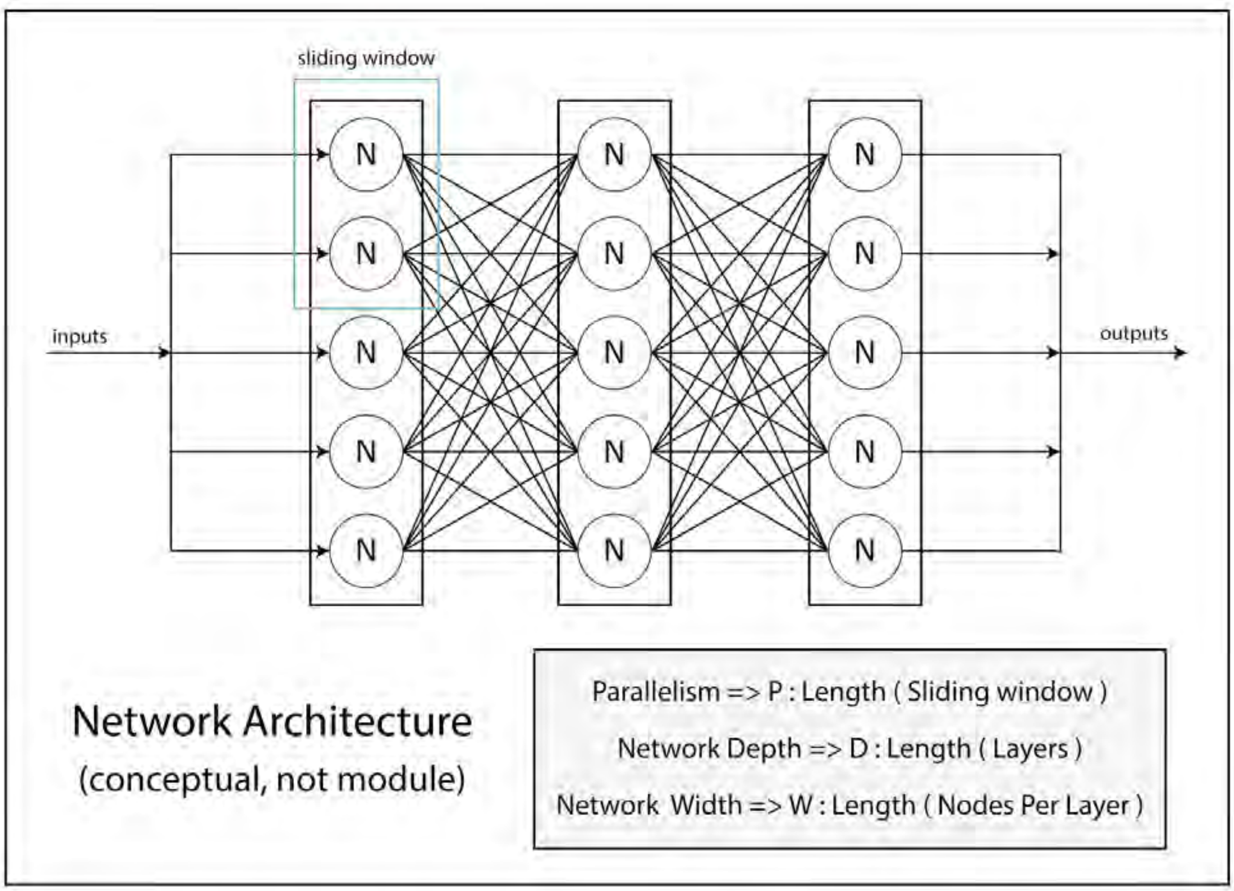
Sebastian Bartlett, Josh Noel
Professors Hom, Steinmeyer
MIT 6.111, Fall 2018
FPGA Neural Network



1. Abstract

This project implemented a neural network calculation system on an FPGA using FSM structures. For complex networks this is a difficult computational task in itself with a significant amount of ongoing research. However, interesting usage applications can be solved on smaller network topologies. We achieved our core goal by interfacing this structure with a computer via USB-UART, where it was used to compute the propagation values for an artificial neural network.

2. Symbol Key

T: Maximum number of multiplication operations per layer calculation step
M: Bit width of weights
N: Bit width of inputs
I , W: Maximum number of nodes per layer
D: Network depth (number of layers)
P: Parallelism. For a given layer, this is the maximum number of nodes whose output can be computed in a single timestep.

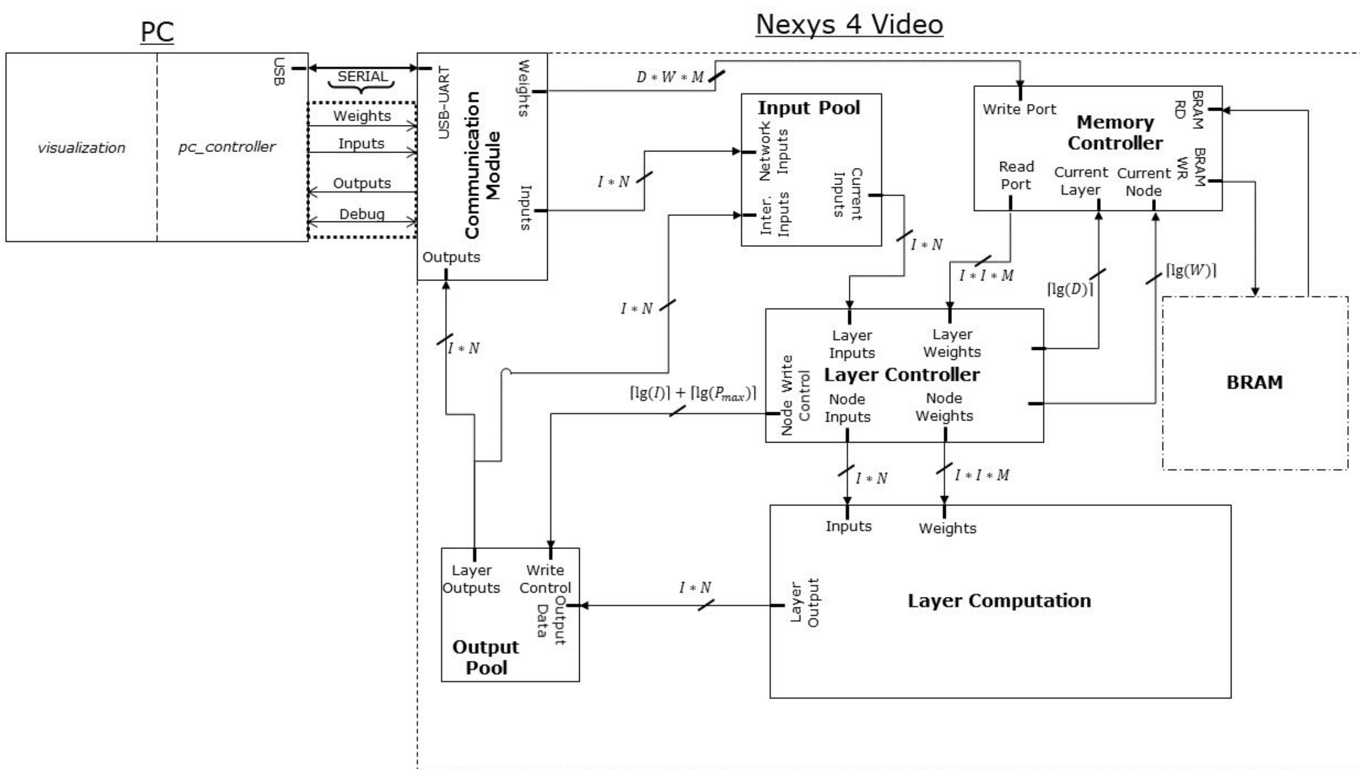
Timestep: When referring to timesteps or steps in reference to the Layer Controller and Layer Computation modules we specifically mean the time required to calculate a single set of node outputs.

For our particular implementation:

T = 740 (DSP slices in Nexys Video board)
M = 24 bits
N = 16 bits
I , W = 24 neurons
D = Runtime dependent [layers] (on network structure given from pc controller)
P = I neurons (entire layer computed simultaneously)

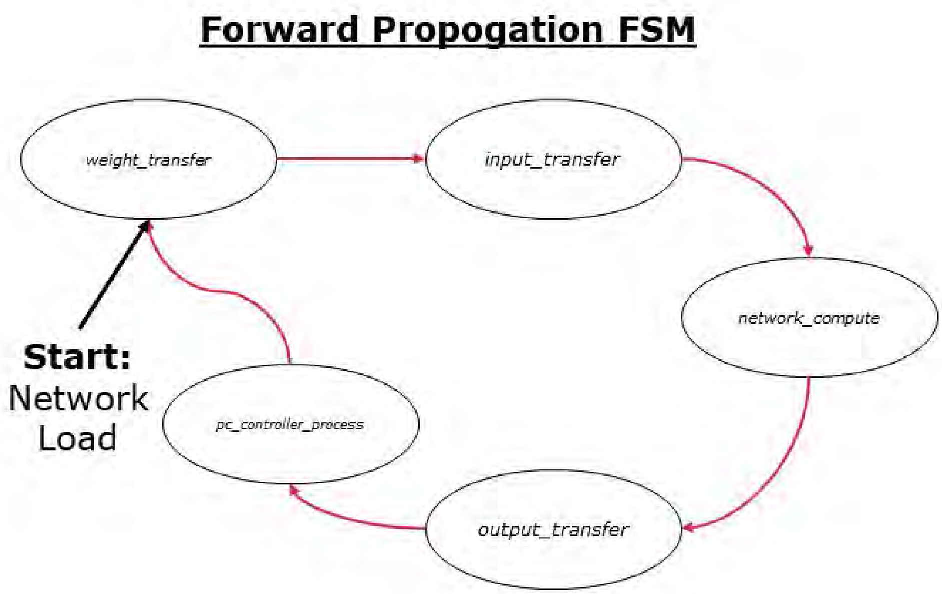
From these values many other necessary constants can be derived as described in the master_params.vh verilog header. For example *PARAM_I_M1 = (I+1)*M* which defines the number of weight and threshold bits needed per node. This is the case because every node has a weight for all nodes in the previous layer. In addition every node has a single threshold value of width M.

3. Implementation



The above diagram depicts a high-level overview of the implemented design. In the following sections we will detail the high-level FSM the design implements. We will then provide detailed functional descriptions of each module. This will be followed by a discussion of our visualization and testing frameworks along with a performance analysis of our final implementation.

3.1. State Machine



The diagram above describes the implemented FSM which computes the outputs of a multi-layer network on the FPGA. We begin in the *weight_transfer* communication state. In this state the weights for the entire neural network (in this case a single layer) are sent across the USB-UART interface from the PC to the FPGA Communication Controller module. The FPGA then stores these weights through the Memory Controller module. Along with these weights additional network topology information is sent. This currently includes the layer's parallelism, "P", and maximum node index.

Once all network information is transferred successfully, we enter the *input_transfer* state. In this state the network input layer values are sent to the FPGA via the UART interface in a similar fashion to the weights. These inputs are then stored in the Input Pool module as opposed to the Memory Controller module.

When all values for the input layer have been transferred, we begin the *network_compute* step. This state contains the bulk of our logic and computation. The parallelism "P", discussed earlier, defines the number of nodes that can be computed in parallel for a given layer. In order to perform parallel computation of the forward propagation values for a given set of active nodes, the Layer Controller module reads in the appropriate inputs from the Input Pool and weights from the Memory Controller, and forwards these values to the Layer Computation module. The number of nodes that can be computed in parallel "P" is given by the expression T / I, where T represents the total number of DSP multipliers, and I represents the number of inputs to any given node in the layer. After this computation is complete, the outputs are written to the output pool. When multiple layers are being computed we signal a transfer from the output pool to input pool and computation restarts. When all final layer outputs are in the output pool, the layer controller signals a transition to the *output_transfer* state.

In the *output_transfer* state, the communication module reads the layer's outputs from the output pool, and sends them to the PC as described in the protocol section. Upon reception of all output data the *pc_controller* can begin using the outputs received.

3.2. Communication Protocol (Josh)

3.2.1. Overview

In order to perform inference on the FPGA we must have the ability to transfer data between the Host PC and FPGA. A common protocol is shared between the PC and FPGA in order to encode all the necessary data. Specifically, we have a process running on a computer that feeds network topology information, including weights, and network inputs to the FPGA. This process is referred to as the *pc_controller*. On the FPGA side the *communication_module* receives this information and decodes it for use. The module also handles the transfer of network outputs back to the PC.

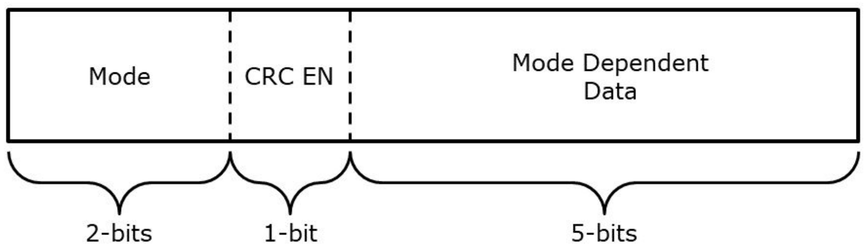
Our serial communication protocol has two actors: a sender and a receiver. For the *weight_transfer* and *input_transfer* states, *pc_controller* is the sender and the *communication_module* is the receiver. The roles are reversed in the *output_transfer* state.

3.2.2. Packet Streams

The protocol itself is defined as follows. All data is sent as a packet stream. A stream is defined to be 1 header packet and 9 data packets where a packet is a single byte. This stream abstraction allows for data to be sent in indepent chunks, and any data that would span multiple chunks is sent across multiple packet streams. Note that any leftover space in a packet stream after data insertion is zero-padded. The decision to make the packet stream 10 packets or 80 bits is that it aligns well with out weight bit-width of 24 bits and input/output bit-width of 16 bits. Excluding the header, there are 72 data bits per stream. This means that a single stream can contain 3 weight values with no wasted space or overflow. For inputs there is an overflow of 8 bits per stream (72 data bits % 16 bit input/outputs = 8 bits). This slight overlap does not add much complexity to input encoding as all data can still be handled on a byte-level granularity.

3.2.3. Packet Stream Headers

Packet Stream Header



Header packets contain metadata for the packet stream as a whole. The first 2 bits specify the mode for the following data packets. The supported modes are, *weight, input, output,* and *debug.* The next bit is whether CRC data for the stream should be expected immediately after the stream ends[1]. The final 5 bits may contain additional metadata dependent on the mode of the stream.

Mode 0 is weight transfer mode, which indicates the data packets are weight packets. The mode dependent header data is the layer id of the weights. Note that the communication protocol for weight transfer contains additional requirements so network metadata can be sent as data packets. Specifically, in the first packet stream for each layer (i.e. whenever the layer id portion of the weight header changes), the first 6 data packets must be the parallelism and maximum node id for the layer respectively. Note that 3 data packets per information is 24-bits the same width as a weight. This ensures that no special encode or decode logic must exist to locate this data within a stream, however the *pc_controller* and *communication_module* must know to treat the first two "weights" for each layer specially. In addition the first actual weight for every node is the node's threshold rather than a weight connecting it to the previous layer.

Mode 1 is input transfer mode and mode 2 is output transfer mode. These types contain the same data, however in input transfer mode data flows from *pc_controller* to *communication_module* and in the opposite direction in output transfer mode. In this mode the dependent data is just zeroed. The input packets flow PC to FPGA and output packets from FPGA to PC.

Mode 3 is debug mode, indicating debug packets which are used to transfer debug data from FPGA to PC. In this mode the dependent data is an error code. Different error codes define different interpretations of the debug data packets.

As described in annotation 1, crc check is currently disabled, however this the protocol still defines its use as follows. If CRC_EN=1 every packet stream will be followed by a corresponding CRC bit sequence. This sequence will be checked on the receiver by recalculating the CRC of the concatenation [packet_stream, CRC]. If this does not yield 0 an ERROR header packet is sent. Otherwise a SUCCESS packet is sent (All 0 bits). Note that this means communication with CRC_EN=1 is necessarily two-way as confirm messages must be sent from

[1] Note that the CRC_EN bit is always disabled in our current implementation as we did not encounter any errors during data transfer. However, the bit was left as it allows for easy extension in the future if some error correction must be implemented.

the receiver after every packet stream from the sender. Note that an LED on the FPGA is lit if any CRC error is detected or error packet received. If any error packet is received by the sender, it restarts transmission of the current state from the beginning.

3.3. Communication Module (FPGA)

This module will implement the FPGA end of the communication protocol described above. The input will be the TXD port (C4) of the FTDI FT232R USB-UART bridge. The output is the RXD port (D4). This module will operate at 115,200 baud. Note that the FT232R bridge is rated for up to 12MBaud, but we used a lower baud rate to simplify debugging as well as to better demonstrate the performance implications of a high communication overhead. These implications will be discussed further in section 7.

The implementation is similar to Lab 2B and Lab 5C. For this reason the uart interface was abstracted away in the modules *uart_rx* and *uart_tx* for receiving and transmission respectively. *uart_rx* was Sebastian's Lab5C module modified to buffer an entire packet stream before sending it for decode. *uart_tx* was based on lab2b, but it takes in a packet stream and transmits at the necessary baudrate. Note that the 16x supersample clock generator was also abstracted into the *uart_clkdiv* module. The *communication_module* itself handles the decoding and encoding of packet streams for the previously described uart interfaces.

For decoding weight streams, every 3 data packets are combined to form a single 24-bit weight. The current layer_id is extracted from the header and also output from this module. The module must keep some state regarding the last layer received in order to decode the network metadata described in the communication protocol section. If it detects that it has received the first packet stream for a new layer, it will treat the first two weights as the layer's parallelism and maximum node id respectively. These values are stored into output registers and sent to the *parallelism_pool* and *max_node_id_pool.* Note that the fact the first weight per node is a threshold is completely ignored by the communication module to cut down on complexity. It simply assumes every node has *I+1* weights. Once all weights for a single node are received the communication module asserts a signal that the weight buffer is full. This signal is handled by the memory controller as described in section 3.4. This process continues as long as weight packets are received.

Input decoding is much simpler than weight decoding. Even though a single input may span multiple packet streams, all streams are buffered contiguously in the ordered received. This places the split input bytes consecutively. The communication module expects *I* inputs, so once it receives this many it asserts to the input pool that it should read the input buffer.

Note than sending an invalid number of weight or input bits such that it only partially fills the communication module weight or input buffer is disallowed by the communication protocol.

