uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Isabela Mona Pasca

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Neural Network Digital Hardware Implementation

TITRE DE LA THÈSE / TITLE OF THESIS

E. Petriu

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

D. Inkpen

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

T. Kwasniewski

V. Groza

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Neural Network Digital Hardware Implementation

by

Isabela Mona Pasca

A thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

in partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering

School of Information Technology and Engineering

Faculty of Engineering

University of Ottawa

December 2006

Your file Votre référence
ISBN: 978-0-494-49261-1
Our file Notre référence
ISBN: 978-0-494-49261-1

To my family

Abstract

This thesis presents a digital hardware implementation of an artificial neuron with learning ability using the QuartusII 5.1sp1 web edition software on Altera's University Program Development Board (UP2). The learning method implemented is neither backpropagation nor conjugate gradient, but the weight simultaneous perturbation. By combining this method with a pulse density system and using a Field Programmable Gate Array, an interesting artificial neuron hardware architecture is obtained. Finally, two applications of the neuron implementation are presented: an analog function and a digital function.

Acknowledgment

First of all, I would like to thank my thesis supervisors Dr. Emil M. Petriu and Dr. Diana Inkpen for their continuous support and encouragement during the whole period of my studies.

My warmest thanks go to my son Ionut and my husband Codrin, for their love, support, encouragement, and understanding; for the sacrifices they had to endure because of this thesis; and especially their patience during my work.

I am grateful to my brother-in-law Sever who provided me valuable technical suggestions and advices.

Finally, I would like to express my deep gratitude and appreciations to my mother for her unfailing love and encouragement during my whole life.

Table of Contents

List of Figures

List of Acronyms

ECG	Electrocardiography
EEG	Electroencephalography
PE	Processing element
NN	Neural networks
VLSI	Very large–scale integration
CCD	Charge-coupled devices
VHSIC	Very high speed integrated circuit
VHDL	Very high speed integrated circuit (VHSIC) hardware description language.
HDL	Hardware description language
CPS	Connections per second
LFSR	Linear feedback shift register
PRBS	Pseudo-random binary sequence
FPGA	Field programmable gate arrays
RAM	Random access memory cell
.vwf	Vector waveform file

Chapter 1

Introduction

1.1. Background

Two basic concepts have marked the development of the artificial neural networks. The first one was the use of a statistical mechanism to explain the certain class of recurrent network (an associative memory) [1]. The second concept was the introduction of the backpropagation algorithm for training multilayer neural networks [2].

Fast advances in digital computer technology permitted the hardware implementation of neural networks, a development that was not possible at the beginning of the artificial neural network field. These days, a large number and a wide variety of applications integrate neural networks [3].

For example, in robotics neural networks increased the visual capability of robots and controlled their trajectories and manipulators. Robots equipped with neural networks are also used in automobile automatic guidance systems, in manufacturing process control, and in many aerospace applications [3].

In medicine, neural networks are used to analyse breast cancer cells; to analyse electrocardiography (ECG) and electroencephalography (EEG) results; to design prostheses or

to optimize the transplant time of human organs; to improve the quality of medical management; and to achieve better test results in emergencies [3].

In telecommunications, specific neural networks are designed for real time translation of spoken language, for customer payment processing systems, or for compressing image data.

As they could increase the manufacturing productivity, reduce risks, and eliminate human errors in many fields like banking, defence, electronics, financial securities and transportation, the use of neural networks has been growing rapidly in the last years [3].

1.2. Artificial Neural Networks

The human brain consists of approximately 10^{11} biological neurons. Every biological neuron has four major components (figure 1.1) [3]:

- the dendrites, which bring the electrical input signals into the cell body,
- the body, which sums and thresholds the input signals from dendrites,
- the axon, which is the single long fibber that carries out the output signals from the body of the neuron to other neurons, and
- the synapse, which is the contact between the axon of one cell and a dendrite of another biological neuron.

There are approximately 10^4 connections between one specific neuron and other neurons.

When a neuron has enough energy, it "fires". This process means that the neuron sends an electrical pulse from its body to the next neuron, through its axon and its synaptic junction. Then, the destination neuron fires if it has enough energy at its inputs and the process goes on and on [3].

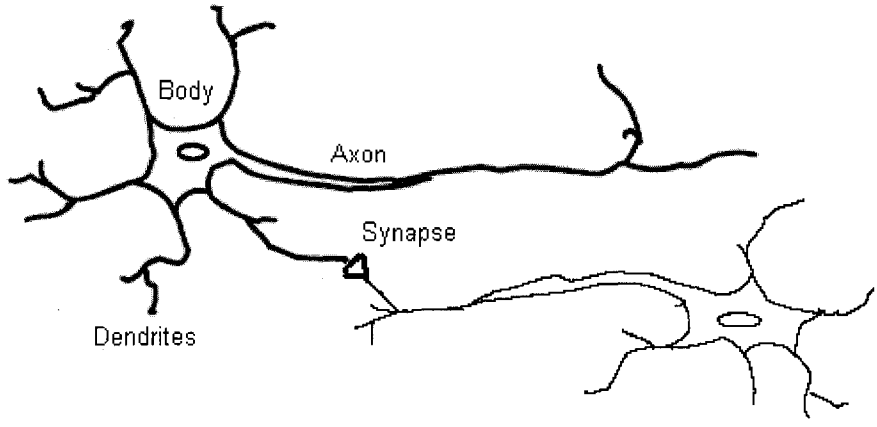
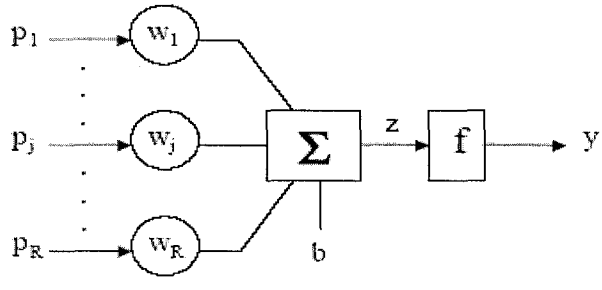


Figure 1.1. Biological neuron (from [3])

Even though the biological neurons respond slower than electrical circuits (milliseconds compared to picoseconds), the brain is able to solve many tasks in less than one second because of their essential property of parallelism, which means that all the neurons are operating at the same time.

