Chapter 5
A Reconfigurable Hardware for Artificial Neural Networks*

Abstract. Artificial Neural Networks (ANNs) is a well known bio-inspired model that simulates human brain capabilities such as learning and generalization. ANNs consist of a number of interconnected processing units, wherein each unit performs a weighted sum followed by the evaluation of a given activation function. The involved computation has a tremendous impact on the implementation efficiency. Existing hardware implementations of ANNs attempt to speed up the computational process. However these implementations require a huge silicon area that makes it almost impossible to fit within the resources available on a state-of-the-art FPGAs. In this chapter, we devise a hardware architecture for ANNs that takes advantage of the dedicated adder blocks, commonly called MACs to compute both the weighted sum and the activation function. The proposed architecture requires a reduced silicon area considering the fact that the MACs come for free as these are FPGA's built-in cores. The hardware is as fast as existing ones as it is massively parallel. Besides, the proposed hardware can adjust itself on-the-fly to the user-defined topology of the neural network, with no extra configuration, which is a very nice characteristic in robot-like systems considering the possibility of the same hardware may be exploited in different tasks.

5.1 Introduction

Artificial Neural Networks (ANNs) are useful for learning, generalization, classification and forecasting problems [3]. They consists of a pool of relatively simple processing units, usually called artificial neurons, which communicates with one another through a large set of weighted connections. There are two main network topologies, which are feed-forward topology [3], [4] where the data flows from input to output units is strictly forward and recurrent topology, where feedback connections are allowed. Artificial neural networks offer an attractive model that allows one to solve hard problems from examples or patterns. However, the computational process behind this model is complex. It consists of massively parallel non-linear

* This chapter was developed in collaboration with Rodrigo Martins da Silva.

N. Nedjah and L. de Macedo Mourelle, *Hardware for Soft Computing and Soft Computing for Hardware*, Studies in Computational Intelligence 529,
DOI: 10.1007/978-3-319-03110-1_5, © Springer International Publishing Switzerland 2014

calculations. Software implementations of artificial neural networks are useful but hardware implementations takes advantage of the inherent parallelism of ANNs and so should answer faster.

Field Programmable Gate Arrays (FPGAs) [7] provide a re-programmable hardware that allows one to implement ANNs very rapidly and at very low-cost. However, FPGAs lack the necessary circuit density as each artificial neuron of the network needs to perform a large number of multiplications and additions, which consume a lot of silicon area if implemented using standard digital techniques.

The proposed hardware architecture described throughout this chapter is designed to process any fully connected feed-forward multi-layer perceptron neural network (MLP). It is now a common knowledge that the computation performed by the net is complex and consequently has a huge impact on the implementation efficiency and practicality. Existing hardware implementations of ANNs have attempted to speed up the computational process. However these designs require a considerable silicon area that makes tem almost impossible to fit within the resources available on a state-of-the-art FPGAs [1], [2], [6]. In this chapter, we devise an original hardware architecture for ANNs that takes advantage of the dedicated adder blocks, commonly called MACs (short for Multiply, Add and Accumulate blocks) to compute both the weighted sum and the activation function. The latter is approximated by a quadratic polynomial using the least-square method. The proposed architecture requires a reduced silicon area considering the fact that the MACs come for free as these are FPGA's built-in cores. The hardware is as fast as existing ones as it is massively parallel. Besides, the proposed hardware can adjust itself on-the-fly to the user-defined topology of the neural network, with no extra configuration, which is a very nice characteristic in robot-like systems considering the possibility of the same piece of hardware may be exploited in different tasks.

The remaining of this chapter is organized as follows: In Section 5.2, we give a brief introduction to the computational model behind artificial neural networks; In Section 5.3, we show how we approximate the sigmoid output function so we can implement the inherent computation using digital hardware; In Section 5.4, we provide some hardware implementation issues about the proposed design, that makes it original, efficient and compact; In Section 5.5, we present the detailed design of the proposed ANN Hardware; Last but no least, In Section 5.6, we draw some useful conclusions and announce some orientations for future work.