Once a debug packet with all 0 data packets is received, the communication module asserts that transfer in is completed and the layer controller begins working.

This module will also handle the sending of output data from the FPGA to the computer once the *layer_controller* asserts that the last layer's outputs are in the output pool. These outputs will be read from the output pool, translated to packet streams, and each stream is then sent through *uart_tx*.

The main lesson learned through this project is in regards to this module. As I was testing communication it would have been very useful to create a test that combined only this module with the memory controller and input pool, and then used this test actually integrated with the computer. I wrote a test similar to this clsoer to the end of the project that would simply send back to the computer the weights or inputs it had received. This tests not only the communication and decoding aspect, but also it's interaction with the elements it needs to store persistent data to. This exposed a couple of bugs that would have been nice to find much earlier in the process. My main takeaway from this experience is to begin testing modules closer to the the environment they will be used in as soon as possible rather than relying on tests in isolation to prove correctness of the system as a whole.

3.4. Memory Controller (Josh)

As we have defined our maximum network width and depth we can derive how much memory is needed for the storage of weights. As described in the symbol key, we have a maximum network width of 24 and depth of 4. Every node has *24 weights + 1 threshold* worth of bits to store through the memory controller. Multiplying all of this by a weight width of 24-bits yields:

$$24weightwidth * (24netwidth * 4netdepth * (24netwidth + 1threshold)) = 57{,}600\ bits$$

The Nexys Video has 13 MBits of fast BRAM, so this is well within the limitations of the board. As described in section 3.2, the communication module sends the weights and thresholds a single node at a time (*I* weights + 1 threshold). This yields a write port width of:

$$24weightwidth * (24netwidth + 1) = 600\ bits$$

We synthesized a BRAM IP core with this port width, and depth defined as the maximum number of nodes in a network. In regards to the memory controller implementation itself it has two distinct operating modes: writing and reading.

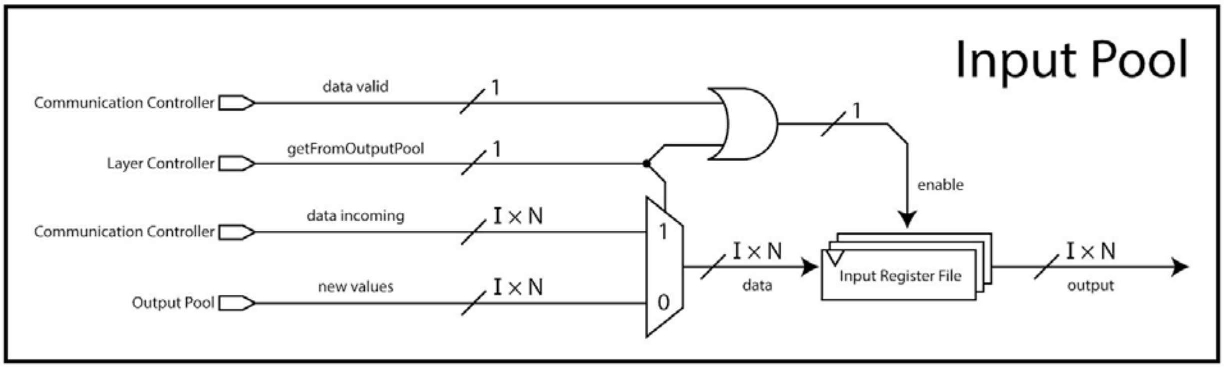
For writing the memory controller keeps an internal counter tracking the number of nodes that have been written so far. It then writes to this index in BRAM. The resulting memory layout is such that any reads will be from consecutive indices, as all nodes within a layer will have been stored one after another.

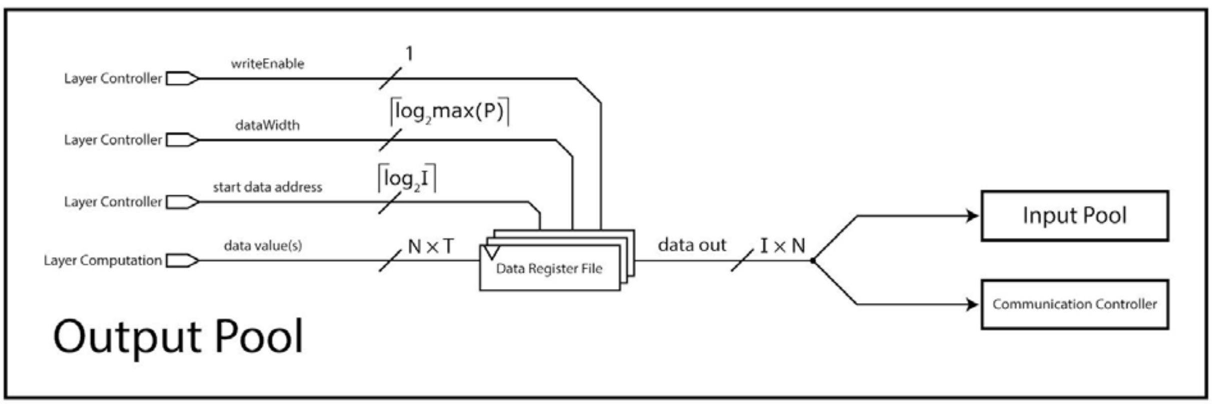
For reading the layer computation module expects to receive weights and thresholds for *P* (parallelism) nodes in the current layer. For this reason two output shift registers are created *read_weights* and *read_thresholds*. In order to read all necessary values from memory, a single layer read request is broken up into a BRAM read request for *P* nodes in the layer. The initial read address is calculated from the signals *read_layer_id* and *read_node_id*. Since data is written layer-by-layer we can reconstruct the start index for a given layer and node as:

*read_layer_id*MAX_NET_WIDTH + read_node_id*

Once a single node read response is returned from BRAM, the first weight is shifted into the output threshold shift register and the remaining weights shifted into the output weight shift register. *P* defines the parallelism of the current layer, but if *MAX_NET_WIDTH % P != 0* then it is possible that less than *P* reads must be done for the final nodes in a layer. For this reason the layer controller checks the current node id being read against the maximum node id for the layer. Note that this id verification functionality is not needed in the current implementation given that we have enough DSP slices to support calculating an entire layer per computation step. Thus, *P* will never cause a read past *max_node_id* as *P = LayerWidth.* Once all consecutive BRAM reads are completed the output shift registers are full and a *read_rdy* signal is asserted.

3.5. Pool Module (Josh)



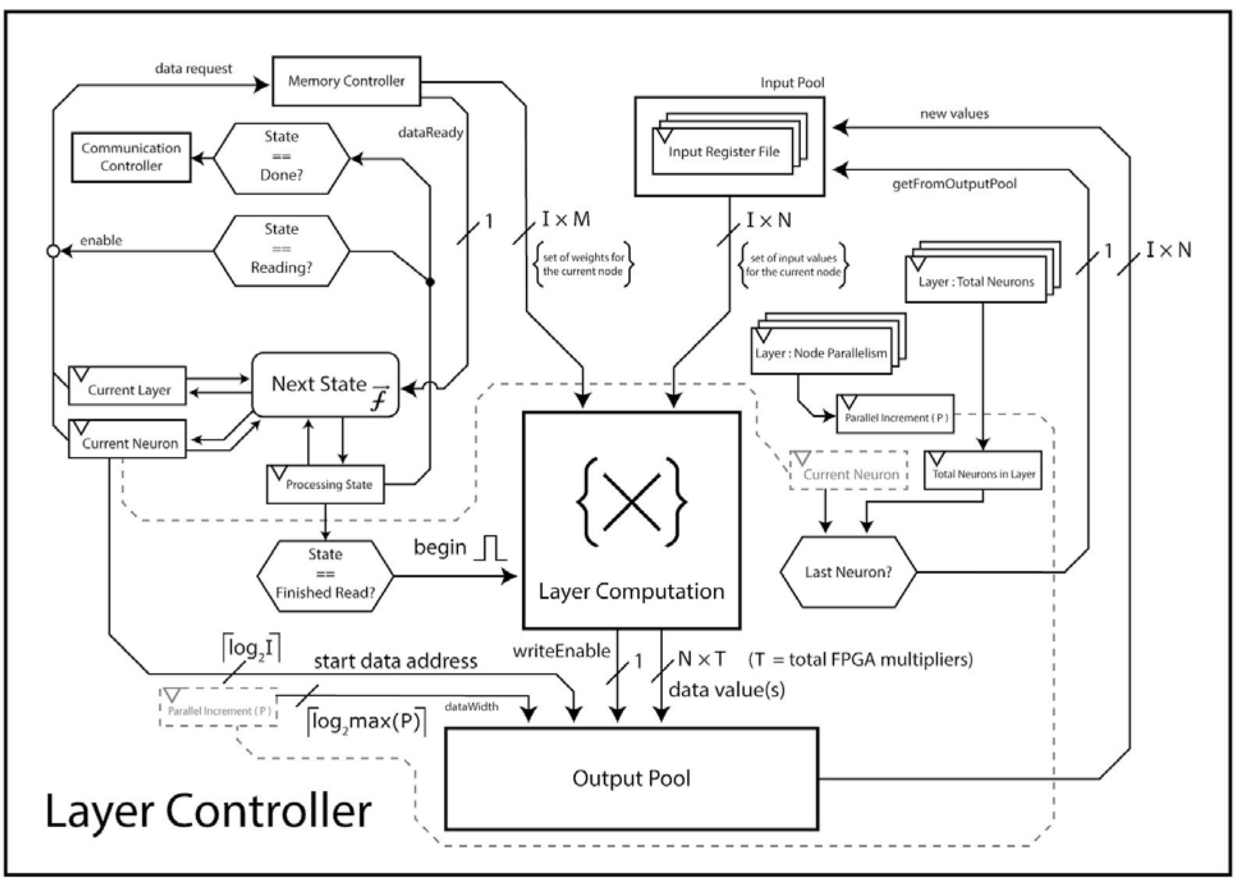


The Input Pool module encapsulates the input data buffer that is used to calculate values in the Layer Computation module. The write inputs to this register file come from the Output Pool and Communication Controller modules, given that the buffer can store initial values received from the host computer or hidden layer values stored in the Output Pool buffer after computation is complete. In the latter case, these output values are passed to the input pool since they are required to calculate the next layer of neuron values. The register file data write is enabled when either the data valid signal from the communication controller or the getFromOutputPool signal from the Layer Controller is asserted.

In addition to the input pool, two other pools are written to by the communication module, but for the purpose of storing network metadata. Specifically, a parallelism pool and max node id pool were created. As described in the communication section, these are values that occur once per received layer. The index to which the data should be written is defined by the current *layer_id* output of the communication module. The communication module asserts the *write_en* signal to these pools once per node received. As the *layer_id* is the same for all nodes in the same layer, multiple writes of the same data occur for every layer, but this has no negative effects. To read from these pools the *index* input is used to extract a value from the internal buffers and store it in an output register. Reads are initiated by the layer controller in order to read from *memory controller* as this module needs to know the parallelism and maximum node id of the layer being read.

The Output Pool serves as the buffer for Layer Computation outputs. As opposed to the Input Pool, which modifies the entire register file on a write operation, the output pool uses the data start address and data width values from the Layer Controller to determine which registers to write to when writeEnable is asserted. This occurs because the output pool was designed to allow for partial computation of layers where some outputs will be sent after one computation step, and the rest of the outputs will be sent across some number of computation timesteps. Note, however, that although the output pool supports this functionality, the rest of the system currently always calculates a full layer per computation step. This change occurred as with the increased number of DSP slices on the Nexys Video, we had enough multipliers to do so. The data from this register file is forwarded to the Input Pool upon layer computation completion and the Communication Controller once all layers finish.

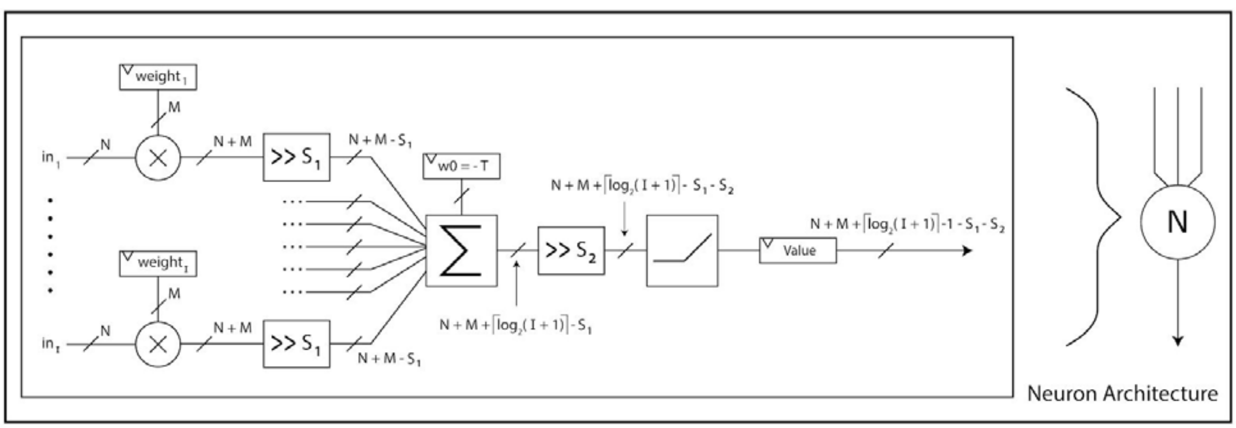
3.6. Layer Controller (Sebastian)

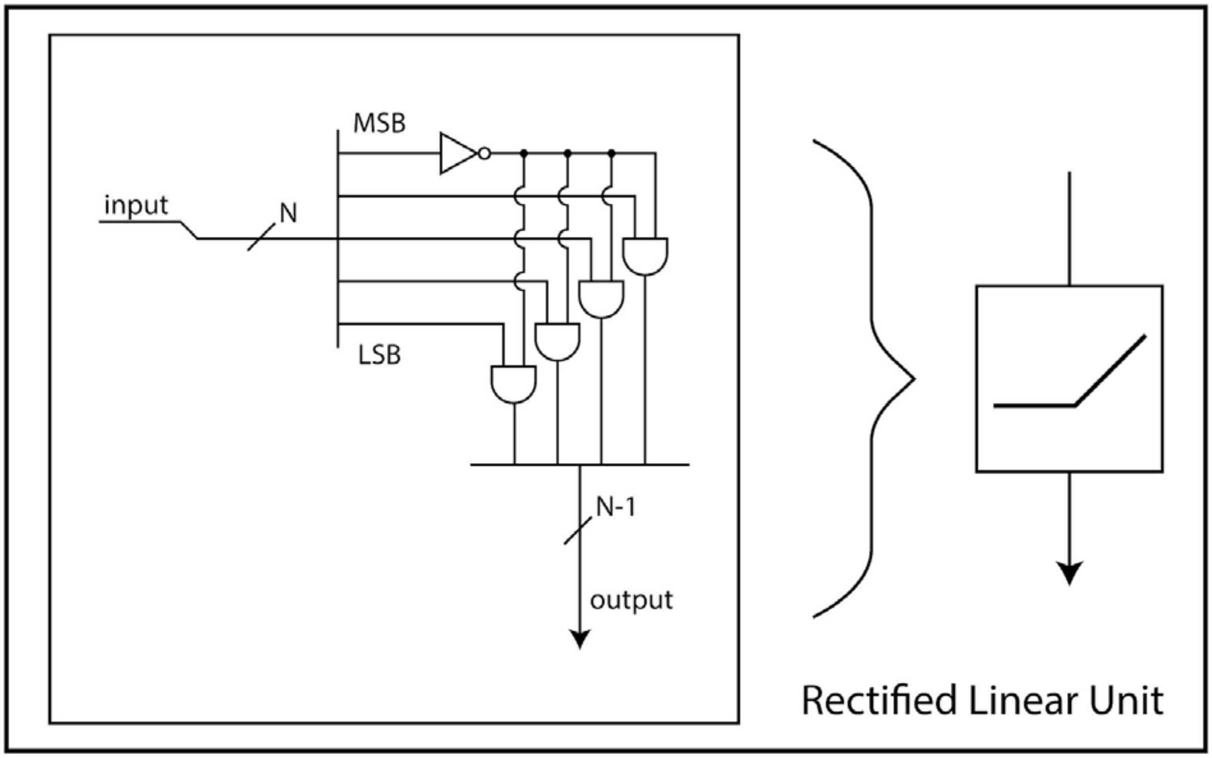


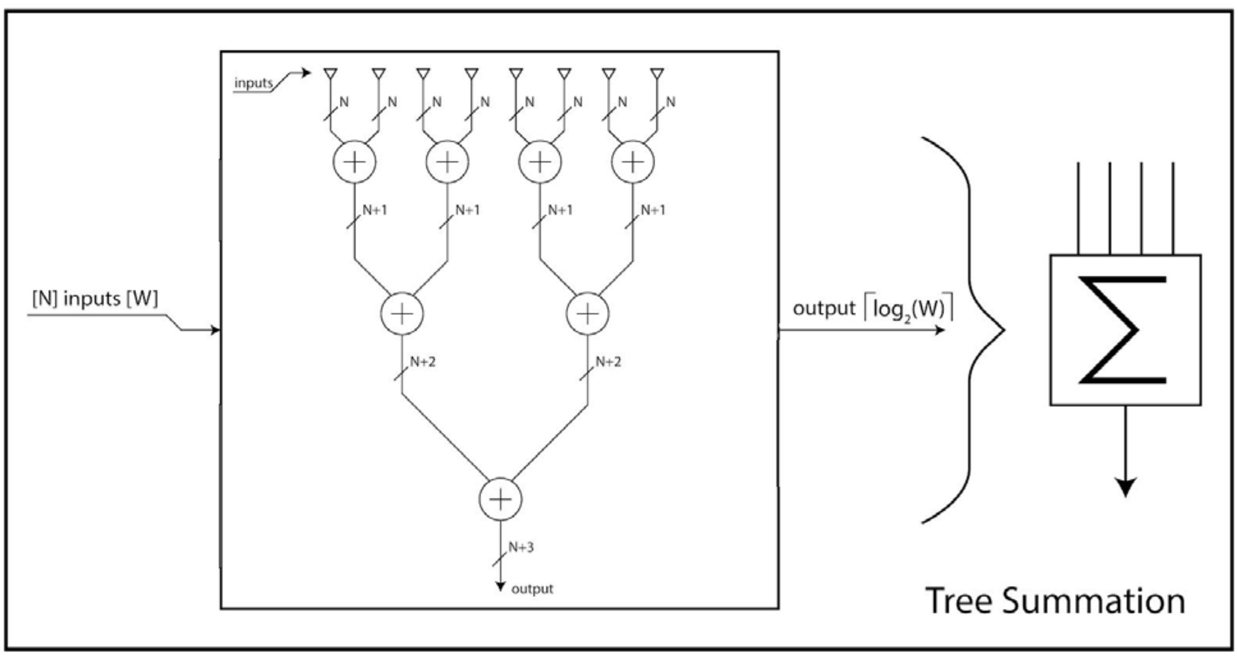
The Layer Controller module acts as the primary control FSM for calculating the neural network layers. To begin, the network depth is loaded from the Communication Controller via the *commCTL_networkDepth* bus and corresponding ready signal. During this data latching the requests for layer architecture data and neuron weight data for the primary layer are initiated via *memCTL_getLayerDataRequest* and *memCTL_getNeuronDataRequest* signals. These values are latched once the requests have valid data on their respective buses and active ready signals. The Layer Controller then waits until the Input Pool has neuron input data ready for calculation, which forms the *S_WAITING_INPUTS* state of the FSM. Once these conditions are met, the Layer Controller sends a pulse to the Layer Computation module (*layerComputation_startPulse*) to initiate the calculations for the layer. The Layer Controller FSM waits for the layer computation to finish (state *S_WAITING_LAYERCOMP*), and if the current layer being considered is the last layer, a signal is asserted for the communication controller to fetch data outputs from the Output Pool (*commCTL_isStateDone*). If the layer being processed is an intermediate ("hidden") layer, the layer counter is incremented, and requests for the next layer architecture and neuron weights are sent to the memory controller. The FSM then returns to the input waiting state as previously delineated.