Similar to the biological neuron, the artificial basic neuron, also called processing element (PE), consists of the following (figure1.2):



where:: $y = f(w_1 \times p_1 + \cdots + w_j \times p_j + \cdots + w_r \times p_r + b)$

Figure 1.2. Artificial neuron (from [3])

- $[p_1 \ldots p_R]$ - the inputs (corresponding to the dendrites from the biological neuron)
- $[w_1 \ldots w_R]$ - the weights (corresponding to the strength of a synapse from the biological neuron)

- b – the bias (can be treated as a weight whose input is always 1. Some authors use the name of offset instead bias).
- $y = f(w_1 \times p_1 + \cdots + w_j \times p_j + \cdots + w_r \times p_r + b)$ – the output of PE (represents the signal on the axon of the biological neuron)
- $z = (w_1 \times p_1 + \cdots + w_j \times p_j + \cdots + w_r \times p_r + b)$ – the summed output, often referred to as the net input
- f - an internal state of the neuron, called the activation function, which is a function of the inputs it receives. It is a nonlinear transfer function that produces the scalar output value y. Four of the most commonly used transfer functions are shown in figure1.3.

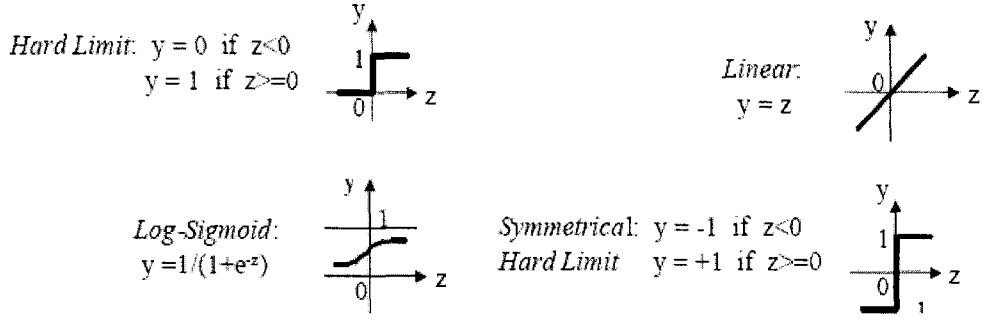


Figure 1.3. Examples of neuron transfer functions (from [3])

The artificial neuron performs the same basic tasks as the biological neuron:

- processes the input data,
- arranges this data into some manner, and
- carries out the information to the output.

Artificial neurons are interconnected to form a network. The network topology is defined from the way in which neurons are connected.

Most artificial neural networks (ANN) have a layered topology (figure 1.4) that consists of:

- an input layer that receives external inputs from outside of the network,
- an output layer whose output is the network output,
- one or more hidden layers that are the inner layers of a network, which do not receive inputs from outside and do not send out information of the network.

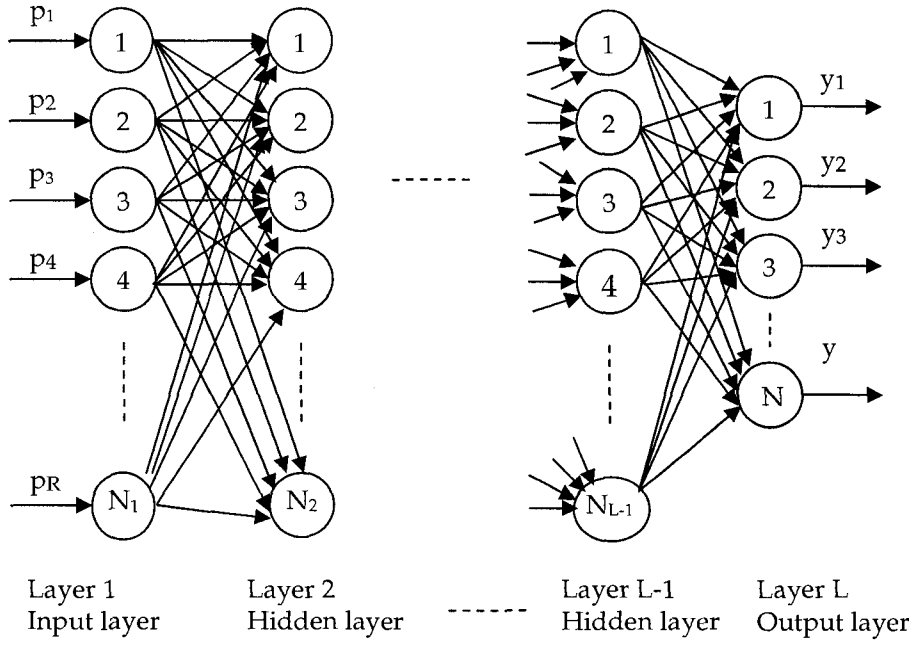


Figure 1.4. Artificial neural network with L Layers (adapted from [4])

There are two topologies of artificial neural networks (ANN) [8]:

- Feed-forward neural networks, in which the data flows from the input to the output layers in a strictly feed-forward manner and no feedback is allowed.
- Recurrent neural networks, in which the data flows from the input to the output layers through both feed-forward and feedback connections. Recurrent neural

networks are used for systems that have some important dynamical properties, which need to be captured through feedback connections.

The fundamental idea of an artificial neural network is that first it has to be trained with an input data set and then, the network should correctly recognize a similar but different input data set, which the network has not been presented before.

