Modeling a Perceptron Neuron Using Verilog Developed Floating-Point Numbering System and Modules for Hardware Synthesis

Dr. Afsaneh Minaie, Utah Valley University

Afsaneh Minaie is a professor of Computer Engineering at Utah Valley University. She received her B.S., M.S., and Ph.D. all in Electrical Engineering from University of Oklahoma. Her research interests include gender issues in the academic sciences and engineering fields, Embedded Systems Design, Mobile Computing, Wireless Sensor Networks, Nanotechnology, Data Mining and Databases.

Mr. Ephraim Nielson, Utah Valley University

Modeling a Perceptron Neural Network Using Verilog Developed Floating-Point Numbering System and Modules for Hardware Synthesis

Abstract

The purpose of a capstone design project is to provide graduating senior students the opportunity to demonstrate understanding of the concepts they have learned during the course of their studies. As with many engineering programs, students of the computer engineering program at Utah Valley University (UVU) conclude their degree programs with a semester capstone design experience. This paper presents the details of a sample project that a student has done in this capstone course.

This senior design project implements the perceptron neural network using Systems Verilog HDL module development for FPGA synthesis. A floating-point binary numbering system is developed with which all arithmetic operation modules are designed. Benefits of hardware implemented neural networks include the parallelization of computational processes that are not provided in software implementations of such networks. All modules included in the network are simulated using Altera's ModelSim platform and synthesized on Altera's DE2-115 Development Board.

Introduction

Neural networks are a type of machine learning algorithm that were created with the intention to mimic the biological function of neurons in the brain [1]. In this biological sense, the primary purpose of neurons is to process and communicate information with each neuron conveying information through "synapses" to other neurons. These are the fundamental units of the nervous system and are extensively large with over 100 billion neurons and 100 trillion synapses in the human brain alone. These interconnections form networks to accomplish their sensory and motor tasks efficiently and effectively over time as networks and neurons are refined and strengthened [1] [2] [3].

The effectiveness and efficiency of the nervous system in the human body has yet to be matched by any human fabrication, however engineers are working towards the creation of systems that more closely emulate the function of the brain to create more intelligent systems [1] [4]. Some scientist work towards a direct, synthetic emulation of neurons. These neurons mimic the operation of the human neuron including the types of excitation, and subsequent electric spiking patterns. The second type of network is more of a functional ability of a system to "learn" as the human brain does. This system, while not a replication of biological brain function and cognition, is widely used and is implemented in many search engine optimizations, digital data organization techniques, data processing, image classification, voice recognition, and much more. These are what are commonly referred to as neural networks in technology – or artificial neural networks.

Just as the biological neurons in the brain are developed over time, artificial neural networks can "learn" based on the accumulation of past and present data available to the network rather than making computations from application specific developed algorithms.

While artificial neural networks are not as expansive as their human inspiration, they are, nonetheless, large networks that can be very computationally expensive. Resources allocated to computing and training a neural network can be further demanding as most of these networks are currently implemented in software rather than hardware and are subject to the constraints of reduced instruction set computing (RISC) [5]. While this allows for flexibility in the development of specific types of neurons and neural networks, it is slower to develop a trained model in comparison with hardware implementations. The goal of this work is to develop a neuron that will be implemented in hardware, flexible in implementation for further expansion, utilizes parallel computation, and is developed from basic hardware such as registers, latches, adders, comparators, and inverters.

The perceptron is one of the early developed neural networks and implements a supervised learning style. This type of learning develops the neuron by training a set of inputs towards a certain output or target. After repeated iterations of training inputs towards the target (called epochs), a well-developed perceptron's output converges or approaches convergence with the target output that is provided and then can be used for classification of organized data. Like the biological neuron, the supervised neural network has only one output or axon [7]. Thus, any sized vector of inputs can be targeted towards a specific scaler output which the neuron is expected to converge.

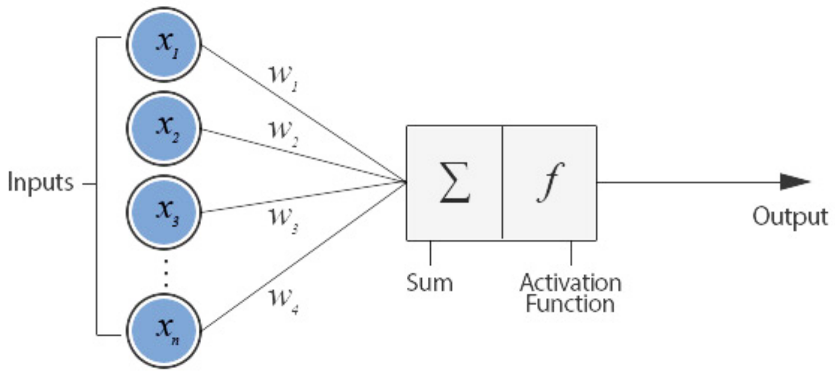


Figure 1 - Illustration of a single perceptron with the two basic layers of the network: input and activation layers [9]

An early model of the perceptron neuron was introduced by Frank Rosenblatt in 1958 [8]. The various networks developed by Rosenblatt, while more complex systems have been developed, is the inspiration for the development of artificial neural networks by computer scientists. A simple single-layered perceptron is shown in Figure 1. This neuron is made up of two main parts: The Input and the General Neuron. The input portion reads in the data, **x** – a vector of inputs {x_1, x_2, x_3, …, xn} and multiplies each input by a weight {w_1, w_2, w_3, … w_n}. The general body of the neuron then adds the weighted inputs and a bias. The scaler result is passed through an output function called the "activation function". The scaler output of a trained network can be used for data classification or for propagation to an input of a larger network in a method called pooling [10]. A mathematical model of the perceptron is shown in Equation 1 [7].

$$a = f(\sum_{i=1}^{n} x_{\mathrm{i}} \cdot w_{\mathrm{i}} + \mathrm{b}) \qquad (1)$$

The value of the synaptic weights prior to training is normally randomized and is unimportant as the value of the weights and bias are gradually changed through repeated iterations, or epochs, to converge the output of the neuron with the target output. [11].