As inputs, the Layer Controller receives the network depth, parallelism values, maximum neuron index for the given layer, as well as flags from other modules indicating the ready status of weights, inputs, and valid data outputs. The Layer Controller outputs the data ready signal to the Input Pool so that values are cycled from the Output Pool, parameters for the Memory Controller to fetch the appropriate data values, request signals to assert data fetch, as well as flags to the Communication Controller and Layer Computation modules regarding data-flow validity and timing, respectively.

3.7. Layer Computation (Sebastian)







The Layer Computation module serves as the central data processor for the project. The inputs of this module include the incoming weights, thresholds, and inputs for the neurons in the current layer of computation, as well as a start pulse from the Layer Controller. To begin, the set of weights and inputs are multiplied in pairs, effectively calculating the dot product between the two value vectors. These values are then right shifted by a quantity “S1”, in order to reduce the hardware needed for summation as well as reduce the probability of an overflow result. The results of this shift are stored in the *multiplicationResult* bit array. The values are then added together using an array of one tree adder per neuron *TreeAdder24(0-23)*, and then right shifted by another quantity "S2" to reduce the bit width of the output values to be the same as the input values, given the cyclic implementation of calculating layers in this hardware implementation. Once all values are computed, they are sent to the Output Pool for further data routing along with an enable signal to indicate that valid data is being sent across the *outputPool_finishedVals* bus. Once the values are latched by the Output Pool, the Layer Computation module waits for the next set of weights and inputs to be sent, and initiates the next round of calculation once the *startPulse* signal from Layer Controller is asserted. The module uses a FSM to keep track of arithmetic steps, consisting of states *S_WAITING*, *S_SUM*, and *S_DONE*.

4. Neural Net Implementation / Visualization (Sebastian)

The visualization software was critical to showcasing the functionality of the system. This code interfaced with the Lab 6 code provided by staff in 6.034 to use our FPGA solution. An encoding function was created to translate the Lab API to a format consisting of list of lists for layer inputs, weights, and thresholds ("neuronal biases"), that could then be sent over USB UART communication. The biggest challenge in modifying the lab code was changing the activation function from a sigmoid curve to a rectified linear unit, in order to match the FPGA neural net implementation. The data sets used would quickly cause the well-known "dying ReLU" problem to occur, at which point no further training could be performed. Forcing data inputs and weights to be integers to match the FPGA implementation caused this problem to worsen. Solving this issue took a large portion of time, proving to be non-trivial. The solution used was implementing leaky ReLUs as opposed to rigid ReLUs, and then iteratively reducing the leakage factor to determine hyperparameters such as training rate that would allow for the net to be successfully trained with regular ReLUs. In addition, rather than training on the FPGA using only integers, the training was instead performed on the host computer using floating point arithmetic. Once the training was complete the network weights and thresholds were converted into integers. The FPGA would then perform inference on the training space, and the data received was displayed on the screen in a two-dimensional chart.

5. PC_Controller (Josh)

The implementation of the PC side of the communication protocol was done near the end of the project similar to the visualization code. The code for it can be seen in the *app.py* and *uart.py* files. Since the FPGA side was finished at this point the implementation in python was not very difficult. The underlying UART implementation is based off of the code given in lab5c with slight updates to support user-specified COM port selection through the command line. The communication protocol was implemented on top of this. In summary, weights and inputs were stored as multidimensional python int arrays, sent from the neural network implementation described above. Two modifications were required on these arrays before transfer. First, all non-layer arrays were zero-padded to length "I" in order to provide a constant length shared by the FPGA and PC. Once this was done, the multidimensional arrays would be flattened. At this point the "send_data_stream" method would be called specifying the data type being sent and the corresponding data array. Python would then iterate over this array creating and sending packet streams. Doing so involved converting the data array to bytes where the number of bytes per entry is stream-type dependent as specified in the communication protocol (i.e. weights are 3 bytes and inputs 2-bytes). Once the entire array is in bytes form the code takes every 72 data bytes, prepends a header byte, and transfers it using pyserial. If there are less than 72 data bytes the stream is zero-padded.

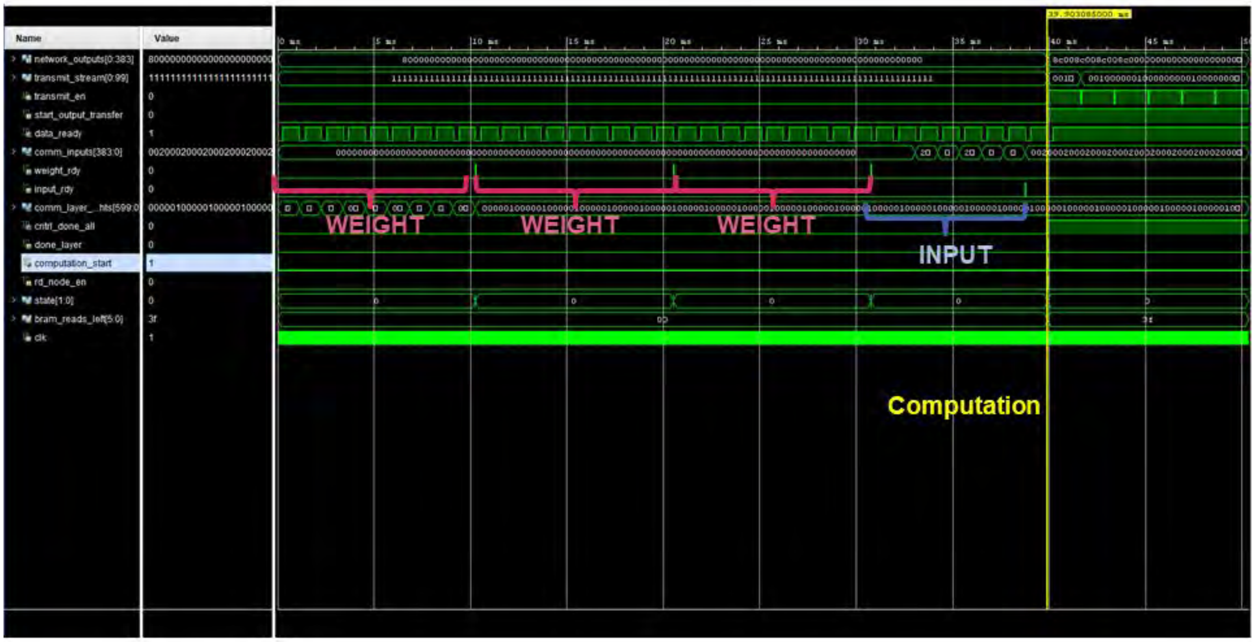
6. Top Level Module Implementation & Debugging (Josh)

Implementation of the top-level module *_top* involved modifying the constraints file with all I/O ports necessary for implementation and debugging as well as connecting the modules described above. As we had a well-defined block diagram, connecting the modules took very little time.

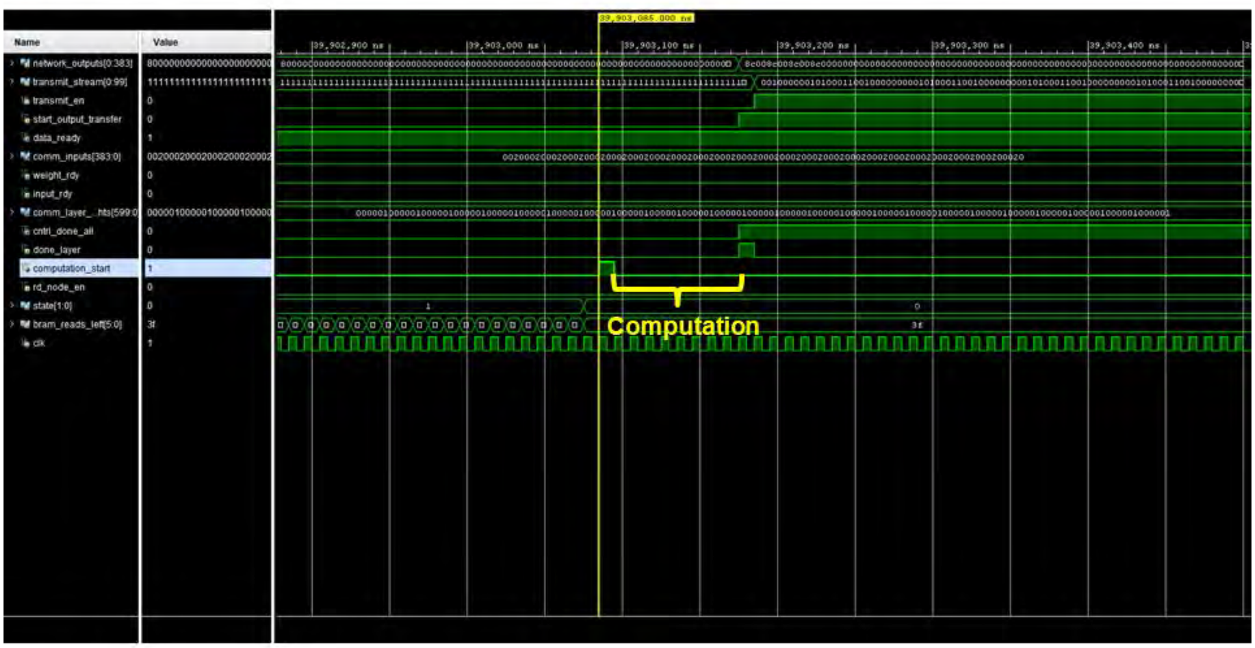
The main hurdles came while debugging all of the modules integrated together at the top level, taking more time than implementing the modules themselves. Though all modules worked in isolation, once connected there were some mismatches in control signals and data layout as some modules were not changed to reflect design updates since the proposal. Along with this, connecting the modules would sometimes expose internal bugs within modules that had not been tested for prior. This full system testing was done partially in simulation using the *full_test.v* testbench. This testbench supported sending packet streams across the *uart_rx* interface and then having the final network outputs flow all the way through a dummy *uart_rx* interface to mimic the PC receiving the outputs. This test would check all aspects of the design, which was very useful. However, when debugging the entire system rather than single modules it was much harder to track down where incorrect outputs were coming from.

app.py contained testing code in addition to the code necessary to support visualization. This eased the process of constructing meaningful test data as *app.py* provides python constructs, while in the simulation testbench, *full_test*, packet streams had to be manually constructed.

7. Performance



Depiction in simulation of the entire process from the simulated transmission of weights and inputs, computation of outputs, and transmission of these outputs.



Same simulation as the first picture, zoomed into the computation step.

The simulated receive and transmission rate is 1:1 with real usage since the uart_clkdiv module was driven by the simulation clock, and the *uart_tx* module assumes a 100MHz system clock when transmitting at 115KBaud. Note how transmission of weights and inputs takes ~40ms, while computation takes about 10 clock cycles with a 1ns clock. Increasing the baudrate would help relieve this issue.

Appendix A: Verilog Implementation

(Note: lines prefaced with "..." are continuations of code in the previous line, and do not denote line breaks in the source code file.)

Top Level Module

```verilog
`timescale 1ns / 1ps
`include "master_params.vh"

module _top (
    input CLK_100MHZ,
    input UART_TX_IN,
    input CPU_RESETN,
    output UART_RX_OUT,
    output [6:0] LED
    );


    // STATE
    localparam STATE_INIT = 0; // PC is sending weights and inputs
    localparam STATE_CALC = 1; // FPGA is calculating network outputs
    localparam STATE_OUT = 2;
    reg state;
    reg network_inputs_rec;
    reg network_weights_rec;



    ////// Communication module wires
    wire [`PARAM_E_EM-1:0] debug_outputs;
    wire [`PARAM_I_N-1:0] comm_inputs;
    wire [`PARAM_I_M1-1:0] comm_layer_weights;
    wire comm_input_rdy, comm_weight_rdy;
    wire comm_transfer_in_done, comm_transfer_out_done;
    wire network_outputs_done, debug_outputs_done;
    wire [`BTW_MAX_NET_DEPTH-1:0] comm_layer_id;
    wire [`BTW_MAX_PARALLELISM-1:0] comm_layer_parallelism;
    wire [`BTW_MAX_NET_WIDTH-1:0] comm_layer_max_node_id;
    wire [`BTW_MAX_NET_DEPTH-1:0] comm_network_depth;
    wire [2:0] comm_debug_state;

    ////// Memory Controller Wires
    wire mem_rd_layer_en, mem_rd_neuron_en;
    wire mem_rd_weights_rdy;
    wire [`PARAM_I_I_M-1:0] mem_rd_weights;
```

```
    wire [`PARAM_I_M-1:0] mem_rd_thresholds;

    ////// Output Pool Wires
    wire cntrl_rd_new_layer; // Pulse to update cntrl_parallelism and cntrl_max_node_id
    wire [`PARAM_I_N-1:0] output_pool_data;

    ////// Parallel Pool Wires
    wire [`BTW_MAX_PARALLELISM-1:0] parallelism_rd;
    wire parallelism_pool_rdy;

    ////// Max Node Pool Wires
    wire [`BTW_MAX_NET_WIDTH-1:0] max_node_id_rd;
    wire max_node_id_pool_rdy;

    ////// Input Pool Wires
    wire [`PARAM_I_N-1:0] input_pool_rd;
    wire input_pool_rd_rdy;


    //// Layer Computation wires
    wire computation_done, computation_start;
    wire [`PARAM_I_N-1:0] computation_output;

    ////// Layer Controller wires
    wire cntrl_done_all;
    wire [`BTW_MAX_NET_DEPTH-1:0] cntrl_cur_layer_id;
    wire [`BTW_MAX_NET_WIDTH-1:0] cntrl_cur_node_id;
    wire [`BTW_MAX_PARALLELISM-1:0] cntrl_parallelism;
    wire [`BTW_MAX_NET_WIDTH-1:0] cntrl_max_node_id;

    // System wires
    wire sysclk = CLK_100MHZ;
    wire btn_cpu_resetn;
    debounce debounce_reset(.clock(sysclk), .reset(0), .noisy(~CPU_RESETN), .clean(btn_cpu_resetn));
    wire reset = btn_cpu_resetn | comm_transfer_out_done;

    // DEBUG wires
    assign LED[0] = comm_debug_state[0];
    assign LED[1] = comm_debug_state[1];
```

```
    assign LED[2] = comm_transfer_in_done;
    assign LED[3] = cntrl_done_all;
    assign LED[4] = comm_transfer_out_done;
    assign LED[5] = output_pool_data[7:0] != 0;
    assign LED[6] = input_pool_rd[7:0] == 1;



    memory_controller memory_controller_instance(.clk(sysclk),
.reset(reset),
            .write_weights_rdy(comm_weight_rdy),
.rd_layer_en(mem_rd_layer_en), .rd_node_en(mem_rd_neuron_en),
            .write_layer_weights(comm_layer_weights),
.write_layer_id(comm_layer_id),
            .read_layer_id(cntrl_cur_layer_id),
.read_node_id(cntrl_cur_node_id),
.read_parallelism(cntrl_parallelism),
            .read_weights(mem_rd_weights),
.read_thresholds(mem_rd_thresholds),
.read_weights_rdy(mem_rd_weights_rdy));

    communication_module comm_module_instance(.clk(sysclk),
.reset(reset), .uart_tx_in(UART_TX_IN),
.uart_rx_out(UART_RX_OUT),
                .network_outputs(output_pool_data),
.debug_outputs(debug_outputs),
                .debug_outputs_done(debug_outputs_done),
                .network_inputs(comm_inputs),
.input_rdy(comm_input_rdy),
                .layer_weights(comm_layer_weights),
.weight_rdy(comm_weight_rdy), .layer_id(comm_layer_id),
                .layer_parallelism(comm_layer_parallelism),
.layer_max_node_id(comm_layer_max_node_id),
                .network_depth(comm_network_depth),
.in_transfer_done(comm_transfer_in_done),
                .start_output_transfer(cntrl_done_all),
.out_transfer_done(comm_transfer_out_done),
                .debug_state(comm_debug_state));


    LayerController layer_controller_instance(.clk(sysclk),
```

```
.reset(reset),
                    .commCTL_networkDepth(comm_network_depth),
.commCTL_networkDepthReady(comm_transfer_in_done),
                    .parallelPool_parallelism(parallelism_rd),
.parallelPool_parallelismReady(parallelism_pool_rdy),

.neuronIndicesPool_maxNeuronIndex(max_node_id_rd),
.neuronIndicesPool_maxNeuronIndexReady(max_node_id_pool_rdy),

.memCTL_weightsAreReady(mem_rd_weights_rdy),
.inputPool_inputsAreReady(1),
.layerComputation_dataReady(computation_done),
                    .outputPool_dataWidth(cntrl_parallelism),
.outputPool_maxNeuronIndex(cntrl_max_node_id),

.inputPool_getFromOutputPool(cntrl_rd_new_layer),
                    .memCTL_currentLayer(cntrl_cur_layer_id),
.memCTL_currentNeuron(cntrl_cur_node_id),
.memCTL_parallelism(cntrl_parallelism),

.memCTL_getLayerDataRequest(mem_rd_layer_en),
.memCTL_getNeuronDataRequest(mem_rd_neuron_en),
                    .commCTL_isStateDone(cntrl_done_all),
.layerComputation_startPulse(computation_start)
                    );
    LayerComputation layer_computation_instance (.clk(sysclk),
.reset(reset), .startPulse(computation_start),
                    .incomingWeights(mem_rd_weights),
//topmost weight is "w0 = - T" Seperated out by memory
controller and sent in threshold port  //this is currently for
just one node, if we want to do parallelism this needs to be
P{}'d (consecutive weights must be sent for each node)
                    .incomingThresholds(mem_rd_thresholds),
                    .incomingInputs(input_pool_rd),

.outputPool_finishedVals(computation_output),
                    .outputPool_writeEnable(computation_done)
                    );

    ////// Input pool wires
    input_pool input_pool_instance(.clk(sysclk),
```

```
.reset(reset),
                         .comm_write_en(comm_input_rdy), .cntrl_pool_transfer(cntrl_rd_new_layer),
                         .comm_data(comm_inputs), .output_pool_data(output_pool_data),
                         .read_data(input_pool_rd), .read_data_rdy(input_pool_rd_rdy));

    //// Parallelism pool
    pool#(.DATA_WIDTH(`BTW_MAX_PARALLELISM)) parallelism_pool_instance(.clk(sysclk), .reset(reset),
                        .write_en(comm_weight_rdy), .write_data(comm_layer_parallelism),
                        .read_en(mem_rd_layer_en), .index(cntrl_cur_layer_id), .read_data(parallelism_rd),
                        .data_rdy(parallelism_pool_rdy));

    //// Max Node Index pool
    pool#(.DATA_WIDTH(`BTW_MAX_NET_WIDTH)) max_id_pool_instance(.clk(sysclk), .reset(reset),
                   .write_en(comm_weight_rdy), .write_data(comm_layer_max_node_id),
                   .read_en(mem_rd_layer_en), .index(cntrl_cur_layer_id), .read_data(max_node_id_rd),
                   .data_rdy(max_node_id_pool_rdy));


    //// Output Pool
    output_pool output_pool_instance(.clk(sysclk), .reset(reset),
                    .write_en(computation_done), .parallelism(cntrl_parallelism),
                    .max_node_id(cntrl_max_node_id),
                    .write_data(computation_output), .cur_outputs(output_pool_data));

endmodule

