ECE 1718 Assignment #2

Group #1
March 19, 2023

Members:
Teng Shu shuteng 1003533651
Yih Cheng cheng442 1009206878
Yang Chen chenya26 999551783
Chen Liang liangc78 1006621443

Milestone #1

Task 1: Isolate the convolution function

A convolution function has been implemented with the ability to do convolution between two-row vectors or to do convolution of a one-row vector with a column vector. The 'img' vector in the convolution layer needs to be preprocessed since the elements used for convolution in these two arrays were not aligned. It is expected that this preprocessing of data may slow down the calculation.

Task 2: Prepare your code for importing into Simulink

There were compile errors encountered during code importing, and it is caused by Simulink having trouble compiling a gcc file, and there is no feasible fix found after researching.

As the instruction mentioned that “If preferred, the whole design and training process of the neural network can be done in MATLAB” and “Porting your trained neural network from MATLAB to Simulink”, it was decided to implement the whole training and testing process of the network on MATLAB, then implement the forward_pass on Simulink which can be used for prediction.

Optimization of the network by pruning was also done in this part. One of the dense layers has been completely removed, to make the number of weights needed for dense_w from 980*120 to 980*10. Forward_pass, backward_pass and update_weights were all updated accordingly to make sure the network is operating. In the end, the pruned network can reach an accuracy of 88.83% with a much smaller number of weights required.

Task 3: Assemble and simulate your CNN block diagram

The inference portion of the network was implemented in Simulink, and simulated using the test set. The expected class and predicted class were both displayed and manually compared. Most of the predictions were matching with the expectation, which illustrates that the network is properly functioning.

The time necessary to run the code - MATLAB

Tic toc function in MATLAB was used to measure functions that were expected to be slowest. The results were shown in the table below.

Function name	Time of execution [s]
read_test_data	2.124306
read_train_data	7.954934
initialise_weights	0.000317

forward_pass	11494.703387
backward_pass	4172.023893
update_weights	3.480888

Since the whole process was done in MATLAB which is not as fast as Cpp, it is expected that it will take way more time than the Cpp version. The final accuracy is 89%, which proves that the MATLAB network is working as expected.

The table shows that the forward_pass function is consuming most of the simulation time.

The time necessary to run the code - Simulink

Simulink profiler was used to analyze the execution time of the network. The results were shown in the table below.

Function name	Time of execution [s]
give_img	0.070
read_test_data	0.510
forward_pass_hw	3.350
convolution_layer	3.348
For Each Subsystem	3.346
convolution_layer_pre	3.212
sigmoid	0.013
filter_dim	0.006
conv	0.006
j	0.004
i	0.002
out	0.002
For Each	0.000
convolution_layer_post	0.001

max_pooling_layer	0.001
dense_layer1	0.000
softmax	0.000

It can be observed that the function ‘convolution_layer_pre’ consumes most of the simulation time. This is the function that prepares the data from ‘img’ and ‘conv_w’ and creates two vectors that are suitable for convolution. This observation matches the expectation in task 1 which expects the preprocessing of data may slow down the computation.

Milestone #2

Task 1: Choose and justify the selection of the number format

Numbers should be signed since weights can be either positive or negative.
Based on the experience from assignment 1, the precision of 2e-8 can be sufficiently large for a reasonable prediction.
There may be some benefits to using different number formats across different portions of CNN, but it takes lots of effort to investigate the most suitable format for each portion of CNN, so it only makes sense to do this for large networks.

Number format - double-favoring accuracy

This is the double-precision floating point format. It can represent a number with higher precision, but the calculation process would be more complex than fixed point format which makes the calculation of it possibly slower. It is double precision, so it uses twice as much as bits needed for the other two formats.
The double format can represent numbers within the range of -1.7e308 to + 1.7e308, with a precision of 5e-324.

Number format - fixed point number(Q24.8 format)

This is a fixed point format, so it provides less precision using the same amount of bits compared to the floating point format, and the range of the format is also smaller. However, the instruction stated that “storage/transfer of integer operands and compute of integer operations fit well with implementations where efficiency is desired”, meaning computation of fixed point numbers would be faster.
This chosen fixed point number format can represent numbers within the range of -8388608 to 8388607, with the precision of 2e-8.

