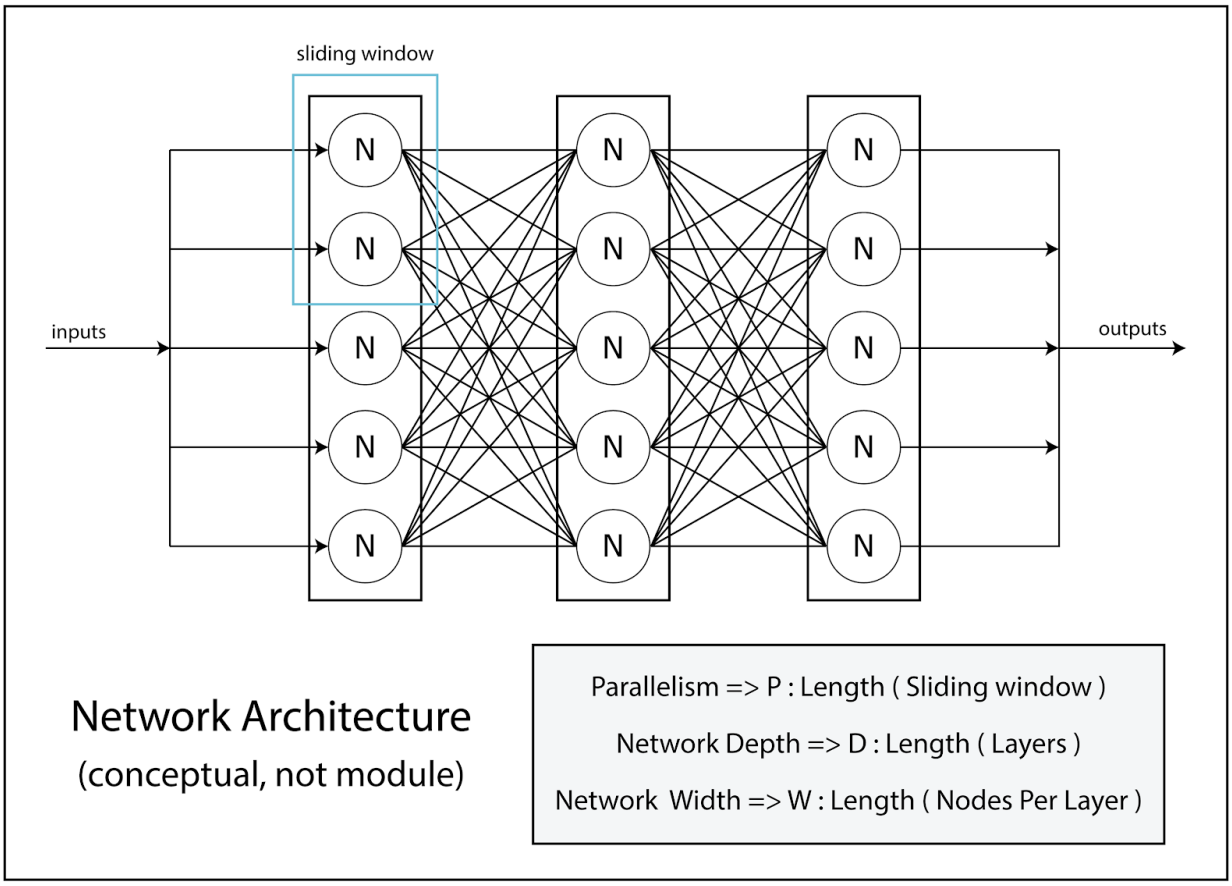
Sebastian Bartlett, Josh Noel
Professors Hom, Steinmeyer
MIT 6.111, Fall 2018

FPGA Neural Network



1. Abstract

This project aims to implement a neural network layer 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. Our core goal is to interface this structure with a computer via USB-UART, where it will be used to compute the propagation values for a larger artificial neural network. A secondary core goal is then using this structure as a matrix multiplication accelerator. This requires no design changes, as the computation of a single layer can be formulated as a matrix multiplication. Possible challenges for this project include determining appropriate network depth, width, and topology to adhere to FPGA computation, memory, and bandwidth constraints. Our primary stretch goal is to train the network to play retro video games such as those designed for the Nintendo Entertainment System (1985). This may be achieved by interfacing the FPGA with a game system emulator on a host CPU. The emulator will output the game-state feature vector to the FPGA, which will then use the neural network FSM to drive controller outputs. Layer weights would be trained using a custom-programmed FPGA-accurate neural net

simulator on a computer. Even further stretch goals include training the network to play more than one game, exploring different network topologies enabled by runtime-selectable weight masks, introducing one convolutional layer, or utilizing the matrix multiply accelerator as a graphics coprocessor.

