COMPUTER ARCHITECTURE

ELL 782
Assignment 2
Report

Instructor Name: Prof. Smruti R. Sarangi

Submission Date: 17 Nov 2022

Submitted By:
Mohit Kumar (2022EET2095);
Mohit Kumar (2022EET2092);

CNN IMPLEMENTATION IN MyHDL

Programming language – Python

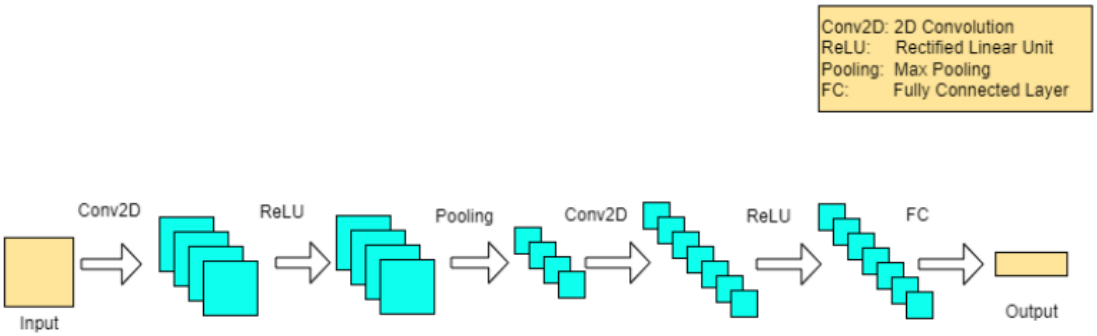
Library – MyHDL

CNNs are the most popular variants of DNNs in use today. They have vast applications in image recognition, image classification, object detection, facial detection etc.

We have four kinds of layers in a CNN:

- Convolutional layer
- Fully connected layer
- ReLU layer (Rectified Linear Activation Unit)
- Pooling layer.

The ReLU and pooling layers are nonlinear and are computationally light while the fully connected layer is computationally heavy.
Here is the design of CNN as provided in the assignment document.



Probably the most integral part of CNN are the adders and multipliers blocks.

We have assumed that all numbers are sufficiently small so that they can be represented in 8 bits 2's complement representation. The range thus of the input numbers becomes:

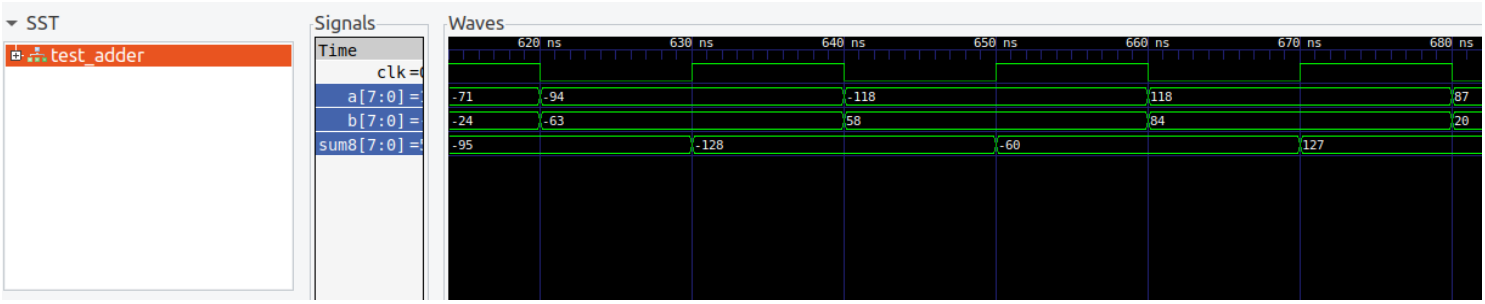
Minimum: -128; Maximum: 127

1. 8-bit Adder

To add 2 8-bit numbers, we have implemented a combinational adder block which accepts 2 8-bit numbers and calculates the sum of these 2 8-bit numbers.
This is done efficiently by using 2 4-bit Carry Look Ahead adder blocks with carry being propagated from lower bit CLA to upper bit CLA.

NOTE: The adder also supports saturating to the upper and lower limits if the sum crosses these bounds. For instance, if we add 127 + 127, the sum is 254, however that is out of bounds, thus the adder returns 127 I.e., the upper bond of the 8-bit adder.

GTKWave for 8-bit Adder with test input data (with saturating output when out of bounds).