The operation of an artificial neural network consists of two important steps: training (or learning) and testing [13]. In the training process, the neural network is fed with a given input set, and it modifies the connection weights such that the error between the output of the network and the desired output is very small. The learning process takes a great deal of time as it requires an artificial neural network to be presented with many examples. The testing process shows how well the artificial neural network has learnt.

The power of an artificial neural network is given by the large number of interconnections (weights) between neurons and the adaptive nature of them.

In conclusion, the neurons are able to store information in their weights. This information is used to determine the output of the network for any set of input data. During the learning process the weights are modified over time in order for the system to accomplish the desired task.

1.3. Neural Network Implementations

The definition of neural networks given by the inventor of the first neurocomputer, Dr. Robert Hecht-Nielson, is as follows: "a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs" [5].

Neural networks are usually implemented as a software program on an ordinary digital computer. In this way the weights and activation functions are calculated serially, thus consuming a great deal of time. In real life there are many applications of neural networks that

request a short time to learn and respond. Unfortunately, the software implementation of ANN cannot utilize the fundamental property of parallelism found in biological neural networks. To cope with this drawback, artificial neural networks are implemented using hardware elements such as FPGAs [7, 8, 9, 10].

The rapid development of technology, particularly of VLSI circuits, makes it possible to implement different ANNs with different learning algorithms. Even though software implementations are characterized by high precision (given by the possibility of using floating point variables with a large number of decimals) and large dynamic range of neural network data (only limited by the maximum floating point number that can be stored in a computer), this thesis has chosen a hardware implementation, which represents signals and weights with limited precision (because it uses only integer variables or floating point variables with one or two decimals) and large noise (inherited to electronic circuits). The hardware implementation has the advantage of being much faster than a software implementation. This is because a software implementation uses serial computations, whereas in a hardware implementation the processes are parallel, as in a biological neural network, and are much faster [3].

Hardware implementations of neural networks consist of interconnected circuits that perform the basic operations needed by a neural network to learn. These operations are: multiplication, addition/subtraction, thresholding, and weight storage.

Until now, the most known hardware implementation technologies are: analog implementations, digital implementations, hybrid implementations, and optical implementations.

1.3.1. Analog Implementations

In analog implementations, signals are represented not by codes or numbers but by physical variables like voltage, current, pulse frequency, charge, or time duration [11]:

- Voltage representation: capacitors are used to store signals, active components sum voltages, and the transmission of signals between components is easily done through wires.
- Current representation: summing signal operations are done by adding currents based on Kirchhoff's law, but the transmission of signals is more complicated because the original current is never exactly equal to its replicas. New techniques that use translinear circuits and pseudoconductances eliminate some of the drawbacks of this implementation.
- Pulse frequency representation: uses pulses of fixed amplitude and duration. This is the main representation of signals in ANN.
- Charge representation: summing and subtraction operations require time sampling and are made using charge-coupled devices (CCDs), whereas multiplication is a more complicated operation in this case and is easier performed by first translating signals to voltage or current representation.

Some of the disadvantages of the traditional analog implementations are [11]:

- Data communications over long lines are very sensitive to random noise, such as thermal noise.
- Relatively poor precision of analog components due to manufacturing process variations, temperature variations, and aging. For example, capacitors lose their charge over time, which negatively influence the storage of weights in an ANN.
- Sensitive to non-random sources of noise, such as power supply noise and crosstalk, i.e. noise coming from other signals generated on chip.

1.3.2. Digital Implementations

Compared with analog implementations, digital implementations have the advantages of being much simpler and having a high signal-to-noise ratio. Also, digital circuits are much

more flexible and are easily cascade-able in comparison to analog designs. Moreover, at the present time, the fabrication price of digital implementations is quite low and many digital synthesis tools are available. Digital implementations are quite reliable [12].

In digital circuits, summation is realized by adders, whereas the non-linear transfer function can be implemented by either using small lookup tables or a dedicated circuit. Usually, multiplication is made by using timed multiplexers and is the most area-consuming operation. There are two forms of storage elements: static and dynamic, e.g. RAM cells [12].

Disadvantages of traditional digital implementations include: limited interconnectivity (even though latest FPGAs contain several tens of thousands of gates, implementations of ANN might require more), limited precision (variables are represented with maximum 10 bits), possible high power dissipation, and limited number of available input/output pins per chip [12, 18].

1.3.3. Optical Implementations

Analog and digital neural network implementations can manage large numbers of neurons, but their many two-dimensional interconnections consume a great deal of circuit area.

On the other hand, biological neural networks are fully three-dimensional, exploiting the extra dimension at all scales from microns to centimetres [14].

Optical implementations of artificial neural networks offer the possibility of three-dimensional interconnections. In some type of crystals as much as 10^{10} interconnections per cubic centimetre can be achieved.

One of the advantages of the optical implementations is the lower power dissipation compared to analog and digital implementations [14].

In optical artificial neural networks information is coded in light intensity, thus all variables have only positive values. This is a drawback, because usually neuron network algorithms are based on negative and positive weights.

There are two approaches in optical implementations. One is to transform light intensities into electrical signals and further process them with standard electronics. The second approach is to do all processing with optical devices [14].

In optics, summation is performed by using simple lenses that can provide fan-in. It is easier to add light than to subtract. Subtraction can be achieved by destructive interference of light, but this is a difficult operation because it requires mutually coherent light beams and very high physical stability of the components. Multiplication is accomplished by passing an optical beam through a semitransparent medium. The intensity of the transmitted (or reflected) beams is the product of the original intensity times the medium's transmittance (or reflectance) [14].

Even though they can somehow solve the number of interconnections problem by using the third-dimension, optical technologies have their drawbacks: they are quite expensive; they have not yet become sufficiently sophisticated; and the number of neurons that can be implemented is still quite low [13].

1.3.4. Hybrid Implementations

Fully analog implementations do not offer a reliable storage of weights and are very sensitive to a variety of noises. On the other hand, due to limited interconnectivity and precision, fully digital implementations cannot achieve the natural parallelism and compactness of the biological neural networks.