This work comprises of the implementation of a two input, single-layered perceptron with no bias. A floating-point numbering system is developed, and all components and arithmetic processes are designed for the floating-point system. Platform for hardware design is done using Altera's Quartus Prime 17.0 software package and coded in the system Verilog hardware description language (HDL). Each module created is case tested by testbenches developed and simulated using Altera's ModelSim software. Further hardware synthesis is performed by use of the JTAG programmer included in Alter's Quartus Prime 17.0 software and synthesized on a field programmable gate array (FPGA) using Altera's DE2-115 Development Board.

Floating-Point Numbers

A floating-point numbering system developed at XXX University is used for the data processing in the perceptron neuron. It is based on the structure of the IEEE 754 Single-Precision floating-point numbering system shown in Figure 2 where the floating-point number comprises of a sign, exponent, and fraction portion of a 32-bit word. The formatting of the number is similar to the IEEE protocol where the 32-bit word comprises of a sign bit in the most significant position of the word followed by 8 bits representing the exponent and the remaining 23 bits dedicated to the fraction of the number. The system, in contrast to IEEE's signed magnitude floating-point numbering system, will be based on a 2's compliment numbering system where the concatenation of the sign bit and the 23 bits of the fraction, {S, F [22:0]}, make up a 2's compliment number with a value between -1_{10} and $0.99999988079071044921875_{10}$.

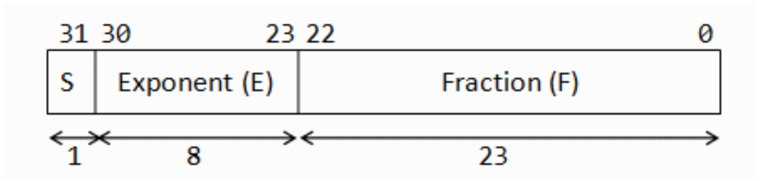


Figure 2 – IEEE 754 protocol for single-precision floating point numbers [12].

The exponent of the floating-point number, in contrast to IEEE's exponent format, is also based on 2's compliment numbers and falls between a range of -128 and 127. Thus, the range of numbers allowed to be represented using this floating-point numbering system is $-1.701412 \times 10^{38}{}_{10}$ to $1.701412 \times 10^{38}{}_{10}$ with a resolution of $3.50325 \times 10^{-46}{}_{10}$. This give a level of accuracy that is very precise compared to fixed-point numbers. This numbering system has a wider range and more accurate resolution compared to IEEE 754 Single-Precision standard. The equation associated with this numbering system is shown in equation 2.

$$(-S + F) \cdot 2^E \qquad (2)$$

Where E is the 2's compliment of a number with a range between $-128_{10} < E < 127_{10}$ and the fraction, F, has the range of $1.1920928955078125 \times 10^{-7}{}_{10} < F < 0.99999988079071044921875_{10}$. The value of S is in the 2^0's place and is either a value of 1_2 or 0_2. Properly formatted numbers require the most significant bit of the fraction, F [22] and the sign bit, S, to be of a different polarity. Thus, if S is equal to 1, then F [22] must be equal to 0, and if S is equal to 0 then F [22] must be equal to 1. If a module that hereafter results in a number where the S bit and the F [22] bit are of the same polarity, S == F [22], then a process takes place where the concatenation of {S, F [22:0]} is shifted 1 bit to the left and the exponent, E, is decremented by a value of 1 so that the accurate value of the number is not lost. While the left shift of {S, F [22:0]} results in a multiplication of the {S, F [22:0]} number by 2, that multiplication is compensated for a subtraction in the Exponent since 2^{-1} is equal to a division by 2.

Another stipulation of correctly formatting numbers is that the value of the whole floating-point number must be of a value between -1_{10} and 1_{10}. All inputs to the perceptron are assumed to already be properly formatted and normalized, however, all data from the arithmetic modules developed demonstrate robust handling of unnormalized data and ensure proper formatting at the output port. Algorithmic State Machine (ASM) charts associated with each arithmetic module contains the conditional block "Fnorm" that checks for proper floating-point normalization and formatting. An example of this can be seen in the arithmetic theory section where the ASM chart for the adder is included in Figure 7.