```

Communication Module

```verilog
`include "master_params.vh"

module communication_module(
    input clk, //CLK_100MHZ
    input reset,
    input uart_tx_in, //UART_TX_IN,
    input start_output_transfer,
    input [`PARAM_I_N-1:0] network_outputs,
    input [`PARAM_E_EM-1:0] debug_outputs,
    input debug_outputs_done,
    output uart_clk, // Output subdivided clock signal for testing
    output uart_rx_out, //UART_RX_OUT,
    output reg input_rdy,
    output reg weight_rdy,
    output reg in_transfer_done,
    output reg out_transfer_done,
    output reg [`PARAM_I_N-1:0] network_inputs,
    output reg [`PARAM_I_M1-1:0] layer_weights, // We output a single node's weights  +cutoff per write
    output reg [`BTW_MAX_NET_DEPTH-1:0] layer_id,
    output reg [`BTW_MAX_PARALLELISM-1:0] layer_parallelism,
    output reg [`BTW_MAX_NET_WIDTH-1:0] layer_max_node_id,
    output reg [`BTW_MAX_NET_DEPTH-1:0] network_depth,
    output [2:0] debug_state
);

    localparam STATE_HEADER_WAIT = 0;
    localparam STATE_CRC_WAIT = 1;
    localparam STATE_IN_DONE = 2;
    localparam STATE_OUTPUT_TRANSFER = 3;
    localparam STATE_DEBUG_TRANSFER = 4;

    //// Note: Modes from packet header. Must match comm. protocol
    localparam MODE_WEIGHT = 2'd0;
    localparam MODE_INPUT = 2'd1;
    localparam MODE_OUTPUT = 2'd2;
    localparam MODE_DEBUG = 2'd3;

    parameter VERIFY_STOP = 1; // Whether UART_RX module
```

```
should verify a stop bit before accepting a packet


    //// UART RX Setup
    wire clk_serial_sample;
    wire verify_stop = VERIFY_STOP;
    // Contains a valid, complete packet stream
    wire [`BITS_PER_STREAM-1:0] packet_stream;
    wire packet_stream_rdy;
    //TDO synch reset signal with sample slk
    uart_clkdiv uart_clkdiv_instance(.clk(clk), .reset(reset),
.clk_out(clk_serial_sample));
    uart_rx uart_rx_instance(.clk(clk),
.uart_clk(clk_serial_sample), .reset(reset),
.signal(uart_tx_in), .verifyStopBit(1),
.data_out(packet_stream), .ready(packet_stream_rdy));
    assign uart_clk = clk_serial_sample;
    ////// UART TX Setup

    wire transmit_done;
    reg transmit_en;
    reg start_output_transfer_pulse;
    reg last_transmit_en;
    wire tx_reset = reset | (last_transmit_en == 0 &&
transmit_en == 1);//(start_output_transfer_pulse;
    reg [`TOTAL_BITS_PER_STREAM-1:0] transmit_stream;
    uart_tx uart_tx_instance(.clk(clk), .reset(tx_reset),
.en(transmit_en), .start_data(transmit_stream),
.xmit_data(uart_rx_out), .done(transmit_done));
    // Read in ouputs stream by stream
    // Read in debug stream by stream

    //// Internal Setup
    reg [2:0] state;
    reg [`CLOG2(`STREAMS_PER_OUTPUTS+1)-1:0]
output_streams_left;
    reg [`PARAM_I_N-1:0] cur_network_outputs;
    reg [`CLOG2(`PARAM_I_M1):0] weight_bits_left;
    reg [`CLOG2(`PARAM_I_N):0] input_bits_left;

    wire [1:0] header_mode = packet_stream[1:0];
```

```
    wire       header_crc =  packet_stream[2];
    wire [4:0] header_data = packet_stream[7:3];
    assign debug_state = {layer_parallelism[0], layer_max_node_id[0]};

    reg prev_layer_id;
    wire layer_first_weight_stream = (header_data != prev_layer_id);

    integer i;
    function [`TOTAL_BITS_PER_STREAM-1:0] data_to_stream (input [`BITS_PER_STREAM-1:0] data);
        for(i = 0; i < `PACKETS_PER_STREAM; i = i + 1) begin
            data_to_stream[8*(i+1)+2*i -: 8] = data[8*(i+1)-1 -: 8];
            data_to_stream[8*(i+1)+2*i+1] = 1;
            data_to_stream[8*i+2*i] = 0;
        end
    endfunction

    always @(posedge clk) begin
        last_transmit_en <= transmit_en;
        if (reset) begin
            state <= STATE_HEADER_WAIT;
            weight_bits_left <= `PARAM_I_M1;
            input_bits_left <= `PARAM_I_N;
            input_rdy <= 0;
            weight_rdy <= 0;
            in_transfer_done <= 0;
            prev_layer_id = -1;
            transmit_en <= 0;
            last_transmit_en <= 0;
            network_inputs <= 0;
            layer_weights <= 0;
            network_depth <= 0;
            output_streams_left <= 0;
            out_transfer_done <= 0;
            start_output_transfer_pulse <= 0;
            transmit_stream <= -1;
            layer_parallelism <= 0;
            layer_max_node_id <= 0;
```

```
        end else if(start_output_transfer_pulse || output_streams_left > 0) begin
            if (transmit_en == 0 && start_output_transfer_pulse == 1) begin
                start_output_transfer_pulse <= 0;
                // Starting new stream transfer
                transmit_stream <= data_to_stream({cur_network_outputs[`BITS_PER_STREAM-9:0], 6'd0, MODE_OUTPUT});
                cur_network_outputs <= cur_network_outputs >> (`BITS_PER_STREAM-8);
                transmit_en <= 1;
            end else if (transmit_en == 1 && transmit_done == 1) begin
                // transmission done
                transmit_en <= 0;
                output_streams_left <= output_streams_left - 1;
                if (output_streams_left == 1) begin
                    // Last packet just finished
                    out_transfer_done <= 1;
                end else begin
                    // Intermediate packet finished, so reset uart_tx for the next one
                  start_output_transfer_pulse <= 1;
                end
            end
        end else if (start_output_transfer && out_transfer_done == 0) begin
            // Start transfer process by reading in outputs
            //start_output_transfer_pulse <= 1;
            cur_network_outputs <= network_outputs >> (`BITS_PER_STREAM-8);
            output_streams_left <= `STREAMS_PER_OUTPUTS;
            transmit_stream <= data_to_stream({network_outputs[`BITS_PER_STREAM-9:0], 6'd0, MODE_OUTPUT});
            transmit_en <= 1;
        end else if (packet_stream_rdy && transmit_en == 0) begin
            case (state)
```

```
            STATE_HEADER_WAIT: begin
                if (header_crc)
                    state <= STATE_CRC_WAIT;
                case (header_mode)
                MODE_WEIGHT: begin
                    if (weight_bits_left >= `BITS_PER_STREAM - 8) begin
                        // We only need to read parallelism and max_node_id as first data value sin first weight packet stream for layer

                        // We are transferring an entire packet and must wait for at least 1 more stream to read the whole layer
                        if (layer_first_weight_stream) begin
                            prev_layer_id <= header_data;
                            layer_parallelism <= packet_stream[8+:`WEIGHT_WIDTH];
                            layer_max_node_id <= packet_stream[8+`WEIGHT_WIDTH+:`WEIGHT_WIDTH];
                            layer_weights[`PARAM_I_M1-1 -:`WEIGHT_WIDTH] <= {packet_stream[`BITS_PER_STREAM-1 -:`WEIGHT_WIDTH]};
                            weight_bits_left <= weight_bits_left - (`BITS_PER_STREAM - (8+2*`WEIGHT_WIDTH));
                        end else begin
                            layer_weights <= {packet_stream[`BITS_PER_STREAM-1:8], layer_weights[`PARAM_I_M1-1:`BITS_PER_STREAM-8]};
                            if (weight_bits_left == `BITS_PER_STREAM-8) begin
                                weight_bits_left <= `PARAM_I_M1;
                                weight_rdy <= 1;
                            end else begin
                                weight_bits_left <= weight_bits_left - (`BITS_PER_STREAM - 8);
                            end
                        end
                    end else begin
                        // This is the last stream in the
```

```
current layer
                        if (weight_bits_left < `BITS_PER_STREAM - 8) begin
                            layer_weights <= {packet_stream[8+`STREAM_WEIGHT_OVERLAP-1:8], layer_weights[`PARAM_I_M1-1:`STREAM_WEIGHT_OVERLAP]};
                             //weight_bits_left <= weight_bits_left - `STREAM_WEIGHT_OVERLAP;
                        end else begin
                            layer_weights <= {packet_stream[`BITS_PER_STREAM-1:8], layer_weights[`PARAM_I_M1-1:`BITS_PER_STREAM-8]};
                            //weight_bits_left <= weight_bits_left - (`BITS_PER_STREAM - 8);
                        end
                        weight_rdy <= 1;
                        weight_bits_left <= `PARAM_I_M1;
                    end
                    layer_id <= header_data;
                end
                MODE_INPUT: begin
                    if (input_bits_left >= `BITS_PER_STREAM - 8) begin
                        // We are transferring an entire packet and must wait for at least 1 more stream to read the whole layer
                        network_inputs <= {packet_stream[`BITS_PER_STREAM-1:8], network_inputs[`PARAM_I_N-1:`BITS_PER_STREAM-8]};
                        input_bits_left <= input_bits_left - (`BITS_PER_STREAM - 8);
                    end else begin
                        // This is the last stream in the current layer
                        if (input_bits_left < `BITS_PER_STREAM - 8) begin
                            network_inputs <= {packet_stream[8+`STREAM_INPUT_OVERLAP-1:8], network_inputs[`PARAM_I_N-1:`STREAM_INPUT_OVERLAP]};
                        end else begin
                            network_inputs <=
```

```
{packet_stream[`BITS_PER_STREAM-1:8], network_inputs[`PARAM_I_N-1:`BITS_PER_STREAM-8]};
            end
            input_rdy <= 1;
            input_bits_left <= `PARAM_I_N;
          end
        end
        MODE_DEBUG: begin
          network_depth <= layer_id;
          in_transfer_done <= 1;
        end
        default: ; // TODO: This is some kind of transmission error as we only have 3 valid input modes
        endcase
      end
      STATE_CRC_WAIT: begin
        // Begin parallel CRC check by passing crc bits to checker and latching the response
        state <= STATE_HEADER_WAIT;
      end
      // TODO: Output states

      default: ;
      endcase
    end else begin
      if (input_rdy) input_rdy <= 0;
      if (weight_rdy) weight_rdy <= 0;
    end
  end

endmodule

```

Constant Functions

```
`ifndef CONSTANT_FUNCS_H
`define CONSTANTS_FUNCS_H

// Constant clog2 hack as verilog has a $clog2 bug that breaks its use in localparameter and initial blocks
`define CLOG2(x) \
   ((x <= 2) ? 1 : \
    (x <= 4) ? 2 : \
    (x <= 8) ? 3 : \
    (x <= 16) ? 4 : \
    (x <= 32) ? 5 : \
    (x <= 64) ? 6 : \
    (x <= 128) ? 7 : \
    (x <= 256) ? 8 : \
    (x <= 512) ? 9 : \
    (x <= 1024) ? 10 : \
    (x <= 2048) ? 11 : \
    (x <= 4096) ? 12 : \
    (x <= 8192) ? 13 : \
    (x <= 16384) ? 14 : \
    (x <= 32768) ? 15 : \
    (x <= 65536) ? 16 : \
    -1)

`define CEIL(x,y) (((x)+(y)-1)/(y))

`define MIN(x,y) ((x) < (y) ? (x) : (y))

`endif
```

Layer Computation

```verilog
`include "master_params.vh"

/*
we're doing T mults in a step
those T mults can be broken down into a set of {inputs,weights}
therefore total number of inputs we have = total number of weights sent = T
how those inputs and weights are grouped is determined by P = T / I
*/
module LayerComputation
    (
    input clk,
    input reset,
    input startPulse,
    input [`PARAM_I_I_M-1:0] incomingWeights,
    input [`PARAM_I_M-1:0] incomingThresholds,
    input [`PARAM_I_N-1:0] incomingInputs,

    output reg [`PARAM_I_N-1:0] outputPool_finishedVals,
    output reg outputPool_writeEnable
    );

    wire [`PARAM_I_I_N-1:0] inputDuplicates = {`MAX_NET_WIDTH{incomingInputs[`PARAM_I_N-1:0]}};

    // Parameters //

    localparam S_WAITING = 2'd0;
    localparam S_SUM = 2'd1;
    localparam S_DONE = 2'd2;
    // Debugging
    //assign outputPool_finishedVals = incomingInputs;
    //assign outputPool_writeEnable = 1;

    // Registers //

    reg [1:0] state = S_WAITING; //"Processing State"

    reg [`PARAM_N_M_S1_I_I-1:0] multiplicationResult;
```

```verilog

    reg sumStartPulse = 0;
    wire sumOutputReady;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput0;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput1;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput2;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput3;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput4;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput5;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput6;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput7;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput8;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput9;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput10;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput11;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput12;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput13;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput14;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput15;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput16;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput17;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput18;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput19;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput20;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput21;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput22;
    wire signed [`PARAM_N_M_S1+3:0] sumOutput23;

    // Tree Adders //

    /* module TreeAdder24
    (
    input clk,
    input reset,
    input [`PARAM_N_M_S1_I-1:0] data,
    input inputReady,
    output outputReady,
    output reg [`PARAM_N_M_S1+3:0] dataOut
    ); */

    TreeAdder24 n0
```

```
(clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*0)+:`PARAM_N_M_S1_I],sumStartPulse,sumOutputReady,sumOutput0);
    TreeAdder24 n1 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*1)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput1);
    TreeAdder24 n2 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*2)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput2);
    TreeAdder24 n3 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*3)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput3);
    TreeAdder24 n4 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*4)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput4);
    TreeAdder24 n5 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*5)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput5);
    TreeAdder24 n6 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*6)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput6);
    TreeAdder24 n7 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*7)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput7);
    TreeAdder24 n8 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*8)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput8);
    TreeAdder24 n9 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*9)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput9);
    TreeAdder24 n10 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*10)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput10);
    TreeAdder24 n11 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*11)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput11);
    TreeAdder24 n12 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*12)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput12);
    TreeAdder24 n13 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*13)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput13);
```

```
    TreeAdder24 n14 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*14)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput14);
    TreeAdder24 n15 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*15)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput15);
    TreeAdder24 n16 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*16)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput16);
    TreeAdder24 n17 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*17)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput17);
    TreeAdder24 n18 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*18)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput18);
    TreeAdder24 n19 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*19)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput19);
    TreeAdder24 n20 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*20)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput20);
    TreeAdder24 n21 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*21)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput21);
    TreeAdder24 n22 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*22)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput22);
    TreeAdder24 n23 (clk,reset,multiplicationResult[(`PARAM_N_M_S1_I*23)+:`PARAM_N_M_S1_I],sumStartPulse,,sumOutput23);

    function signed [`PARAM_N_M_S1:0]shift1 (input signed [`PARAM_N_M:0] data);
        shift1 = data[`PARAM_N_M-1:`SHIFT_1];
    endfunction

    function signed [`PARAM_N_M_LG_S1_S2-1:0] shift2 (input signed [`PARAM_N_M_LG_S1-1:0] data);
        shift2 = data[`PARAM_N_M_LG_S1-1:`SHIFT_2];
    endfunction