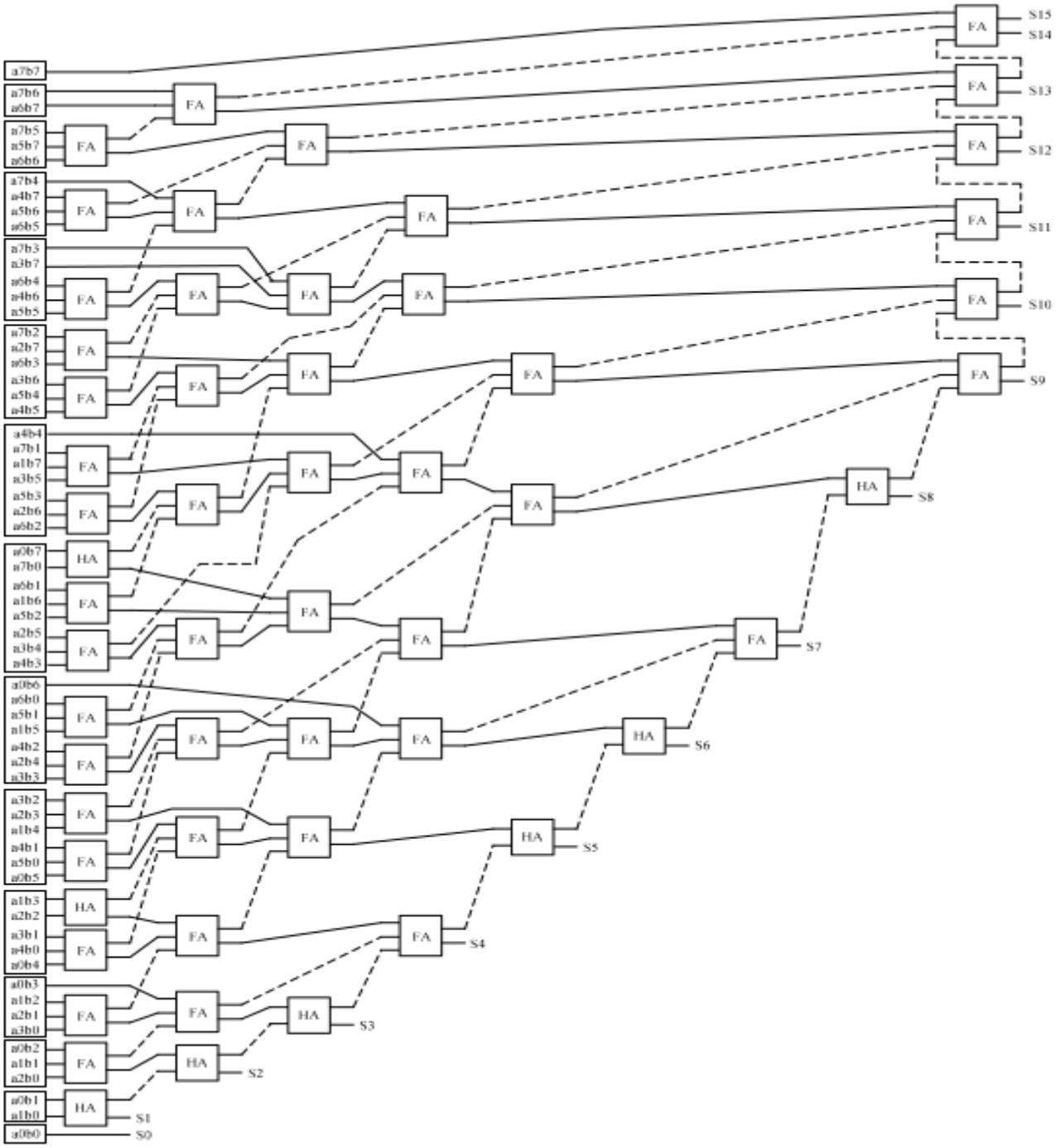
As can be seen from the figure above if we try to pass –94 and –63 to adder, the result is out of bounds, thus the adder returns –128, which is the lower bound for 8-bit signed number.