Number format - integer - favoring efficiency

This is integer format, which would have the same pros as the fixed point number format, and would be even easier for type casting. It can represent a wider range of numbers compared to the fixed point number. However, it is the least precise number format among these three formats.
It can represent numbers within the range of -2147483648 to 2147483647, with a precision of 1.

Performance analysis

The performance analysis was done in C++ due to the MATLAB training being too time-consuming. The function being altered is convolution as suggested by the instruction. All values are passed in as double, and then will convert the data into the target format, finish all the calculations of convolution, then the result will be converted back to double and returned.

Due to the data conversion process, the computation time cannot accurately represent the actual time spent on computation, especially for the fixed point format since the conversion to/from fixed point needs some extra calculation that can be easily done in hardware by shifting bits but has to be performed as complex arithmetic calculation due to shifting on double format is not supported in C/C++.

Data type	Computation time [min:s:ms]	Classification accuracy
float(single)	46.02	0.846667
double	43.48	0.908333
fixed point (Q24.8 format)	1:12.33	0.83
int	0:39.71	0.0883333

The table above shows the computation time and classification accuracy for each of the data types. Due to the data conversion time, float and fixed point formatting is spending more time than double format. Int format is taking the least amount of time even converting time is included.

The order of classification accuracy is as expected. Double is the most precise format, resulting in the highest classification accuracy, followed by the float. Fixed point format is not as precise as double or float, so has lower classification accuracy than float. Int has the worst precision, which does not meet the minimum required precision and results in unacceptable classification accuracy.

Task 2: Implement the bfloat16 data number format

Bfloat16 in MATLAB

A function that converts data to bfloat16 type was defined in MATLAB. This function uses “CustomFloat(x, 16, 7)” to convert data into the desired format, which has 16 bits in total and 7 bits of mantissa.

Due to the command “CustomFloat” returning an object which is a way more complicated type than just a numeric value, the execution time was also significantly longer than using double as data types.

The bfloat16 type was tested on the MATLAB CNN code and only converted the data type while doing convolution. Due to the slow calculation of the CustomFloat type, it was unrealistic to test it on the whole training and testing process, so it was decided to only test it on the inference of 20 images just to prove that this data type is precise enough for the network.

The execution result stated that the accuracy was 0.9 for these 20 images, which shows that the accuracy would be very acceptable, and maybe better compared to the floating-point one and the fixed-point one chosen in the previous milestone. However, these inferences take about 20 hours which means using the MATLAB function for bfloat16 conversion would be unrealistic to be used in practice.

Bfloat16 in Simulink

The exponential field of bfloat16 also has a 127 offset, which is the same as the offset of single-precision floating point, so there is no offset change that needs to be calculated from single to bfloat16. It was realized that bfloat16 and single are having the same amount of bits for sign and exponent, so ‘bit slice’ blocks were used to convert single to bfloat16 by just slicing the unnecessary bits.

Inference in Simulink was tested on bfloat16, and the output matches the expected classification, which means the conversion works properly. The computation time was way better than simulating bfloat16 in MATLAB, and the accuracy was still reasonable.

The impact of the implementation method on the computation times

The impact of implementation methods on the computation times could be huge, like the bfloat16 simulation in MATLAB. “CustomFloat” type in MATLAB is represented by objects, which is very complex and inefficient considering it is only representing one value.

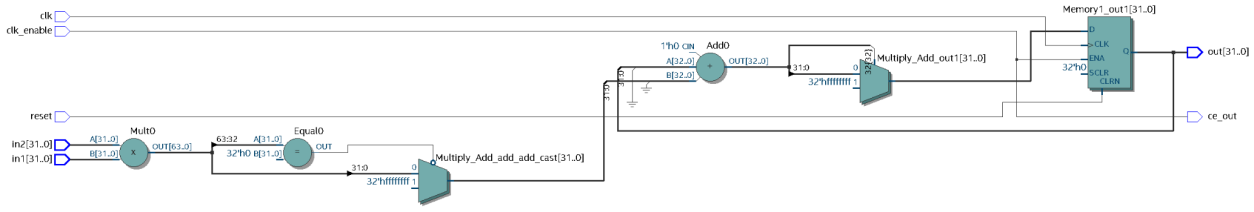
The fastest way of implementation is likely bit-manipulation. This is in the low level so would be very fast, but also can be extremely complex to achieve non-trivial goals.