```

```
    function signed [`INPUT_WIDTH-1:0] relu (input signed [`PARAM_N_M_LG_S1_S2-1:0] data);
        relu = data[`PARAM_N_M_LG_S1_S2-1] ? 'b0 : data[`PARAM_N_M_LG_S1_S2-2:0];
    endfunction

    wire signed [`PARAM_N_M_S1-1:0] test_mul, test_mul2, test_mul5;
    wire [(2*`PARAM_N_M_S1)-1:0] test_mul3;
    wire signed [`PARAM_N_M_S1_I-1:0] test_mul4;
    wire signed [`PARAM_N_M_S1+3:0] test_sum;
    wire signed [`PARAM_N_M_S1+10:0] test_sum3;
    wire signed [`WEIGHT_WIDTH-1:0] test_sum1;
    wire signed [`PARAM_N_M_LG_S1_S2-1:0] test_sum2;
    wire signed [`PARAM_N_M_S1-1:0] muls [`PARAM_I_I-1:0];
    // Loop indices //

    integer i_dotProd = 0;
    genvar i_test;
    integer test_idx = 574; // index into multiplication for loop
    integer test_idx_2 = 23; // index into corresponding adder tree

    // Sequential Logic //
    /**The first is wrong? Copies an extra msb from inputDuplicates*/
    assign test_mul = {{(`PARAM_N_M_S1-`INPUT_WIDTH){'b0}}, inputDuplicates[(test_idx*`INPUT_WIDTH)+:`INPUT_WIDTH]};
    assign test_mul2 = $signed(incomingWeights[(test_idx*`WEIGHT_WIDTH)+:`WEIGHT_WIDTH]);//{{(`PARAM_N_M_S1-`INPUT_WIDTH){'b0}}, inputDuplicates[15:0]};
    assign test_mul3[0+:`PARAM_N_M_S1] = shift1($signed(inputDuplicates[(test_idx*`INPUT_WIDTH)+:`INPUT_WIDTH]) * $signed(incomingWeights[(test_idx*`WEIGHT_WIDTH)+:`WEIGHT_WIDTH]));
    assign test_mul4 = multiplicationResult[(`PARAM_N_M_S1_I*test_idx_2)+:`PARAM_N_M_S1_I];// input to sum23
```

```
    assign test_mul5 =
multiplicationResult[(`PARAM_N_M_S1*test_idx)+:`PARAM_N_M_S1];
    assign test_sum = sumOutput23;
    assign test_sum1 =
incomingThresholds[(`WEIGHT_WIDTH*test_idx_2)+:`WEIGHT_WIDTH];
    assign test_sum2 = shift2($signed({1'b0, sumOutput23})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*test_idx_2)+:`
WEIGHT_WIDTH]);
    assign test_sum3 = relu(shift2($signed({1'b0,
sumOutput23})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*test_idx_2)+:`
WEIGHT_WIDTH]));

    generate
        for(i_test = 0; i_test < `PARAM_I_I; i_test = i_test +
1) begin
            assign muls[i_test] =
multiplicationResult[(i_test*`PARAM_N_M_S1)+:`PARAM_N_M_S1];
        end
    endgenerate



    always @(posedge clk) begin
        if(reset) begin
          state <= S_WAITING;
          outputPool_finishedVals <= -1;
          outputPool_writeEnable <= 0;
        end
        else case(state)
            S_WAITING: begin
                outputPool_writeEnable <= 0;
                if(startPulse) begin //start multiplication
                    //indexing scheme = [(i*WIDTH)-1 :
(i-1)*WIDTH ]
                    for(i_dotProd = 0; i_dotProd < `PARAM_I_I;
i_dotProd = i_dotProd + 1) begin

multiplicationResult[(i_dotProd*`PARAM_N_M_S1)+:`PARAM_N_M_S1]
<=
shift1($signed(inputDuplicates[(i_dotProd*`INPUT_WIDTH)+:`
```

```
INPUT_WIDTH]) *
$signed(incomingWeights[(i_dotProd*`WEIGHT_WIDTH)+:`
WEIGHT_WIDTH]));

//{{(`PARAM_N_M_S1-`INPUT_WIDTH){'b0}},
inputDuplicates[(i_dotProd*`INPUT_WIDTH)+:`INPUT_WIDTH]};//
shift1($signed(inputDuplicates[(i_dotProd*`INPUT_WIDTH)+:`
INPUT_WIDTH]) *
$signed(incomingWeights[(i_dotProd*`WEIGHT_WIDTH)+:`
WEIGHT_WIDTH]));
                        //(multiplicationResult)i = (input)i *
(weight)i
                    end
                    sumStartPulse <= 1;
                    state <= S_SUM;
                end
            end
            S_SUM: begin
                sumStartPulse <= 0;
                if (sumOutputReady) begin
                    // testing weight input by sending back
truncated weights

outputPool_finishedVals[(`INPUT_WIDTH*0)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput0})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*0)+:`WEIGHT_WIDTH]))
;

outputPool_finishedVals[(`INPUT_WIDTH*1)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput1})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*1)+:`WEIGHT_WIDTH]))
;

outputPool_finishedVals[(`INPUT_WIDTH*2)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput2})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*2)+:`WEIGHT_WIDTH]))
;

outputPool_finishedVals[(`INPUT_WIDTH*3)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput3})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*3)+:`WEIGHT_WIDTH]))
```

```
;

outputPool_finishedVals[(`INPUT_WIDTH*4)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput4})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*4)+:`WEIGHT_WIDTH]))
;

outputPool_finishedVals[(`INPUT_WIDTH*5)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput5})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*5)+:`WEIGHT_WIDTH]))
;

outputPool_finishedVals[(`INPUT_WIDTH*6)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput6})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*6)+:`WEIGHT_WIDTH]))
;

outputPool_finishedVals[(`INPUT_WIDTH*7)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput7})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*7)+:`WEIGHT_WIDTH]))
;

outputPool_finishedVals[(`INPUT_WIDTH*8)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput8})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*8)+:`WEIGHT_WIDTH]))
;

outputPool_finishedVals[(`INPUT_WIDTH*9)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput9})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*9)+:`WEIGHT_WIDTH]))
;

outputPool_finishedVals[(`INPUT_WIDTH*10)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput10})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*10)+:`WEIGHT_WIDTH])
);

outputPool_finishedVals[(`INPUT_WIDTH*11)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput11})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*11)+:`WEIGHT_WIDTH])
);
```

```
outputPool_finishedVals[(`INPUT_WIDTH*12)+:`INPUT_WIDTH] <= relu(shift2($signed({1'b0, sumOutput12})) + $signed(incomingThresholds[(`WEIGHT_WIDTH*12)+:`WEIGHT_WIDTH]));
outputPool_finishedVals[(`INPUT_WIDTH*13)+:`INPUT_WIDTH] <= relu(shift2($signed({1'b0, sumOutput13})) + $signed(incomingThresholds[(`WEIGHT_WIDTH*13)+:`WEIGHT_WIDTH]));
outputPool_finishedVals[(`INPUT_WIDTH*14)+:`INPUT_WIDTH] <= relu(shift2($signed({1'b0, sumOutput14})) + $signed(incomingThresholds[(`WEIGHT_WIDTH*14)+:`WEIGHT_WIDTH]));
outputPool_finishedVals[(`INPUT_WIDTH*15)+:`INPUT_WIDTH] <= relu(shift2($signed({1'b0, sumOutput15})) + $signed(incomingThresholds[(`WEIGHT_WIDTH*15)+:`WEIGHT_WIDTH]));
outputPool_finishedVals[(`INPUT_WIDTH*16)+:`INPUT_WIDTH] <= relu(shift2($signed({1'b0, sumOutput16})) + $signed(incomingThresholds[(`WEIGHT_WIDTH*16)+:`WEIGHT_WIDTH]));
outputPool_finishedVals[(`INPUT_WIDTH*17)+:`INPUT_WIDTH] <= relu(shift2($signed({1'b0, sumOutput17})) + $signed(incomingThresholds[(`WEIGHT_WIDTH*17)+:`WEIGHT_WIDTH]));
outputPool_finishedVals[(`INPUT_WIDTH*18)+:`INPUT_WIDTH] <= relu(shift2($signed({1'b0, sumOutput18})) + $signed(incomingThresholds[(`WEIGHT_WIDTH*18)+:`WEIGHT_WIDTH]));
outputPool_finishedVals[(`INPUT_WIDTH*19)+:`INPUT_WIDTH] <= relu(shift2($signed({1'b0, sumOutput19})) + $signed(incomingThresholds[(`WEIGHT_WIDTH*19)+:`WEIGHT_WIDTH]));

```

```
outputPool_finishedVals[(`INPUT_WIDTH*20)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput20})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*20)+:`WEIGHT_WIDTH])
);

outputPool_finishedVals[(`INPUT_WIDTH*21)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput21})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*21)+:`WEIGHT_WIDTH])
);

outputPool_finishedVals[(`INPUT_WIDTH*22)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput22})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*22)+:`WEIGHT_WIDTH])
);

outputPool_finishedVals[(`INPUT_WIDTH*23)+:`INPUT_WIDTH] <=
relu(shift2($signed({1'b0, sumOutput23})) +
$signed(incomingThresholds[(`WEIGHT_WIDTH*23)+:`WEIGHT_WIDTH])
);
                    outputPool_writeEnable <= 1;
                    state <= S_WAITING;
                end
            end
            default: state <= S_WAITING;
        endcase
    end
endmodule
```

Layer Controller

```
`include "master_params.vh"

module LayerController
    (
    input clk,
    input reset,

    input [`BTW_MAX_NET_DEPTH-1:0] commCTL_networkDepth,
    input commCTL_networkDepthReady,

    input [`BTW_MAX_PARALLELISM-1:0] parallelPool_parallelism,
    input parallelPool_parallelismReady,

    input [`BTW_MAX_NET_WIDTH-1:0] neuronIndicesPool_maxNeuronIndex,
    input neuronIndicesPool_maxNeuronIndexReady,

    input memCTL_weightsAreReady,
    input inputPool_inputsAreReady,

    input layerComputation_dataReady,

    output [`BTW_MAX_PARALLELISM-1:0] outputPool_dataWidth,
    output [`BTW_MAX_NET_WIDTH-1:0] outputPool_maxNeuronIndex,

    output inputPool_getFromOutputPool,

    output [`BTW_MAX_NET_DEPTH-1:0] memCTL_currentLayer,
    output [`BTW_MAX_NET_WIDTH-1:0] memCTL_currentNeuron,
    output [`BTW_MAX_PARALLELISM-1:0] memCTL_parallelism,
    output reg memCTL_getLayerDataRequest, //for layer params
    output reg memCTL_getNeuronDataRequest, //for neuron weights, needs to keep in mind the parallelism above

    output commCTL_isStateDone,
    output reg layerComputation_startPulse // "begin" pulse
    );

    // Parameters //

    localparam S_WAITING_INPUTS = 2'd0;
```

```
    localparam S_WAITING_LAYERCOMP = 2'd1;
    localparam S_DONE = 2'd2;

    //indices to architectureReady:
    localparam ARCH_PARALLELISM = 0;
    localparam ARCH_MAXNEURON = 1;

    // Registers //

    reg [1:0] state = S_WAITING_INPUTS; //"Processing State"

    reg networkDepthLoaded = 0;
    reg [`BTW_MAX_NET_DEPTH-1:0] networkDepth = {(`CLOG2(`BTW_MAX_NET_DEPTH)){1'b0}}; //NOTE: actually needs to be depth -1, since again, it's 0 indexed

    //microarchitectural ready states, all need to be 1 to enable the system
    reg [1:0] architectureReady = 2'b0;

    reg [`BTW_MAX_NET_DEPTH-1:0] currentLayer = {(`CLOG2(`BTW_MAX_NET_DEPTH)){1'b0}}; // set to 0
    reg [`BTW_MAX_NET_WIDTH-1:0] currentNeuron = {(`CLOG2(`BTW_MAX_NET_WIDTH)){1'b0}}; // set to 0

    reg [`BTW_MAX_NET_WIDTH-1:0] maxNeuronIndex; //"Total Neurons in Layer"
    reg [`BTW_MAX_PARALLELISM-1:0] parallelism; //"Parallel Increment (P)"

    // Wires //

    // We add parallelism because this should be true if we are calculating last neuron
    // currentNeuron is first neuron in set being calculated
    wire isLastNeuron = (currentNeuron + parallelism >= maxNeuronIndex) & architectureReady[ARCH_MAXNEURON]; //sent to inputPool
    wire isLastLayer = ((currentLayer == networkDepth-1) || networkDepth == 0)& networkDepthLoaded;

```

```
    // Assign statements //

    assign commCTL_isStateDone = state == S_DONE;

    assign inputPool_getFromOutputPool = isLastNeuron & layerComputation_dataReady & (~isLastLayer);
    assign outputPool_dataWidth = parallelism; //todo check if P valid here or?
    assign outputPool_maxNeuronIndex = maxNeuronIndex; //todo check if M valid here or?

    assign memCTL_currentLayer = currentLayer;
    assign memCTL_currentNeuron = currentNeuron;
    assign memCTL_parallelism = parallelism;

    // nextState Functions //

    //We've sent the requests and have our layer architecture, wait for inputs to be ready
    task doState_WaitingInputs;
    begin
        if(memCTL_weightsAreReady && inputPool_inputsAreReady) begin
            layerComputation_startPulse <= 1;
            state <= S_WAITING_LAYERCOMP;
        end

        //reset previous requests
        memCTL_getLayerDataRequest <= 0;
        memCTL_getNeuronDataRequest <= 0;
    end
    endtask

    //We've sent a request to the layer computation module, wait for it to finish calculating
    task doState_WaitingLayerComp;
    begin
        layerComputation_startPulse <= 0;
        if(layerComputation_dataReady) begin //if the computation is done
            if(isLastNeuron) begin
```

```
                if(isLastLayer) state <= S_DONE;
                else begin
                    //update layer
                    currentLayer <= currentLayer + 1;
                    currentNeuron <= {(`CLOG2(`BTW_MAX_NET_WIDTH)){1'b0}}; // set to 0

                    //request next layer architecture
                    memCTL_getLayerDataRequest <= 1; //is set to 0 in next state

                    //request weights for the new neuron-layer combo
                    memCTL_getNeuronDataRequest <= 1; //is set to 0 in next state
                    state <= S_WAITING_INPUTS; //wait for weights to refresh
                end

                //reset layer architecture flags
                architectureReady <= 2'b0;
            end
            else begin
                currentNeuron <= currentNeuron + parallelism;

                //request weights for the new neuron-layer combo
                memCTL_getNeuronDataRequest <= 1; //is set to 0 in next state
                state <= S_WAITING_INPUTS; //wait for weights to refresh
            end
        end
    end
    endtask

    // Sequential Logic //

    always @(posedge clk) begin
        if(reset) begin
          state <= S_WAITING_INPUTS;
```

```
          currentLayer <= 0;
          currentNeuron <= 0;
          architectureReady <= 2'b0;
          memCTL_getLayerDataRequest <= 0;
          memCTL_getNeuronDataRequest <= 0;
          layerComputation_startPulse <= 0;
          networkDepthLoaded <= 0;
        end
        else if(~networkDepthLoaded) begin //global parameter needs to be set
            if(commCTL_networkDepthReady) begin
                networkDepth <= commCTL_networkDepth;
                networkDepthLoaded <= 1; //never gets unset

                //send initial architecture (layer data) request
                memCTL_getLayerDataRequest <= 1;
                memCTL_getNeuronDataRequest <= 1;
            end
        end
        else if(!(&{architectureReady})) begin //if any bit of architectureReady is 0 (i.e. if the architecture is not ready)
            //reset previous request
            memCTL_getLayerDataRequest <= 0;
            memCTL_getNeuronDataRequest <= 0;

            //Load neuron indices if RDY
            if(neuronIndicesPool_maxNeuronIndexReady) begin
                maxNeuronIndex <= neuronIndicesPool_maxNeuronIndex;
                architectureReady[ARCH_MAXNEURON] <= 1;
            end

            //Load parallelism values if RDY
            if(parallelPool_parallelismReady) begin
                parallelism <= parallelPool_parallelism;
                architectureReady[ARCH_PARALLELISM] <= 1;
            end
        end
        else case(state)
            S_WAITING_INPUTS: doState_WaitingInputs;
```

```
            S_WAITING_LAYERCOMP: doState_WaitingLayerComp;
            S_DONE: ; //intentionally empty state, waits for
external reset to begin next computation
            default: state <= S_WAITING_INPUTS;
        endcase
    end
endmodule
```