2. 8-bit Multiplier

We have implemented a multiplier unit that takes as input 2 8-bit numbers and generates a 16-bit result. However, keeping in mind the limitations of our architecture (8-bit) the 16-bit output is transformed into an 8-bit output, with the following conditions:

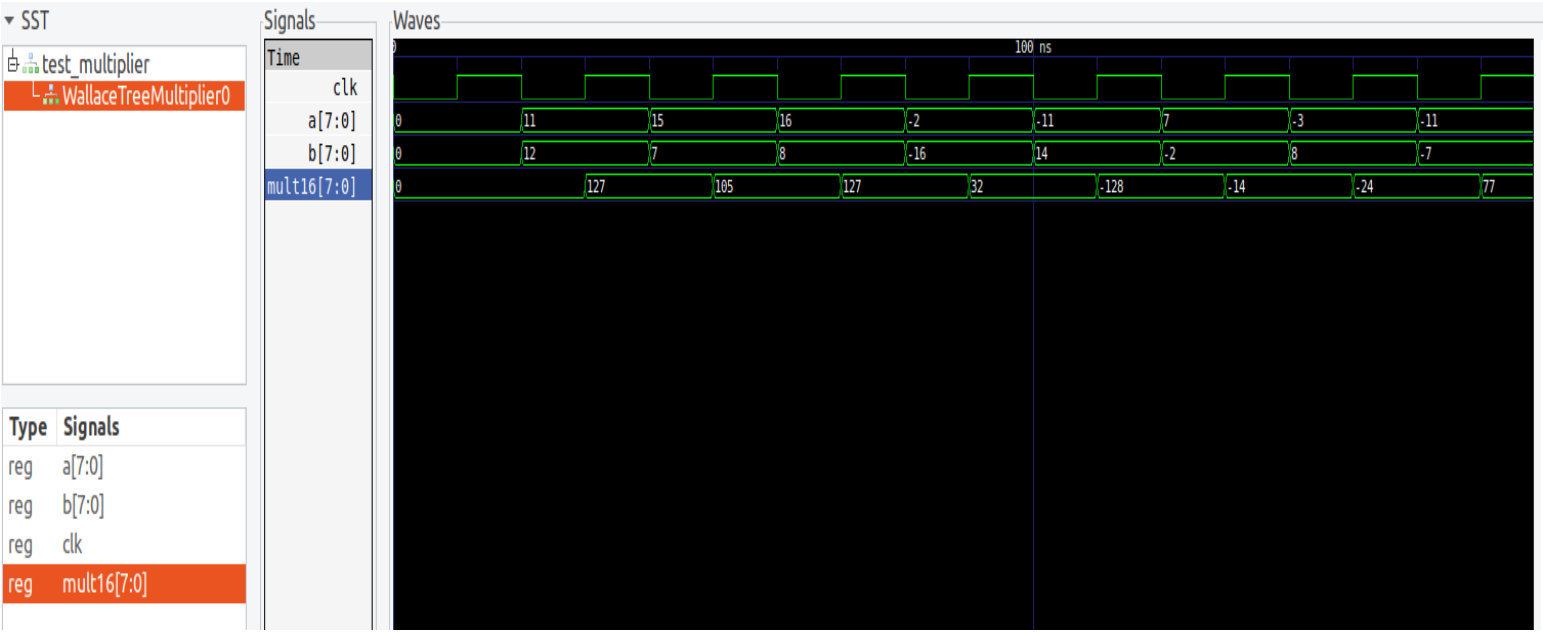
- If the 16-bit multiplied output is less than –128, the 8-bit output (to be returned) is set to –128.
- If the 16-bit multiplied output is more than 127, the 8-bit output (to be returned) is set to 127.
- For all other cases 16-bit output can be represented in 8 bits, thus 8-bit transformed output is returned.

Here is the block diagram of the Wallace Tree multiplier that is implemented.



This is the design followed for 8 bit multiplication, there are pre and post processing units implemented inside the Wallace Tree multiplier block which ensure that the output remains in bounds of 8 bit signed number.

