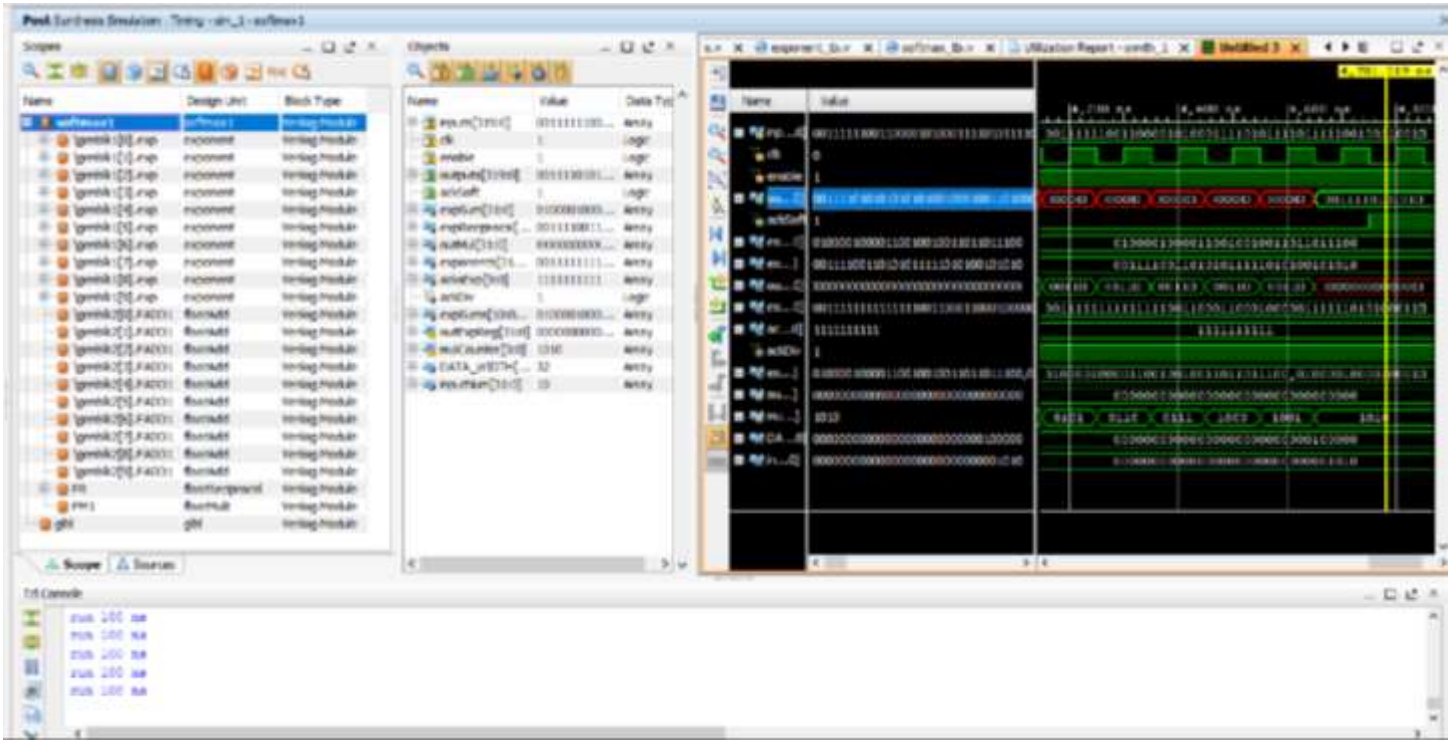
```
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs*                | 22417 |     0 |     53200 | 42.14 |
|   LUT as Logic             | 22417 |     0 |     53200 | 42.14 |
|   LUT as Memory            |     0 |     0 |     17400 |  0.00 |
| Slice Registers            |  1894 |     0 |    106400 |  1.78 |
|   Register as Flip Flop    |  1894 |     0 |    106400 |  1.78 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |     1 |     0 |     26600 | <0.01 |
| F8 Muxes                   |     0 |     0 |     13300 |  0.00 |
+----------------------------+-------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typica

1.1 Summary of Registers by Type
```

4- Timing Diagram:

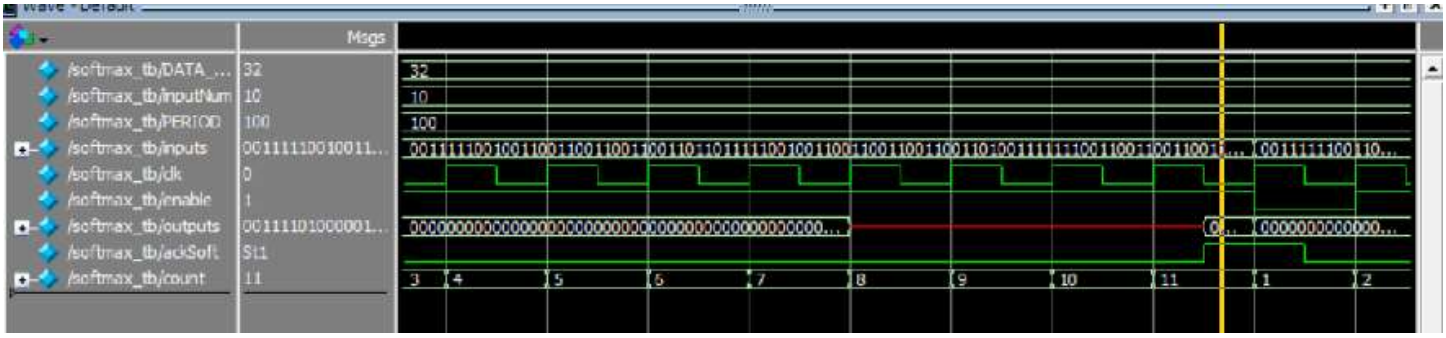


Design 3

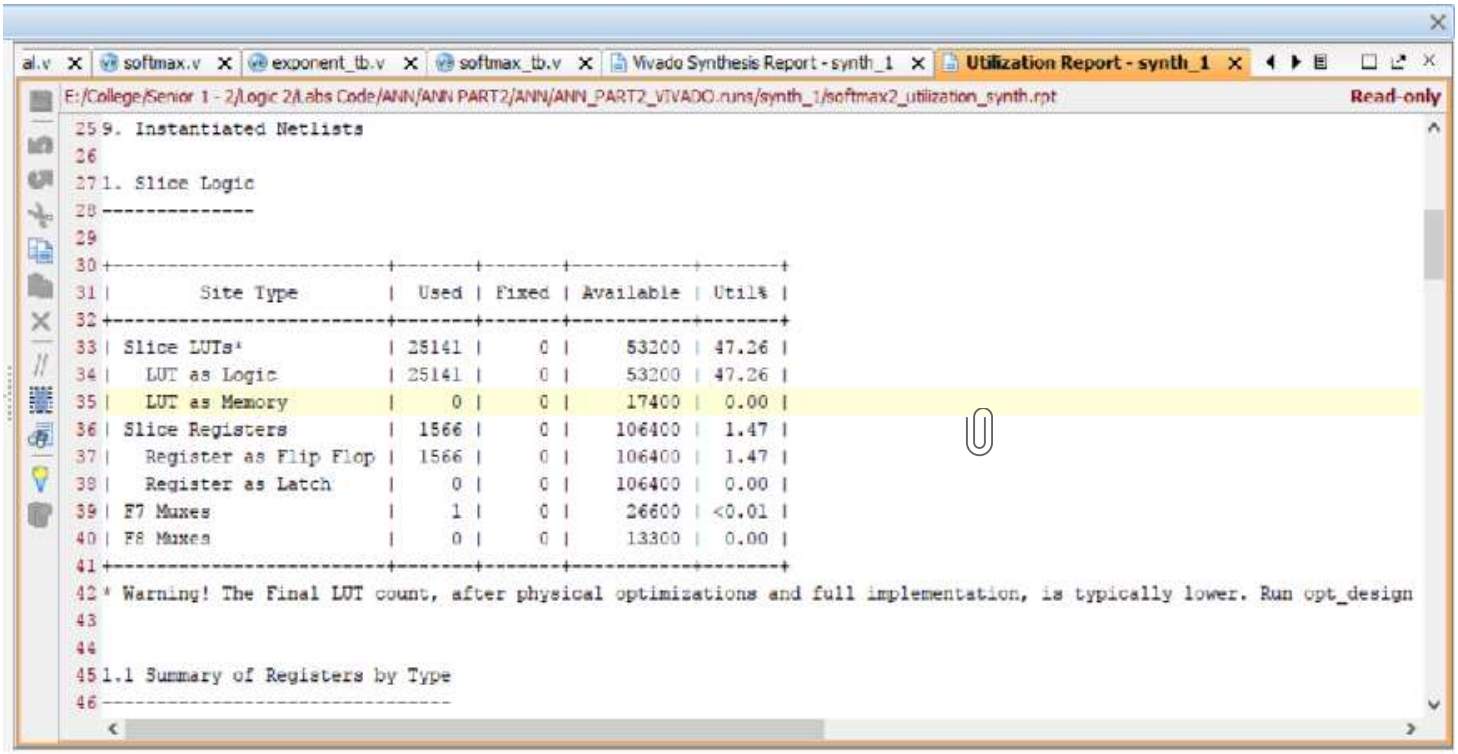
Design Description: Design Description: 10 exponent units, 10 floating adder , 10 multiplier, 1 division unit.

5- Clock Cycles : 11 cycles as shown. ackSoft is triggered

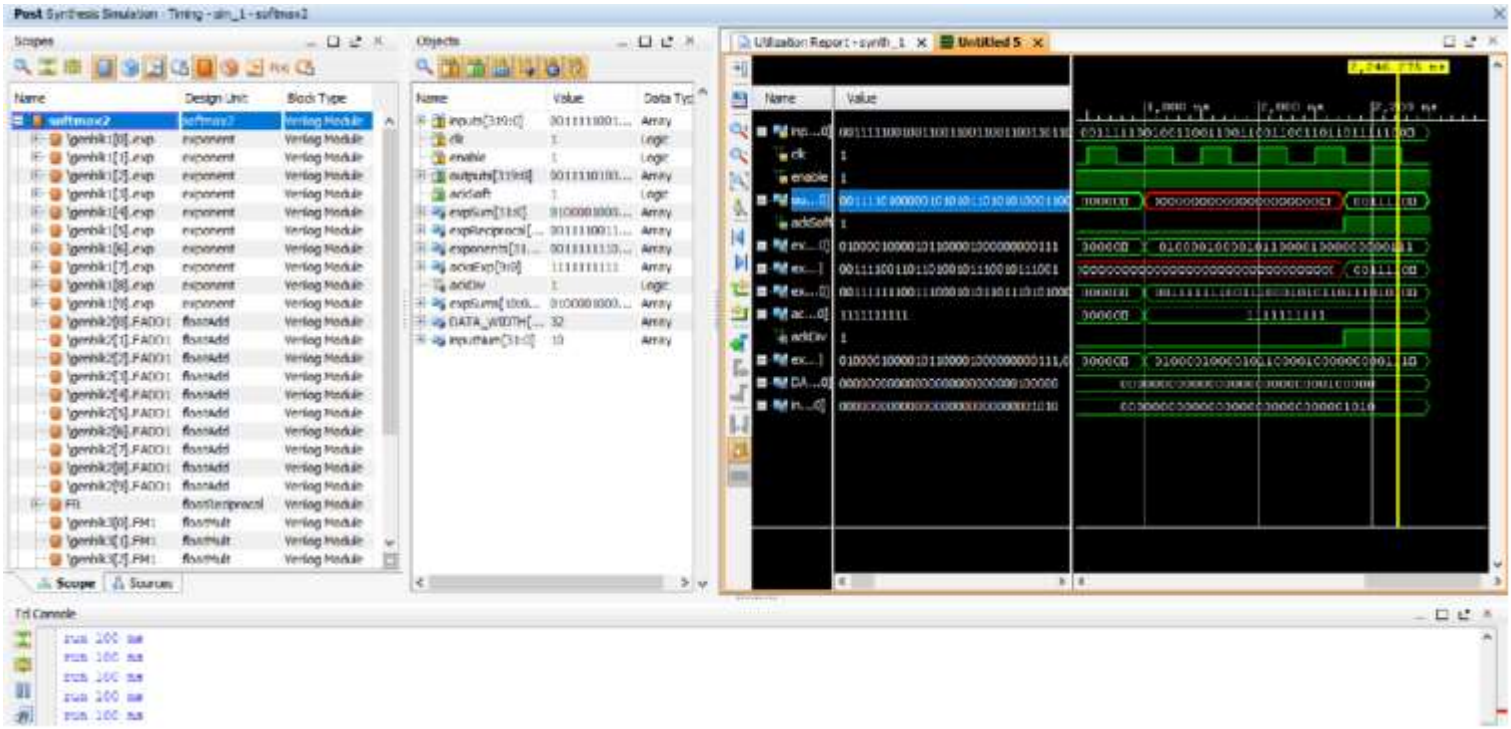
Description: Same as Design 2 but 10 parallel multipliers are used to multiply with the Reciprocal in one clock cycle.