Perceptron Design and Development

Design of the perceptron is broken down into module that perform a specific function for the network. Equations 3-6 demonstrate the mathematical operations necessary for the perceptron in this work where *n* is the current epoch, *x* is the input, w is the weight, and *i* designates which input and weight are designated during that epoch.

$$i[n] = \sum_{i=1}^{2} (x_i[n] \cdot w_i[n]) \qquad (3)$$

$$f(i[n]) = \begin{cases} -1 & for\ i[n] \leq -1 \\ 0.5 \cdot i[n] - 0.5 & for -1 < i[n] < 0 \\ 0 & for\ i[n] = 0 \\ 0.5 \cdot i[n] + 0.5 & for\ 0 < i[n] < 1 \\ 1 & for\ i[n] \geq 1 \end{cases} \qquad (4)$$

$$r(Target[n],\ f(i[n])) = (Target[n] - f(i[n])) \cdot f'(i[n]) \qquad (5)$$

$$w_i[n+1] = \begin{cases} w_i[n] + LearnRate \cdot r\big(Target[n], f(i[n])\big) \cdot x_i[n] \\ w_1[1] = Weight_1 \\ w_2[1] = Weight_2 \end{cases} \tag{6}$$

While a trained model needs only equations 3 and 4, training the perceptron utilizes all the equations listed. When in training mode, an input of the *Target* value of the perceptron is required. Other initialized values not taken as inputs to the perceptron are *Weight$_1$, Weight$_2$,* and *LearnRate* that are included in the Perceptron module. These values can be changed according to the desired value in code. Traditional initialization techniques of the weights are done by randomly selecting a number, however, in this work, the weights are predetermined to verify correct perceptron output and project validation.

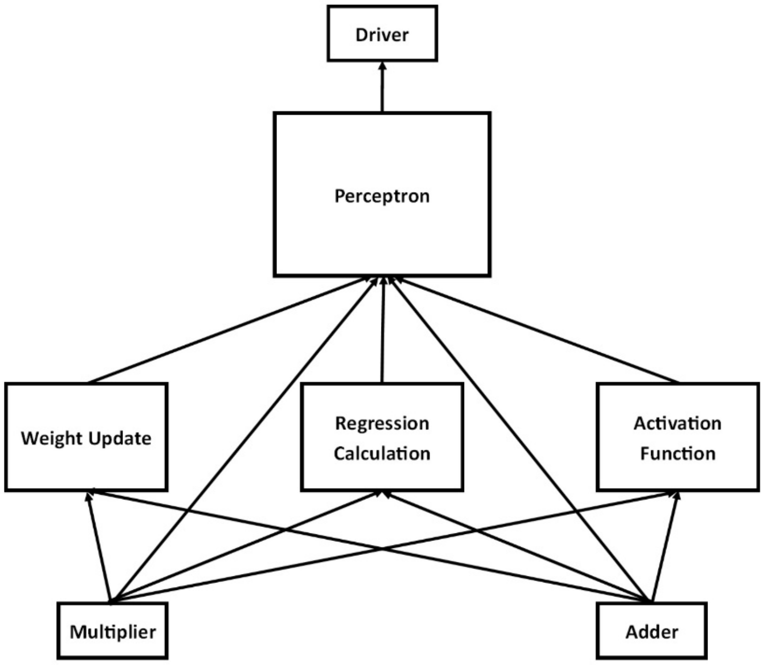


Figure 3 – Hierarchy of perceptron neural network project design

Equation 3 performs the input function of the perceptron as discussed in Figure 1 while Equation 4 is the activation function used in this work. Equation 4 is a linear approximation of the inverse tangent function for the classification of linearly separable data. Equation 5 calculates the regression of the perceptron from the produced output and the target output. Regression calculation includes the first derivative of the activation function as a parameter. In the case of the perceptron as it is designed, *f'(i[n])* = 0.5 for -1 < *i[n]* < 0 and for 0 < *i[n]* < 1.Therefore, the calculation of the first derivative is skipped as all values are normalized to be within -1 and 1. To be accurate, this should be undefined when *i(n)* = 0, however, for brevity, this is not taken into consideration and will be a focused in future work. Equation 6 updates each weight included in the perceptron for the next epoch.

The fundamental module for all other modules are the arithmetic modules of the multiplication and addition of two floating-point numbers. All other modules comprise of instantiations of the multiplier and adder to perform their specific function. The hierarchy diagram designating the separation of functions for the perceptron module is shown in Figure 3.