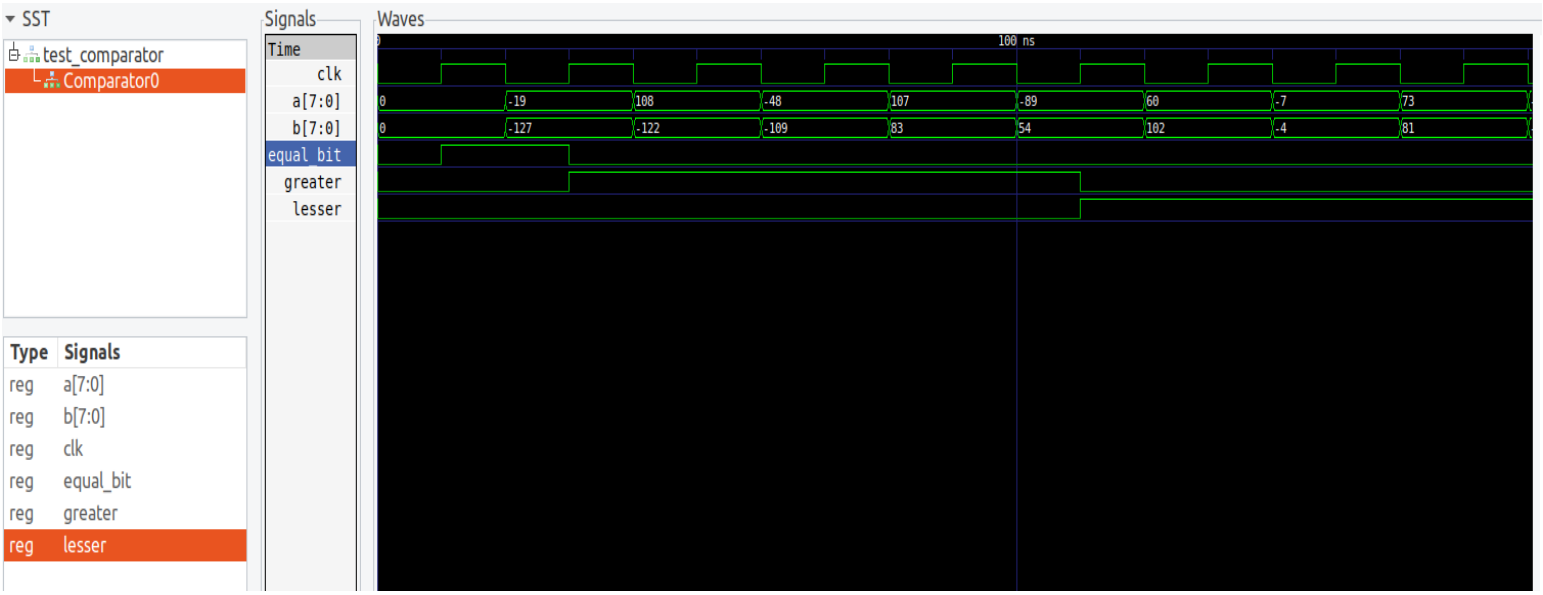
GTKWave for 8-bit Multiplier with test input data (with saturating output when out of bounds)



As can be seen in the vcd file above, when we multiply 11 and 12 the output is 132 which is more than the upper bound 127, thus the multiplier saturates to 127, similarly, -11 and 14 gives –154, thus the multiplier saturates to –128.