6- Synthesis Report: 25141/53200 47.26% utlization



4- Timing Diagram



Completion of part 4: Verification that the design is working

In this section I will discuss how I verified that the design is working using python scripts and Floating Point to IEEE converter. https://www.h-schmidt.net/FloatConverter/IEEE754.html

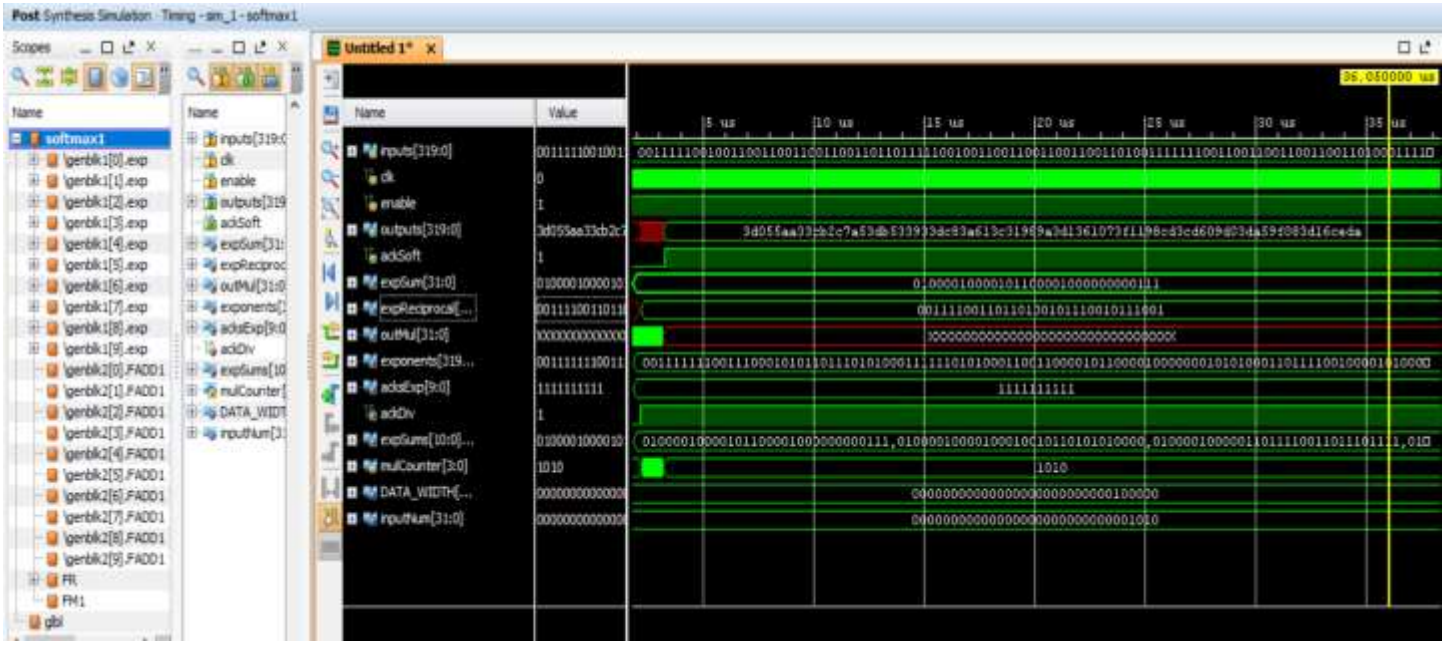
```
The 10 inputs we will use to verify are 0.2 -0.2 1.2 1.3 -0.9 0.3 3.1 -0.02 1.11
0.323
```

The input is then run through the softmax.py file and the input binary string which is the input of the module in the post synthesis simulation is written to log.txt.

In our case the input string is

00111110010011001100110011001101101111100100110011001100110011001101001111111001100110011001
10011010001111111010011001100110011001100110101111110110011001100110011001100011111101001100
10011001100110100100000001000110011001100110011001101011110010100011110101110000101000111111
10001110000101000111101100111110101001010110000001000010

In the figure below is the output of the system after the ackSoft signal is High.



Hexadecimal result:
3d055aa33cb2c7a53db533933dc83a613c31989a3d1361073f1198cd3cd609d03da59f083d16ceda

Binary result:

00111101000001010101101010100011001111001011001011000111101001010011110110110101001100111001001100111101110010000011101001100001001111000011000110011000100110100011110100010011011000100000111001111110001000110011000110011010011110011010110000010011101000000011110110100101100111111000010000011110100010110110011101101101010

The binary result is then used in the split.py file to get 10 IEEE 32 bit floating point numbers.

```
The result of splitting this string is ['00111101000001010101101010100011',
'00111100101100101100011110100101', '00111101101101010011001110010011',
 '00111101110010000011101001100001', '00111100001100011001100010011010',
'00111101000100110110000100000111',
 '00111111000100011001100011001101', '00111100110101100000100111010000',
'00111101101001010011111100001000',
 '00111101000101101101001110111011010']
Which corresponds to these numbers
[0.0325571410358, 0.0218237135559, 0.0884772762656, 0.097767598927, 0.0108396057039,
0.0359812043607, 0.56873780489, 0.0261277258396, 0.0808697342873, 0.0368183627725]
```

Now after running the softmax.py we get two results , first is the softmax numbers if we use math.exponent, and softmax if we use taylor.

Here is the results If we use actual exponent function.

```
RealSoftmax is [0.03182415 0.02133236 0.086507   0.09560502 0.01059334 0.03517112
           0.57837666 0.02553948 0.07906144 0.03598943]
```

Here is the results if we use taylor exponent used in the design.

```
TaySoftmax is [0.03255714 0.0218237  0.08847725 0.09776757 0.0108396  0.0359812
           0.56873776 0.02612771 0.08086971 0.03681836]
```

The Tay softmax results are very similar to the real softmax considering the approximation we used for the exponent function. And also the Tay softmax results are the same as the output of the softmax module in the post synthesis simulation.