5.2 ANNs Computational Model

We now give a brief introduction to the computational model used in neural networks. Generally, is constituted of few layers, each of which includes several neurons. The number of neurons in distinct layers may be different and consequently the number of inputs and that of outputs may be different [3].

The model of an artificial neuron requires n inputs, say $I_1, I_2, \ldots, I_n$ and the synaptic weights associated with these inputs, say $w_1, w_2, \ldots, w_n$. The weighted sum a, which, also called activation of the neuron, is defined in (5.1). The model usually

includes an output function $nout(.)$ that is applied to the neuron activation before it is fed forwardly as input to the next layer neurons.

$$a = \sum_{j=1}^{n} w_j \times f_j \tag{5.1}$$

The non-linearity of the neuron is often achieved by the output function, which may be the hyperbolic tangent or sigmoid [3]. In some cases, $nout(a)$ may be linear.

A typical ANN operates in two necessary stages: *learning* and *feed-forward computing*. The learning stage consists of supplying known patterns to the neural network so that the network can adjust the involved weights. Once the network has learned to recognize the provided patterns, the network is ready to operate, performing the feed-forward computing. In this stage, the network is supplied with an input data or pattern, which may or not be one of those given in learning stage and verify how the network responds with output results. This allows one to know whether the neural network could recognize the input data. The precision of the net in recognizing the new input patterns depends on the quality of its learning stage and on its generalization. As we have previously mentioned, here we are only concerned with the implementation of feed-forward computing stage.

5.3 Approximation of the Output Function

Unlike the activation function, which includes operations that can easily and efficiently implemented in hardware, the out function requires a special care before the computation involved can be modeled in hardware. Without loss of generality, we chose to use the sigmoid output function. Note that the same treatment applies to the hyperbolic function too. To allow an efficient implementation of the sigmoid function defined in (5.2), in hardware, we proceeded with a parabolic approximation of this function using the least-square estimation method.

$$sigmoid(a) = \frac{1}{1+e^{-a}} \tag{5.2}$$

The approximation proceeds by defining $nout(a) = C \times a^2 + B \times a + A$ as a parabolic approximation of the sigmoid of (5.2), just for a short range of the variable a. We used the least-square parabola to make this approximation feasible. Many attempts were performed to try to find out the best range of a for this approximation, so that the parabola curves fits best that of $sigmoid(a)$. We obtained the range $[-3.3586, 2.0106]$ for variable a, taking into account the calculated coefficients $C = 0.0217$, $B = 0.2155$ and $A = 0.4790$ for the parabolic approximation. Thus, the approximation of the sigmoid function is as defined in (5.3):

$$nout(a) = \begin{cases} 0 & a < -3.3586 \\ 0.0217 \times a^2 + 0.2155 \times a + 0.4790 & a \in [-3.3586, 2.0106] \\ 1 & a > 2.0106 \end{cases} \tag{5.3}$$

5.4 Implementation Issues

An Artificial Neural Network is a set of several interconnected neurons arranged in layers. Let L be the number of layers. Each layer has its own number of neurons. Let m_i be the number of neurons in layer i. The neurons are connected by the synaptic connections. Some neurons get the input data of the network, so they are called *input* neurons and thus compose the input layer. Other neurons export their outputs to the outside world, so these are called output neurons and thus compose the output layer. Neurons placed on the layer 2 up to layer $L-1$ are called the hidden neurons because they belong to the hidden layers. In Fig. 5.1, we show a simple example of an ANN. The output of each neuron, save output neurons, represents an input of all neurons placed in the next layer.

The computation corresponding to a given layer starts only when that of the corresponding previous layer has finished. Our ANN hardware has just one *real* layer of neurons, constitutes of k neurons, where k is maximum number of neurons per layer, considering all layers of the net. For instance, for the net of Fig. 5.1, this parameter is 3. This single real layer or physical layer is used to implement all layers of the network. As only one layer operates at a time, this allows us to minimize drastically the silicon area required without altering the response time of the net. For instance, considering the net of Fig. 5.2, the first stage of the computation would use only 2 neurons, then in the second stage all three physical neurons would be exploited and in the last stage, only one neuron would be useful. So instead of having 6 physically implemented neurons, our hardware requires only half that number to operate. ANN hardware treats the nets layers as virtual.