For these reasons, hybrid implementations combine both digital and analog technologies, so that their advantages can be summed up and achieve better performances.

1.3.5. Discussion

In order to obtain a high accuracy of results, a small circuit area and a high processing speed, when designing an ANN hardware implementation we should concentrate on

optimization of the balance of the many trade-offs involved in selecting the classes and sizes of the architecture, the learning algorithms, and the types of circuits.

In the last years, many ANN implementations have been built and some of them are available as commercial products. However, their performance is still far behind the performance of the biological neural networks [12].

1.4. Thesis Organization and Main Contributions

The thesis is organized into the following chapters:

Chapter 2 presents a basic introduction to the pulse technique and describes a few basic examples of pulse representation found in literature.

Chapter 3 introduces the simultaneous perturbation method and its advantages versus the widely used backpropagation method.

Chapter 4 focuses on the digital hardware implementation of a neuron. First, the block diagram of the neuron is introduced. Then, the specification and a description of the functionality of the units that compose the neuron are presented in detail. Finally, the simulation results for each unit are presented.

Chapter 5 describes two applications of this artificial neuron implementation. First, the analogical function $y = 1 - x$, and second the digital function XOR are implemented.

Chapter 6 summarizes the work done in this thesis and briefly suggests some directions for further research developments.

Appendix A contains the VHDL code of the state machine used in the neuron implementation.

Appendix B presents the examples of the input neuron implementation schematic diagram.

Appendix C summarizes each block with their signals used in the artificial neuron implementation.

The main contribution of this thesis is the study and development of a gate-level prototyping environment for modular hardware ANN architectures using an internal stochastic pulse data representation. We have used QuartusII 5.1sp1 web edition integrated development environment to design, compile and simulate the developed neuron circuits and test neural network examples before their FPGA implementation.

Chapter 2

The Stochastic Pulse Data Representation Technique

As stated in the previous chapter, analog signals can be represented by physical variables like pulse frequency. A representative selection of the existing pulse encoding techniques is shown in figure 2.1 [6, 16]:

- *Pulse Amplitude Modulation* – when the pulsed waveform is modulated in time and follows the variation of the given signal.
- *Pulse Width Modulation* – when the width of individual digital pulses follows the variation of the given signal.
- *Pulse Phase or Delay Modulation* – when two signals are used to represent each neural state, and the phase difference between these two waveforms follows the variation of the given signal, that is, the delay between the occurrences of two pulses is modulated.
- *Pulse Frequency Modulation* – when the signal is represented as the instantaneous frequency of the digital pulses, whose widths are equal.

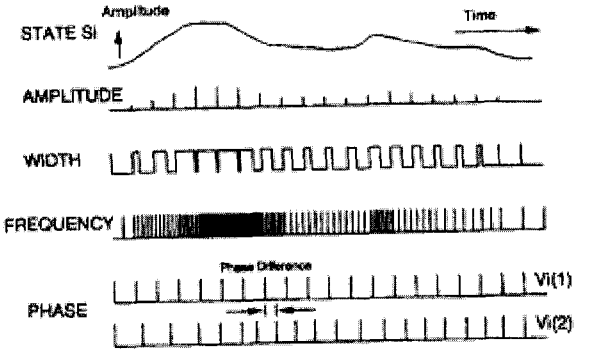


Figure 2.1. Pulse encoding technique (from [16])

This thesis uses a stochastic coded pulse frequency modulation technique. The fundamental characteristic of this technique is that it is based on the similarities between Boolean algebra and statistical algebra. The two basic arithmetic operations, summation and multiplication, can be easily implemented with simple digital gates. An ANN implementation based on the stochastic pulse technique presents a fairly good computational accuracy, a high packing density of electronic circuits, and is very well suited for FPGAs [26].

2.1. Data Representation

The probabilities of streams of random pulses were first presented by von Neumann in 1956. Beginning with 1960, several similar stochastic data processing concepts were reported, such as noise computer, random-pulse machine, and stochastic computing [17].

In the random-pulse machine theory the analog variables are represented by the mean rate of random-pulse streams. This particular representation can be viewed as the probability modulation of a random-pulse carrier by a deterministic analog variable [24, 39].

ANN is one of the applications of stochastic computing. In the stochastic pulse technique signals are represented by probabilities and are encoded in random pulse streams. These pulses are of fixed amplitude and duration (digital signals) and carry information [15].

For example, a number is represented by a long probabilistic bit-stream whose density of bits set to 1 is proportional to its numeric value. The main advantage of stochastic computing is the easy way in which the basic operations can be implemented. For instance, only a single 2-input logic gate is necessary for implementing the multiplication of two probabilistic bit-streams [10].

There are two stochastic representations [8, 9]:

- Unipolar – where a number A in the interval $[0, N]$ is mapped to a random binary variable with the generating probability of A, named P_A, given by:

$$P_A(A=1) = \frac{A}{N} \qquad (1)$$

For example, if $N = 1$ then the number 0 is converted to the bit-stream $00\ldots0$ and the number 1 to the bit-stream $11\ldots1$. The information carried in a stochastic bit-stream A corresponds to $P(A = 1) = P_A$.

- Bipolar – where a number A in the interval $[-N, N]$ is mapped to a random binary variable with the generating probability of A named P_A, given by:

$$P_A(A=1) = \frac{A}{2N} + \frac{1}{2} \qquad (2)$$

For example, if $N=1$ then number -1 is converted to the bit-stream $00\ldots0$ and the number 1 to the bit-stream $11\ldots1$. In the bipolar format, the information carried in a stochastic bit-stream A corresponds to $2P(A = 1) - 1 = 2P_A - 1$.

Since these two stochastic representations have quite different features, they are used according to the particularities of each neural network topology. If there is a need for a more accurate neural network, then a bipolar representation should be used, whereas for a faster computing neural network a unipolar representation is used [8, 9].