Part 4

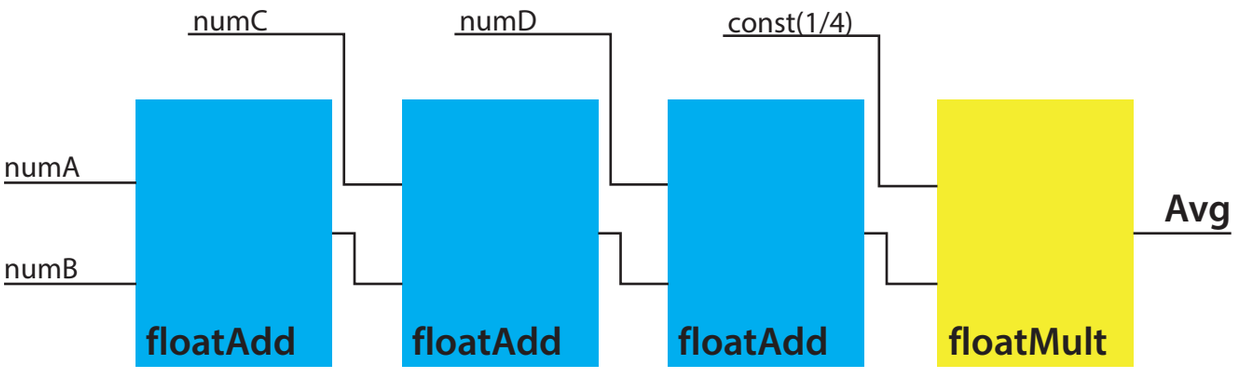
Average Pool

Ahmed Abdulkader

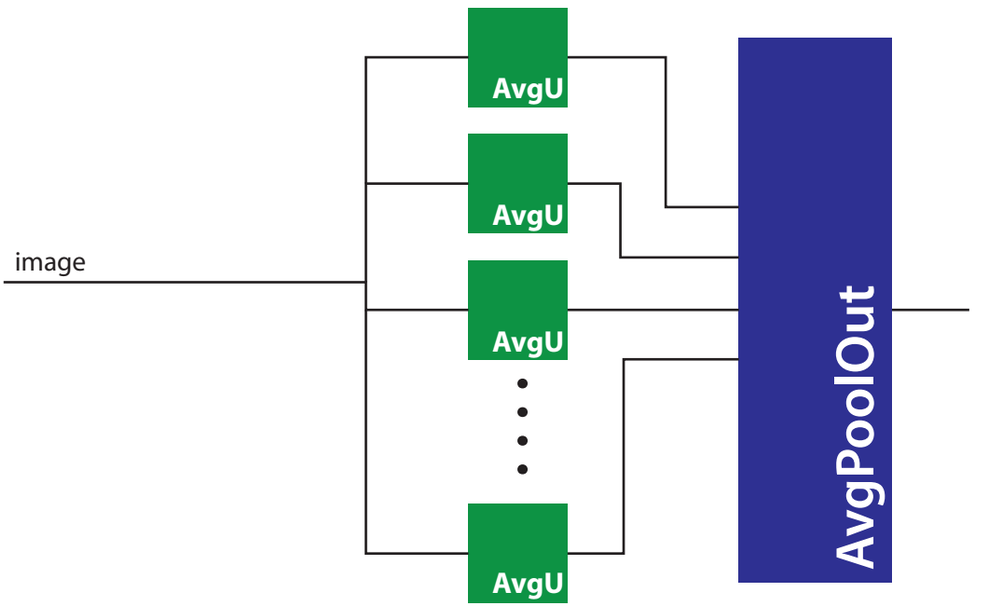
Part 4-Average Pooling

1) Block Diagrams

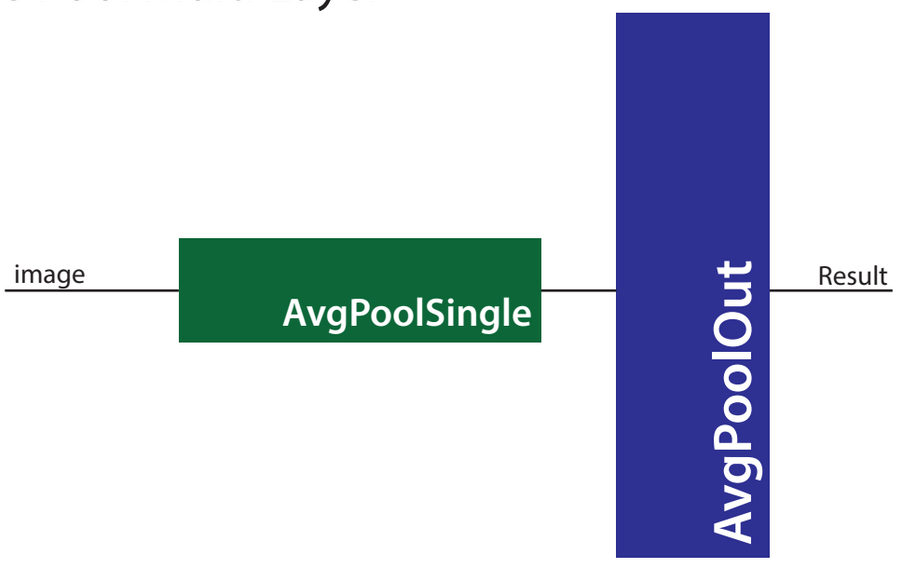
Averaging Unit



Average Pool layer



Average Pool Multi Layer



2) Verilog Codes

AvgUnit.v

AvgPoolSingle.v

AvgPoolMulti.v

3) Testbench Codes

AvgUnit_tb.v

AvgPoolSingle_tb.v

AvgPoolMulti_tb.v

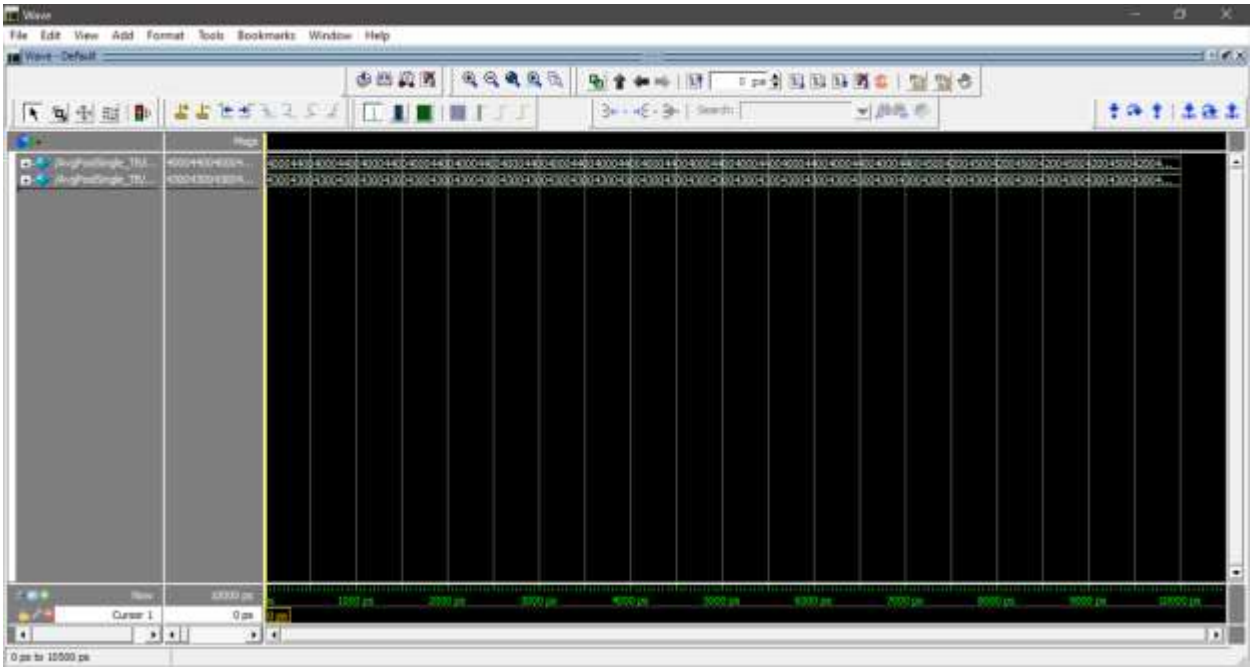
AvgPoolMulti_tb2.v

4) Screenshots:

a. Simulation (Waveform/TCL console):