Besides reducing the number of neurons that are actually implemented in hardware, our design takes advantage of some built-in cores that come for free in nowadays FPGAs. This blocks are called MACs (Multiply, add and Accumulate), which are usually used in DSPs (Digital Signal Processing) and their architecture is shown in Fig. 5.2. The MACs blocks are perfect to perform the weighted sum.

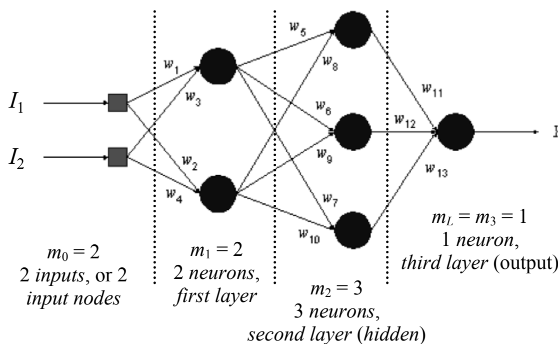

Fig. 5.1 Neural network with two inputs, three layers and one output Y

Recall that $nout(a)$ of (5.3) is the actual neuron output function our ANN hardware will perform. Observe that the computation involved in this function is sum of products (quadratic polynomial) and so the MACs can be used in this purpose to. Actually we use the same block of the neuron to compute the output function.

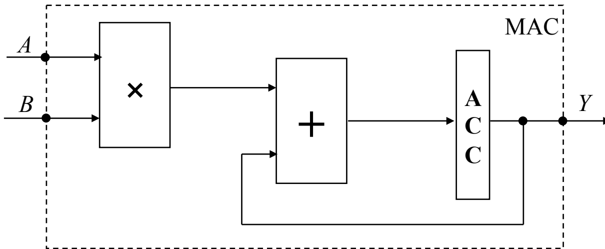

Fig. 5.2 Built-in MACs blocks in FPGAs

5.5 ANN Hardware Architecture

The ANN hardware interface is illustrated in Fig. 5.3, wherein two other components are included: LOAD CONTROLLER and CLOCK SYSTEM. The former may be any outside system able to setup the neural network topology and to load all necessary data in the ANN hardware. This include the number of inputs, the number of layers, the number of neurons per layer and that of outputs, besides, the net inputs and the definitive weights. Our ANN hardware is organized in a neural control unit (UC) and Neural arithmetic and logic unit (ALU).

Neural UC encompasses all control components for computing all neural network feed-forward computation. It also contains the memories for storing the net's inputs in the INPUT MEMORY, the weights in the WEIGHT MEMORY, the number of inputs and neurons per layer in the LAYER MEMORY and the coefficients of the output function in the OUTPUT FUNCTION MEMORY as described in (5.3). Fig. 5.4 and Fig. 5.5 depict, respectively, two parts of the neural UC.

During the loading process, which commences when $LCStart = 1$, the LOAD CONTROLLER sets signal *DataLoad* and selects the desired memory of the neural UC by signals $Load_0$ and $Load_1$ (see Fig. 5.3, Fig. 5.4 and Fig. 5.5).

The counters that provide addresses for memories are entirely controlled by the LOAD CONTROLLER. Signal *JKClk* is the clock signal (from CLOCK SYSTEM in Fig. 5.4) that synchronizes the actions of those Counters and of the LOAD CONTROLLER. This one fills each memory through the 32-bit *DATA* in loading process.