Master Parameters

```
`ifndef MASTER_PARAMS_H
`define MASTER_PARAMS_H

`include "constant_funcs.vh"

// Basic parameters
`define NODES_PER_STREAM 9 //for communication module
// `define WRITE_CNTRL 15 // TODO: Width of address wire from comm module to
`define MAX_NET_WIDTH 24 // I
`define MAX_NET_DEPTH 4 // D
`define INPUT_WIDTH 16    // N
`define WEIGHT_WIDTH 24   // M
`define SHIFT_1 3 //S1
`define SHIFT_2 1 //S2
`define DEBUG_WIDTH 8     // E
`define MAX_DEBUG_MSGS 9  // E_max
`define BITS_PER_STREAM 80
`define DSP_SLICES 740 // Based on # DSP slices
// Same as PARAM_I_I_M1 `define MAX_WEIGHT_BITS_PER_STEP `MAX_NET_WIDTH*`MAX_NET_WIDTH*`WEIGHT_WIDTH+`MAX_NET_WIDTH // This is the logical max weights. With 24 width, 4 depth, we know the output of min. `MIN(`DSP_SLICES * `WEIGHT_WIDTH, `MAX_NET_WIDTH*`MAX_NET_WIDTH*`WEIGHT_WIDTH+`MAX_NET_WIDTH)
`define MAX_PARALLELISM `MAX_NET_WIDTH //theoretical max is $sqrt(`DSP_SLICES) = 27, but 24 now as we only do entire layers in a compute steps

// Bit widths
`define BTW_MAX_NET_WIDTH $clog2(`MAX_NET_WIDTH)
`define BTW_MAX_NET_DEPTH $clog2(`MAX_NET_DEPTH)
`define BTW_MAX_PARALLELISM $clog2(`MAX_PARALLELISM)
`define BTW_MAX_WEIGHTS_PER_STEP $clog2(`MAX_WEIGHT_BITS_PER_STEP)
`define BTW_BITS_PER_STREAM $clog2(`BITS_PER_STREAM)

// Derived Parameters
`define PARAM_I_M (`MAX_NET_WIDTH*`WEIGHT_WIDTH)   // I*M
`define PARAM_I_N (`MAX_NET_WIDTH*`INPUT_WIDTH)    // I*N
`define PARAM_I_D_N (`MAX_NET_WIDTH*`MAX_NET_DEPTH*`INPUT_WIDTH)    // I*N
```

```
`define PARAM_E_EM (`DEBUG_WIDTH*`MAX_DEBUG_MSGS)  // E*E_max
// Subtract 24 to account for header, parallelism, max_node data in first data packet
`define STREAM_WEIGHT_OVERLAP ((`PARAM_I_M1-(`BITS_PER_STREAM-(8+24+24))) % (`BITS_PER_STREAM-8) == 0 ? `BITS_PER_STREAM-8  : (`PARAM_I_M1-(`BITS_PER_STREAM-(8+24+24))) % (`BITS_PER_STREAM-8)) // Overlap into last packet stream for weight data
`define NODES_PER_STREAM_OVERLAP `STREAM_WEIGHT_OVERLAP % 8
`define STREAM_INPUT_OVERLAP (`PARAM_I_N % (`BITS_PER_STREAM-8) == 0) ? `BITS_PER_STREAM-8 : (`PARAM_I_N % (`BITS_PER_STREAM-8)) // Overlap into last packet stream for inputs data
`define STREAMS_PER_OUTPUTS (`CEIL(`PARAM_I_N, `BITS_PER_STREAM-8))
`define PACKETS_PER_STREAM (`BITS_PER_STREAM/8)
`define TOTAL_BITS_PER_STREAM (`PACKETS_PER_STREAM*10) // Bits per stream including start and stop bits
// Derived Parameters
`define PARAM_I_I (`MAX_NET_WIDTH * `MAX_NET_WIDTH) //interconnect of 2 layers
`define PARAM_I_I_N (`MAX_NET_WIDTH*`PARAM_I_N)
`define PARAM_I_M1 (`MAX_NET_WIDTH*(`WEIGHT_WIDTH+1))   // I*(M+1) := Weight/Threshold bits per node
`define PARAM_I_I_M (`MAX_NET_WIDTH*`PARAM_I_M)   // I^2*(M) := Weight bits per layer
`define PARAM_I_I_M1 (`MAX_NET_WIDTH*`PARAM_I_M1)   // I^2*(M+1) := Weight/Threshold bits per layer
`define PARAM_N_M (`INPUT_WIDTH +`WEIGHT_WIDTH) // N+M
`define PARAM_N_M_S1 (`PARAM_N_M - `SHIFT_1) // N+M-S1
`define PARAM_N_M_S1_I (`PARAM_N_M_S1 * `MAX_NET_WIDTH) // I(N+M-S1) //all the weights for one node
`define PARAM_N_M_S1_I_I (`PARAM_N_M_S1_I * `MAX_NET_WIDTH) // I(N+M-S1) //all the weights for one layer
`define PARAM_N_T (`INPUT_WIDTH*`NUM_MULTIPLIERS)


`define PARAM_N_M_LG_S1 (`PARAM_N_M + `CLOG2(`MAX_NET_WIDTH+1) - `SHIFT_1)
`define PARAM_N_M_LG_S1_S2 (`PARAM_N_M +
```

```
`CLOG2(`MAX_NET_WIDTH+1) - `SHIFT_1 - `SHIFT_2)

// BRAM write widths
`define BRAM_WIDTH `PARAM_I_M1 // For now we read in a single node's weights per step
//`define BRAM_BTW_ADDR $clog2(`MAX_NET_DEPTH*(`MAX_NET_WIDTH*`MAX_NET_WIDTH*`WEIGHT_WIDTH +`MAX_NET_WIDTH))
`define BRAM_BTW_ADDR $clog2(`MAX_NET_DEPTH*`MAX_NET_WIDTH)

`endif

```

Memory Controller

```
`timescale 1ns / 1ps

`include "master_params.vh"


// Current implementation reads and writes a single node's weights per timestep
// During writing phase, expects all nodes in a layer to have weights sent in-order
// TODO: Do we have to buffer write requests?
module memory_controller(
    input clk,
    input reset,
    input write_weights_rdy,
    input rd_layer_en,
    input rd_node_en,
    input [`PARAM_I_M1-1:0] write_layer_weights, // Note that we currently expect a single node's weights + cutoff per write.
    input [`BTW_MAX_NET_DEPTH-1:0] write_layer_id,
    input [`BTW_MAX_NET_DEPTH-1:0] read_layer_id,
    input [`BTW_MAX_NET_WIDTH-1:0] read_node_id,
    input [`BTW_MAX_PARALLELISM-1:0] read_parallelism,
    output reg [`PARAM_I_I_M-1:0] read_weights,
    output reg [`PARAM_I_M-1:0] read_thresholds,
    output reg read_weights_rdy);


    localparam READS_PER_REQ = `MAX_NET_WIDTH;// Layer controller will send for each computation step. Currently this is all nodes in a layer.
    localparam WRITES_PER_NODE = 1; // Port width set to write all of a node's data at a time
    localparam STATE_WAITING = 0; // We are not waiting for BRAM to finish reading. Inputs are monitored
    localparam STATE_READING = 1; // BRAM is reading into output buffer. All inputs ignored except reset
    localparam STATE_WRITING = 2;


    //wire bram_start_addr = read_layer_id * `PARAM_I_M1 +
```

```
read_node_id*`WEIGHT_WIDTH + read_node_id;
    wire [`BRAM_BTW_ADDR-1:0] bram_start_addr = read_layer_id * `MAX_NET_WIDTH + read_node_id;
    reg [`BRAM_BTW_ADDR-1:0] bram_cur_addr;
    reg [$clog2(READS_PER_REQ):0] bram_reads_left;
    wire [`BRAM_WIDTH-1:0] cur_rd_weights;

    reg [1:0] state;
    reg last_wr_layer;
    reg cur_wr_layer;
    reg [`BRAM_BTW_ADDR-1:0] bram_wr_offset;
    reg [$clog2(WRITES_PER_NODE):0] bram_writes_left;
    reg [`PARAM_I_M1-1:0] cur_wr_weights;
    wire dummy_bram_busy;

    wire bram_en = bram_writes_left > 0 || bram_reads_left > 0;

    blk_width24_depth4 bram_inst (
      .clka(clk),            // input wire clka
      .rsta(reset),          // input wire rsta
      .ena(bram_en),      // input wire ena
      .wea(bram_writes_left > 0),              // input wire [0 : 0] write enable
      .addra(bram_cur_addr), // input wire [14 : 0] addra
      .dina(cur_wr_weights[`BRAM_WIDTH-1:0]), // input wire [374 : 0] dina
      .douta(cur_rd_weights),// output wire [374 : 0] douta
      .rsta_busy(dummy_bram_busy)  // dummy reset busy. Assume we don't care about this for now
    );

    always @(posedge clk) begin
        if (reset) begin
            read_weights_rdy <= 0;
            state <= STATE_WAITING;
            cur_wr_layer <= 0;
            last_wr_layer <= 1;
            bram_writes_left <= 0;
            bram_reads_left <= 0;
            bram_cur_addr <= 0;
```

```
        end else begin
            case (state)
            STATE_WAITING: begin
                if (write_weights_rdy) begin
                    // Write layer weights `PARAM_I_M bits to BRAM
                    // Reset layer offset if we are writing nodes in a new layer
                    if (last_wr_layer != write_layer_id) begin
                        bram_wr_offset <= 0;
                        last_wr_layer <= write_layer_id;
                    end
                    cur_wr_layer <= write_layer_id;
                    cur_wr_weights <= write_layer_weights;
                    bram_writes_left <= WRITES_PER_NODE;
                    state <= STATE_WRITING;
                end else if (rd_node_en) begin
                    // Addresses BRAM using layer_id, node_id, and parallelism
                    bram_cur_addr <= bram_start_addr;
                    bram_reads_left <= READS_PER_REQ;
                    // Prefill output weight and thresholds to account for layers that do not read to fill the buffer
                    // NOTE: that this neuron id check only works because we do a full layer per computation step
                    if (read_node_id == 0) begin
                        read_weights <= 0;
                        read_thresholds <= 0;
                    end
                    state <= STATE_READING;
                end else begin
                    if (read_weights_rdy) read_weights_rdy <= 0;
                end
            end
            STATE_READING: begin
                // delay first read from bram by a cycle due to delay
                if (bram_reads_left != READS_PER_REQ) begin
                    if (bram_reads_left == 0) begin
                        state <= STATE_WAITING;
```

```
                        read_weights_rdy <= 1;
                    end
                    // least significant weight of every node is the cutoff
                    read_weights <= {cur_rd_weights[`BRAM_WIDTH-1:`WEIGHT_WIDTH], read_weights[`PARAM_I_I_M-1:`BRAM_WIDTH-`WEIGHT_WIDTH]};
                    read_thresholds <= {cur_rd_weights[`WEIGHT_WIDTH-1:0], read_thresholds[`PARAM_I_M-1:`WEIGHT_WIDTH]};
                    bram_cur_addr <= bram_cur_addr + 1;//`BRAM_WIDTH;
                end
                bram_reads_left <= bram_reads_left - 1;
            end
            STATE_WRITING: begin
                if (bram_writes_left == 1) state <= STATE_WAITING;
                cur_wr_weights <= cur_wr_weights >> `BRAM_WIDTH;
                bram_writes_left <= bram_writes_left - 1;
                bram_cur_addr <= bram_cur_addr + 1;//`BRAM_WIDTH;
            end
            default: ;
            endcase
        end
    end

endmodule
```

Pool

```
`timescale 1ns / 1ps

`include "master_params.vh"

// General pool module to support input pool and parallelism/max_node_id pools
// This can be split into two specialized pool as some functionality only exists for the former/latter
module pool #(DATA_WIDTH = `BTW_MAX_PARALLELISM) (
    input clk,
    input reset,
    input write_en, // Initial write enable
    input [DATA_WIDTH-1:0] write_data, // Initial write data
    input read_en, // read enable. asserts index is valid
    input [`BTW_MAX_NET_DEPTH-1:0] index, // write/read index
    output reg data_rdy,
    output reg [DATA_WIDTH-1:0] read_data // output data
    );

    reg [(`MAX_NET_WIDTH*DATA_WIDTH)-1:0] storage;
    always @(posedge clk) begin
        if (reset) begin
            storage <= 0;
            data_rdy <= 0;
        end else begin
            if (write_en) begin
                storage[index*DATA_WIDTH +: DATA_WIDTH] <= write_data;
             end else if (read_en) begin
                read_data <= storage[index*DATA_WIDTH +: DATA_WIDTH];
                data_rdy <= 1;
             end else if (data_rdy) begin
                data_rdy <= 0;
             end
        end
    end
endmodule

module input_pool(
    input clk,
```

```
    input reset,
    input comm_write_en, // Initial write enable
    input cntrl_pool_transfer, // Transfer output pool to input pool
    input [`PARAM_I_N-1:0] comm_data, // Initial write data
    input [`PARAM_I_N-1:0] output_pool_data, // Intermediate data to overwrite
    output reg [`PARAM_I_N-1:0] read_data, // output data
    output reg read_data_rdy
    );

    always @(posedge clk) begin
        if (reset) begin
            read_data <= 0;
            read_data_rdy <= 0;
        end else begin
            if (comm_write_en | cntrl_pool_transfer) begin
                read_data <= (cntrl_pool_transfer) ? output_pool_data : comm_data;
                read_data_rdy <= 1;
            end else if (read_data_rdy) read_data_rdy <= 0;
        end
    end
endmodule

module output_pool(
    input clk,
    input reset,
    input write_en,
    input [`BTW_MAX_PARALLELISM-1:0] parallelism,
    input [`BTW_MAX_NET_WIDTH-1:0] max_node_id,
    input [`PARAM_I_N-1:0] write_data,
    output reg [`PARAM_I_N-1:0] cur_outputs,
    output reg done_layer // Calculated by layer controller. Output here should matche for debugging
    );

    integer i;
    reg [`BTW_MAX_NET_WIDTH-1:0] cur_write_node;
    always @(posedge clk) begin
        if (reset) begin
```

```
            cur_outputs <= 1;
            cur_write_node <= 0;
            done_layer <= 0;
        end else if (write_en) begin
            //cur_outputs <= {{(`PARAM_I_N-24-8){1'b1}}, {3'b0, cur_write_node}, {3'b0, parallelism}, {3'b0, max_node_id}, 8'b00};
            //done_layer <= 1;
            // partial layer output write to cur_outputs based on data and parallelism
            for (i = 0; i < `MAX_NET_WIDTH; i = i + 1) begin
                if (i >= cur_write_node && i < cur_write_node + parallelism && i <= max_node_id) begin
                    // Update new node outputs
                    cur_outputs[i*`INPUT_WIDTH +: `INPUT_WIDTH] <= write_data[`INPUT_WIDTH*(i-cur_write_node)+:`INPUT_WIDTH];
                end else if (i >= cur_write_node) begin
                    // Zero future node outputs. This handles zeroing outputs that are never written
                     cur_outputs[i*`INPUT_WIDTH +: `INPUT_WIDTH] <= 'b0;
                end
            end
            if (cur_write_node + parallelism <= max_node_id) begin
                cur_write_node <= cur_write_node + parallelism;
            end else begin
                cur_write_node <= 0;
                done_layer <= 1;
            end
        end else if (done_layer) begin
            done_layer <= 0;
        end
    end
endmodule
```

Tree Adder

```
`include "master_params.vh"

/*

SPECIFICATION:

assert inputReady
next cycle data has to be ready to latch (must be held only for that cycle)
when outputReady is asserted the data will be available on dataOut
(same clock cycle) and will remain until inputReady is asserted again

reset does not need to be called before this, and there is no mechanism
to clear garbage data because it is not needed in the use-case

*/

module TreeAdder24
    (
    input clk,
    input reset,
    input [`PARAM_N_M_S1_I-1:0] data,
    input inputReady,
    output outputReady,
    output reg [`PARAM_N_M_S1+3:0] dataOut
    );

    //Tree-layout adder, log delay
    //sumA layer 0
    reg signed [`PARAM_N_M_S1:0] sumA00;
    reg signed [`PARAM_N_M_S1:0] sumA01;
    reg signed [`PARAM_N_M_S1:0] sumA02;
    reg signed [`PARAM_N_M_S1:0] sumA03;
    reg signed [`PARAM_N_M_S1:0] sumA04;
    reg signed [`PARAM_N_M_S1:0] sumA05;
    reg signed [`PARAM_N_M_S1:0] sumA06;
    reg signed [`PARAM_N_M_S1:0] sumA07;

```

```
    //sumA layer 1
    reg signed [`PARAM_N_M_S1+1:0] sumA10;
    reg signed [`PARAM_N_M_S1+1:0] sumA11;
    reg signed [`PARAM_N_M_S1+1:0] sumA12;
    reg signed [`PARAM_N_M_S1+1:0] sumA13;

    //sumA layer 2
    reg signed [`PARAM_N_M_S1+2:0] sumA20;
    reg signed [`PARAM_N_M_S1+2:0] sumA21;

    //sumA layer 3
    reg signed [`PARAM_N_M_S1+3:0] sumA3;


    //sumB latch 0
    reg signed [`PARAM_N_M_S1:0] sumB00;
    reg signed [`PARAM_N_M_S1:0] sumB01;
    reg signed [`PARAM_N_M_S1:0] sumB02;
    reg signed [`PARAM_N_M_S1:0] sumB03;

    //sumB latch 1
    reg signed [`PARAM_N_M_S1:0] sumB10;
    reg signed [`PARAM_N_M_S1:0] sumB11;
    reg signed [`PARAM_N_M_S1:0] sumB12;
    reg signed [`PARAM_N_M_S1:0] sumB13;


    //sumB layer 1
    reg signed [`PARAM_N_M_S1+1:0] sumB20;
    reg signed [`PARAM_N_M_S1+1:0] sumB21;

    //sumB layer 2
    reg signed [`PARAM_N_M_S1+2:0] sumB3;

    reg active = 0;

    reg [3:0] cycleCount = 3'b0;
    assign outputReady = cycleCount == 3'd5;

    always @(posedge clk) begin
        if(reset | inputReady) begin
```

```
            cycleCount <= 3'b0;
            active <= inputReady;
        end
        else if(active) begin
            cycleCount <= cycleCount + 1;
            if(cycleCount == 3'd4) active <= 0; //will be 5 next round, we are done

            //sumA data latch
            sumA00 <= data[`PARAM_N_M_S1 *(0) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(1) +: `PARAM_N_M_S1];
            sumA01 <= data[`PARAM_N_M_S1 *(2) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(3) +: `PARAM_N_M_S1];
            sumA02 <= data[`PARAM_N_M_S1 *(4) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(5) +: `PARAM_N_M_S1];
            sumA03 <= data[`PARAM_N_M_S1 *(6) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(7) +: `PARAM_N_M_S1];
            sumA04 <= data[`PARAM_N_M_S1 *(8) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(9) +: `PARAM_N_M_S1];
            sumA05 <= data[`PARAM_N_M_S1 *(10) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(11) +: `PARAM_N_M_S1];
            sumA06 <= data[`PARAM_N_M_S1 *(12) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(13) +: `PARAM_N_M_S1];
            sumA07 <= data[`PARAM_N_M_S1 *(14) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(15) +: `PARAM_N_M_S1];

            //sumB data latch
            sumB00 <= data[`PARAM_N_M_S1 *(16) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(17) +: `PARAM_N_M_S1];
            sumB01 <= data[`PARAM_N_M_S1 *(18) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(19) +: `PARAM_N_M_S1];
            sumB02 <= data[`PARAM_N_M_S1 *(20) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(21) +: `PARAM_N_M_S1];
            sumB03 <= data[`PARAM_N_M_S1 *(22) +: `PARAM_N_M_S1] + data[`PARAM_N_M_S1 *(23) +: `PARAM_N_M_S1];

            //sumB layer 1 ( latch to keep pipeline uniform )
            sumB10 <= sumB00;
            sumB11 <= sumB01;
            sumB12 <= sumB02;
            sumB13 <= sumB03;
```

```

            //sumA layer 1
            sumA10 <= sumA00+sumA01;
            sumA11 <= sumA02+sumA03;
            sumA12 <= sumA04+sumA05;
            sumA13 <= sumA06+sumA07;

            //sumA layer 2
            sumA20 <= sumA10+sumA11;
            sumA21 <= sumA12+sumA13;

            //sumA layer 3
            sumA3 <= sumA20+sumA21;

            //sumB layer 2
            sumB20 <= sumB10 + sumB11;
            sumB21 <= sumB12 + sumB13;

            //sumB layer 3
            sumB3 <= sumB20 + sumB21;

            //final Sum (layer 4)
            dataOut <= sumA3 + sumB3;
        end
    end
endmodule