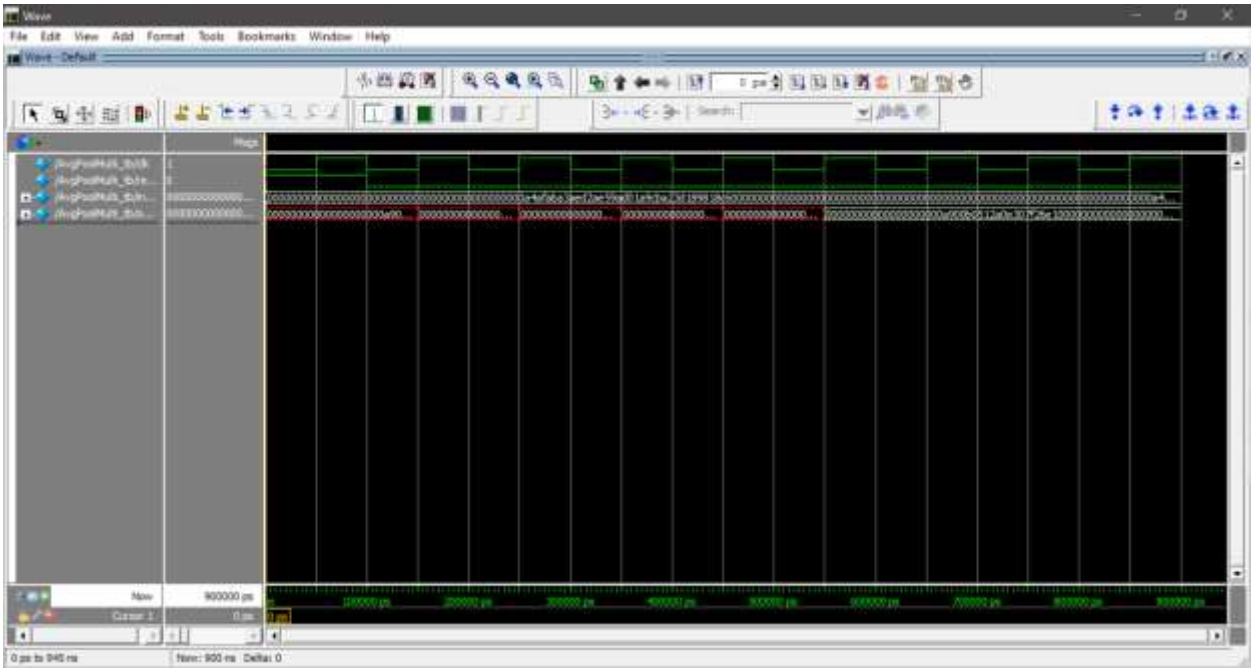
i. AverageUnit

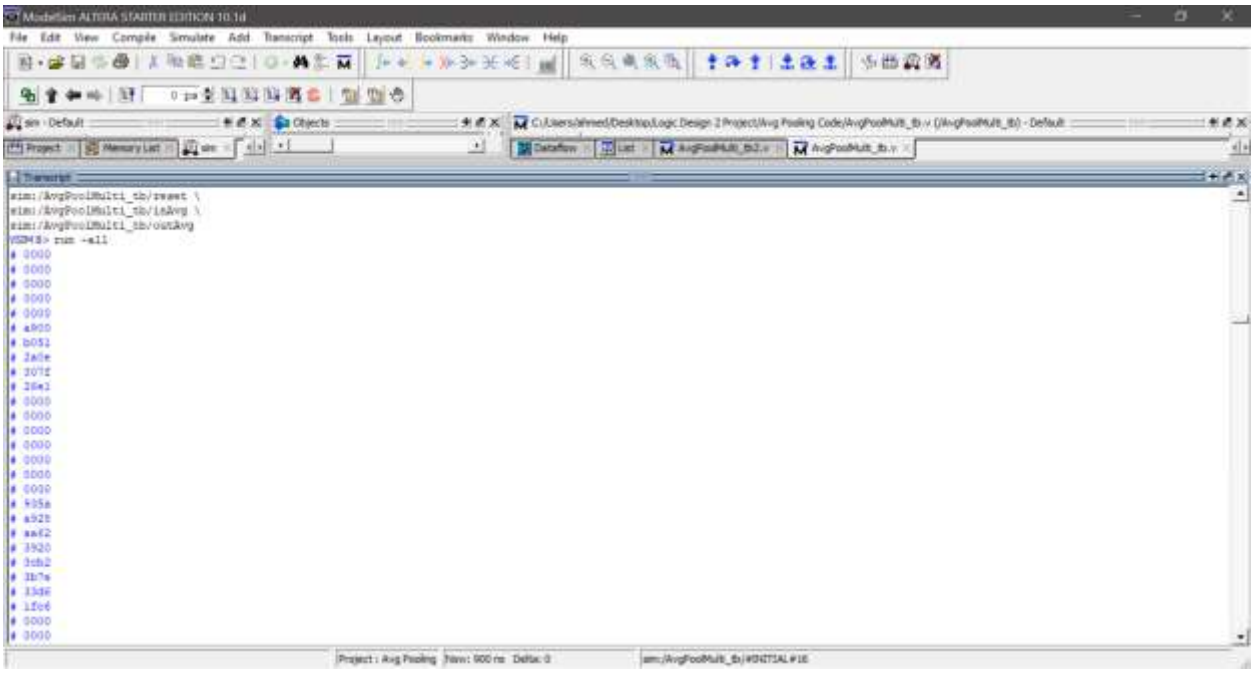
ii. Single Average Layer



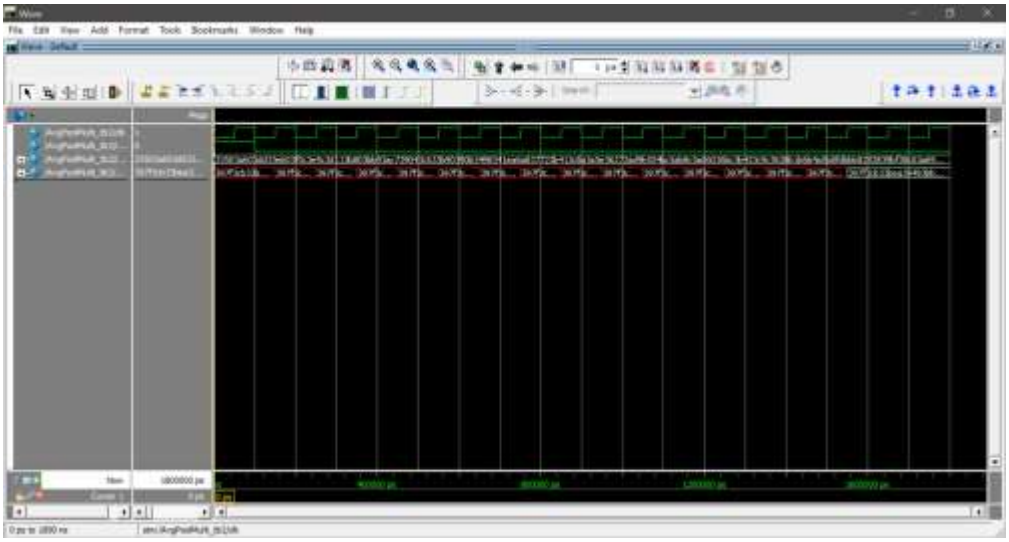
iii. Multi Average Layer

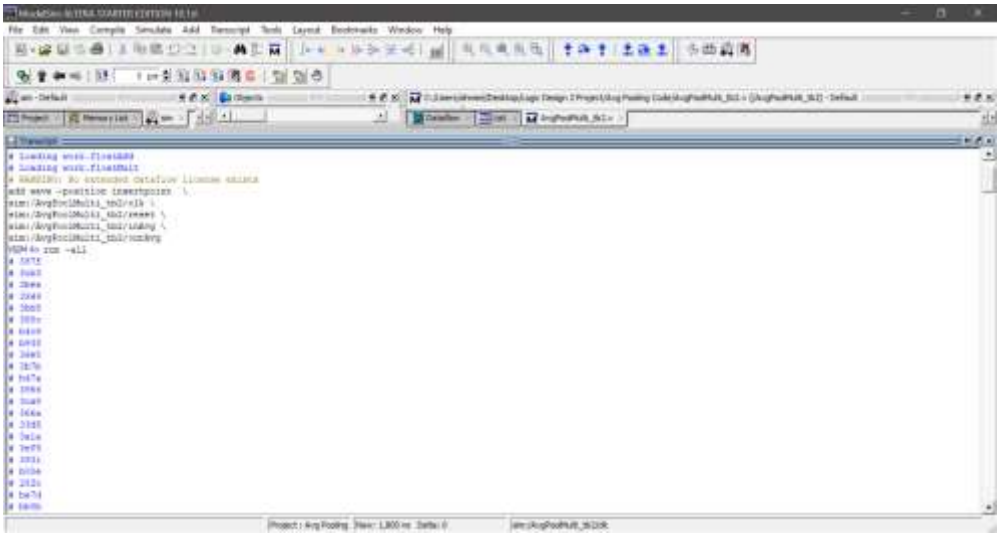
1. Test Layer 1





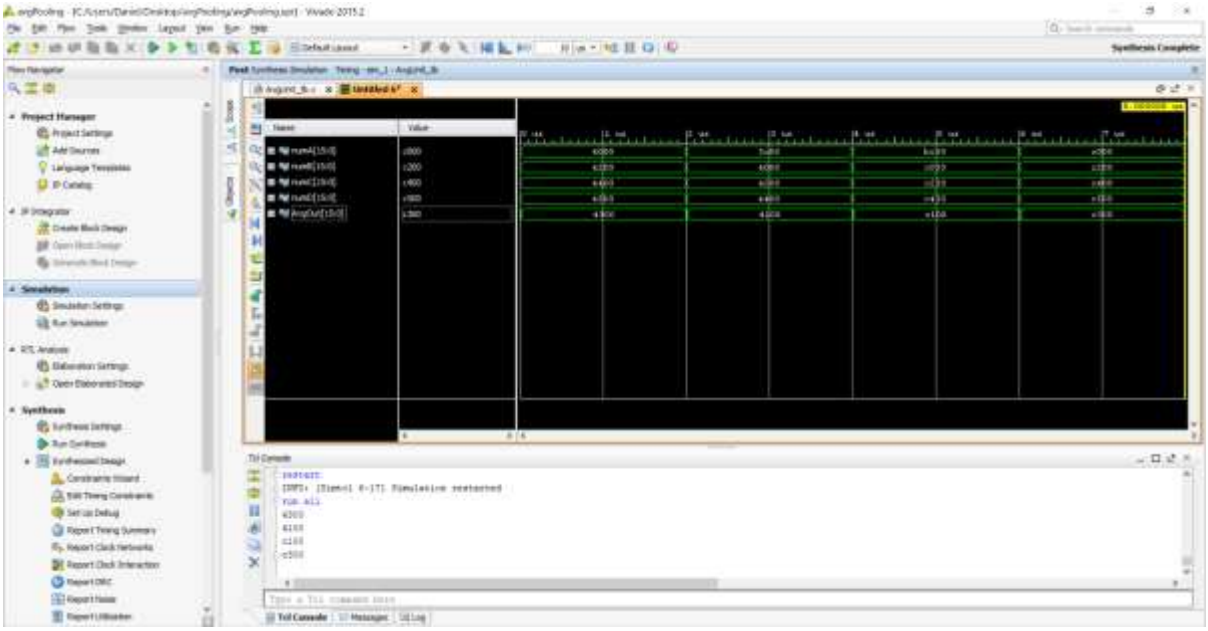
2. Test Layer 2





All results were verified using the simulated Average Pool Layer run on python. The Simulations are run on two test benches (TB1 and TB2). You can find the python script in the folder Python Tests called ‘AvgPoolTest1.py’ and ‘AvgPoolTest2.py’. All result comparisons are also listed out on the Convolution1Test sheet on the attached CNNtest excel file (CNNtest.xlsx).

b. Post Synthesis Timing for the basic avgUnit:



(as agreed upon post timing for larger layers of hardware are not achievable with our hardware capabilities)

5) Clock Cycle Count:

a. Average Unit

0 Clock Cycles, The circuit is combinational