Milestone #3

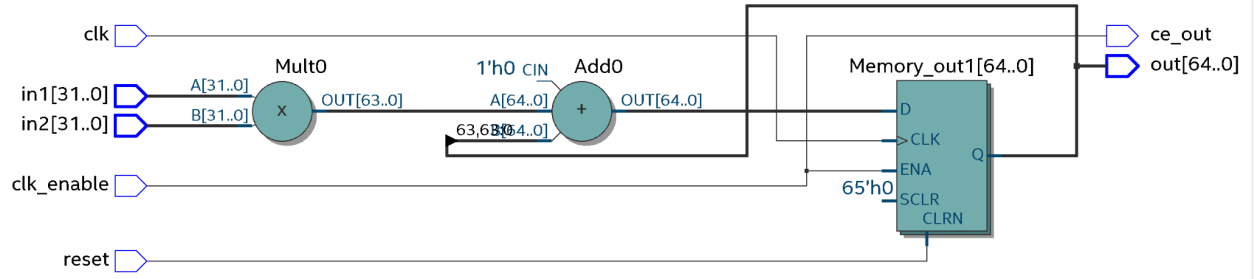
Task 1: Generate the HDL Code

There are 4 different number formats types of MAC unit RTL used for this comparison: bfloat16, integer, fixed point, and standard floating point.

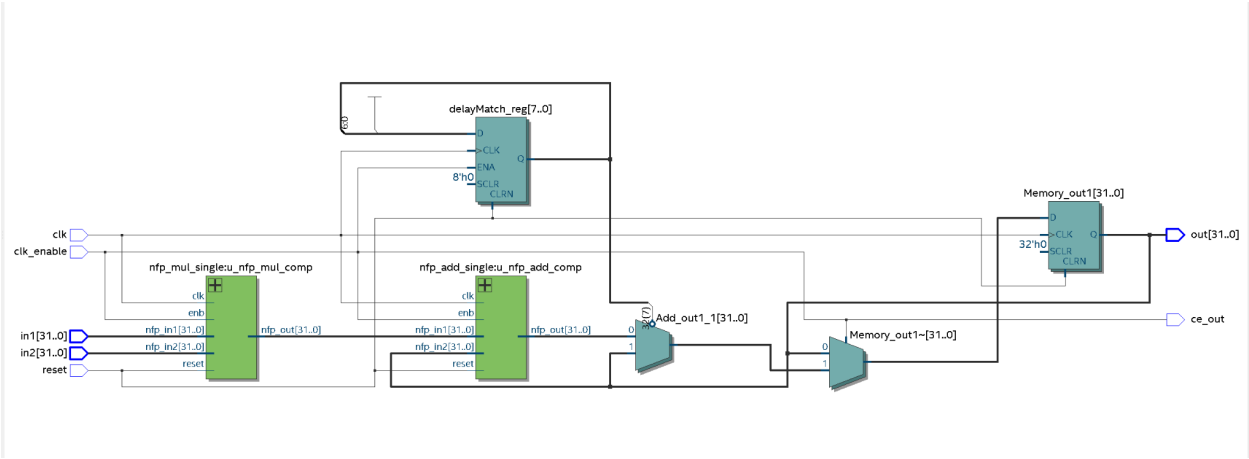
Sample int MAC RTL block view:



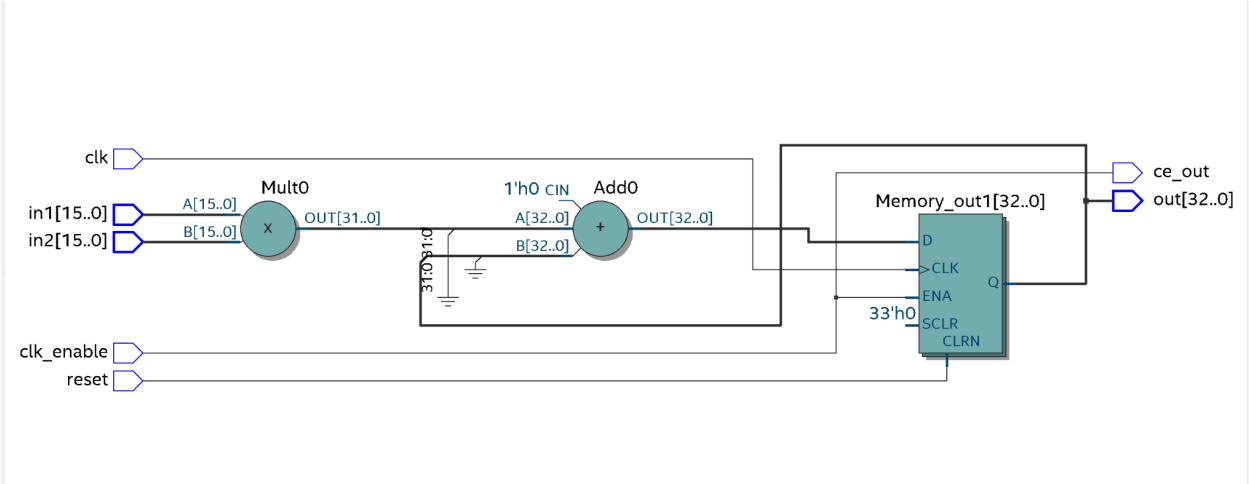
Sample fixed point MAC RTL block view:



Sample float MAC RTL block view:



Sample bfloat16 MAC RTL block view:



Task 2: Compare the MAC unit RTL's for various number formats

Sample int MAC synthesis result:

ALMs used	Registers used
47	32

Integers use the simplest arithmetic units, so it is the simplest MAC and uses the least amount of resources which is expected. However, the precision of the integer is far from what is needed so this MAC cannot be used for CNN.

Sample fixed point MAC synthesis result:

ALMs used	Registers used

56	65

The big difference between fixed point MAC and integer MAC is that the output data type for integer MAC remains as an integer, while for fixed point MAC it doubles the number of bits used for the decimal part. If want to keep the data type consistent, data conversion would be needed. For fixed point type, this will take some extra calculation, while for integer type, only bit slicing will be needed.
Another small difference is that in integer MAC, the previous accumulator value is sliced and represented as a 32-bit integer, while in the fixed point (Q24.8) MAC, the previous accumulator value is represented as Q48.16 fixed point type. This is likely just a design decision by Simulink.

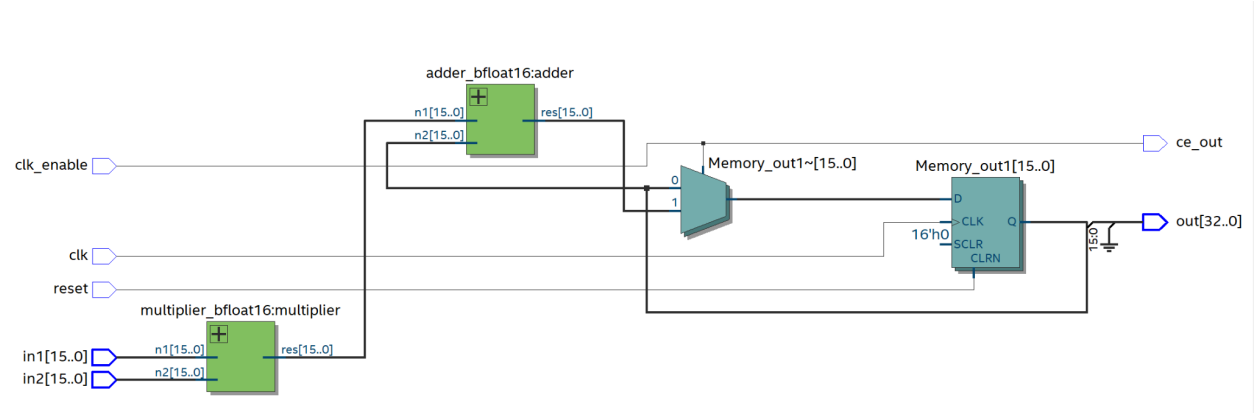
Sample float MAC synthesis result:

ALMs used	Registers used
463	1131

Float MAC takes about 10 times more resources than integer/fixed point MAC. Given that the fixed point Q24.8 can provide reasonable precisions, float type would be unrecommended due to the resources needed and the complexity due to the floating point arithmetic.