3. 8-bit Comparator

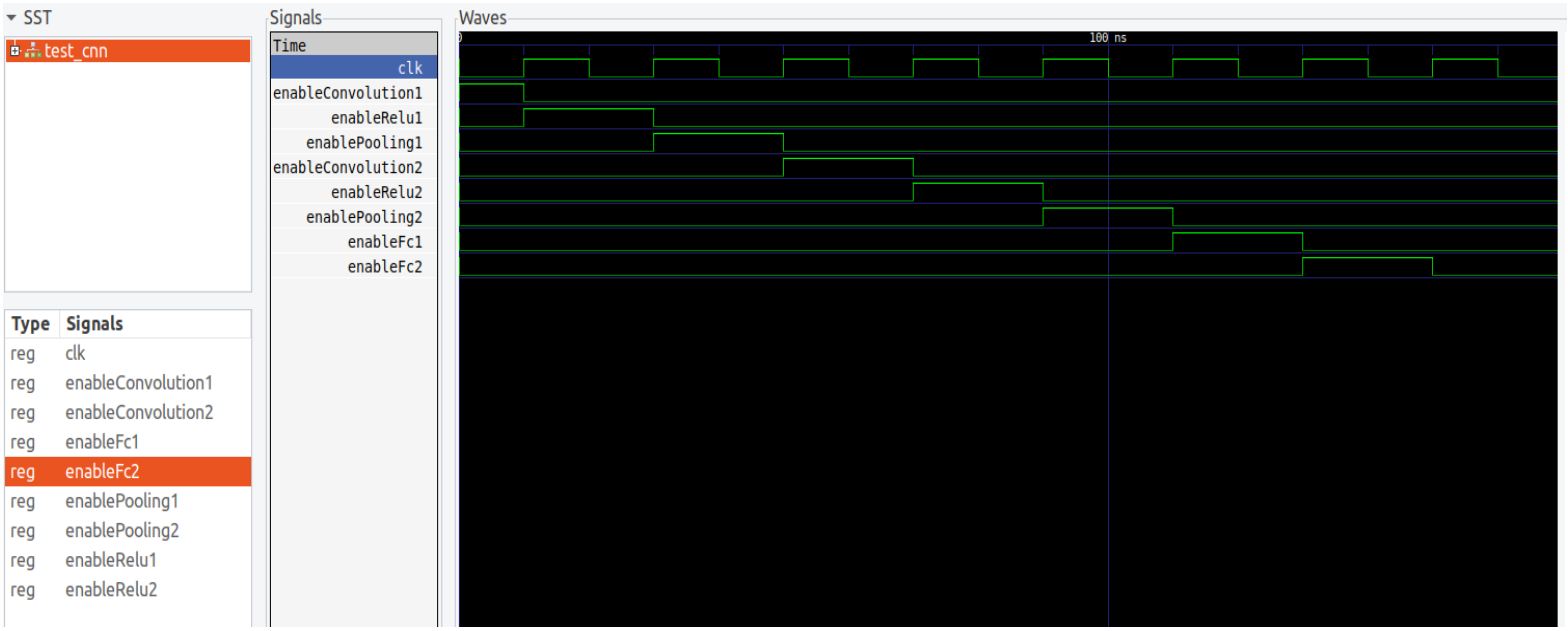
We have designed an 8-bit comparator, which accepts 2 8-bit numbers and 3 bits: equal, greater and lesser, these bits are self-explanatory.
We compare the first number with the second and the result of comparator is set in these 3 bits.
This is demonstrated in vcd file below:



CNN Model

The designed CNN model has various enable signals like enableConvolution1, enableReslu1, enablePooling1, enableConvolution2, enableRelu2, enablePooling2, enableFc1, enablefc2.

The various enable signals ensure the correct flow of the CNN network and that is visible from the vcd file shown above.



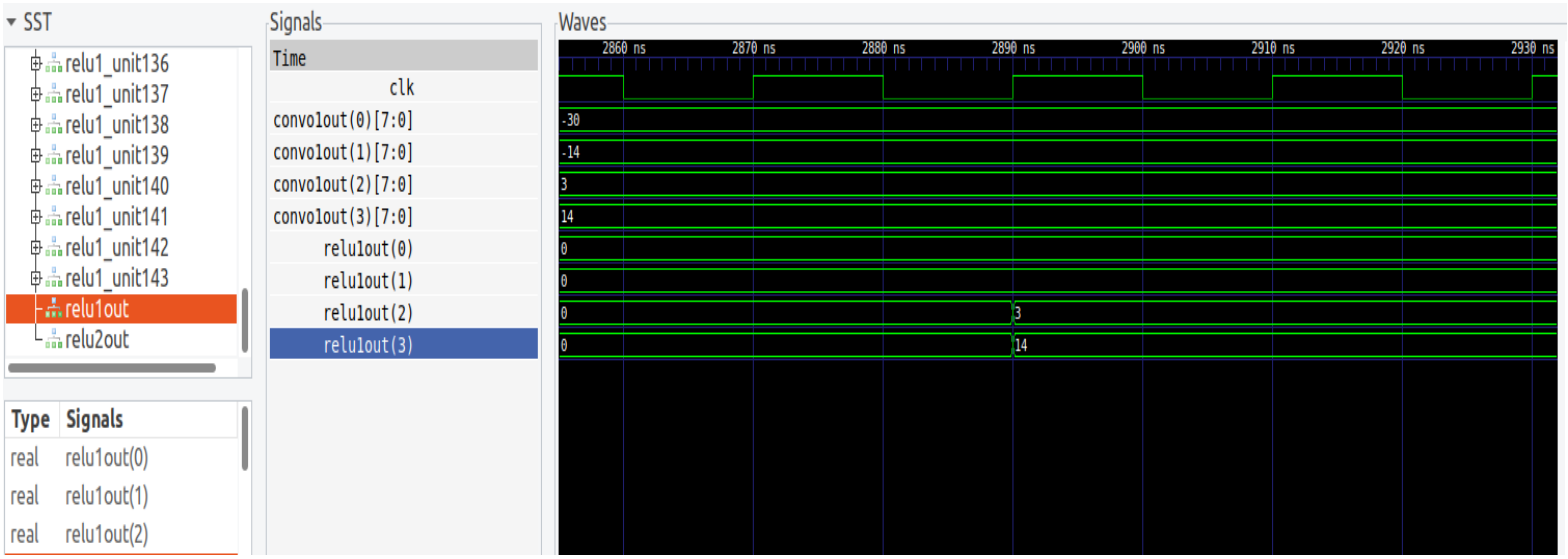
Now, we have attached the output of each layer starting from convolution layer 1.