b. Single Pool Layer

0 Clock Cycles, All Combinational Units in Parallel

c. Multi Pool Layer

Clock Cycles equal to the depth

6 for the first AvgPool

16 for the second AvgPool

6) Synthesis Utilization Report

Avg Unit

Site Type	Used	Fixed	Available	Util%
CLB LUTs*	748	0	242400	0.31
LUT as Logic	748	0	242400	0.31
LUT as Memory	0	0	112800	0.00
CLB Registers	0	0	484800	0.00
Register as Flip Flop	0	0	484800	0.00
Register as Latch	0	0	484800	0.00
CARRY8	18	0	30300	0.06
F7 Muxes	0	0	121200	0.00
F8 Muxes	0	0	60600	0.00
F9 Muxes	0	0	30300	0.00

Avg Pool Layer

Site Type	Used	Fixed	Available	Util%
CLB LUTs*	146848	0	242400	60.58
LUT as Logic	146848	0	242400	60.58
LUT as Memory	0	0	112800	0.00
CLB Registers	0	0	484800	0.00
Register as Flip Flop	0	0	484800	0.00
Register as Latch	0	0	484800	0.00
CARRY8	3528	0	30300	11.64
F7 Muxes	0	0	121200	0.00
F8 Muxes	0	0	60600	0.00
F9 Muxes	0	0	30300	0.00