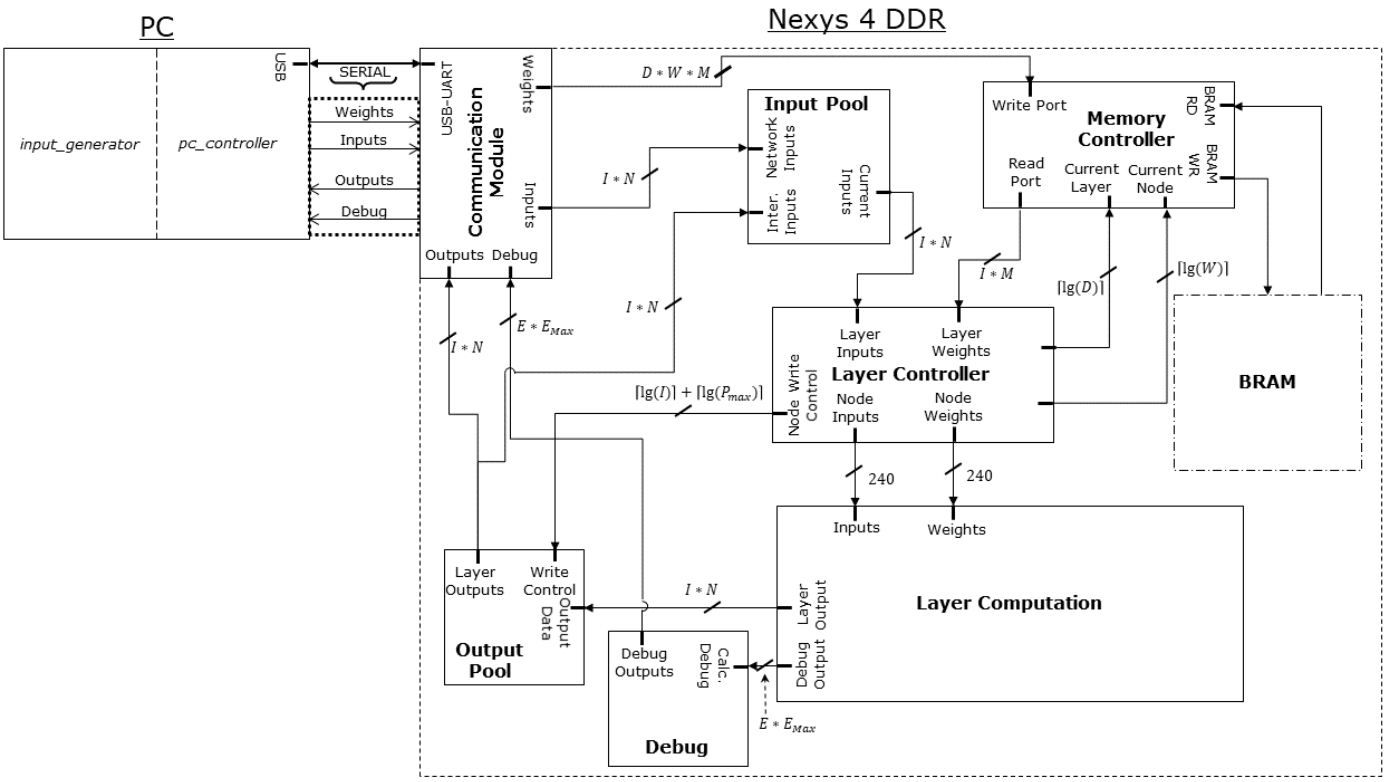
2. Timeline

Given that the initial block diagram for the project is complete, our next step is to begin implementation. We plan to divide the work for all modules excluding the Layer Controller. Modules such as the Communication Controller and Input / Output Pools can be created separately as we have collaborated on the interfaces of each. While the interface for the Layer Controller is known, it is the most complex module in the project. As such we plan to implement this module collaboratively. Specifically, we aim to finish implementing the basic functionality of this module by the end of next week. The other modules can be implemented independently, which we aim to functionally complete on November 16. Once all modules are completed we will begin the testing phase as described in the Testing section. We expect debugging to be intensive, and are allotting 2 weeks for this (ending on November 30). Once we have a working implementation, we will interface the network structure with the host computer. This will involve modifying the *pc_controller* used for testing, and will entail training of the larger host-driven neural network structure. The remaining time will therefore be dedicated to achieving this functionality.