The following sections describe the architecture of the basic signal processing components for a stochastic arithmetic required in an artificial neural network. The typical operations needed to implement an ANN are digital to stochastic pulse conversion, stochastic pulse to digital conversion, multiplication, and summation.

2.2. Digital Data to Stochastic Pulse and Stochastic Pulse to Digital Data Conversions

Two conversions are introduced:

- Digital data to stochastic pulse conversion – a digital value is converted to a bit-stream.
- Stochastic pulse to digital data conversion – the converter generates an estimate for the probability of the bit-stream.

2.2.1. Digital Data to Stochastic Pulse Conversion

The digital data to stochastic pulse conversion is made by comparing the given digital value, which is stored in a register, with a random number. If the random number is less or equal than the given digital value, the output of the comparator is set to a high level. Otherwise, a low level is set [31, 32, 36]. Figure 2.2 shows the schematic diagram of a digital data to stochastic pulse converter.

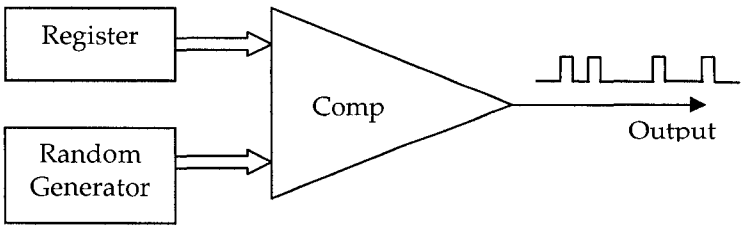


Figure 2.2. Digital data to stochastic pulse conversion (adapted from [26])

The probability P of a high-level output is defined as:

$$P(Output = 1) = \frac{Digitalvalue}{2^n}$$

The stochastic pulse stream at the output of the comparator is a Bernouilli's sequence with the probability of a high-level output equal to the given digital number stored in the register. According to Bernouilli's theorem, the mean of the addition of the Bernouilli's random variables tends to the above probability. Thus, the more pulses a circuit processes, the best accuracy is obtained when representing a digital value by means of a stochastic pulse stream.

A random pulse generator is one of the fundamental elements of a digital stochastic processing system [26].

Linear Feedback Shift Registers (LFSR)

Random numbers needed for the above conversion are obtained by using a linear feedback shift register (LFSR), which can be implemented with standard digital components [26].

There are known two main techniques to generate pseudo-random number sequences using linear feedback shift registers:

- *The Galois law* and
- *The Fibonacci law*

The *Galois* technique consists of a shift register, the content of which is modified at every step by a binary-weighted value of the output stage [8].

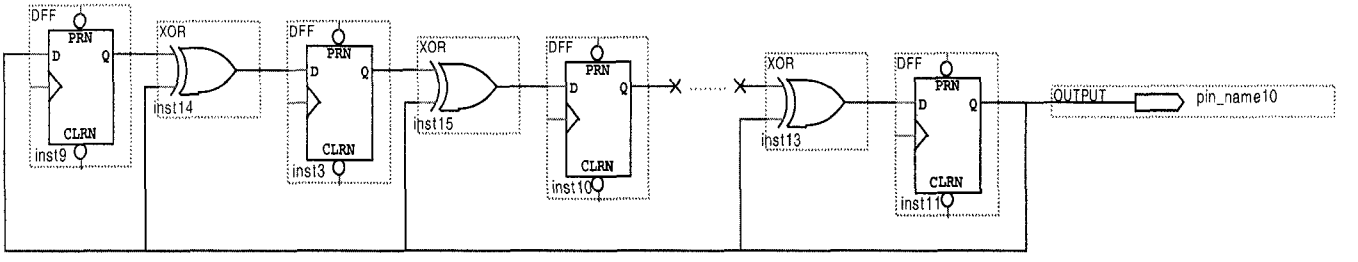


Figure 2.3. The Galois LFSR random number generation (adapted from [8])

The *Fibonacci* technique consists of a shift register in which a binary-weighted modulo-2 sum of the taps is fed back to the input. The modulo-2 sum of two one-bit binary numbers yields *0* if the two numbers are identical, and *1* if not [8].

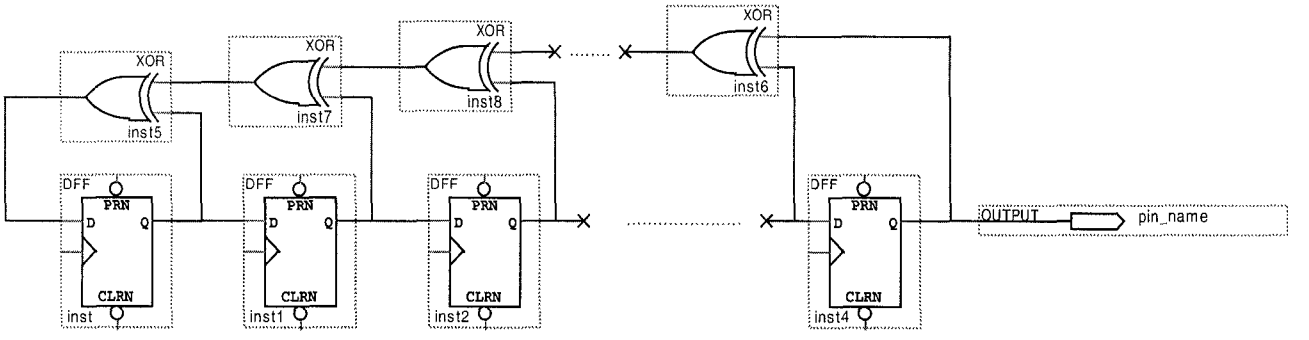


Figure 2.4. The Fibonacci LFSR random number generation (adapted from [8])

Galois or Fibonacci LFSR random number generation can be used to generate multiple pseudorandom bit sequences. The variables of such pseudorandom generators are the length of the LFSR and the taps from which the sequences are produced [8].