Sample bfloat16 MAC synthesis result:

ALMs used	Registers used
166	16



Milestone #4

Both full_adder_1bit and half_adder_1bit were implemented so they can be used in more complicated adders and multipliers. Both full_adder_1bit and half_adder_1bit were made by a few logic gates.

Task 1: Implement and compare the performance of different adder structures

Implementation - carry ripple adder:

The carry ripple adder was implemented by combining full_adder_1bit and half_adder_1bit. The LSB from operands will be sent to the half_adder, and the rest of the operands' bits will be sent to the full_adder. Carries from the previous 1bit adder will be passed to the next 1bit adder.

Performance analysis - carry ripple adder:

The detailed RTL viewer diagram is attached at the end of this pdf.

ALMs used	40

The carry ripple adder was imported into Simulink to perform one addition in parallel with a Simulink adder, and profiled by the Simulink profiler to analyze the performance.
The carry ripple adder takes about 0.037s to complete one addition, while the Simulink adder takes negligible time (0s reported by the profiler). This is likely because the Simulink is only simulating the behavior of the carry ripple adder, and it can only evaluate values for each sub-paths sequentially. This means Simulink simulation performance cannot show any benefit from concurrency and will be only affected by the complexity of the module.

Implementation - carry skip adder:

There was a carry_skip_adder_4bit implemented in a similar fashion as carry_ripple_adder, with only some extra logic to detect if the carry calculation can be skipped or not.
The carry_skip_adder (32bit version) was implemented by combining four carry_skip_adder_4bit together with carries passed properly.

Performance analysis - carry skip adder:

The detailed RTL viewer diagram is attached at the end of this pdf.

ALMs used	53

The carry skip adder was also imported into Simulink to perform one addition in parallel with a Simulink adder, and profiled by the Simulink profiler to analyze the performance.

The carry ripple adder takes about 0.057s to complete one addition, while the Simulink adder takes negligible time (0s reported by the profiler). It takes more time to simulate than to carry ripple adder. As explained above, this is likely due to the complexity of the carry skip adder being slightly greater than the carry ripple adder. And due to the lack of concurrency, the benefit of the carry skip adder, which is a shorter critical path, is completely inhibited by Simulink.

Task 2: Implement and analyze the performance of the Wallace Tree multiplier

Implementation:

Wallace tree multiplier does multiplication by compressing partial products. full_adder_1bit and half_adder_1bit were used as 3-to-1 compressor and 2-to-1 compressor. Compression was divided into different stages. Sums and carries from the current stage adders will be considered only in the next stage to maximize the concurrency (by only doing additions of signals having similar propagation delay at the same time).
After each compression stage, the carry bit will be passed into the next digit. The compression will keep going until every bit position only has one or two bits that need to be added. In the end, a proper adder will be selected for each bit position to do the last one or two bits addition to get the product.

There were lots of signals that needed to be tracked and connected to different adders, so it would be very difficult and inefficient to manually write the code for the Wallace tree multiplier. It was decided to write a Python script to auto-generate the Verilog code for the Wallace tree multiplier with any arbitrary input width.

The main benefit of the Wallace tree multiplier is its short critical path.
Assuming the simplest 32-bit multiplier uses regular adders to add all the partial products in a row, then each bit position will have some dependency from the previous bit position since there may be a carry from the previous bit position. Also due to adding multiple bits for each bit position, each bit addition will have some dependency from the previous bit addition since there may be a carry from the previous calculation. In total, the worst-case propagation delay can be approximately 32*32 of the propagation delay of a single adder.
For a 32-bit Wallace tree multiplier, there will be 7 stages needed for this calculation (reported by the Python code), so the propagation delay for compression is just 7 of the propagation delay of a single adder. To calculate the product, a 32-bit adder will be used in the end, so the worst case is 32 of the propagation delay of a single adder. In total, the worst-case propagation delay would be just 7+32 of the propagation delay of a single adder.

The downside would be the complexity of the module. There will be lots of extra adders being used while the compression, which means more difficult to debug as there will be more intermediate signals, and more resources needed.

Performance analysis:

The detailed RTL viewer diagram is attached at the end of this pdf.