```

UART

```
`include "master_params.vh"

// Modifications from base:
// Parameterized uart interface to support variable packet stream size

//115200 * 16 oversampling = 1843200 samples / sec
//100000000 hz / N = 1843200 hz
//N ~=~ 54 (54.253)
// TODO: test max baud
module uart_clkdiv //to account for non blocking assignment timing
    (input clk,
    input reset,
    output clk_out);

    localparam TARGET = 'd54;
    reg [8:0] count; //9 bits needed to store values up to and including TARGET

    assign clk_out = (count == 9'b0);
    always @(posedge clk) begin
        if (reset) begin
            count <= 0;
        end else begin
            if(count == TARGET-1) count <= 9'b0;
            else count <= count + 1;
        end
    end
endmodule

// 10 packets per stream. 8 bits per packet
`define BITS_PER_STREAM 80
//2ms hold constraint:
//115200 hz ^-1 = 5.425 E -7 sec / cycle
//5.425 E -7 * NUM_CYCLES = .002 sec
//VERIFy_SAMPLES = NUM_CYCLES ~=~ 230 ( = 230.4 )  (8 bits needed to store)

module uart_rx
    (input clk,
```

```
    input uart_clk,
    input reset,
    input signal,
    input verifyStopBit, //for debugging purposes, negative logic
    output reg [`BITS_PER_STREAM-1:0] data_out,
    output reg ready = 0); //THIS IS A PULSE

    localparam VERIFY_SAMPLES = 'd230;

    localparam S_VERIFY_HIGH = 0;
    localparam S_WAITING = 1;
    localparam S_LISTENING = 2;
    localparam S_VERIFYSTOP = 3;

    reg [1:0] state;
    reg [7:0] tempData;
    reg [7:0] timingCounter;
    reg [2:0] bitCounter; //counts how many bits sampled
    reg [3:0] cycleCounter; //counts how many cycles have passed
    reg [`BITS_PER_STREAM-1:0] data;


    always @(posedge clk) begin
        if (reset) begin
            state <= 2'b0;
            tempData <= 8'b0;
            timingCounter <= 12'b0;
            bitCounter <= 3'b0;
            cycleCounter <= 3'b0;
            data <= 'b0;
            data_out <= -1;
            ready <= 0;
        end else if (uart_clk) begin
            case(state)
                S_VERIFY_HIGH: begin
                    ready <= 0;
                    if(signal) begin
                        if(timingCounter == VERIFY_SAMPLES) begin //will be 307 on next cycle
```

```
                            state <= S_WAITING; //we have verified we are not in the middle of data transmission, go to wait state
                            if(cycleCounter >= 'd9) begin
                               cycleCounter <= 3'b0;
                            end
                            timingCounter <= 12'b0;
                        end
                        else timingCounter <= timingCounter + 1;
                    end
                    else timingCounter <= 12'b0;
                end
                S_WAITING: begin
                    if(~signal) begin //signal has gone low
                        if(timingCounter == 'd7) begin //will be 8 the next cycle, start sampling
                            state <= S_LISTENING;
                            timingCounter <= 'd0;
                        end
                        else timingCounter <= timingCounter + 1;
                    end
                    else timingCounter <= 12'b0; //likely noise, keep waiting for data transmission start
                end
                S_LISTENING: begin
                    if(timingCounter == 12'd15) begin
                        timingCounter <= 'd0;
                        tempData <= {signal,tempData[7:1]};
                        if(bitCounter == 'd7) begin // this cycle is the 8th bit, we're done
                            state <= S_VERIFYSTOP;
                            bitCounter <= 3'b0;
                        end
                        else bitCounter <= bitCounter + 1;
                    end
                    else timingCounter <= timingCounter + 1;
                end
                S_VERIFYSTOP: begin
                    if(timingCounter == 12'd15) begin
```

```
                        timingCounter <= 12'b0;
                        if(verifyStopBit | signal) begin//stop bit found, data valid (or debug enabled)
                            data <= {tempData,data[`BITS_PER_STREAM-1:8]};
                            tempData <= 0;
                            if(cycleCounter == 'd9) begin
                                ready <= 1; //this is the 10th byte, send ready pulse
                                data_out <= {tempData,data[`BITS_PER_STREAM-1:8]};
                                state <= S_VERIFY_HIGH;
                            end
                            else state <= S_WAITING;
                            cycleCounter <= cycleCounter + 1;
                        end
                        else state <= S_WAITING;
                    end
                    else timingCounter <= timingCounter +1;
                end
                default: state <= S_VERIFY_HIGH;
            endcase
        end else if (ready == 1) ready <= 0;
    end
endmodule

module uart_tx(
    input clk,
    input reset,
    input en,
    input [`TOTAL_BITS_PER_STREAM-1:0] start_data,
    output reg xmit_data,
    output reg done);

    localparam SEND_RATE = 16;
    localparam DELAY = 100; // UART cycles to wait before sending data. Possible bug in delay from computer send inputs to waiting for outputs
    localparam N = 868;

    reg signed [`TOTAL_BITS_PER_STREAM-1:0] shift_data;
```

```
    reg [`CLOG2(`TOTAL_BITS_PER_STREAM)-1:0] cntr;
    reg [9:0] downsampler; // xmit_clk is the 16x sample clock from receiver. Need to send slower.
    reg [9:0] delay;
    reg [9:0] trans_cntr; // 100 Mhz * n = 115200 hz => n = 100000000/115200 = 868

    always @(posedge clk) begin
        if (reset) begin
            xmit_data <= 1;
            cntr <= `TOTAL_BITS_PER_STREAM;
            shift_data <= start_data;
            done <= 0;
            downsampler <= 0;//SEND_RATE;
            delay <= DELAY;
            trans_cntr <= N-1;
        end else if (trans_cntr == 0) begin//(xmit_clk && cntr > 0) begin
            trans_cntr <= N-1;
            if (delay == 0) begin
                //if (downsampler == 0) begin
                    //downsampler <= SEND_RATE;
                    cntr <= cntr - 1;
                    xmit_data <= shift_data[0];
                    shift_data <= shift_data >>> 1;
                    if (cntr == 1) done <= 1;
                // end else begin
                   // downsampler <= downsampler - 1;
                // end
            end else
                 delay <= delay - 1;
        end else if (done) begin
           done <= 0;
        end else if (en) begin
           trans_cntr <= trans_cntr - 1;
        end
    end

endmodule
```

Appendix B: Python Visualization Code

(Note: Solutions to the 6.034 lab assignment have been intentionally omitted. These are not key files in interfacing the FPGA neural network with the "training.py" script that is given as part of the 6.034 starting lab material, which has been modified to suit this project.)

Main Script
(app.py)

```python
from comm import pc_uart
from ctypes import c_uint8
import random
import sys
import time
import labinterface

TESTPRINT = False
def TP(*args):
    if TESTPRINT:
        TP(*args)
# Matched with communication_module.v for MODE values expected in packet headers
class stream_types(): #ENUM
    WEIGHTS = 0
    INPUTS = 1
    OUTPUT = 2
    DEBUG = 3

MAX_NET_WIDTH = 24
MAX_NET_DEPTH = 4
NUM_LAYERS= 1
CRC_EN = 0
WEIGHT_WIDTH = 24
INPUT_WIDTH = 16
NUM_NODES = [15]
parallelism = [15]

# ensure this is 0 padded in actual implementation
test_weights = [NUM_NODES[i] * [(MAX_NET_WIDTH+1)*[1]] for i in range(len(NUM_NODES))]
max_node = [NUM_NODES[i]-1 for i in range(len(NUM_NODES))]
# 24 nodes, each has weight array [cutoff=1, weight_0 = 1, ..., weight_23 = 1]
test_inputs = 12*[0x20,0x20] #range(15)
uart_ifc = None

def make_header(data_type, header_data):
    header = (data_type | (CRC_EN << 2) | (header_data << 3)).to_bytes(1, byteorder='little')
    return header
```

```python

def get_size(data_type):
    if data_type == stream_types.WEIGHTS:
        return 3
    if data_type == stream_types.INPUTS:
        return 2
    if data_type == stream_types.DEBUG:
        return 1

def ceil_div(a, b):
    return -(-a // b)

def create_send_stream(data_type, header_data, data_bytes):
    global uart_ifc
    header_bytes = make_header(data_type, header_data)
    packet_stream = header_bytes + data_bytes
    uart_ifc.send_stream(packet_stream)

# def send_data_array(data_type, header_data, data_arr):
#     data_i = 0
#     # check data being sent is a multiple of packet size
#     packets_per_num = get_size(data_type)
#     increment = 9 // packets_per_num
#     overlap = 9 % packets_per_num
#
#     while data_i + increment <= len(data_arr):
#         data_bytes = [i.to_bytes(get_size(data_type), byteorder='little') for i in data_arr[data_i:data_i+increment]]
#         if overlap > 0:
#             overlap_start = data_i + increment
#             overlap_end = data_i + increment + overlap
#             data_bytes += [i.to_bytes(get_size(data_type), byteorder='little') for i in
#                 data_arr[overlap_start:overlap_end]]
#             data_i += overlap
#         TP(str(data_bytes))
#         create_send_stream(data_type, header_data, data_bytes)
#         data_i += increment
```

```
#     # handle leftover
#     leftover = len(data_arr) - data_i - 1
#     if leftover > 0:
#         TP(leftover)
#         data = data_arr[data_i:data_i+leftover] + (9-leftover)*[0]
#         TP(str(data))
#         create_send_stream(data_type, header_data, data)

def send_data_array(data_type, header_data, data_arr):
    # check data being sent is a multiple of packet size
    data_bytes = b''.join([i.to_bytes(get_size(data_type), byteorder='little', signed=True) for i in data_arr])
    TP("DATA (" +str(len(data_arr)) + ") = " + str(data_arr))
    TP("DATA_BYTES (" + str(len(data_bytes)) + ") = " + str(data_bytes))
    overlap = len(data_bytes) % 9
    i = 0
    while i + 9 < len(data_bytes):
        stream_data_bytes = data_bytes[i:i+9]
        TP("STREAM_DATA_BYTES = " + str(stream_data_bytes))
        create_send_stream(data_type, header_data, stream_data_bytes)
        i += 9
    # handle leftover
    if overlap > 0:
        TP("overlap = " + str(overlap))
        data = data_bytes[-overlap:] + (9-overlap)*b'\x00'
        TP(str(data))
        create_send_stream(data_type, header_data, data)
def get_outputs():
    global uart_ifc
    outputs = b''
    while len(outputs) < ceil_div(MAX_NET_WIDTH*INPUT_WIDTH, 8):
        streams = uart_ifc.rec_stream()
        if streams:
            for stream in [streams[i:i+10] for i in range(0, len(streams), 10)]:
                TP(str(stream[1:].hex()))
                outputs += stream[1:]
```

```python
        else:
            return None
    #grouped_bytes = [[outputs[i+1:i+2] + outputs[i:i+1]] for i in range(0,len(outputs),2)]
    grouped_bytes = [[outputs[i:i+1] + outputs[i+1:i+2]] for i in range(0,len(outputs),2)]
    TP('[{}]'.format(', '.join(x[0].hex() for x in grouped_bytes)))
    outputs = [int.from_bytes(b''.join(g), byteorder='little', signed=True) for g in grouped_bytes]
    return outputs[0:24]
def main():
    global uart_ifc
    if len(sys.argv) < 2:
        TP("Please pass part of port name to connect to. E.g. COM7")
        return -1
    uart_ifc = pc_uart()
    if uart_ifc.setup(str(sys.argv[1])) == False:
        TP("Exiting....")
        return -1
    # WEIGHTS
    try:
        TP("Generating weights")
        #while(True):
        #    uart_ifc.send_stream(b''.join([int(0).to_bytes(1, byteorder='little', signed=True)] + 9*[int(0).to_bytes(1, byteorder='big')]))
        #    time.sleep(0.5)
        ## generate weight array
        weight_data = [NUM_LAYERS*[]]
        for i in range(NUM_LAYERS):
            weight_data[i] = [parallelism[0], max_node[0]]
            for j in range(NUM_NODES[i]):
                weight_data[i] += test_weights[i][j]
        TP("Sending weights" + str(test_weights))
        for layer_id in range(NUM_LAYERS):
            send_data_array(stream_types.WEIGHTS, layer_id, weight_data[layer_id])
        TP("Sending inputs")
        send_data_array(stream_types.INPUTS, 0, test_inputs)
```

```python
        TP("Sending debug send_done packet")
        send_data_array(stream_types.DEBUG, 0, [0])
        TP("Waiting for outputs...")
        # Wait for outputs
        outputs = get_outputs()
        TP("outputs = " + str(outputs))
    finally:
        TP("Closing serial port")
        uart_ifc.close()
def verify_TestNet():
    #inputs = [12*[1<<5,1<<5]]
    inputs = [[0,0]]
    #weights = 2*[7 * [(24)*[1]] + (24-7)* [24*[0]] for i in range(1)]
    weights = [[[-27, 40, 0], [-27, 40, 0], [-27, 40, 0]], [[24, 24, 24], [24, 24, 24], [0, 0, 0]], [[40, 40, 0], [0, 0, 0], [0, 0, 0]]]
    #thresh = 2*[24 * [1] + (24-7) * [24*[0]] for i in range(1)]
    thresh = [[17, 17, 17], [-6, -6, 0], [-11, 0, 0]]
    depth = len(thresh)
    inputs = [l + (24-len(l))*[0] for l in inputs]
    weights = [l + (24-len(l))*[0] for n in weights for l in n]
    thresh = [l + (24-len(l))*[0] for l in thresh]
    newNet = labinterface.TestNet(inputs, weights, thresh, depth)
    TP("Running on FPGA")
    run(inputs, weights, thresh, len(weights[0]), depth-1)
    newNet.printFstLayer()
    TP('-----------------')
    newNet.printResults()
def run_rand():
    global uart_ifc
    if len(sys.argv) < 2:
        TP("Please pass part of port name to connect to. E.g. COM7")
        return -1
    uart_ifc = pc_uart()
    if uart_ifc.setup(str(sys.argv[1])) == False:
        TP("Exiting....")
```

```
        return -1
    # WEIGHTS
    try:
        newNet = labinterface.TestNet()
        TP("Generating weights")
        #while(True):
        #    uart_ifc.send_stream(b''.join([int(0).to_bytes(1, byteorder='little', signed=True)] + 9*[int(0).to_bytes(1, byteorder='big')]))
        #    time.sleep(0.5)
        ## generate weight array
        # reverse weight to have correct ordering relative to lsb
        #weight_data.reverse()
        for layer_id in range(NUM_LAYERS):
            combined = [num_nodes, num_nodes-1] # parallelism and max node
            for i,l in enumerate(newNet.weights[layer_id]):
                combined += [newNet.thresholds[layer_id][i]] + l
                TP(str(combined))
            TP("Sending weights: " + str(combined))
            send_data_array(stream_types.WEIGHTS, layer_id, combined)
        TP("Sending inputs")
        newNet.printFstLayer()
        send_data_array(stream_types.INPUTS, 0, newNet.values[0])
        TP("Sending debug send_done packet")
        send_data_array(stream_types.DEBUG, 0, [0])
        TP("Waiting for outputs...")
        # Wait for outputs
        outputs = get_outputs()
        TP("outputs = " + str(outputs))
        TP("Actual Results: ")
        newNet.printResults()
        TP("Results match? " + newNet.checkResults(outputs))

    finally:
        TP("Closing serial port")
        uart_ifc.close()
```

```python
def run(inputs, weights, thresholds, num_nodes, depth, port_name = "COM8"):
    single_layer = depth == 1
    max_node = [num_nodes-1 for i in range(depth)]
        # WEIGHTS
    if uart_ifc is None:
        setup()
    try:
        TP("Generating weights")
        # reverse weight to have correct ordering relative to lsb
        #weight_data.reverse()
        for layer_id in range(depth):
            combined = [num_nodes, num_nodes-1] # parallelism and max node
            for i,l in enumerate(weights[layer_id]):
                combined += [thresholds[layer_id][i]] + l + (24-len(l)) * [0]
            TP("Sending weights: " + str(combined))
            send_data_array(stream_types.WEIGHTS, layer_id, combined)
        TP("Sending inputs")
        inputs = inputs + (24-len(inputs)) * [0]
        TP(str(inputs))
        send_data_array(stream_types.INPUTS, 0, inputs)
        TP("Sending debug send_done packet")
        send_data_array(stream_types.DEBUG, 0, [0])
        TP("Waiting for outputs...")
        # Wait for outputs
        outputs = []
        if not single_layer:
            for i in range(MAX_NET_DEPTH):
                outputs.append(get_outputs())
        else:
            outputs.append(get_outputs())
        print("outputs (final first) = " + str(outputs))
        return outputs

    except:
        TP("Closing serial port")
        uart_ifc.close()
```

```python
def setup(port_name = "COM8"):
    global uart_ifc
    if port_name == "":
        TP("Please pass part of port name to connect to. E.g. COM7")
        return -1
    uart_ifc = pc_uart()
    if uart_ifc.setup(port_name) == False:
        TP("Exiting....")
        return -1
if __name__ == "__main__":
    setup()
    main()
    #t = labinterface.TestNet.run_working(0)
    #t[0].printResults()
    #verify_TestNet()
    #run_rand()
```

Communications
(comm.py)

```python
import serial.tools.list_ports
import sys
import time
from time import sleep

TESTPRINT = False
def TP(*args):
    if TESTPRINT:
        TP(*args)
#Version 2.7 or Above?
if sys.version_info[0] >2:
    version3 = True
    kwargs = {'newline':''}
else:
    version3 = False
    kwargs = {}
def get_usb_port(port_name):
    usb_port = list(serial.tools.list_ports.grep("USB-Serial Controller"))
    if len(usb_port) == 1:
        print("Automatically found USB-Serial Controller: {}".format(usb_port[0].description))
        return usb_port[0].device
    else:
        ports = list(serial.tools.list_ports.comports())
        port_dict = {i:[ports[i],ports[i].vid] for i in range(len(ports))}
        usb_id=None
        for p in port_dict:
            print("{}:   {} (Vendor ID: {})".format(p,port_dict[p][0],port_dict[p][1]))
            if port_name in port_dict[p][0]:
                print("\tAbove Port Matches Argument: "+str(port_name))
                usb_id = p
        if usb_id== None:
            return False
        else:
            print("USB-Serial Controller: Device {}".format(p))
            return port_dict[usb_id][0].device
class pc_uart:
```

```
    def __init__(self):
        self._serial = None
    def setup(self, port_name):
        s = get_usb_port(port_name)
        if s:
            self._serial = serial.Serial(port = s,
                baudrate=115200,
                parity=serial.PARITY_NONE,
                stopbits=serial.STOPBITS_ONE,
                bytesize=serial.EIGHTBITS,
                timeout=None)
            print(self._serial)
            print("Serial Connected!")
            if self._serial.isOpen():
                 print(self._serial.name + ' is open...')
            return True
        else:
           print("USB-Serial Controller Not Found")
           return False
    def rec_stream(self):
        data = []
        data = self._serial.read(10)
        TP("stream: " + str(data.hex()))
        if len(data) > 0:
            return data
        else:
            print("ERR: Received invalid packet stream")
            return None
        #sleep(3.0)
    def send_stream(self, stream):
        assert len(stream) == 10
        TP("STREAM_BYTES: " + str(stream))
        time.sleep(0.005) # hold constraint
        TP("# packets sent: " + str(self._serial.write(stream)))
    def close(self):
        self._serial.close()