2.2.2. Stochastic Pulse to Digital Data Conversion

A given bit-stream cannot directly be converted to a binary digital number, but an estimate for the probability can be generated. A digital value can be recovered from a stochastic pulse stream by estimating its mean value in a long enough sequence. Statistically, the mean value estimator has a Gaussian distribution. This distribution has a mean value equal to the mean value of the discrete Bernoulli sequence μ, and a standard deviation σ that decreases with the root square of the length N of that sequence [27]:

$$Mean\ value\ estimator : AN\left(\mu, \frac{\sigma}{\sqrt{N}}\right)$$

The estimation is done by an up/down counter because probabilities can be both positive and negative. The stochastic pulse to digital data conversion, shown in figure 2.5, can be performed by counting the bits set to logic 1 in the stochastic stream X, whose sign is $sgn(X)$. NPx is the estimated number of bits in bit-stream X [9].

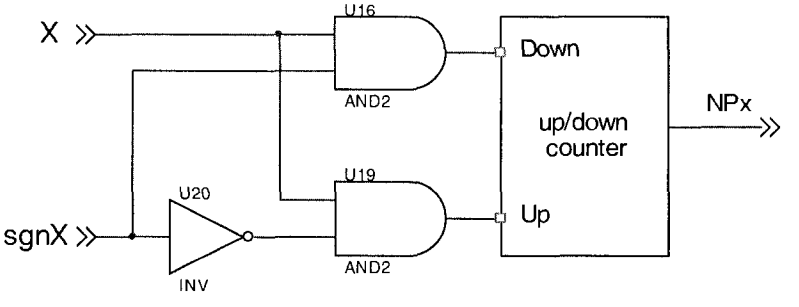


Figure 2.5. Stochastic pulse to digital data conversion (adapted from [9])

2.3. Stochastic Multiplication

This section presents the stochastic multiplication operation for the two above mentioned representations: unipolar and bipolar.

2.3.1. The Stochastic Multiplication of Two Bit-Stream Pulses with Unipolar Representation

The stochastic multiplication of two bit-stream pulses with unipolar representation is performed by an AND gate [33, 34, 35]. The input stochastic pulse streams have to be uncorrelated.

Figure 2.6 shows a stochastic multiplication of two input pulse-stream signals with unipolar representation, whose probabilities are P_A and P_B. The output pulse-stream has the probability P_C equal to the product of the input probabilities, which means that the output pulse-stream probability will have a high-level value only when both input pulse-stream probabilities have a high level value [26].

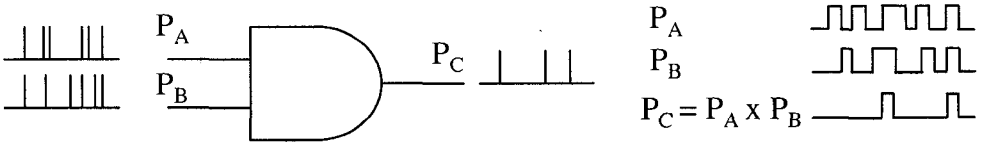


Figure 2.6. A stochastic multiplier using AND gate (adapted from [29])

For example, given the numbers A, B, and C, with $0 < A < V$ and $0 < B < V$ and where $AB = C$, the value of number C/V can be obtained by "AND"-ing the bit-streams that represent the numbers A and B. The probabilities of these bit-streams are P_A, P_B and P_C, respectively. The AND operation is defined as:

$$P_C = P_A \times P_B \qquad (3)$$

Using the equation (1) from section 2.1 for each A, B, C numbers, we obtain:

$$P_A = \frac{A}{V}, \qquad P_B = \frac{B}{V}, \qquad P_C = \frac{C}{V}, \qquad (4)$$

Introducing equations (4) in equation (3) and scaling these values with $1/V$ we obtain:

$$C = A \times B$$

2.3.2. The Stochastic Multiplication of Two Bit-Stream Pulses with Bipolar Representation

The stochastic multiplication of two bit-stream pulses with bipolar representation is performed by an XNOR gate.

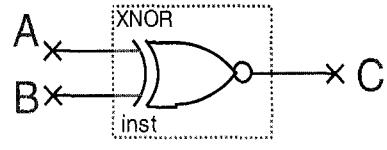


Figure 2.7. The XNOR gate used for the multiplication of two bit stream with bipolar representation

Figure 2.7 shows a stochastic multiplication of two input pulse-stream signals with bipolar representation, whose probabilities are P_A and P_B. The output pulse-stream probability will have a high-level value only when the input pulse-stream probabilities have different values [26].

For example, given the numbers A, B, and C, with $-V < A < V$ and $-V < B < V$ and where $AB = C$, the value of number $C/2V$ can be obtained by "XNOR"-ing the bit-streams that represent the numbers A and B. The probabilities of these bit-streams are P_A, P_B and P_C, respectively. The XNOR operation is defined as:

$$P_C = P_A \times P_B + (1 - P_A)(1 - P_B) \qquad (5)$$

Using the equation (2) from section 2.1 for each A, B, C numbers, we obtain:

$$P_A = \frac{A}{2V} + \frac{1}{2}, \qquad P_B = \frac{B}{2V} + \frac{1}{2}, \qquad P_C = \frac{C}{2V} + \frac{1}{2}, \tag{6}$$

Introducing equations (6) in equation (5) we obtain:

$$C = A \times B$$

2.4. Stochastic Addition

So far, all the operations were performed using basic gates, thus we would be tempted to try to implement a stochastic addition with an OR gate. Unfortunately, using a simple OR gate the result of a stochastic addition of two input pulse-stream signals with unipolar representation, whose probabilities are P_A and P_B and the output pulse-stream having the probability P_C is not actually an addition, but it is described the following formula: $P_C = P_A + P_B - P_A \times P_B$. Thus, the OR gate stochastic adder has the disadvantage of having a sigmoidal saturation property when a pulse overlapping occurs. The OR gate is only suitable for implementations in which low pulse density inputs are used [8].