When the loading process is finished ($LCFinal = 1$), in Fig. 5.3, signal *DataLoad* can be turned off and the LOAD CONTROLLER can set signal *Start* for the commencing of the feed-forward Neural Network computing. When $Start = 1$ (and $DataLoad = 0$), the ANN hardware gets the whole control of its components; so the LOAD CONTROLLER can no longer interfere in the neural UC. This one has a

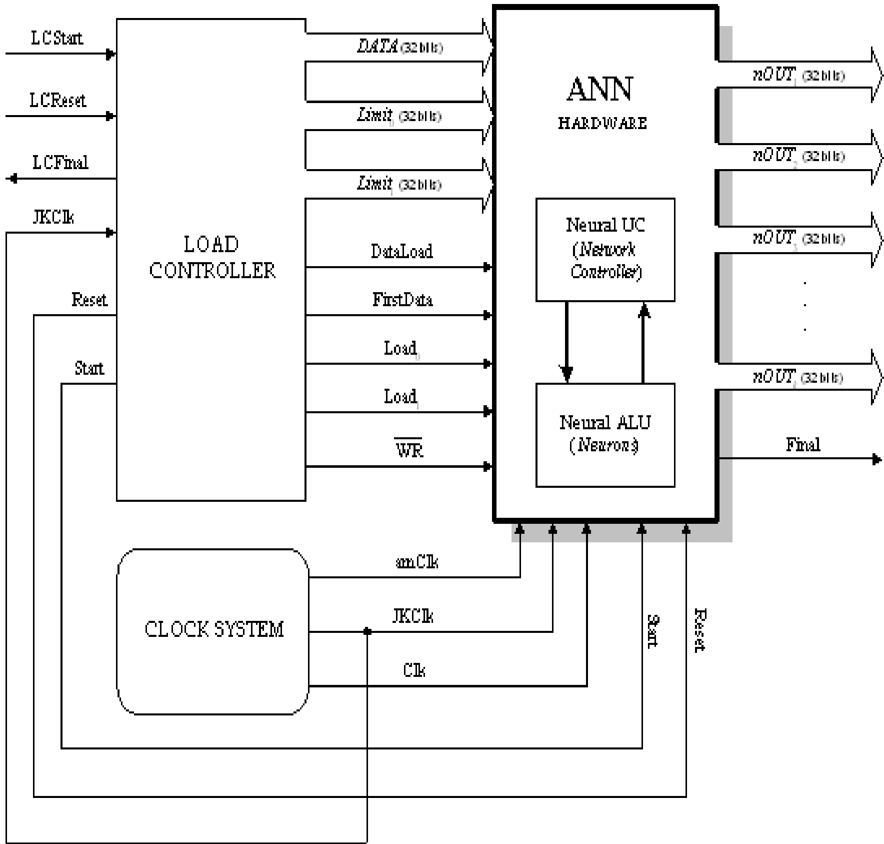

Fig. 5.3 Interface of the ANN hardware

main controller called *Network Controller* (Fig. 5.3) that controls all components of the neural UC (Fig. 5.4 and Fig. 5.5) and also the neural ALU, which is depicted in Fig. 5.6.

During the ANN hardware operation, neural UC by the mean of the network controller, controls the computation of each layer per stage. For each layer of the neural network, all k hardware neurons of the neural ALU of Fig. 5.6 work in parallel even though not necessarily all physical neurons are needed in the layer. Recall that some layers in the ANN hardware may have fewer neurons than k. At this stage, signal *Clk* is now the active clock of the ANN hardware, not signal *JKClk* anymore.

In Fig. 5.6, ADDER MUX decides the actual input for all hardware neurons and it is exploited to multiplex a network input, from the INPUT MEMORY in Fig. 5.4 or the output of a hardware neuron $nOUT_i$, which is an output of a neuron placed in a layer i of the net. While all physical neurons are in operation, the WEIGHT REGISTERS of Fig. 5.6 are already being loaded using signal W (see Fig. 5.4). These are the new