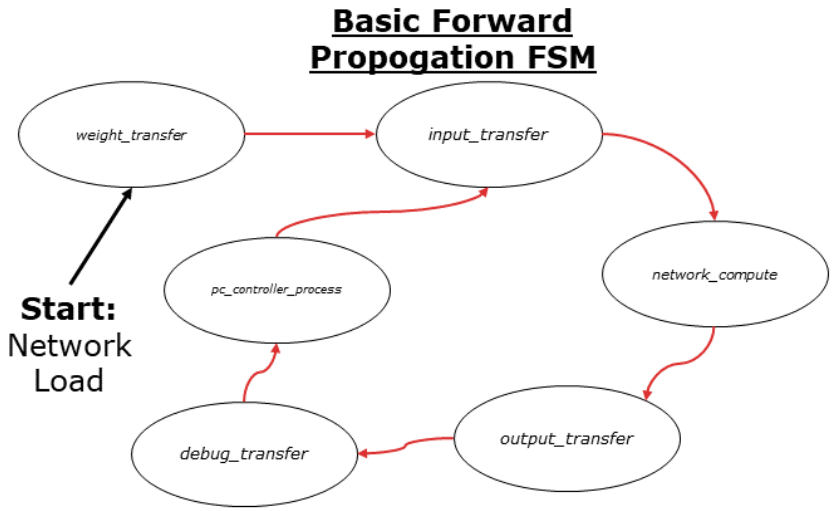
3. Symbol Key

T: Maximum number of multiplication operations per layer calculation step
M: Bit width of weights
N: Bit width of inputs
E: Width of debug message. E_{max} : Maximum number of debug messages
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.

4. Implementation



4.1. State Machine



The high level graphical description of the state machine can be seen in figure above. Specifically, the mechanism for performing forward propagation as follows.

We begin in the *weight_transfer* communication state. In this state the weights for the entire neural network 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 will be sent, such as the parallelism "P", of each layer. As described in the constraints section, there is enough block RAM on the FPGA to store all weights and network characteristics.

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. If any error is detected during this computation, such as overflow or underflow due to shifting, debug information will be sent to the debug module. This information will include relevant data such as the inputs, weights, and outputs for which the error was detected. When all node outputs are in the output pool, the layer controller signals a transition to the *output_transfer* state.

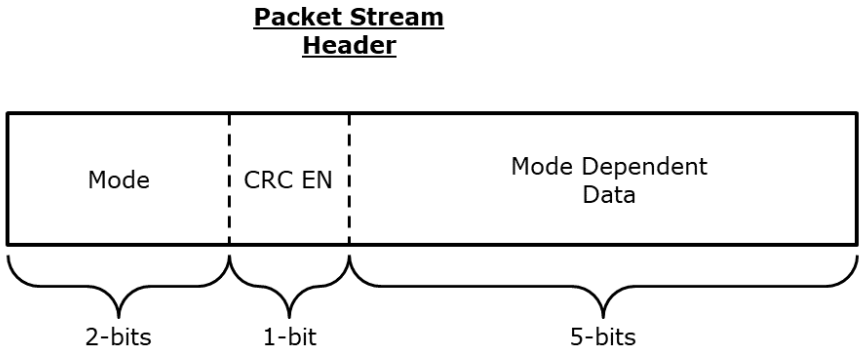
In the *output_transfer* state, the communication module reads the layer's outputs from the output pool, and sends them to the PC as described in the protocol section. Once this is done any debug info is transferred in the *debug_transfer* state. If no errors are found, a debug packet will be sent indicating successful completion. Upon reception of this packet the *pc_controller* can begin using the outputs received. This calculation loop will be performed for each layer in the larger artificial neural network.

4.2. PC-FPGA Communication Protocol

In order to perform inference on the FPGA we must have the ability to transfer data between the Host PC and FPGA. Specifically, we will have a process running on a computer that feeds weights and inputs to the FPGA. This process will be referred to as the *pc_controller*.

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.