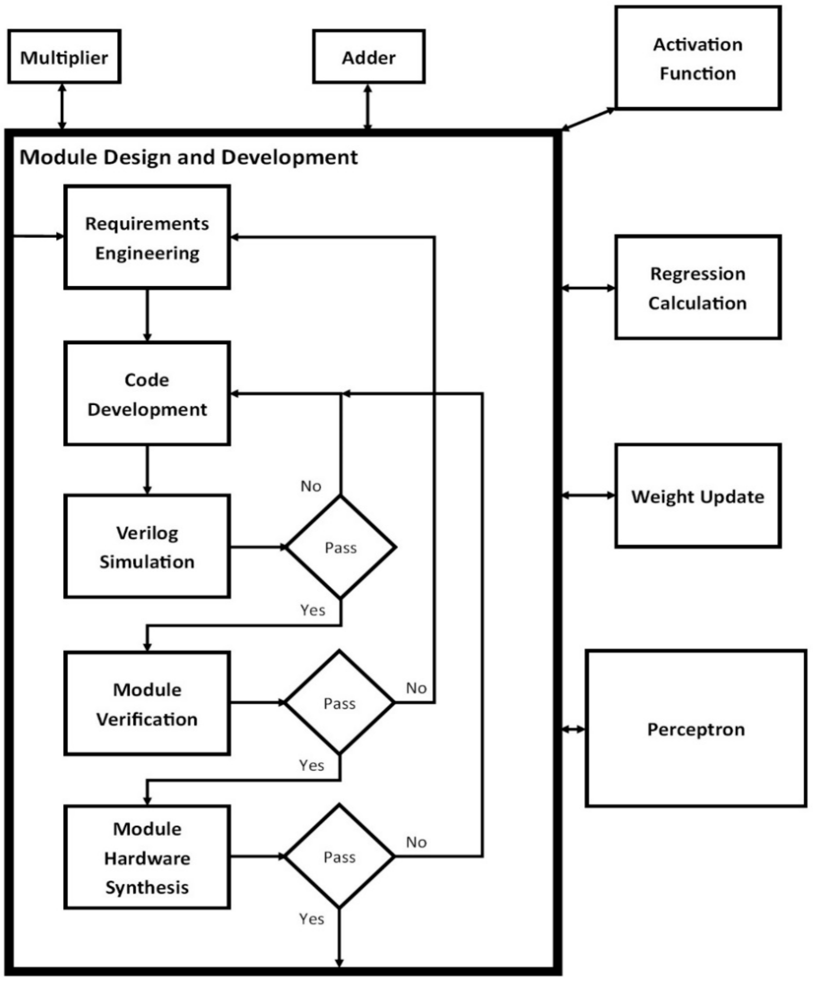


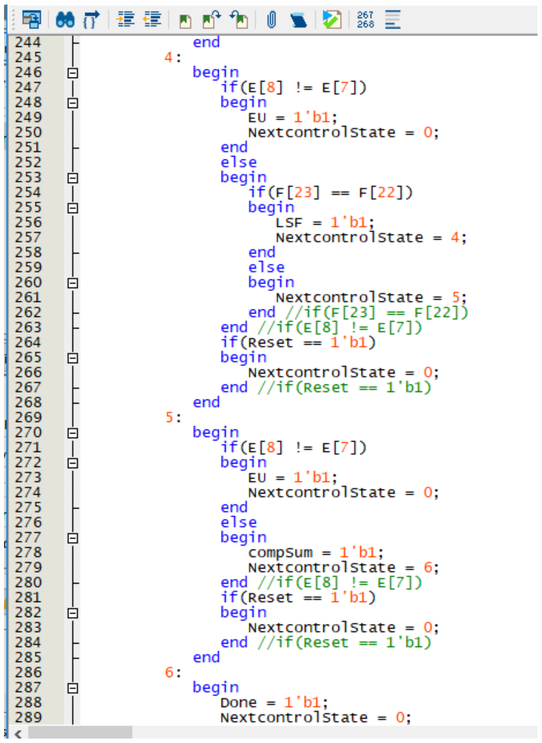
Figure 4 – Flow chart of project design and development with iterative capabilities

A module to drive the perceptron is also developed for hardware synthesis on the DE2-115 Development Board. This module, while not part of the perceptron neural network, consists of a state machine that gets serial input of the perceptron's inputs and target output from the device switches and perceptron controls from the pushbuttons. It then expands the input and target output values to the 32 bits necessary for perceptron processing and subsequently truncates the perceptron output for display.

Development of each module began with requirements engineering, code development, simulation of code, module verification, and synthesis of the module on the DE2-115 FPGA. It also follows an agile approach as shown in Figure 4. This allowed for flexibility in design of each module as requirements changed. For example, the first modules to be developed were the multiplier and adder modules. These modules initially contained no reset function. When the activation function was designed, it was determined that a reset function needed to be included in all submodules included in the activation function module in the case that an error occurred. Therefore, a revisit of the requirements for the multiplier and adder modules were necessary and both recoded, simulated, and verified before proceeding to the design of the regression calculation.

The design of the multiplier is based on bit-shift and add multiplication technique for the concatenation of the sign fractional portions of the multiplicand and multiplier for floating-point numbers of all polarities. Since the exponents of the floating-point numbers are added together during multiplication, an error signal is produced if overflow occurs.

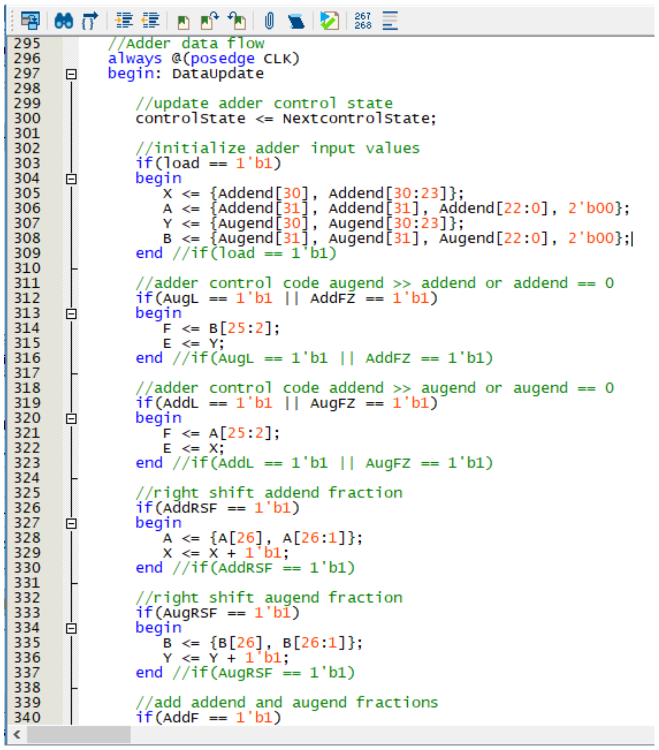
Likewise, the adder module keeps track of both exponent overflow and underflow as the exponents of both the addend and augend must be equal before the concatenations of sign and fractional portions of the floating-point number can be added. A snippet of code used for overflow detection in the adder module's control path is shown in Figure 5 while a snippet of the adder module's data path is shown in Figure 6.



```
            end
        4:
            begin
                if(E[8] != E[7])
                begin
                    EU = 1'b1;
                    NextcontrolState = 0;
                end
                else
                begin
                    if(F[23] == F[22])
                    begin
                        LSF = 1'b1;
                        NextcontrolState = 4;
                    end
                    else
                    begin
                        NextcontrolState = 5;
                    end //if(F[23] == F[22])
                end //if(E[8] != E[7])
                if(Reset == 1'b1)
                begin
                    NextcontrolState = 0;
                end //if(Reset == 1'b1)
            end
        5:
            begin
                if(E[8] != E[7])
                begin
                    EU = 1'b1;
                    NextcontrolState = 0;
                end
                else
                begin
                    compSum = 1'b1;
                    NextcontrolState = 6;
                end //if(E[8] != E[7])
                if(Reset == 1'b1)
                begin
                    NextcontrolState = 0;
                end //if(Reset == 1'b1)
            end
        6:
            begin
                Done = 1'b1;
                NextcontrolState = 0;
```