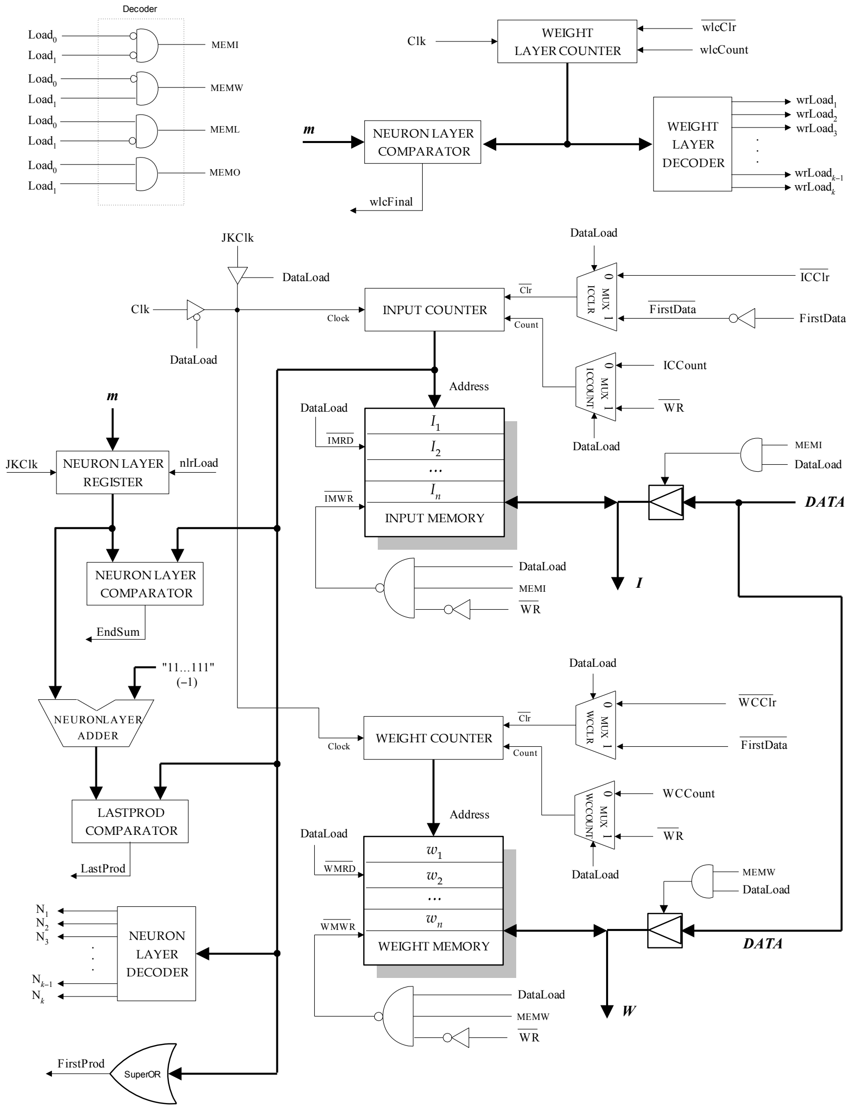

Fig. 5.4 First part of the neural UC

weights, which are the weights of the next layer in the network. Furthermore, in Fig. 5.6, we see a set of tri-state buffers, each of which is controlled by signal N_i, issued by the NEURON LAYER DECODER, in the neural UC of Fig. 5.4. Fig. 5.6 shows the neuron architecture. Each hardware neuron performs the weighted sum followed by the application of the output function $nout(a)$.