Kernel – [[1,2,1];[1,1,1];[1,2,3]]

Here is the vcd file demonstrating the output of the same.

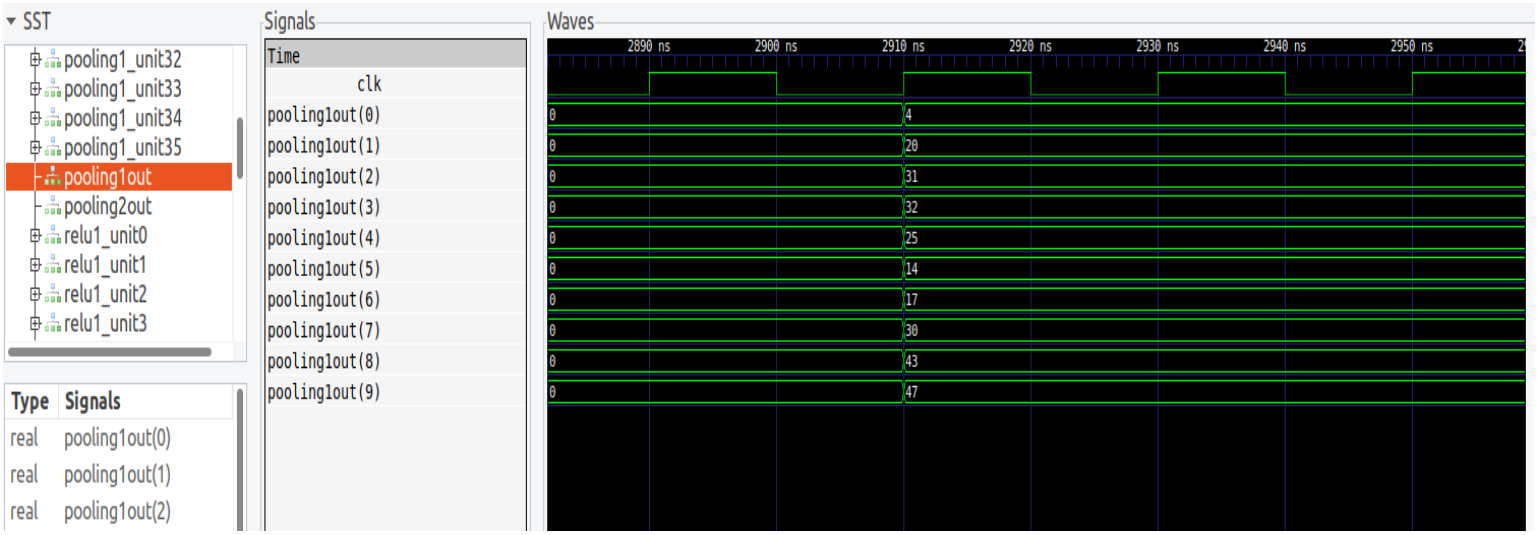
Convolution 1 has 144 neurons thus the convolution layer 1 generates 144 outputs.

The Relu Layer 1 converts all negative convolution outputs to 0 and leaves positive outputs as is.



This is evident from the outputs of Relu unit as compared to convolution outputs negatives are converted to 0 and positives are left as is.

The pooling layer 1 also provides the maximum value of the entire 9 values.