Thus, the stochastic addition is not a simple addition. For example, the sum S of two or more unipolar or bipolar bit-stream signals representing values in the ranges of *[0, 1]* or *[-1, 1]* does not necessarily belong to *[0, 1]* or *[-1, 1]* respectively.

One of the simplest solutions to this problem is to use a multiplexer (figure 2.8) that randomly selects a given input I_i with the probability Sel_i such that $\sum_i Sel_i = 1$ will generate an output with a probability that is a scaled sum of the input probabilities. Using an *N-to-1* multiplexer, we have [8, 35]:

$$P_S = Sel_1 P_{I_1} + Sel_2 P_{I_2} + \cdots + Sel_n P_{I_n}$$

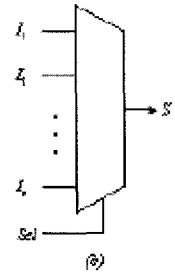


Figure 2.8. The summation of an n- bit stream with bipolar representation: performed by a multiplexer

The summation of two bit-stream signals with unipolar representation with $P_{I_i} = I_i$ is a scaled sum of the inputs [9]:

$$S = Sel_1 I_1 + Sel_2 I_2 + \cdots + Sel_n I_n$$

The summation of two bit-stream signals with bipolar representation is [9]:

$$S = 2P_S - 1 = 2\left(Sel_1 \frac{I_1 + 1}{2} + Sel_2 \frac{I_2 + 1}{2} + \cdots + Sel_n \frac{I_n + 1}{2} \right) - 1 =$$

$$= Sel_1 I_1 + Sel_2 I_2 + \cdots + Sel_n I_n + Sel_1 + Sel_2 + \cdots + Sel_n - 1 =$$

$$= Sel_1 I_1 + Sel_2 I_2 + \cdots + Sel_n I_n$$

In an ANN implementation, the sum of the weighted inputs is obtained by using an *N-to-1* multiplexer, whose inputs are selected randomly by a counter with a modulus equal to the number of inputs. This counter is driven by a single random bit in order to either increment or maintain its state each cycle, resulting in its random output [9].

Another solution to a stochastic addition is the use of a multiplexer (figure 2.9), whose inputs are selected by a random number generator consisting of a linear feedback shift register *(LFSR).* The *LFSR* acts as a "stochastic isolator", which means that it removes unwanted correlations between sequences with similar patterns. The outputs of the multiplexer are

statistically independent because its random scanning inputs are uniformly distributed and have the same probability [17, 33, 38].

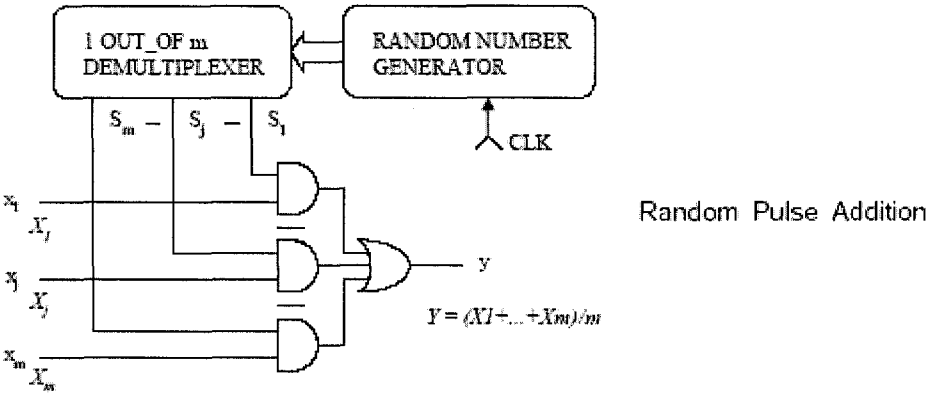


Figure 2.9. The stochastic addition circuit (from [38])

Another solution that uses multiplexers is the voting neuron structure, which is an improved version since it features a variance much smaller than that featured by stochastic neurons. In a conventional voting neuron circuit the neuron architecture fires whenever the number of the excitatory input pulses exceeds that of the inhibitory ones. In [28] the proposed neuron architecture is based on a voting machine, which uses two digital multiplexers and one shift-register, such that the numbering of the output shift-register cells represents the scale on which the difference between the excitatory and inhibitory pulses can be directly read after the neuron computation.

2.5. Advantages and Drawbacks of using the Pulse Density Representation and Stochastic Technique in ANN Implementations

In stochastic pulse density representation, signals are represented by probabilities and encoded in random bit-stream pulses. In this representation the adding and multiplication operations are stochastic processes that use pseudorandom pulse sequences. In this way, both

the product between inputs and weights, and the output of the neuron are represented as probabilities and estimated as average rates of pulse occurrences.

The advantages of ANN implementations based on stochastic computing are:

- Large, dense, fully parallel networks can be made with basic electronic circuits while maintaining a high computational accuracy [10, 15]. For example, multiplication of two probabilistic bit-streams can be accomplished with a single 2-input logic gate.
- They are well suited to VLSI technology, in particular FPGAs, thus offering a high packing density of electronic circuits [15].
- They can handle quantized analog values based on digital circuits [7, 26, 30, 37]. This is because analog variables can be represented by the mean value of the pulse stream [15].
- They are not vulnerable to noisy conditions. For example, variation in the amplitude of signals due to noise does not affect the number of pulses of the transformed signals because the weight of a single pulse in the pulse train signal is always unity.

Some of the disadvantages of ANN implementations based on stochastic computing are:

- The inherent variance of the output of a neuron is very large due to the stochastic computation. For example, when transforming a digital value into a pulse stream, we compare this value with different random numbers, thus we may obtain a different number of pulses for the output signal for each random number.
- Due to the above mentioned variance, the number of clock cycles required to accomplish a given computation (summation, multiplication, etc) is increased because we need to compute the mean value of the pulses [9].