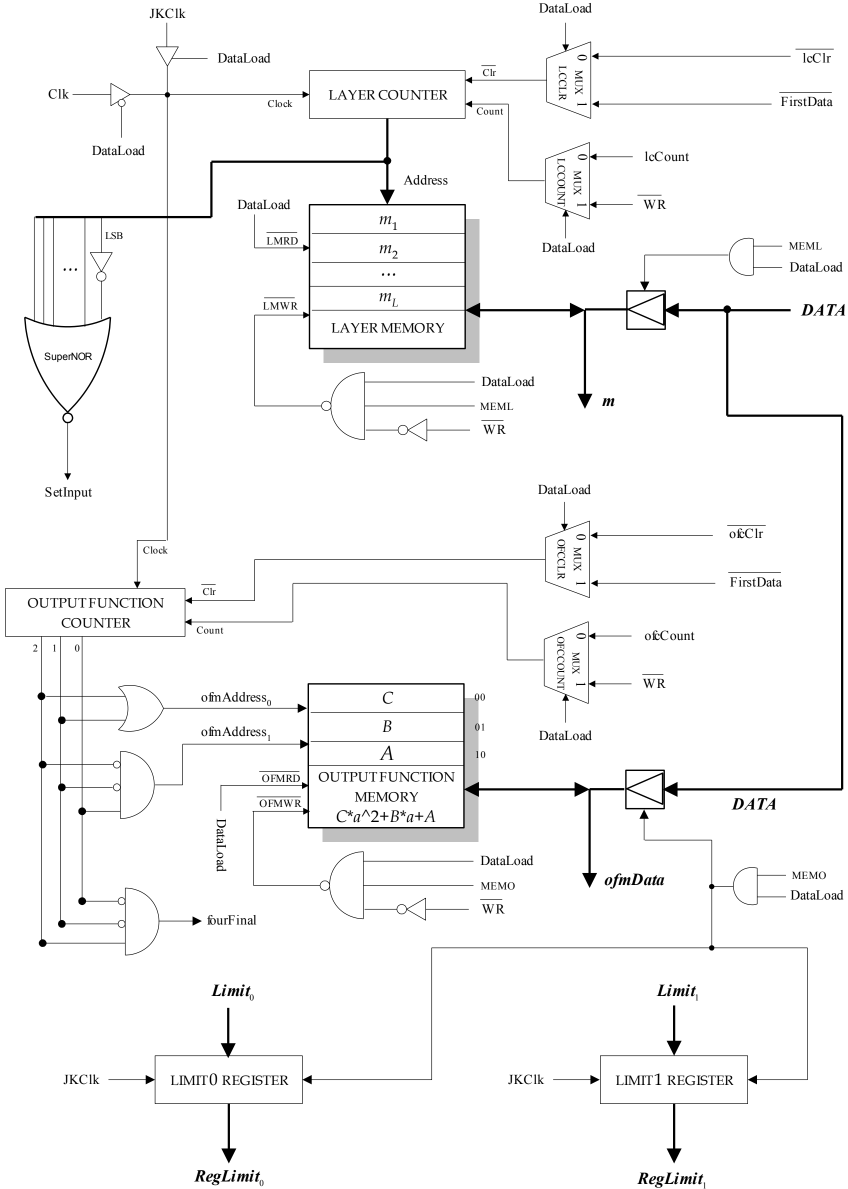

Fig. 5.5 Second part of the neural UC

Observing Fig. 5.4 (neural UC), the INPUT COUNTER, together with NEURON LAYER REGISTER, NEURON LAYER COMPARATOR, NEURON LAYER ADDER and LASTPROD COMPARATOR control the computation of the weighted sum: signal *FirstProd* indicates the first product of the weighted sum and *LastProd*, the last