ALMs used	1763

The Wallace tree multiplier was imported into Simulink to perform one multiplication in parallel with a Simulink product block, and profiled by the Simulink profiler to analyze the performance. The Wallace tree multiplier takes about 0.953s for a single calculation, while the Simulink product block takes about 0.001s. This is likely due to the same reason explained above. The complexity of the 32-bit Wallace tree multiplier is very high and contains lots of blocks so it will take Simulink lots of time to simulate it. And the lack of concurrency completely inhibits the short critical path benefit from the Wallace tree multiplier.

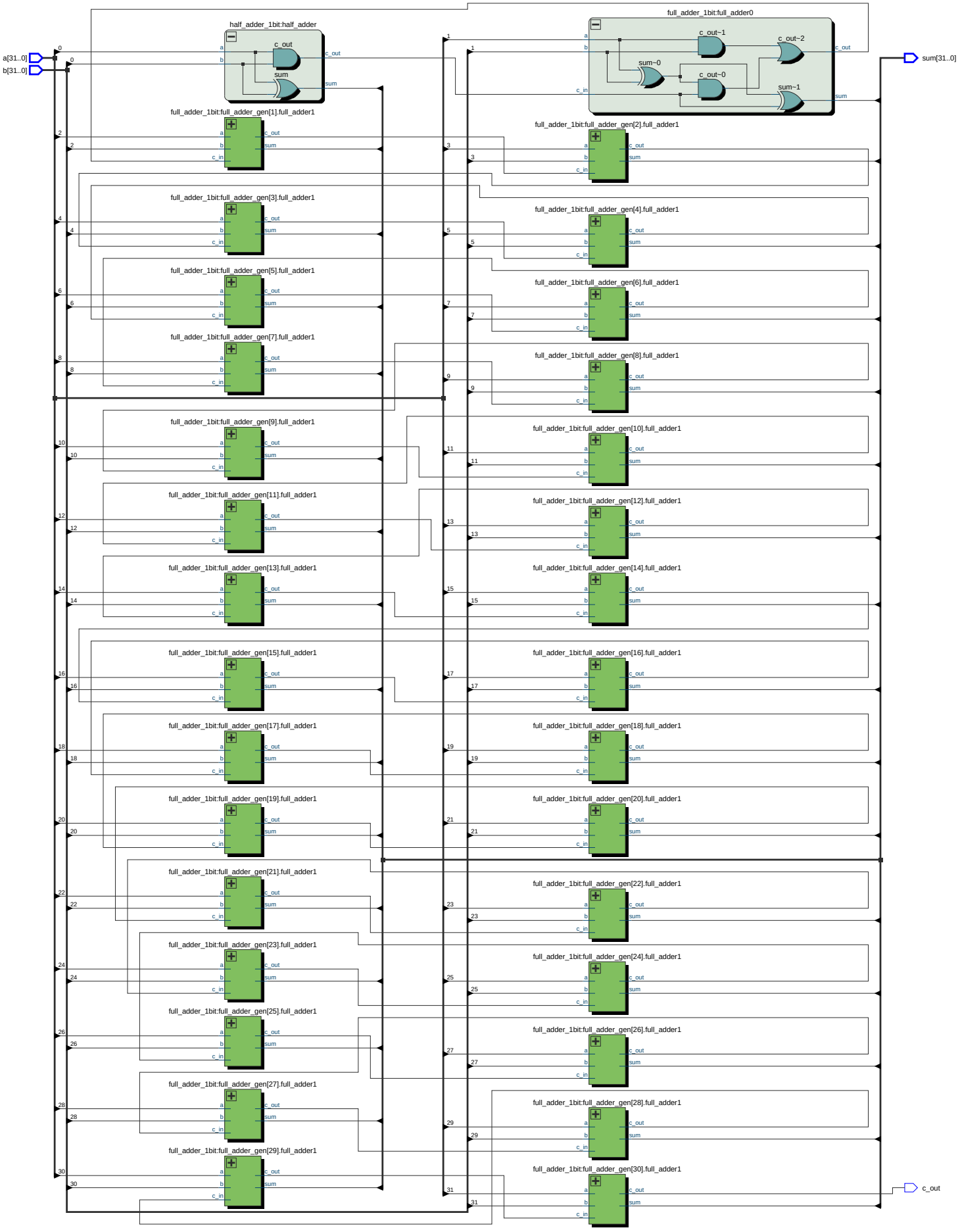
Task 3: Verify correctness

Due to the extremely slow Simulate simulation on the Wallace tree multiplier, it may be unrealistic to replace multipliers and adders in the actual CNN network or the simulation will take forever. It was decided to replace multipliers and adders in a single MAC unit, send random fixed point values to this MAC unit and have another MAC use Simulink's multipliers and adders in parallel so the output value can be compared.