The protocol itself is defined as follows. All data sent will be sent as a packet stream. A stream is defined to be 1 header packet and 9 data packets. Header packets are always 8 bits. The first 2 bits specify the mode for the following data packets. The next bit is whether CRC data for the stream should be expected immediately after the stream ends. The final 5 bits are then dependent on the mode.



Mode 1 is weight transfer mode, which indicates the data packets are weight packets. The mode dependent header data is the layer id of the weights. Mode 2 is input/output transfer mode, indicating input/output data packets. 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.

Data packets vary in size depending on the data being sent, however all data will be sent in 8-bit segments as this is the maximum width supported by the COM port on our laptops. Therefore, a data packet can be sent across multiple serial bytes. We have M-bit weights, so weight packets will be M bits with *2*ceil(M/8)* bits added for start and stop bits. We have N-bit inputs/outputs, so input/output packets will be N bits with *2*ceil(N/8)* start and stop bits. Debug data packets contain any data associated with the error code in the header. A debug data packet must have width determined by $E = max(M,N)$ in order to support transfer of errors associated with both weights and inputs. As we are communicating over UART, M and N may not be perfect multiples of the 8 data bits sent between start,stop bit pairs. We therefore subdivide any M or N bit value over *ceil([M,N]/8)* bytes. Data is transferred LSB first, so the last byte sent will be padded with zeroes if M or N is not a multiple of 8.

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 the receiver after every packet stream from the sender. Note that we will also light an LED on the FPGA 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.

4.3. Communication Module

This module will implement the communication model described above. The input will be the TXD port (C4) of the FT2232 USB-UART bridge. The output is the RXD port (D4). This module will operate at 128Kbaud as discussed in the USB-UART constraints section. The implementation will be similar to Lab 2B and Lab 5C, however the decoding will be as described

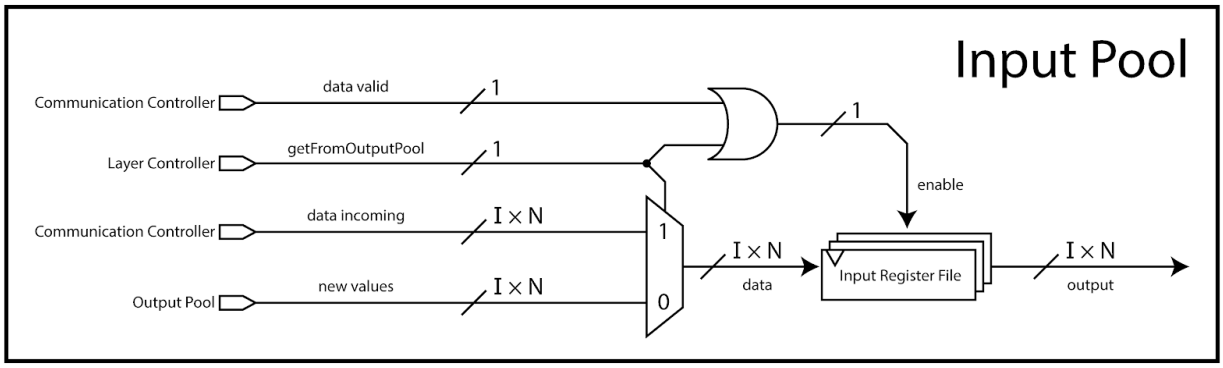
in the protocol depending on the header received. This includes detecting headers and switching the decoding procedure accordingly. The decoding procedure involves packing input and weight bits spread across multiple transmitted bytes, as well as routing the output to the correct module (the memory interface for weights and the input pool for inputs).

This module will also handle the sending of output data from the FPGA to the computer once every layer is done calculating. These outputs will be read from the output pool. Once all outputs are sent debug messages are sent as described in the communication protocol. The final debug message must indicate either success or failure.