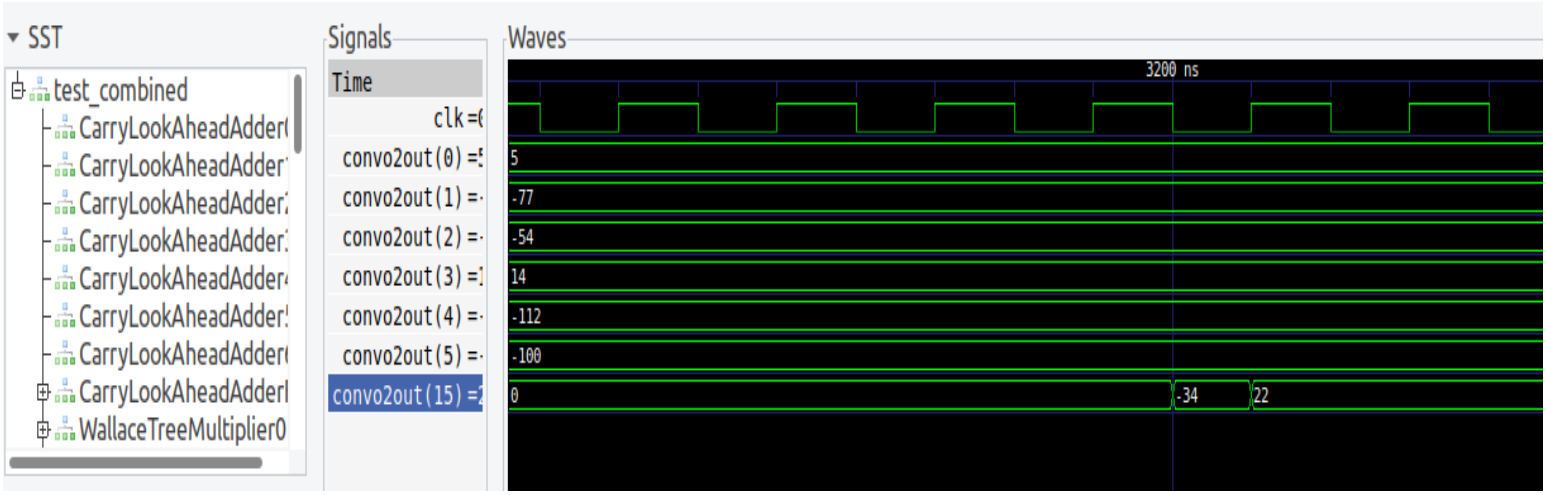


Convolution Layer 2

The convolution layer takes the output from pooling layer 1 and uses thsi matrix to do convolution.

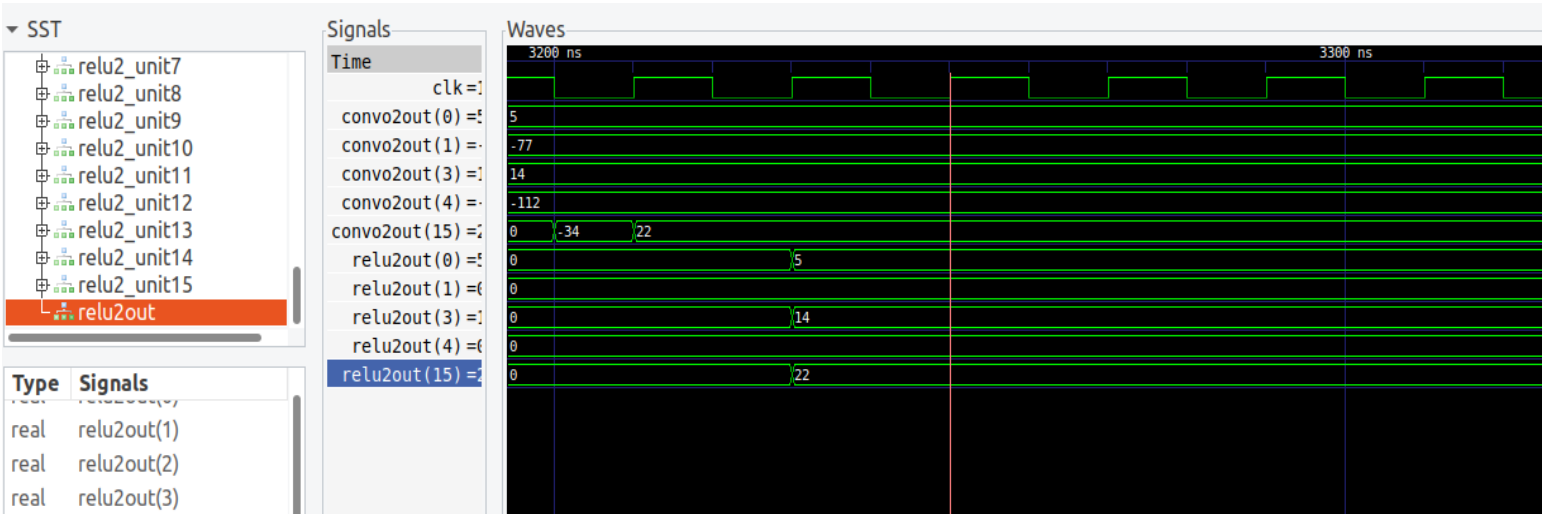
Kernel – [[1,1,-2];[0,2,-3];[2,-3,0]]