Figure 5 – Code Snippet of the Control Path of the Adder Module

Separation of the control and data paths are implemented for every module in the perceptron for the successful creation of a state machine in hardware synthesis. Control paths require that all control lines consist of 1-bit registers and keep their value for one clock cycle. This ensures proper operation and prevents the synthesis of inferred latched. The control path is asynchronous with the clock and executes sequentially in contrast with the data path where registers are updated synchronous with the clock. Data paths can also include latches and control large arrays of registers.



```
//Adder data flow
always @(posedge CLK)
begin: DataUpdate

   //update adder control state
   controlState <= NextcontrolState;

   //initialize adder input values
   if(load == 1'b1)
   begin
      X <= {Addend[30], Addend[30:23]};
      A <= {Addend[31], Addend[31], Addend[22:0], 2'b00};
      Y <= {Augend[30], Augend[30:23]};
      B <= {Augend[31], Augend[31], Augend[22:0], 2'b00};
   end //if(load == 1'b1)

   //adder control code augend >> addend or addend == 0
   if(AugL == 1'b1 || AddFZ == 1'b1)
   begin
      F <= B[25:2];
      E <= Y;
   end //if(AugL == 1'b1 || AddFZ == 1'b1)

   //adder control code addend >> augend or augend == 0
   if(AddL == 1'b1 || AugFZ == 1'b1)
   begin
      F <= A[25:2];
      E <= X;
   end //if(AddL == 1'b1 || AugFZ == 1'b1)

   //right shift addend fraction
   if(AddRSF == 1'b1)
   begin
      A <= {A[26], A[26:1]};
      X <= X + 1'b1;
   end //if(AddRSF == 1'b1)

   //right shift augend fraction
   if(AugRSF == 1'b1)
   begin
      B <= {B[26], B[26:1]};
      Y <= Y + 1'b1;
   end //if(AugRSF == 1'b1)

   //add addend and augend fractions
   if(AddF == 1'b1)
```

Figure 6 – Code Snippet of the Data Path of the Adder Module

Control paths of each module determines the state of data flow according to the condition of the data as determined by the desired function. A visual illustration of these are found in the module's Algorithmic State Machine (ASM) chart. An example of an ASM chart for the Adder module is found in Figure 7. This follows the description of Adder operation below:

- Load (load) the Addend and Augend into the data path if start (St) is high.
- If the Addend is zero,
  - Inform data path to produce the Augend (AddFZ).
- Else if the Augend is zero,
  - Inform the data path to produce the Addend (AugFZ).
- Else if both exponents of the Addend and Augend are equal, then continue to next step.
  - If $E_1 – E_2 > 23$, Addend will become summation (AddL). Skip to last step.
  - If $E_1 > E_2$, check for overflow and right shift F_2, increment E_2 (AugRSF), and repeat step.
  - If $E_2 – E_1 > 23$, Augend will become summation (AugL). Skip to last step.
  - If $E_2 > E_1$, check for overflow and right shift F_1, increment E_1 (AddRSF), and repeat step.
- Add the fractions of both floating-point values (AddF).
- Check for normalization of the fraction and continue (Fnorm).
  - If value is not formatted correctly, left shift F and decrement E (LSF).

    - Check for underflow and repeat.
- Output summation if no underflow has occurred (compSum).