4.4. Memory Controller Module

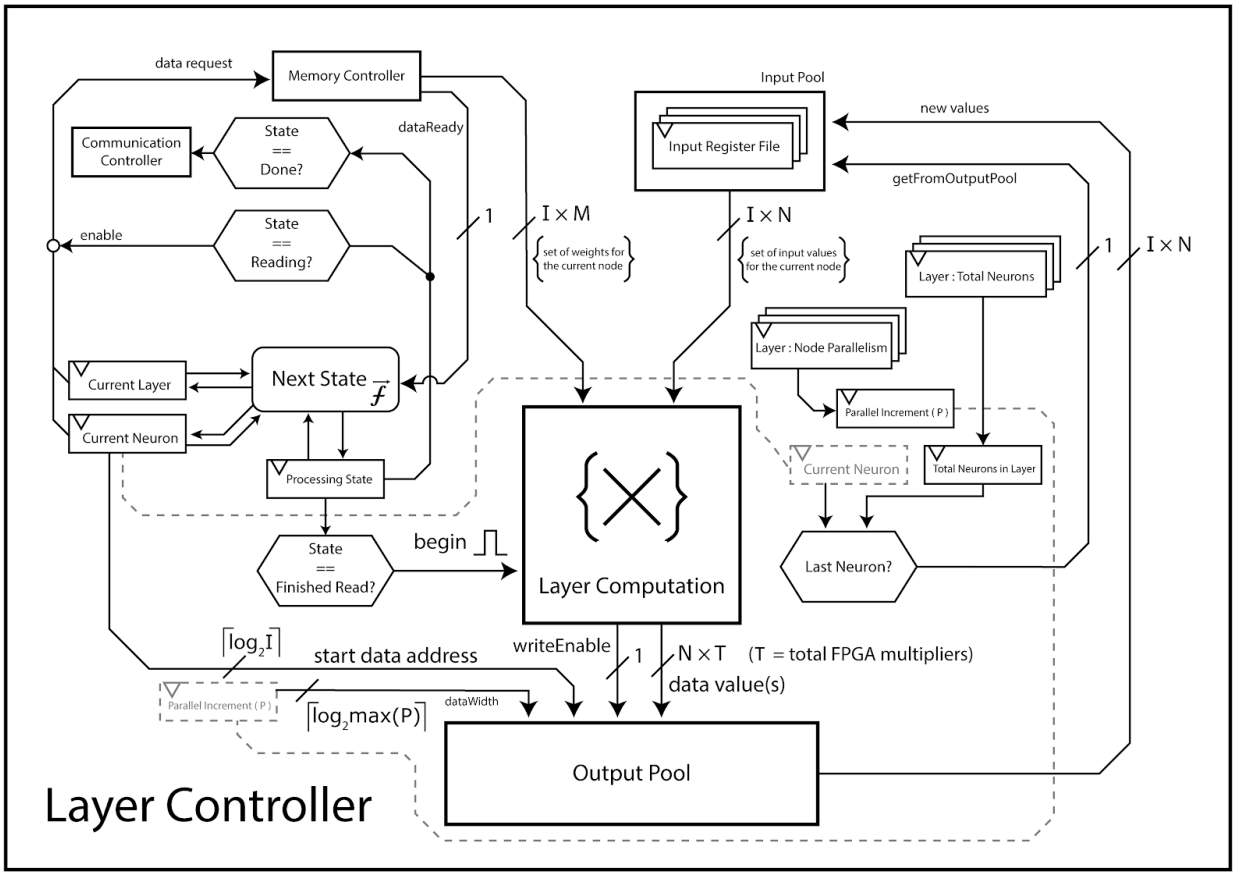
Our main stretch goal is to interface with a game system emulator, which is what we currently base our memory bounds on. As shown in section 6, we estimate our maximum required memory for weights to be about 2,000Kbits. The Nexys 4 DDR has over 4,000 Kbits of fast block RAM. Under our current specification this memory interface will present a read-write interface to block RAM and not DRAM. All read requests will come from the layer controller in order to load weights, and all write requests will come from the communication module as weights are received from the PC. From the reference manual, block RAM has one cycle of read latency which will be accounted for in the layer controller as it reads weights. If any use of DRAM is needed in the future, all requests will be routed through this interface.