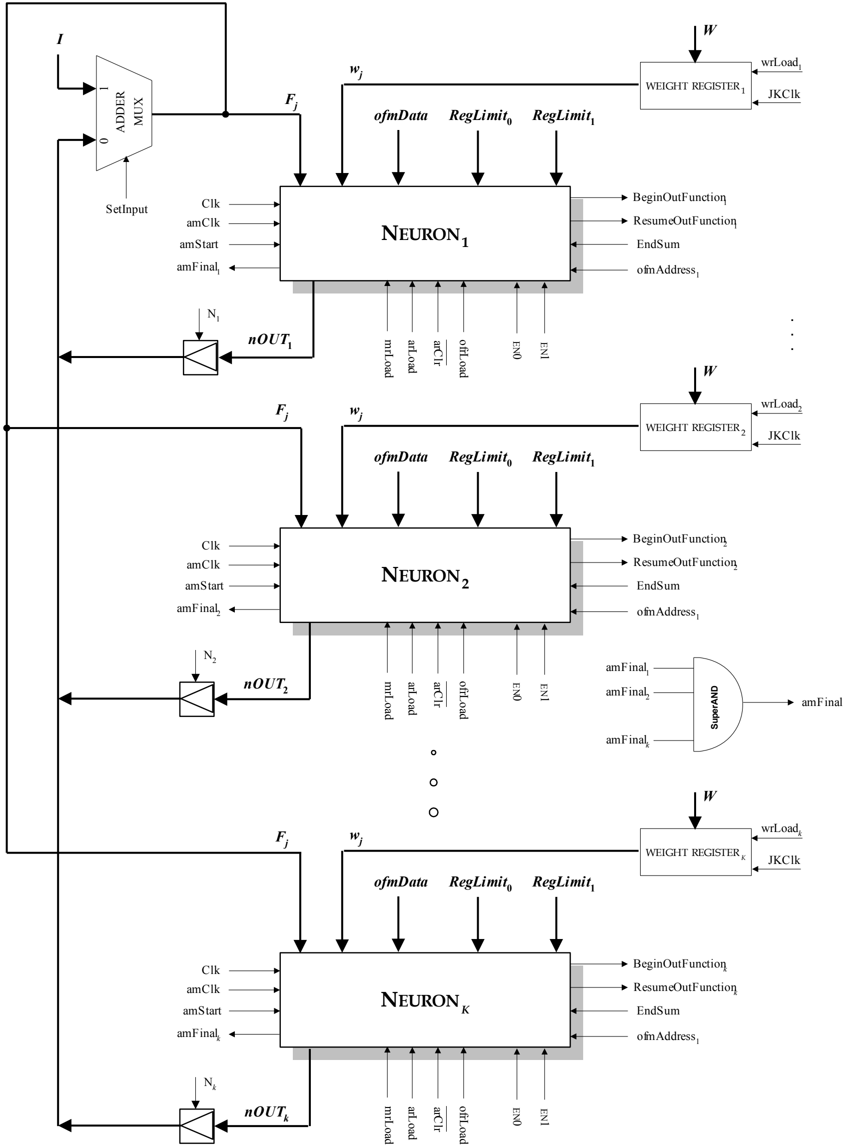

Fig. 5.6 Overall hardware architecture of the Neural ALU

one. SuperOR component is an OR of all input bits. Signal *EndSum* (Fig. 5.4, Fig. 5.5 and Fig. 5.6) flags that the weighted sum has been completed. It also triggers the start of the output function computation. During this stage, the OUTPUT FUNCTION COUNTER (see Fig. 5.5) provides the address to the OUTPUT FUNCTION MEMORY in order to release the coefficients ($C = 0.0217$, $B = 0.2155$ or $A = 0.4790$), through

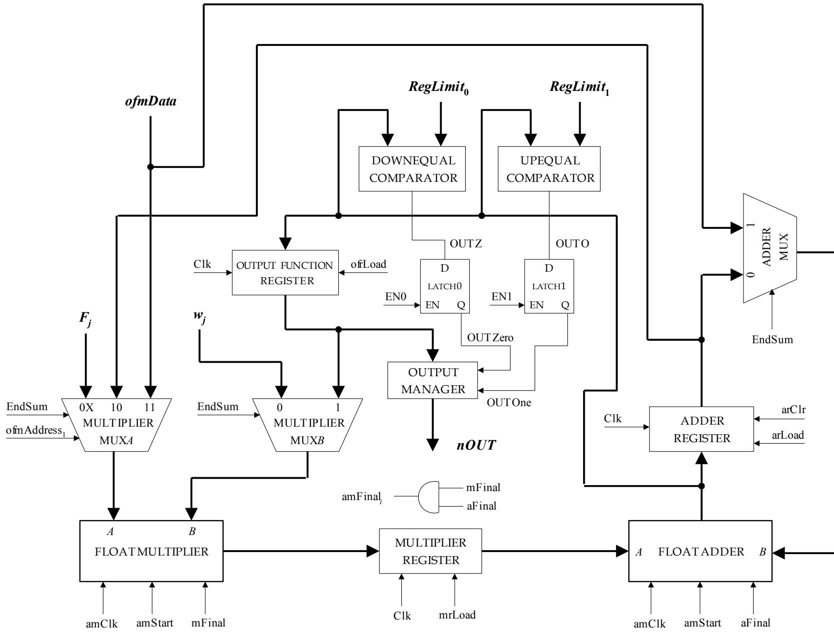

Fig. 5.7 Hardware architecture of the Neuron

ofmData, to the hardware neurons. Signal *fourFinal*, in Fig. 5.5, indicates that the computation of $nout(a)$ has finished.