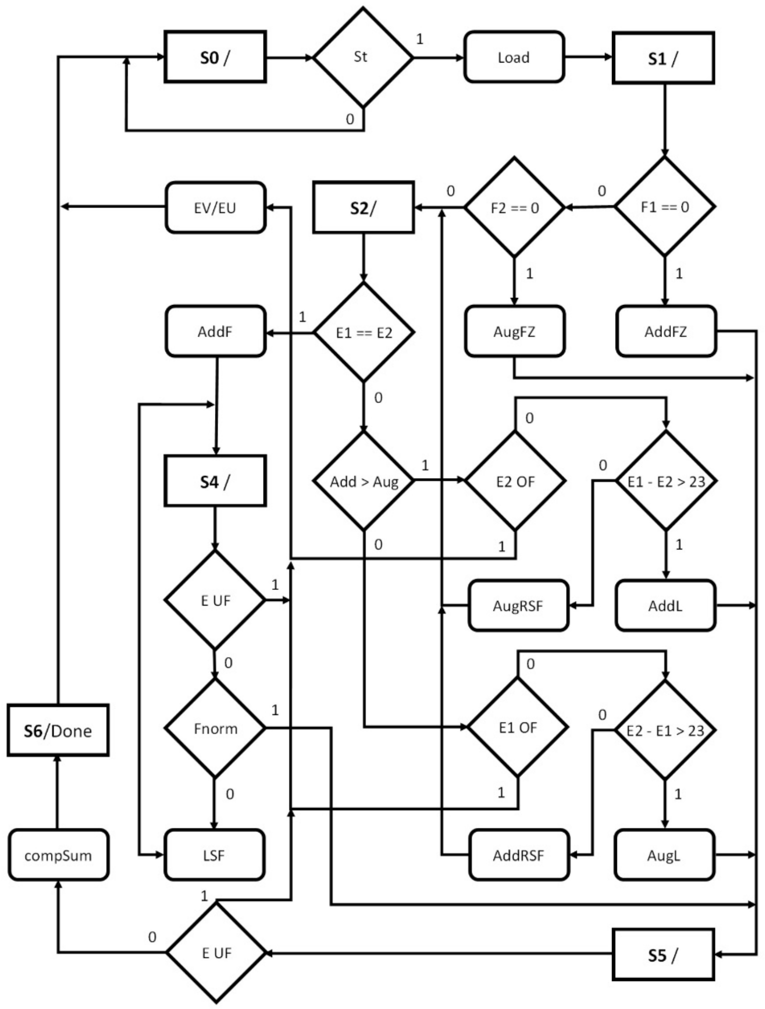


Figure 7 – ASM Chart Illustrating the Control Circuit Operation of the Adder.

In a similar manner to the design of the Adder module, all modules in the perceptron neural network separate control and data paths for operational hardware synthesis. Each module also includes an ASM chart to illustrate the design of the module.

Results

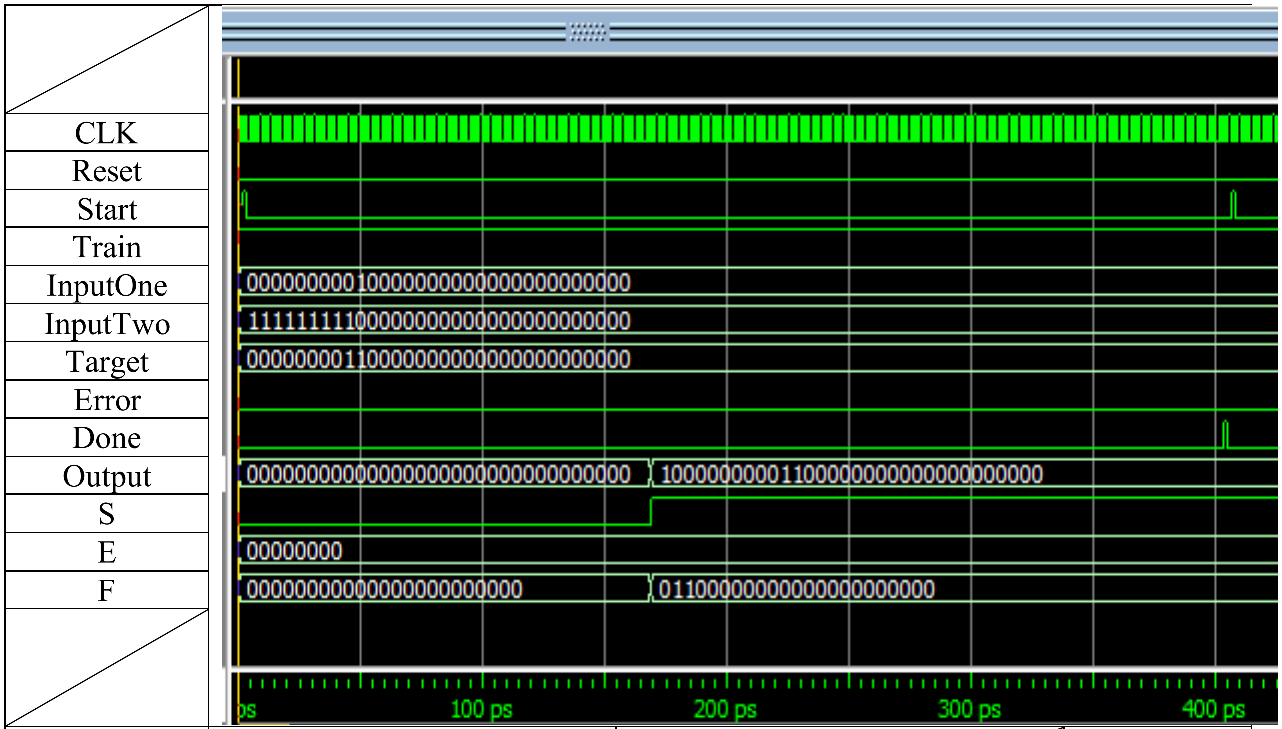
Verification and validation of the perceptron design is carried out by module simulation and synthesis. Simulation is done by creating testbenches in Verilog and verifying the results of the outputs in ModelSim. Synthesis is done by using the JTAG programming toolbox available in the Quartus Prime software platform.