4.5. Input Pool Module



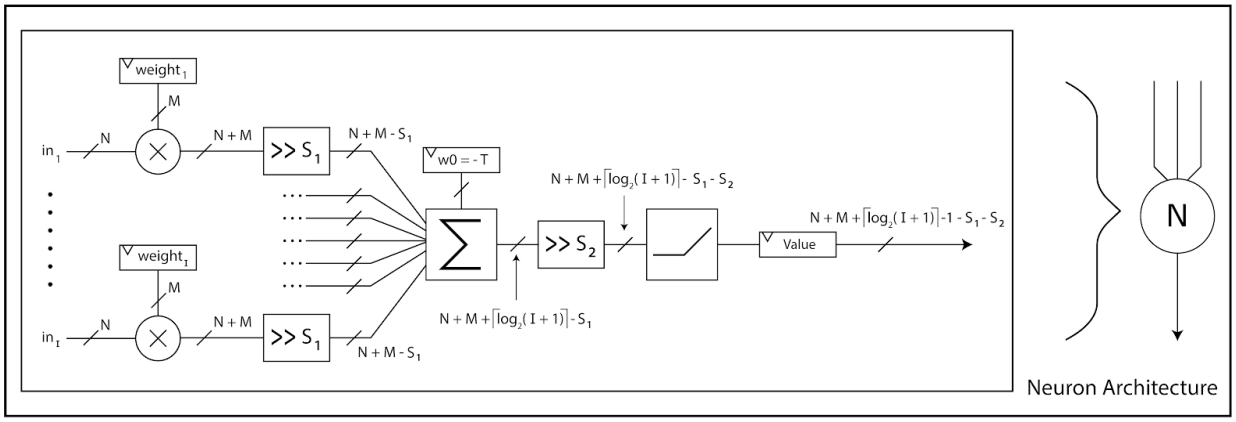
The Input Pool module encapsulates the input data buffer that is used to calculate values in the Layer Computation module. The write inputs to this register file come from the Output Pool and Communication Controller modules, given that the buffer can store initial values received from the host computer 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.

4.6. Layer Controller Module



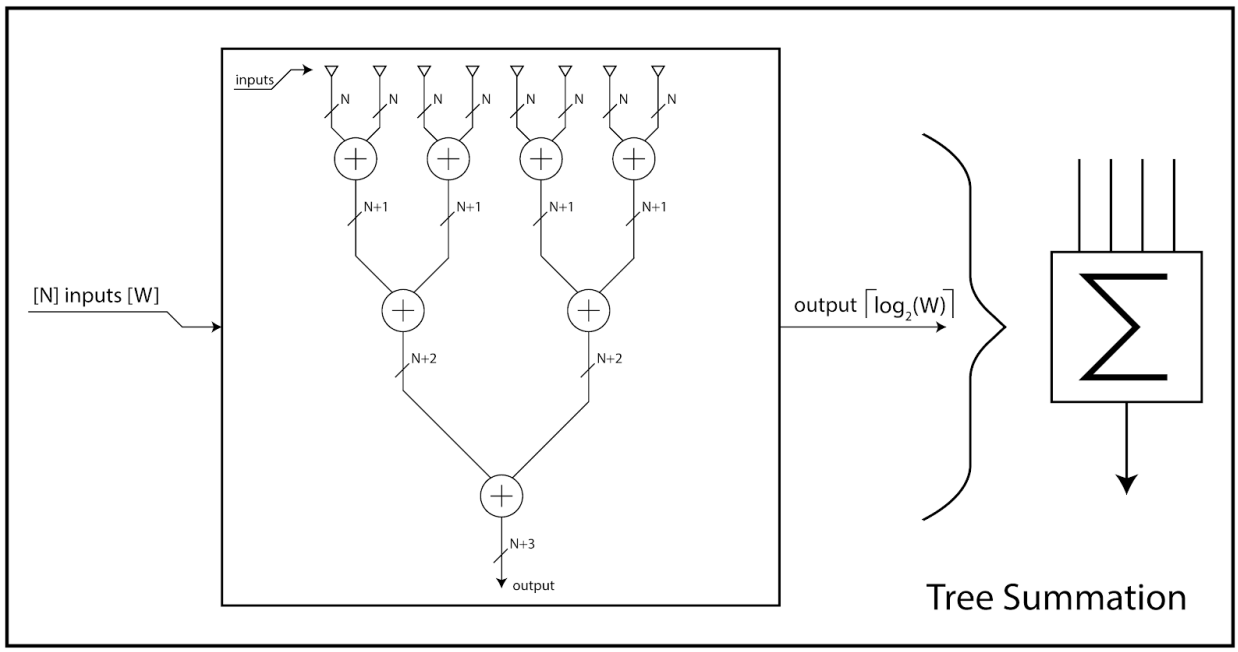
The Layer Controller forms the central processing unit of the neural network implementation. It is responsible for generating the primary control signals that are sent to the Communication Controller, Memory Controller, Layer Computation, Input Pool, and Output Pool modules. This module will keep track of the current neurons being computed using the initial current neuron index as well as the parallel increment value “P”. Upon the start of a processing cycle, the current layer and neuron are passed to the Memory Controller. Once the Memory Controller loads all respective weights to its weight buffer, and the Input Pool asserts the neuron layer inputs are stable, these values get sent to the Layer Computation module for processing. After this processing is complete, the current neuron value will be incremented by P, and a new data request will be sent to the Memory Controller. This procedure will occur for all neurons in the active layer. Once the layer is complete, the current layer register will increment, and the Layer Controller will signal to the Input Pool that it should overwrite its values to those from the Output Pool. Once the final layer is computed, the Communication Controller will be signalled to send the values from the Output Pool to the *pc_controller*.