Here is the vcd that demonstates this:



Relu Layer 2

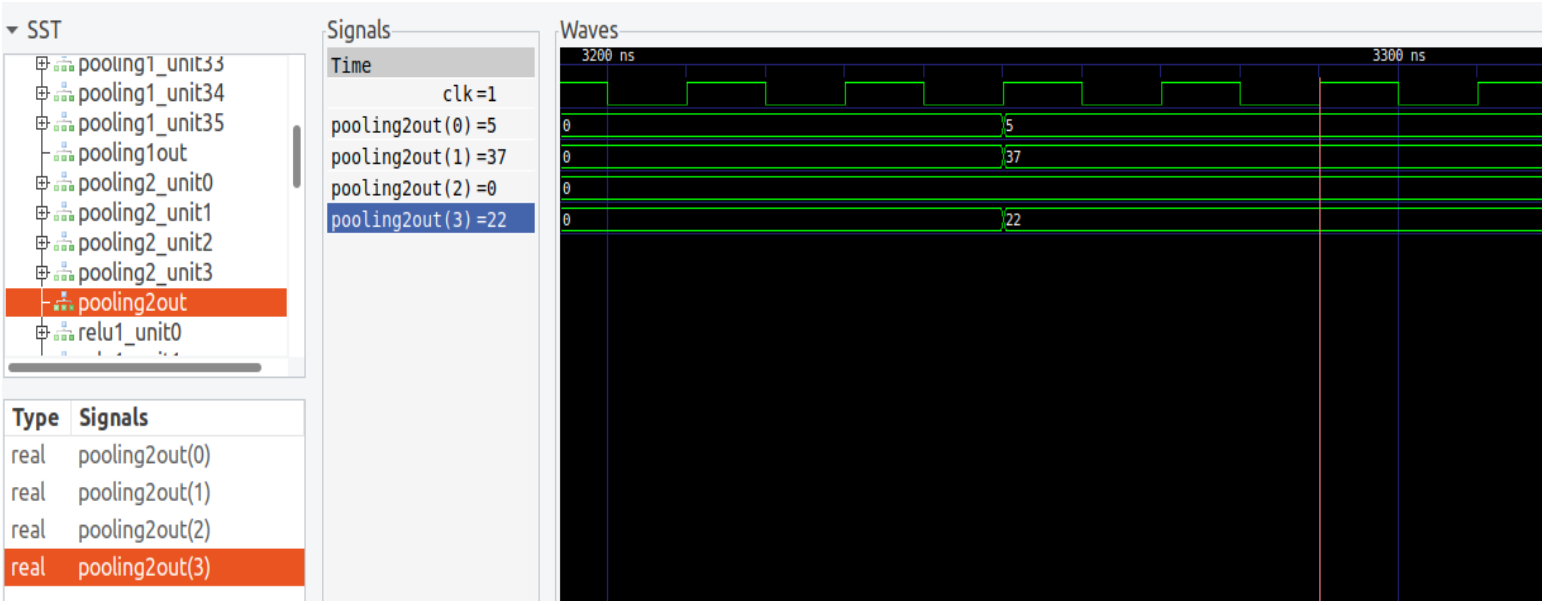
The following is the output of the Relu 2 layer.



Relu layer makes the negative values to 0 and positive values remain as is. It is evident from the vcd file as well.

Pooling Lyer 2

The following is the output of Pooling layer 2.



The pooling here is max pooling and only takes the maximum value out of the 4 values.

Fully Connected Layers 1 and 2

FC1 Weights : [[-2,1,1,1];[1,-2,1,1];[1,1,-2,1];[1,1,1,-2]]

FC2 Weights : [[3,1,1,1];[1,2,1,1];[1,1,3,1];[1,1,1,3]]

Each of the 2 fully connected layers have 4 neurons each and each neuron of layers is connected to each neuron of layer 1, simply put each neuron of layer 1 has 4 weights.

Here is the vcd for Fully Connected Layer.