Testing is not exhaustive and is performed in case tests that check the operation of the module to ensure functionality. This includes the successful performance of its specific task as well as the function of controls. Errors occurring from overflow and underflow are tested to ensure correct module output. Further testing of submodules occurs during perceptron level verification. During simulation predetermined inputs are placed in the ports of modules instantiations while outputs wires are monitored for correct operation. Testing is done by case for each of the modules. Most are designed to test the functionality of the modules as well as ensure proper output of error signals. Figure 8 shows the simulation of the first epoch of the perceptron where Weight$_1$ = 0.25, Weight$_2$ = 0.75, and the LearnRate = 0.5 as referenced to in Equation 6.

Test Case 2: Input$_1$ = 0.5_{10}, Input$_2$ = -0.5_{10}, Expected Output = 1_{10}
Weight$_1$[Epoch 1] = 0.25_{10}, Weight$_2$[Epoch 1] = 0.75_{10}, Learning Rate = 0.5_{10}

Epoch 1

	S	**E**	**F**	**Pass/Fail**
Computed Output:	1	00000000	0110000000000000000000	
Actual Output:	1	00000000	0110000000000000000000	Passed
Control:				Passed
Error:				N/A



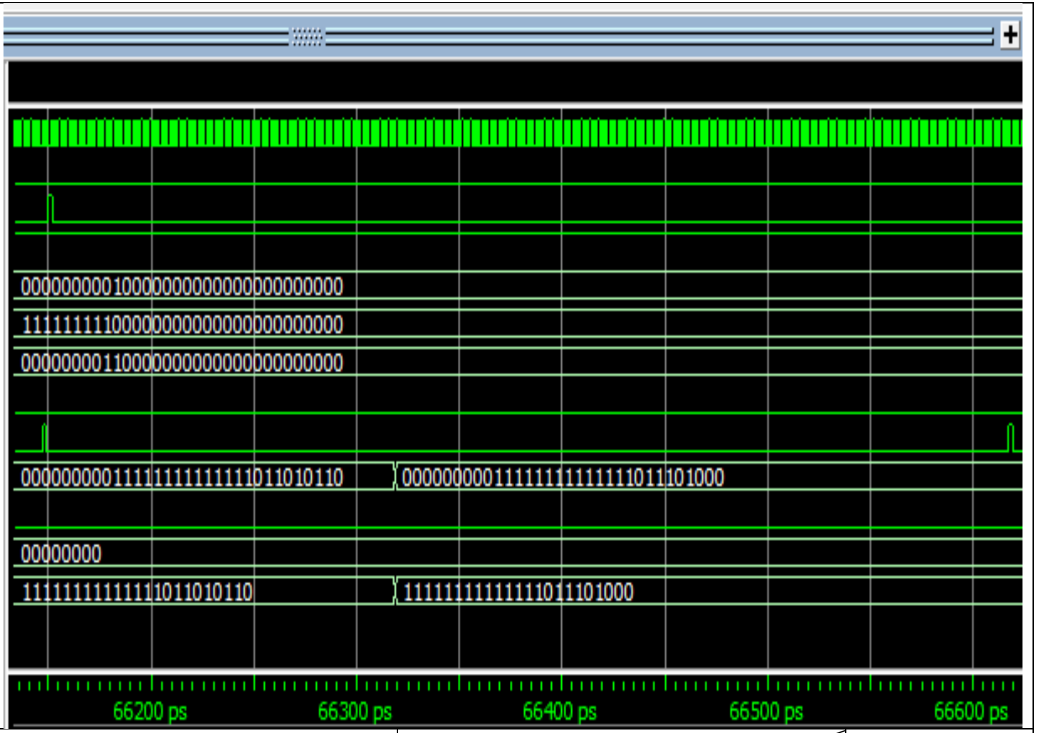
```
InputOne   0000000001000000000000000000000
InputTwo   1111111110000000000000000000000
Target     0000000011000000000000000000000
Output     0000000000000000000000000000000  1000000000110000000000000000000
E          00000000
F          0000000000000000000000           0110000000000000000000
```

Net	**Binary**		**Decimal**
S	1		1
E	00000000		0
F	0110000000000000000000		0.375
Output	$(-S+F)\cdot 2^{E} =$	$(-1+\ 0.375)\cdot 2^{0}$	-0.625

Figure 8 – First Epoch of the Test Case for Perceptron Verification and Operation

This is a case specific test showing the verification of module performance and validation of the system through a gradual convergence of the produced output and the target output according to calculated values monitored. Simulation for the 150th epoch is shown in Figure 9. As can be seen, for an input of x_1 = 0.5, x_2 = -0.5, a target of 1, and a learning rate of 0.5, the output of the perceptron is 0.99996662139892578125 which is close to a value of 1. This demonstrates the convergence of the perceptron's output with the target value as the difference between the target and the perceptron's output is 1.625 during the first epoch and converges to a difference of 0.00003337860107421875 for the 150th epoch.