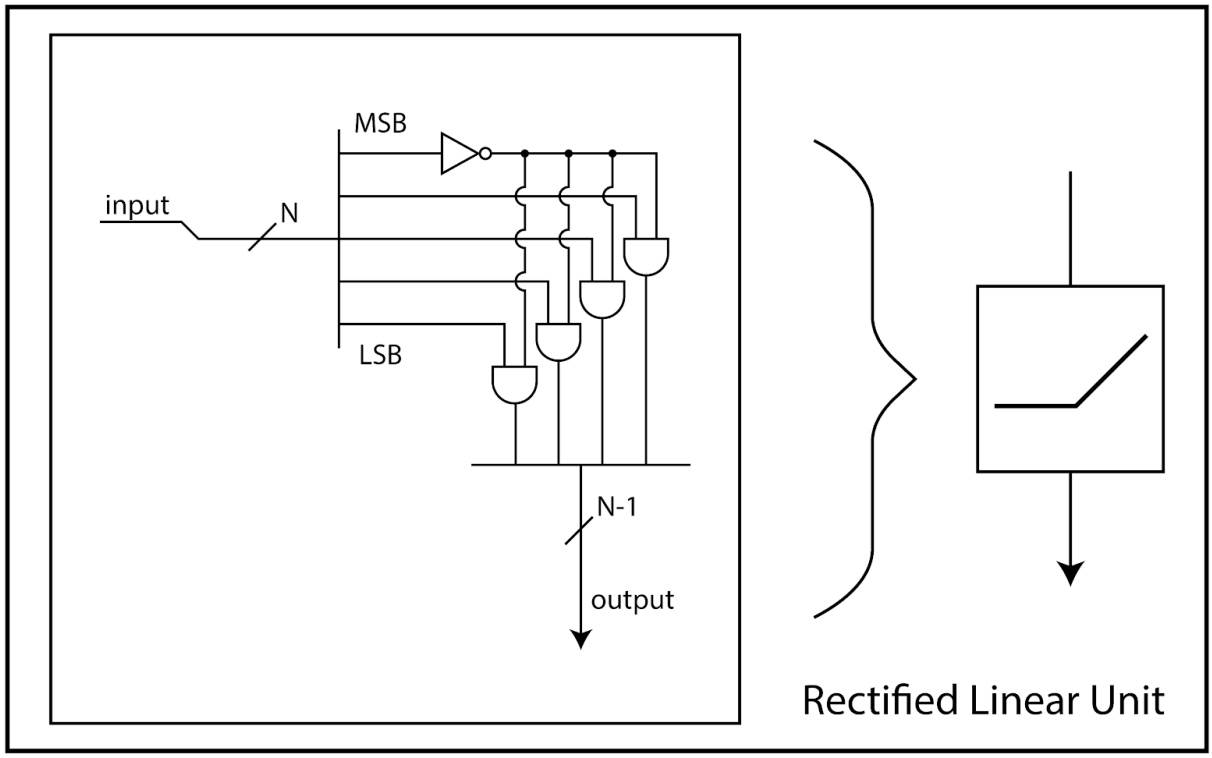
4.7. Layer Computation Module



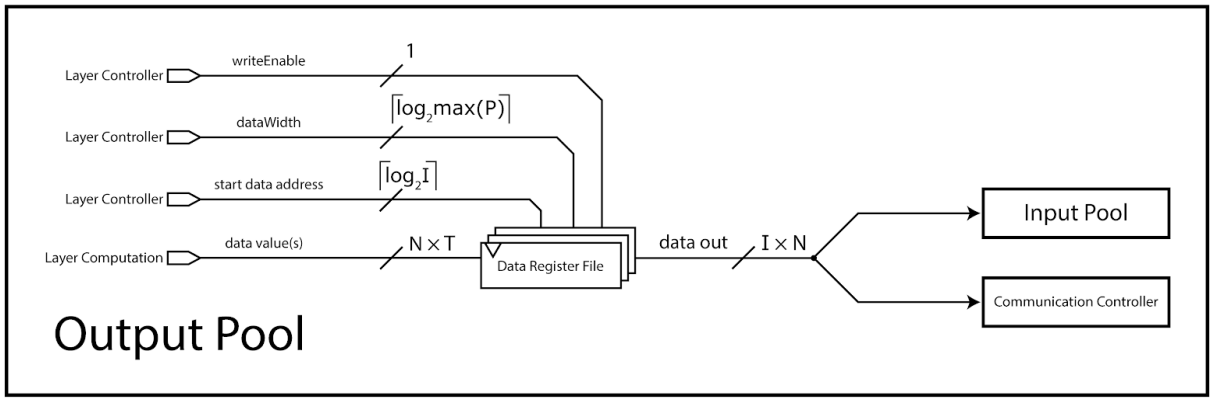
This module will control all DSP slices on the FPGA based on the inputs and control signals from the layer controller. The DSP slices will be interfaced to perform N * M bit multiplications of corresponding inputs and weights respectively. In order to avoid overflow, wires are sized according to the bit widths needed for the extremities of the operations performed. To reduce complexity in the Tree Summation structure, outputs from the multipliers are right shifted by S_1, which is equivalent to only considering the most significant bits of the multiplied value.

At each timestep we must sum the outputs for the neuron's computed in the timestep. The multiplier outputs are routed to the adder trees using multiplexers, with select signals from the layer controller. These select signals are derived from the neuron parallelism value and the indices of multipliers corresponding to each neuron. The output of this summation will be right shifted by S_2 in order to avoid overflow while maintaining manageable bit widths. The Rectified Linear Unit transform is applied to this output, and the final N-bit output is sent to the output pool. The aforementioned S_2 shift introduces the possibility of underflow error, which will be inspected by the debug module.


inputs
N
N+1
N+2
N+3
output
[N] inputs [W]
output ⌈log2(W)⌉
Tree Summation


MSB
input
N
LSB
N-1
output
Rectified Linear Unit

4.8. Output Pool



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. The data from this register file is forwarded to the Input Pool and the Communication Controller for appropriate further processing.

5. Testing

We plan to simulate each module in isolation using dedicated test-benches. Given the complexity of Layer Controller signals and FSM transitions, we will dedicate the most time to testing it. Once we believe the implementation to be functionally correct in simulation we will begin tests with a single layer for which we know the correct output. Though this will have been performed in simulation during the Layer Calculator tests, this procedure will test the