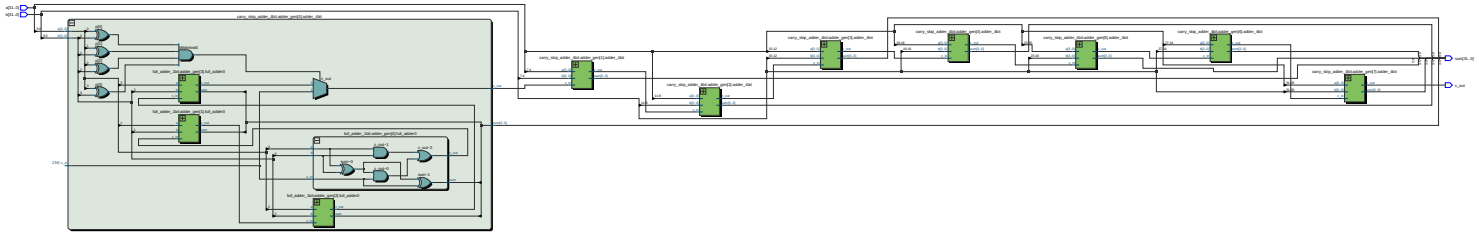
Imported HDL blocks were hard-coded by Simulink to only take integers as input and output type, so some Matlab function of data conversion was implemented to keep the precision of fixed point values.

The results from these two MACs were identical, which means the implementation of adders and multipliers in Verilog was able to generate correct results.

carry_ripple_adder_RTL_viewer


a[31..0]
b[31..0]
sum[31..0]
c_out
half_adder_1bit:half_adder
c_out
sum
full_adder_1bit:full_adder0
c_out~1
c_out~2
sum~0
c_out~0
sum~1
full_adder_1bit:full_adder_gen[1].full_adder1
full_adder_1bit:full_adder_gen[2].full_adder1
full_adder_1bit:full_adder_gen[3].full_adder1
full_adder_1bit:full_adder_gen[4].full_adder1
full_adder_1bit:full_adder_gen[5].full_adder1
full_adder_1bit:full_adder_gen[6].full_adder1
full_adder_1bit:full_adder_gen[7].full_adder1
full_adder_1bit:full_adder_gen[8].full_adder1
full_adder_1bit:full_adder_gen[9].full_adder1
full_adder_1bit:full_adder_gen[10].full_adder1
full_adder_1bit:full_adder_gen[11].full_adder1
full_adder_1bit:full_adder_gen[12].full_adder1
full_adder_1bit:full_adder_gen[13].full_adder1
full_adder_1bit:full_adder_gen[14].full_adder1
full_adder_1bit:full_adder_gen[15].full_adder1
full_adder_1bit:full_adder_gen[16].full_adder1
full_adder_1bit:full_adder_gen[17].full_adder1
full_adder_1bit:full_adder_gen[18].full_adder1
full_adder_1bit:full_adder_gen[19].full_adder1
full_adder_1bit:full_adder_gen[20].full_adder1
full_adder_1bit:full_adder_gen[21].full_adder1
full_adder_1bit:full_adder_gen[22].full_adder1
full_adder_1bit:full_adder_gen[23].full_adder1
full_adder_1bit:full_adder_gen[24].full_adder1
full_adder_1bit:full_adder_gen[25].full_adder1
full_adder_1bit:full_adder_gen[26].full_adder1
full_adder_1bit:full_adder_gen[27].full_adder1
full_adder_1bit:full_adder_gen[28].full_adder1
full_adder_1bit:full_adder_gen[29].full_adder1
full_adder_1bit:full_adder_gen[30].full_adder1
c_out

carry_skip_adder_RTL_viewer



wallace_tree_multiplier_RTL_viewer