```

Lab Interface
(labinterface.py)

```python
import lab6, random, app, sys, time

def NNGet(net):
    """NeuralNet object -> lists"""
    neuronNameToPos = {}
    posToNeuronName = {}
    neurons = [[]]
    currLayerI = 0
    currNodeI = 0
    maxNodeI = 0

    #get neuron list
    currLayer = neurons[0]
    for neuron in net.topological_sort(True):
        for parentNode in net.get_incoming_neighbors(neuron):
            if parentNode in neurons[currLayerI]: #next layer found
                currLayerI += 1
                maxNodeI = max(maxNodeI,currNodeI)
                currNodeI = 0
                neurons.append([])
                break
        if neuron != -1:
            neurons[currLayerI].append(neuron)
            neuronNameToPos[neuron] = (currLayerI,currNodeI)
            posToNeuronName[(currLayerI,currNodeI)] = neuron
            currNodeI += 1

    #for layer in neurons: #zero pad to max size
    weights = [[[0]*maxNodeI for _ in range(maxNodeI)] for _ in range(currLayerI)]
    thresholds = [[0]*maxNodeI for _ in range(currLayerI)]

    #Reference (copied from below)
    #values[1][2] -> layer, neuron
    #weights[1][2][3] -> layer, neuron, prevNeuron
    #thresholds[1][2] -> layer, neuron

    for wire in net.get_wires():
        n0 = wire.startNode
        n1_layerI, n1_nodeI = neuronNameToPos[wire.endNode]
```

```
        n1_layerI -= 1 #shift array data left by 1, since there is nothing before the 0th layer, and this indexes by prev layer
        if n0 == -1: thresholds[n1_layerI][n1_nodeI] = -wire.weight
        else:
            n0_layerI, n0_nodeI = neuronNameToPos[n0]
            assert n0_layerI == n1_layerI #no +1 because we did a -1 above to n1_layerI
            weights[n1_layerI][n1_nodeI][n0_nodeI] = wire.weight

    return (neurons,weights,thresholds,maxNodeI,currLayerI, posToNeuronName)
def FPGAFwdProp(net,inputs,runOnFPGA):
    """ return (final output, dict {neuron -> output}
    this function has to stall until the final outputs are ready"""

    if runOnFPGA is False: return lab6.forward_prop(net,inputs)
    else:
        neurons,weights,thresholds, numNodes, numLayers, posToNeuronName = NNGet(net)

        inputLayer = [int(i) for i in inputs.values()]
        DEBUG = True
        if DEBUG:
            print("n",neurons)
            print("w",weights)
            print("t",thresholds)
            print("i",inputLayer)
            print("params",numNodes,numLayers)
            print("pos -> neuron name",posToNeuronName)

        #_ = input("CALL FPGA HERE") #todo remove this
        # in case we only have single layer
        print(inputLayer)
        netOutputs = [inputLayer]
        port_name = "COM8"
```

```
        if True:
            cur_inputs = inputLayer
            for layer in range(numLayers):
                outputs = app.run(cur_inputs, weights[layer:], thresholds[layer:], numNodes, 1, port_name)
                if type(outputs) == int:
                    break
                netOutputs.append(outputs[0])
                cur_inputs = outputs[0]
                #time.sleep(0.05)
        else:
            netOutputs = app.run(inputLayer, weights, thresholds, numNodes, numLayers-1, port_name)#FPGA_FUNCTIONCALL() #todo, returns a 1d list of raw neuron outputs
        if type(netOutputs) == int:
            print("FPGA forward propogation returned error: " + str(netOutputs))
            sys.exit("Please kill and restart after fixing the error")
        #make 2d list
        print(str(netOutputs))
        #netOutputs = [netOutputs[i:i+numNodes] for i in range(0, len(netOutputs), numNodes)]
        print(str(netOutputs))
        print(str(posToNeuronName))
        #make into dict
        values = {}#{neuron:output}
        for layerI in range(len(netOutputs)):
            for neuronI in range(len(netOutputs[layerI])):
                if (layerI,neuronI) in posToNeuronName:
                    name = posToNeuronName[(layerI,neuronI)]
                    values[name] = netOutputs[layerI][neuronI]

        return (values[net.get_output_neuron()],values)
class working_tests():
    inputs = [
        [12*[1<<5,1<<5]],
    ]
    weights = [
        2*[5 * [(24)*[1]] for i in range(1)],
```

```
    ]
    thresholds = [
        2*[5 * [1] for i in range(1)],
    ]

class TestNet():
    layerWidth = 24
    inputBitWidth = 16
    weightBitWidth = 24
    shift1 = 3
    shift2 = 1
    def __init__(self, inputs = None, weights = None, thresholds = None, depth=2):
        self.layerDepth = depth
        #create values and initialize first layer: values[1][2] -> layer, neuron
        if inputs is None:
            self.values = [[self.getRandomSigned(self.inputBitWidth) for _ in range(self.layerWidth)]]
        else:
            self.values = inputs
        #create weights: weights[1][2][3] -> layer, neuron, prevNeuron
        if weights is None:
            self.weights = [[[self.getRandomSigned(self.weightBitWidth) for _ in range(self.layerWidth)] for _ in range(self.layerWidth)] for _ in range(self.layerDepth)]
        else:
            self.weights = weights
        #create thresholds for each neuron: thresholds[1][2] -> layer, neuron
        if thresholds is None:
            self.thresholds = [[self.getRandomSigned(self.weightBitWidth) for _ in range(self.layerWidth)] for _ in range(self.layerDepth)]
        else:
            self.thresholds = thresholds
        print(self.weights)
        print(self.values)
```

```
        print(self.thresholds)

        for i in range(1,self.layerDepth):
            currLayer = []
            self.values.append(currLayer)
            for currNeuron in range(self.layerWidth):
                currLayer.append(max(0,(sum([(self.weights[i][currNeuron][prevN] * self.values[i-1][prevN]) >> self.shift1 for prevN in range(self.layerWidth)]) >> self.shift2)+self.thresholds[i][currNeuron]))
    # Returns instances of this class containing either a specified or all working test cases
    def run_working(self, test_num = -1):
        # specific test case chosen
        new_inst = []
        if test_num >= 0:
            new_inst = [TestNet(inputs = working_tests.inputs[test_num], weights = working_tests.weights[test_num],
                                  thresholds = working_tests.thresholds[test_num])]
        else:
            for i in range(len(working_tests.inputs)):
                new_inst.append(TestNet(working_tests.inputs[i], working_tests.weights[i], working_tests.thresholds[i]))
        return new_inst

    @staticmethod
    def getRandomSigned(n): return random.randint(-(2**(n-1)),(2**(n-1))-1)

    def getFstLayer(self): return self.values[0]

    def checkResults(self,actualResult): return actualResult == self.values[-1]

    def printLayer(self,n): print("[" + ",".join([str(i) for i in self.values[n]]) + "]")
    def printFstLayer(self): self.printLayer(0)
```

Training

(training.py, modified version of 6.034 starting code)

```python
#!/usr/bin/env python3
# MIT 6.034 Lab 6: Neural Nets
# This file originally written by Joel Gustafson and Kenny Friedman

from __future__ import print_function
from sys import argv
from random import random, randint, shuffle
from time import sleep
from matplotlib import pyplot, cm
import numpy
# import _tkinter
from lab6 import *
from neural_net_api import *
import labinterface
from math import ceil
AMP = 200 #positive data amplitude
STATIC_W = 30 #static weight used
ACCURACY = -20
MAX_STEPS = 50

colormap = cm.get_cmap("plasma")
def multi_accuracy(desired_outputs, actual_outputs):
    pairs = []
    actual_outputs.sort()
    for d_o in sorted(desired_outputs):
        a_o = actual_outputs.pop(0)
        while a_o[0] != d_o[0]:
            a_o = actual_outputs.pop(0)
        pairs.append((d_o, a_o))
    total = sum(accuracy(a[1], b[1]) for a, b in pairs)
    return float(total) / len(pairs)
# Multi-point forward propagation
def multi_forward_prop(net, resolution=1, runOnFPGA = False):
    outputs = []
    data = []
    for i in range(resolution * 5):
        y = float(i) / resolution
        line = []
        for j in range(resolution * 5):
            x = float(j) / resolution
```

```
            #result = forward_prop(net,{'x': x, 'y': y})[0]
            result = labinterface.FPGAFwdProp(net,{'x': x, 'y': y},runOnFPGA)[0]
            if i % resolution == 0 and j % resolution == 0:
                outputs.append(((x, y), result))
            line.append(result)
        data.append(line)
    data = numpy.array(data)
    pyplot.pcolor(data, cmap=colormap)
    # pyplot.pcolor(data)
    pyplot.pause(0.0001)
    sleep(0.05)
    return (sorted(outputs), data)
# Backward propagation
def multi_update_weights(net, desired_outputs, width=5, height=5):
    shuffle(desired_outputs)
    for desired_output in desired_outputs:
        input_values = {'x': desired_output[0][0], 'y': desired_output[0][1]}
        #neuron_outputs = forward_prop(net, input_values)[1]
        neuron_outputs = labinterface.FPGAFwdProp(net,input_values,False)[1]
        net = update_weights(net, input_values, desired_output[1], neuron_outputs)
        #TP([w.weight for w in net.get_wires()])
    return net
def multi_back_prop(net, desired_outputs, resolution=1):
    actual_outputs = multi_forward_prop(net, resolution)[0]
    c = 0
    current_accuracy = multi_accuracy(desired_outputs, actual_outputs)
    TP("ii",c, current_accuracy)
    while current_accuracy < ACCURACY:
        net = multi_update_weights(net, desired_outputs)
        actual_outputs = multi_forward_prop(net, resolution)[0]
        c += 1
        current_accuracy = multi_accuracy(desired_outputs, actual_outputs)
        TP("ii",c, current_accuracy)
```

```python
        if c > MAX_STEPS: break
    return net
# Define neural nets
def get_small_nn(w=None):
    if w is None:
        f = lambda: STATIC_W # 10 * (0.5 - random())
        w = [f() for n in range(9)]
    return NeuralNet(['x', 'y', -1], ['A','B','C']) \
        .join('x', 'A', w[0]).join('x', 'B', w[1]) \
        .join('y', 'A', w[2]).join('y', 'B', w[3]) \
        .join('A', 'C', w[4]).join('B', 'C', w[5]) \
        .join(-1, 'A', w[6]).join(-1, 'B', w[7]).join(-1, 'C', w[8]) \
        .join('C', NeuralNet.OUT)
def get_medium_nn(w=None):
    if w is None:
        f = lambda: STATIC_W #10 * (0.5 - random())
        w = [f() for n in range(20)]
    return NeuralNet(['x', 'y', -1], list('ABCDEF')) \
        .join('x', 'A', w[0]).join('x', 'B', w[1]).join('y', 'A', w[2]) \
        .join('y', 'B', w[3]).join('y', 'C', w[4]).join('x', 'C', w[5]) \
        .join(-1, 'A', w[6]).join(-1, 'B', w[7]).join(-1, 'C', w[8]) \
        .join('A', 'E', w[9]).join('A', 'D', w[10]).join('B', 'D', w[11]) \
        .join(-1, 'D', w[12]).join(-1, 'E', w[13]).join(-1, 'F', w[14]) \
        .join('B', 'E', w[15]).join('C', 'E', w[16]).join('C', 'D', w[17]) \
        .join('D', 'F', w[18]).join('E', 'F', w[19]).join('F', NeuralNet.OUT)
def get_large_nn():
    w = lambda: STATIC_W #10*(0.5-random())
    nn = NeuralNet(['x', 'y', -1], list('ABCDEFGHIJKLMNOPQRSTUVWXYZ'))
    #first layer: A-J (10 neurons)
    for n1 in 'ABCDEFGHIJ':
        nn.join('x', n1, w()).join('y', n1, w())
        #second layer: K-T (10 neurons)
```

```
        for n2 in 'KLMNOPQRST':
            nn.join(n1, n2, w())
    #third layer: U-Y (5 neurons)
    for n3 in 'UVWXY':
        for n2 in 'KLMNOPQRST':
            nn.join(n2, n3, w())
        #final layer: Z (1 neuron)
        nn.join(n3, 'Z', w())
    #define Z as output neuron
    nn.join('Z', NeuralNet.OUT)

    return nn
nets = {'small': get_small_nn, 'medium': get_medium_nn, 'large': get_large_nn}
# Define data sets
# horizontal
# - - - - -
# - - - - -
# - - - - -
# + + + + +
# + + + + +
horizontal = sorted([((0,0),AMP),((0,1),AMP),((0,2),AMP),((0,3),0),((0,4),0),((1,0),AMP),
((1,1),AMP),((1,2),AMP),((1,3),0),((1,4),0),((2,0),AMP),((2,1),AMP),
((2,2),AMP),((2,3),0),((2,4),0),((3,0),AMP),((3,1),AMP),((3,2),AMP),
((3,3),0),((3,4),0),((4,0),AMP),((4,1),AMP),((4,2),AMP),((4,3),0),((4,4),0)])
# diagonal
# + + + + -
# + + + - -
# + + - - -
# + - - - -
# - - - - -
diagonal = sorted([((0,0),0),((0,1),AMP),((0,2),AMP),((0,3),AMP),((0,4),1
```

```
),((1,0),0),
((1,1),0),((1,2),AMP),((1,3),AMP),((1,4),AMP),((2,0),0),((2,1),0),
((2,2),0),((2,3),AMP),((2,4),AMP),((3,0),0),((3,1),0),((3,2),0),
((3,3),0),((3,4),AMP),((4,0),0),((4,1),0),((4,2),0),((4,3),0),((4,4),0)])
# stripe
# - - - - +
# - - - + -
# - - + - -
# - + - - -
# + - - - -
stripe = sorted([((0,0),AMP),((0,1),0),((0,2),0),((0,3),0),((0,4),0),((1,0),0),
((1,1),AMP),((1,2),0),((1,3),0),((1,4),0),((2,0),0),((2,1),0),
((2,2),AMP),((2,3),0),((2,4),0),((3,0),0),((3,1),0),((3,2),0),
((3,3),AMP),((3,4),0),((4,0),0),((4,1),0),((4,2),0),((4,3),0),((4,4),AMP)])
# checkerboard
# - -   + +
# - -   + +
#
# + +   - -
# + +   - -
checkerboard = sorted([((0,0),AMP),((1,0),AMP),((0,1),AMP),((1,1),AMP),((3,3),AMP),((4,3),AMP),
((3,4),AMP),((4,4),AMP),((0,3),0),((1,3),0),((0,4),0),
((1,4),0),((3,0),0),((3,1),0),((4,0),0),((4,1),0)])
# letterL
# + -
```

```
# + -
# + -
# + - - - -
# - + + + +
letterL = sorted([((0,0),0),((1,0),AMP),((2,0),AMP),((3,0),AMP),((4,0),AMP),((0,1),AMP),
                  ((1,1),0),((2,1),0),((3,1),0),((4,1),0),((0,2),AMP),((0,3),AMP),
                  ((0,4),AMP),((1,2),0),((1,3),0),((1,4),0)])
# moat
# - - - - -
# -       -
# -   +   -
# -       -
# - - - - -
moat = sorted([((0,0),0),((0,1),0),((0,2),0),((0,3),0),((0,4),0),((1,4),0),
               ((2,4),0),((3,4),0),((4,4),0),((4,3),0),((4,2),0),((4,1),0),
               ((4,0),0),((3,0),0),((2,0),0),((1,0),0),((2,2),AMP)])
training_data = {'horizontal': horizontal, 'diagonal': diagonal, 'stripe': stripe,
                 'checkerboard': checkerboard, 'letterL': letterL, 'moat': moat}
# Main function for training and heatmap
def start_training(data=None, net=None, resolution=None, fullscreen=False):
    if data == None:
        print('defaulting to diagonal training dataset')
        train = diagonal
    elif data in training_data:
        train = training_data[data]
    else:
        print('training dataset not found, defaulting to diagonal')
        train = diagonal
```

```python
    if net == None:
        print('defaulting to medium net')
        net_fn = get_medium_nn
    elif net in nets:
        net_fn = nets[net]
    else:
        print('net not found, defaulting to medium net')
        net_fn = get_medium_nn
    if resolution == None:
        print('defaulting to resolution of 1')
        resolution = 1
    else:
        try:
            resolution = int(resolution)
            assert resolution > 0
        except Exception:
            print('invalid resolution, defaulting to 1')
            resolution = 1
    pyplot.ion
    pyplot.show()
    if fullscreen:
        time.sleep(1)
    nn = net_fn()
    #print('\nInitial neural net:\n', nn)
    #print('\nIter, Accuracy:')
    try: nn = multi_back_prop(nn, train, resolution)
    except _tkinter.TclError as e:
        print('\nException caught: ', e, '\n')
        Athena_ssh_error_message = ("If you are running this on Athena "
            + "over ssh, try sshing again using the -X flag, which allows "
            + "Athena to display GUI windows on your local desktop.     "
            + "If you want to see the original stack trace instead of this "
            + "error, find the line in training.py that raises this "
            + "RuntimeError and replace it with 'raise e'.")
        raise RuntimeError(Athena_ssh_error_message)
    pyplot.ioff()
```

```python
    print('\nTrained neural net:\n', nn)

    #trained NN, now we make it into integers
    nn.integerize()
    print(nn)

    data = multi_forward_prop(nn, resolution, True)[1]
    pyplot.clf()
    pc = pyplot.pcolor(data, cmap=colormap)
    pyplot.colorbar(pc)
    pyplot.show()
if __name__ == "__main__":
    train = "diagonal"
    net = "medium"
    resolution = 1
    if '-data' in argv:
        train = argv[argv.index('-data') + 1]
    else:
        print('defaulting to diagonal training dataset')
    if '-net' in argv:
        net = argv[argv.index('-net') + 1]
    else:
        print('defaulting to medium net')
    if '-resolution' in argv:
        resolution = argv[argv.index('-resolution') + 1]
    else:
        print('defaulting to resolution of 1')
    start_training(train, net, resolution)
```