broader hardware structure, from sending data across the communication interface to receiving it back on the PC side. Once we can calculate on one layer we will then test our logic for handling multiple layers on the FPGA. With all these tests passing, we will move out of the testing phase and into the implementation phase to use our neural network structure to complete the stretch goals described in the abstract. It is worth noting the foundational goals rely on only one layer being operational.

6. Constraint Calculations

6.1. USB-UART

The FTD2232HQ chip on the Nexys 4 DDR supports up to 12Mbaud, however this is constrained further by the capabilities of the USB port on the PC side. This is 128Kbaud on the host computer.

The weights are only sent once per network, but will be the largest set of inputs stored. As described below we expect to transmit on the order of 2,000,000 bits of weight values. At the given transfer rate this will take $\frac{2{,}000{,}000\ bits}{128{,}000\ baud} \approx 16$ seconds. This excludes headers, start bits, and stop bits. This is about a constant 20 second overhead to load any network onto the FPGA and assumes no errors are encountered in the process. If errors are found to be prominent, error correction bits or packet retransmission can be added to the communication protocol.

On every inference step, a new set of inputs must be loaded. We expect our largest input size to be on the order of 200 inputs. With the given transfer rate this will take $\frac{200\ bits}{128{,}000\ baud} \approx 1.5$ milliseconds.

6.2. Memory

The neural network weights comprise the largest memory demand on the FPGA circuitry. With each weight being M bits, and the layer width, depth, and output layer count being denoted by I, D, and O, respectively, the number of bits needed to store all weights for the fully connected topology is given by $M * (I + I^2 * (D - 2) + I * O)$. If I, D, and O are taken to be 200, 4, and 6, respectively, 2,035,000 bits will be required to store all weights. In particular, O is chosen to be six to represent the four directional buttons and the two input command buttons found on the Nintendo Entertainment System.