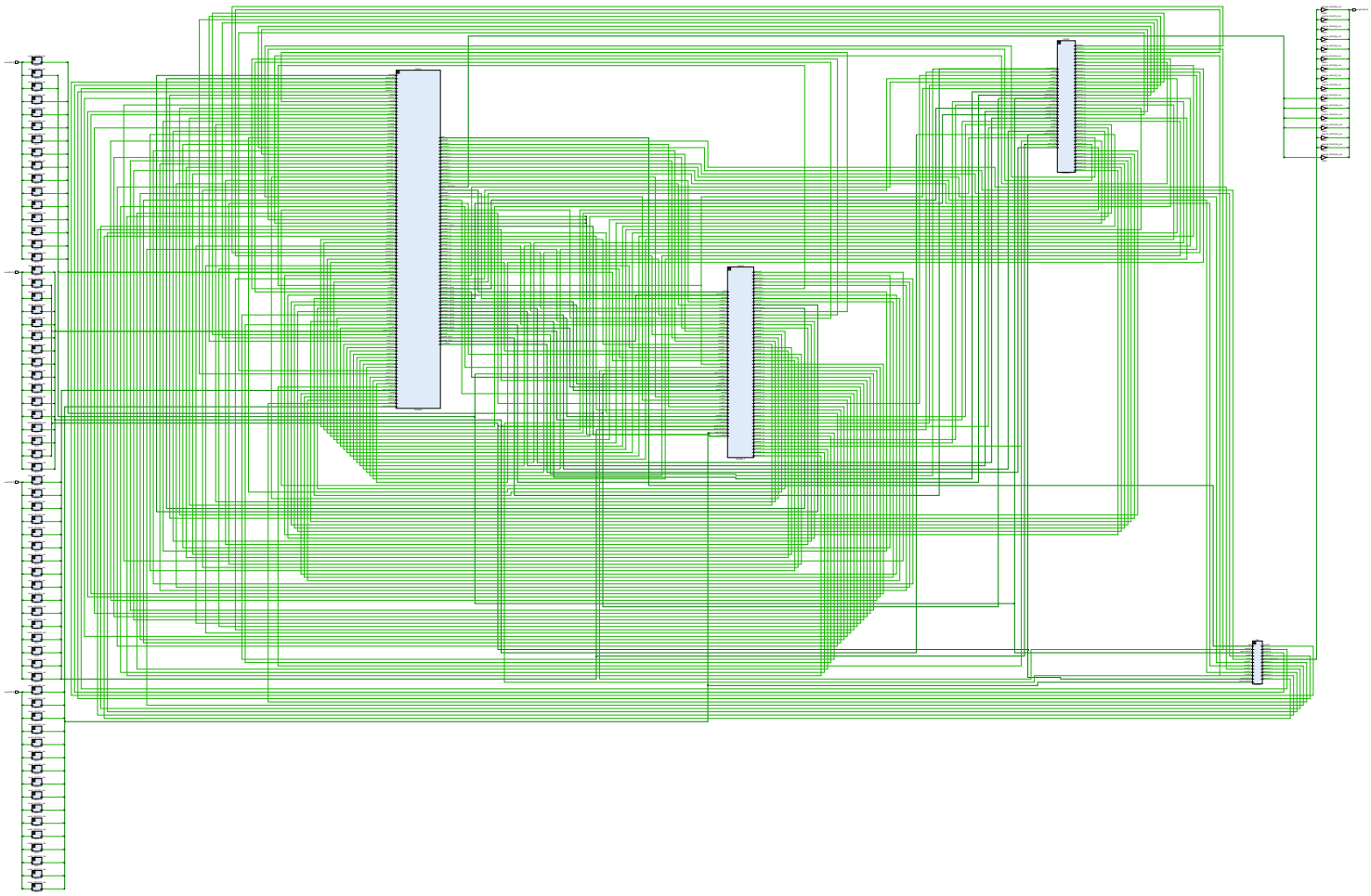
7) Synthesis Schematic and Comments:

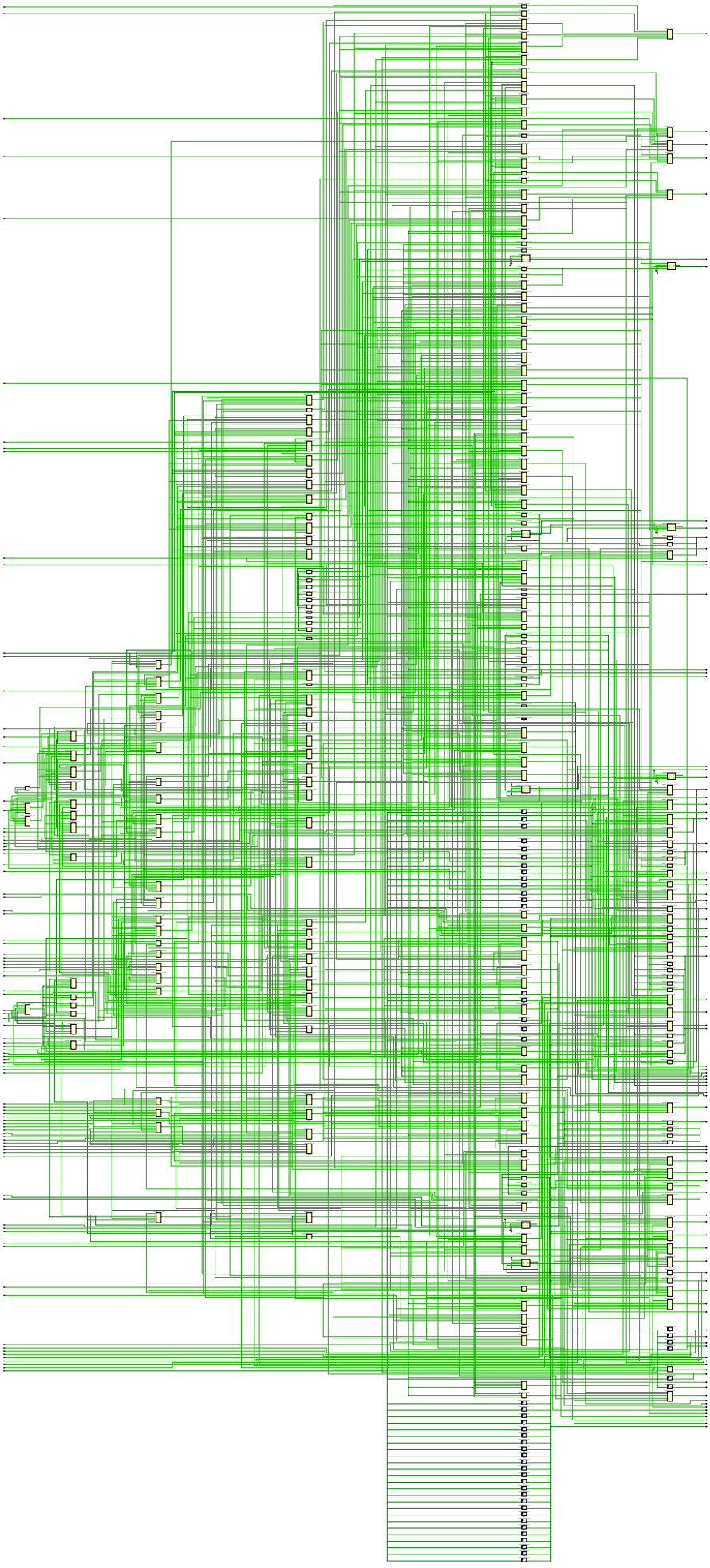
The Synthesis Schematic is attached on the next page.

The least number of LUTs is on the simple averaging unit, that has only 3 adders and a multiplier. The Number of LUTs significantly increases in the single layer because it's consistent of a lot of Avg Units stacked together in parallel. The number does not differ a lot for the multi layer, since it's just a single layer with sequential input, therefore with almost identical LUTs.