Each hardware neuron encloses a MAC block, which consists of a FLOAT MULTIPLIER and a FLOAT ADDER to perform products and sums, respectively. The MULTIPLIER REGISTER allows the LOAD ADDER to works in parallel with FLOAT MULTIPLIER. The ADDER REGISTER accumulates the weighted sum. Recall that all hardware neurons work in parallel (see Fig. 5.6).

At an earlier stage, the LOAD CONTROLLER has loaded $Limit_0 = -3.3586$ and $Limit_1 = 2.0106$ in neural UC so that $RegLimit_0 = -3.3586$ and $RegLimit_1 = 2.0106$ have been obtained. Those float numbers refer to (5.3), wherein $nout(a)$ is 0 if $a < -3.3586$ and 1 if $a > 2.0106$.

In Fig. 5.6, which shows the hardware neuron, DOWNEQUAL COMPARATOR sets $OUTZ = 1$, if $a < -3.3586$ and UPEQUAL COMPARATOR sets $OUTO = 1$, if $a > 2.0106$. These components, intermediated by two latches, control the OUTPUT MANAGER, which decides as to the output of the hardware neuron ($nOUT$): *(i)* If $a \in [-3.3586, 2.0106]$, then $nOUT$ is the result of the second degree polynomial as described in (5.3), which is the content of the OUTPUT FUNCTION REGISTER; *(ii)* If $a < -3.3586$, then the OUTPUT MANAGER provides $nOUT = 0$; *(iii)* If $a > 2.0106$, then $nOUT$ is 0. Components $LATCH_0$ and $LATCH_1$ are used to maintain $nOUT$ stable. Signal $nOUT$ have to be kept during the computation of the weighted sum of a next layer neuron. Furthermore, in Fig. 5.6, signal $amFinal_i$

indicates the end of both a product and sum performed by the neuron. The multiplier and the adder operate in parallel, i.e. when the adder is accumulating the freshly computed product to the partial weighted sum obtained so far, the multiplier is computing the next product. In Fig. 5.5, signal *amFinal* indicates the end of all the computation in all neurons.

In Fig. 5.3, signal *Final* indicates that all computation required in all the layers of the network are completed and the outputs of the network have been obtained. These outputs are available signals $nOUT_1$, $nOUT_2$, …, $nOUT_h$ (see Fig. 5.3 and 5.7), where h is the number of neurons placed in the output layer of the Network, with $h \leq k$.

5.6 Summary

In this chapter, we presented novel hardware architecture for processing an artificial neural network, whose topology can be changed on-the-fly without any extra reconfiguration effort. The design takes advantage of the built-in MACs block that come for free in modern FPGAs. The model was specified in VHDL [5], simulated to validate its functionality. We are now working on the synthesis process to evaluate time and area requirements. The comparison of the performance result of our design will be then compared to both the binary-radix straight forward design and the stochastic computing based design.

References

1. Bade, S.L., Hutchings, B.L.: FPGA-Based Stochastic Neural Networks Implementation. In: IEEE Workshop on FPGAs for Custom Computing Machines, pp. 189–198. IEEE Press, Napa (1994)
2. Brown, B.D., Card, H.C.: Stochastic Neural Computation II: Soft Competitive Learning. IEEE Transactions on Computers 50(9), 906–920 (2001)
3. Hassoun, M.H.: Fundamentals of Artificial Neural Networks. MIT Press, Cambridge (1995)
4. Moerland, P., Fiesler, E.: Neural Network Adaptation to Hardware Implementations. In: Fiesler, E., Beale, R. (eds.) Handbook of Neural Computation, Oxford, New York (1996)
5. Navabi, Z.: VHDL: Analysis and Modeling of Digital Systems, 2nd edn. McGraw Hill (1998)
6. Nedjah, N., Mourelle, L.M.: Reconfigurable Hardware for Neural Networks: Binary radix vs. Stochastic. Journal of Neural Computing and Applications 16(3), 249–255 (2007)
7. Xilinx, Inc. Foundation Series Software, `http://www.xilinx.com`