Test Case 2: Input$_1$ = 0.5_{10}, Input$_2$ = -0.5_{10}, Expected Output = 1_{10} Weight$_1$[Epoch 1] = 0.25_{10}, Weight$_2$[Epoch 1] = 0.75_{10}, Learning Rate = 0.5_{10}				
Epoch 150				
	S	**E**	**F**	**Pass/Fail**
Computed Output:	0	00000000	11111111111111011101000	
Actual Output:	0	00000000	11111111111111011101000	Passed
Control:				Passed
Error:				N/A



CLK	
Reset	
Start	
Train	
InputOne	00000000100000000000000000000000
InputTwo	11111111100000000000000000000000
Target	00000000110000000000000000000000
Error	
Done	
Output	00000000011111111111111011010110 → 00000000011111111111111011101000
S	
E	00000000
F	11111111111111011010110 → 11111111111111011101000

Net	**Binary**		**Decimal**
S	0		0
E	00000000		0
F	11111111111111011101000		0.999966…
Output	$(-S+F)\cdot 2^{E} =$	$(0 + 0.9999\ldots)\cdot 2^{0}$	0.999966…

Figure 9 – 150th Epoch of the Test Case for Perceptron Verification and Operation

The perceptron, while many operations need to be operated in sequence, makes use of the parallel multiplications for the inputs and their weights. Further parallelization of the perceptron is done in the update of the weights during training. In total, the amount of clock cycles for the perceptron as it is designed here is reduced from 308 clock cycles to 222 clock cycles per epoch. This shows an increase in efficiency of the perceptron by over 38.7% when the perceptron is in training mode. Using the 50 MHz clock included on the DE2-115 development board, an epoch for the perceptron in training mode is completed in 4.44 µs as opposed to 6.16 µs using sequential operations as would be found in software applications without accelerated hardware. The 150 epochs, using the DE2-115 development board, are completed in 666.18 µs.

As more inputs are added, the parallelization of available perceptron functions become increasingly important as the multiplication of the inputs and weights can stay constant at 26 clock cycles and the update weight operation of the neural network at 60 clock cycles.

Summary and Concluding Remarks

Capstone courses play a crucial role in Computer Engineering curricula. The principle purpose of a Capstone project course is to offer a summative opportunity for graduating senior engineering students to apply their professional skills and knowledge in a single experience and prepare them for work in industry. Like many engineering programs, students at Utah Valley University complete their requirements for graduation with a semester long capstone design project course. The intention of this course is to apply competencies gained during their first three years toward the solution of a design problem. Our senior design course is structured as a collection of independent student projects. As our students are required to design, build, and troubleshoot a fully functional project, they find this course both challenging and rewarding.

Scope of this senior design project was to demonstrate the development, design, simulation, and synthesis of a neural network that makes a contribution to technology today. Neural networks are an important part of computational intelligence, are widely in use, and show promise for continued growth as technology progresses. As most neural networks are implemented in software, a need is shown for faster computations of these systems as demand for resources increases. Thus, neural networks implemented in dedicated hardware and integrated circuits are becoming increasingly important. Speed is shown in increase as more operations are available for parallelization.

This project was successful and the student commented after finishing his project that “this project was probably one of the more difficult projects I had the opportunity to work on during my years at the university. It was also one of the most rewarding. It required a self-initialized approach to managing time and finding good resources along with the development of hardware development skills in an important field of computational intelligence. The first benefit was a better understanding of neural networks and why they are important. The second is a better understanding of product development and design using HDL tools that are currently in use as well as a deeper knowledge of digital design. The third is an appreciation for failure. Throughout the design process, there were many times when things didn’t work out and I had to go back to the code and redesign certain features of the modules I developed. The mental work it took to problem solving, in turn, gave me a better understanding of hardware design and is a practical tool that I can use in my future career."

References

[1] F. Folowosele, T. J. Hamilton and R. Etienne-Cummings, "Silicon Modeling of the Mihala¸s–Niebur Neuron," *IEEE Transactions on Neural Networks,* vol. 22, no. 12, pp. 1915-1927, December 2011.

[2] D. Johnston, S. Wu and R. Gray, Foundations of Cellular Neurophysiology, Cambridge, MA: MIT Press, 1995.

[3] T. Pearce and J. Williams, "Microtechnology: Meet neurobiology," *Lab Chip,* vol. 7, no. 1, pp. 30-40, January 2007.

[4] J. Wijekoon and P. Dudek, "Compact silicon neuron circuit with spiking and bursting behavior," *Neural Networks,* vol. 21, no. 2-3, pp. 524-534, March-April 2008.

[5] S. Sahin, Y. Bercerkli and S. Yazici, "Neural Network Hardware Implementation Using FPGAs," in *Neural Information Processing*, Berlin, Springer-Verlag, 2006, pp. 1105-1112.

[6] A. R. Omandi and J. C. Rajapakse, FPGA Implementations of Neural Networks, Springer, 2006.

[7] M. T. Hagan, H. B. Demuth and M. Beale, Neural network design, 2 ed., Boston, MA: PWS Publishing, 1996.

[8] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review,* vol. 65, no. 6, pp. 386-408, 1958.

[9] Mathworks, "Neural Network Toolbox," [Online]. Available: https://edoras.sdsu.edu/doc/matlab/toolbox/nnet/model215.html. [Accessed 1 February 2018].

[10] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss and E. S. Chung, "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware," Microsoft Research, 2015.

[11] K. Palchikoff, "Neural Networks: A neural network can approximated any practical non linear function," September 2003. [Online]. Available: http://ffden-2.phys.uaf.edu/212_fall2003.web.dir/Keith_Palchikoff/Neural%20Networks.html. [Accessed 1 February 2018].

[12] Nanyang Technical University, "A Tutorial on Data Representation: Integers, Floating-point Numbers, and Characters," January 2014. [Online]. Available: https://www.ntu.edu.sg/home/ehchua/programming/java/DataRepresentation.html. [Accessed 1 February 2018].

[13] C. H. Roth, L. K. John and B. K. Lee, Digital Systems Design using Verilog, 1 ed., Boston, MA: Cengage Learning, 2014.