8) Answers to questions:

No questions were asked in AvgPool description





Part 5
Integration
Omar Tarek

1) Block Diagram

(note: due to the large size and several constraints the full design was not implemented)

The fully implemented part is Fully Connected layer, it includes FC1 + Tanh + FC2 + SoftMax. The code for the conv part is written but not tested

2) Verilog Codes

IntegrationFC.v

IntegrationConv.v

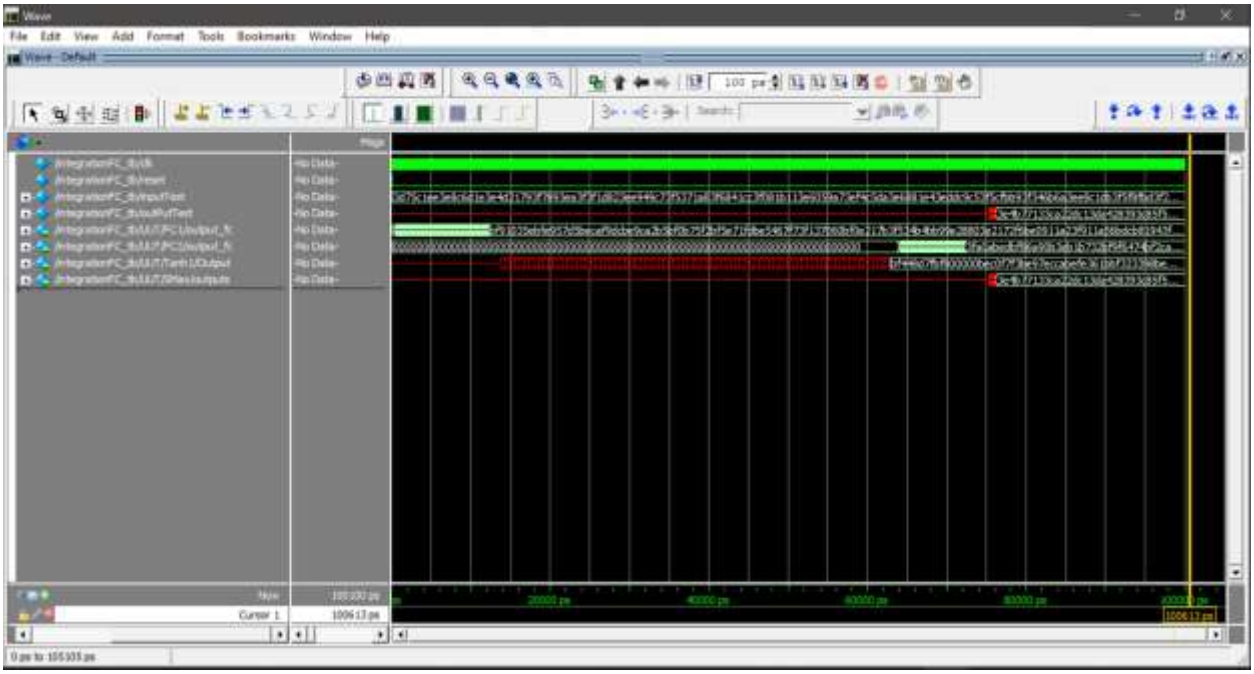
3) Verilog Tesbench

IntegrationFC_tb.v

4) Screenshots:

a. Simulation (Waveform/TCL console):

i. Fully Connected Part



We can see clearly the sequence of activation at the fully connected part with each layer outputting to the next. In the next page we talk about result verification.

Using files supplied in phase 1 (C++) code, I generated a random set of 120 numbers that were input to the module. I ran the module and compared the output with every stage output from the C++ files. They are attached here:

I also laid out the data in excel to make it easier to visualize.

Excel Proof of function:

(compared to output from simulations)

	A	B	C	D	E	F	G	H	I
1	Layer 1 IN		Layer 1 OUT (With Tanh)		Layer 2 Out		Softmax		
2	C++	Sim	C++	Sim	C++	Sim	Python	Python hex	Sim
3	0.06	3d75c1ee	-0.81202	bf91035e	1.47622	3fa0abed	11111001101000000000000000000000	3E6845FB	3e4b7713
4	0.27427	3e8c6d1e	-0.94914	bfe957d5	-1.3465	bf86a90b	11110001011110000000000000000000	3C5E4232	3ca22dc1
5	0.20032	3e4d2179	-0.3769	becaf9dc	0.232178	3eb1b732	11110110000110000000000000000000	3D8600E1	3da42839
6	0.97101	3f7893ea	-0.29673	be9ca2b5	-1.01419	bf9f6474	11110010011010000000000000000000	3C9A3636	3c85f52d
7	0.74654	3f3f1d82	-0.49659	bf0b75f2	-0.81699	bf2cafd0	11110010111011100000000000000000	3CBBBED2	3cec63d9
8	0.45564	3ee949c7	-0.70083	bf5e71fd	1.37648	3f943375	11111001010010000000000000000000	3E52488D	3e3899b7
9	0.82595	3f5371a8	-0.2045	be5467f7	-0.72196	bf045678	11110011001110000000000000000000	3CCE735A	3d0a5ea6
10	0.90725	3f6841cc	0.5197	3f137060	-0.74362	bf1b48ab	11110011001010000000000000000000	3CCA0781	3cfd03f5
11	0.53166	3f081b11	-0.50441	bf0e217b	1.78006	3fd87588	11111010011101000000000000000000	3E9D1FA5	3e9d11f8
12	0.21909	3e6059a7	0.67798	3f534b4b	0.554654	3f1c9246	11110110111000100000000000000000	3DB8FF3A	3dd5dc6f
13	0.47807	3ef4c5da	-0.0003	b99e2880					
14	0.22706	3e6881e4	0.15637	3e2172f6					
15	0.43318	3eddc9c5	-0.13306	be0911a2					

We notice that the numbers are a bit off but all within our range of accuracy. Off by a small fraction due to the short range and convergence condition of the tanh.

The data was copied from the Waveform and the results of the C++ code.

b. Post Synthesis Timing for the basic convUnit:

(as agreed upon post timing for larger layers of hardware are not achievable with our hardware capabilities. Our attempts reached up to **20 hours**, all while stalling the progress bar. It's not achievable on consumer hardware)

5) Clock Cycle Count:

a. Integration FC part:

FC1 -> 122

TanH-> 84*6+2

FC2-> 86

SoftMax-> 11

---

725 clock cycles

6) Synthesis Utilization Report

Integration FC part

Site Type	Used	Fixed	Available	Util%
CLB LUTs*	553993	0	230400	240.45
LUT as Logic	553993	0	230400	240.45
LUT as Memory	0	0	101760	0.00
CLB Registers	10800	0	460800	2.34
Register as Flip Flop	10800	0	460800	2.34
Register as Latch	0	0	460800	0.00
CARRY8	2173	0	28800	7.55
F7 Muxes	207037	0	115200	179.72
F8 Muxes	19927	0	57600	34.60
F9 Muxes	0	0	28800	0.00

7) Synthesis Schematic and Comments:

The synthesis Schematic is attached on the next page
The LUT's produced represent the huge amount of data processed in this file. Synthesis caused a lot of issues but the code is synthesizable.

8) Answers to extra questions:

No extra questions were asked in the description