- Weights and input signals are restricted to the range $[0.0,\ 1.0]$.
- The shape of the activation function realized by the statistical saturation is almost fixed [23].

Chapter 3

Simultaneous Perturbation as Learning Rule

Neurons store information in their weights, which are associated with each communication link. The value of the weights determines the output of the neuron for any of its given input. Neural networks can modify the weights during the learning process, which consists of the adjustment or adaptation of the network weights or parameters over time to enable the system to perform the task it has been designed for.

3.1. Perturbation Algorithms

The most popular algorithm for training multilayer networks is an approximate steepest descendent algorithm, named the backpropagation algorithm, in which the performance index is a mean square error.

However, implementations of large backpropagation learning algorithm based ANNs in analog hardware create serious problems because of the need of an accurate derivative of the activation function in the calculation of the gradients on the error surface.

Another disadvantage of backpropagation learning is that its implementation asks for separate circuitry for the backward pass of the algorithm.

In order to overcome these drawbacks, perturbation algorithms as learning rules that are better suited for hardware implementations have been designed.

The main idea of perturbation algorithms (proposed by Maeda in [7]) is to calculate a direct estimate of the gradients by a slight random perturbation of some network parameters (e.g. weights and inputs), using only the forward pass of the network to measure the resulting network error. By eliminating the complex backward pass, these algorithms yield to much more simpler and robust implementations [19].

There are two types of perturbation learning algorithms:

- Node perturbation - in which each gradient is estimated by perturbing the input value of the neuron. The Madaline-3 rule, the first node perturbation algorithm, was based on an application of the chain rule, which is standard in the deviation of the backpropagation algorithm. The disadvantage of this type of learning is that the node perturbations are performed serially, thus the complexity of the circuits increase significantly as compared with implementations based on backpropagation learning rule [20].
- Weight perturbation - in which each gradient is estimated by perturbing the weight value of the neuron [21]. The weight perturbation algorithm performs better when limited precision weights are used, but on the other hand, even though the addressing and routing circuitry is simpler, the weights are perturbed serially, thus a higher computational complexity is needed [18, 25]. The loss of parallelism can be addressed by perturbing all the weights of the neuron simultaneously and using the resulting error to update the weights. The weights have to be perturbed more than once in order to obtain a fairly estimate of a gradient descent weight, but the number of perturbations is quite small compared to the number of the weights of the network [26].

3.2. Simultaneous Weight Perturbation

The concept of weight perturbation can be understood by observing that the gradient descent weight update can be approximated by finite differences:

$$\Delta w^i = \alpha \frac{J(w + ce^i) - J(w)}{c} \qquad (1)$$

where:

- Δw^i is a modifying quantity for the i-th weight,
- w is a weight vector,
- $J(w)$ denotes the unperturbed error function and $J(w+ce^i)$ denotes the perturbed error function,
- c represents a magnitude of the perturbation,
- α is a positive coefficient (represents the learning coefficient), and
- e^i is the fundamental vector whose i-th component is 1 and the others are 0 [19].

In order to overcome the loss of parallelism, this thesis concentrates on the simultaneous perturbation learning rule instead of the simple weight perturbation.

The simultaneous perturbation learning rule computes the weight updates according to [7, 19, 30, 37]:

$$w_{t+1} = w_t - \alpha \Delta w_t \qquad (2)$$

$$\Delta w_t^{\ i} = s_t^i \frac{J(w_t + cs_t) - J(w_t)}{c} \qquad (3)$$

where:

- w_t is the weight vector, including the threshold of all neurons in a network at the i-th iteration,

- c is the magnitude of the perturbation,
- α is a positive constant (the learning coefficient),
- Δw_t is the modification of the weight vector w_t, with the component i given by Δw_t^i,
- s_i denotes the sign vector. If the sign is $+1$, then a positive perturbation $+c$ is added to the i-th weight, and if the sign is -1, then a negative perturbation $-c$ is used.

The value of the constant c is empirically determined and controls the magnitude of the perturbations.

The sign of $s^i{}_t$ satisfies the following statistical requirements [7, 19, 30, 37]:

- is randomly determined with a zero mean for each iteration.
- is independent of the sign of the j-th element $s^j{}_t$ of the sign vector. This means that a different sign is used for a different weight.
- its characteristic function has periodic zeros:

$$E(s_t^i) = 0, \quad E(s_t^i s_t^j) = 0 (i \neq j) \qquad (4)$$

where E is the expectation.

The error function $J(w)$ is defined as follows:

$$J(w) = |o - d| \qquad (5)$$

where:

- o is the output of the neural network, and
- d is the teaching signal corresponding to the output o.

Usually the error function is defined by a square error, but for simplification we use an absolute error.

In the simultaneous perturbation learning rule, the modifying quantities Δw_t of the weight vectors w_t are calculated using the unperturbed error function $J(w)$ and the perturbed error function $J(w+ce^i)$. This means that two forward operations are performed in the network. In this way we obtain $J(w+ce^i) - J(w)$, which multiplied by s^i/c yields to the right hand side of equation (3). This result, Δw_t^i, is an estimated gradient of the error function with respect to w^i. By repeating the same above calculation for $i = 1,\ldots,n$ all the weights are updated [7, 19, 30, 37].

Expanding the error $J(w_t+cs_t)$ at the weight w_t, there exist w_{s1}, such that

$$J(w_t + cs_t) = J(w_t) + cs_t^T \frac{\partial J(w_t)}{\partial w} + \frac{c^2}{2} s_t^T \frac{\partial^2 J(u(w_{S1}))}{\partial w^2} s_t \qquad (6)$$

Therefore, equation (3) becomes:

$$\Delta w_t^i = s_t^i s_t^T \frac{\partial J(w_t)}{\partial w} + \frac{cs_t^i}{2} s_t^T \frac{\partial^2 J(u(w_{S1}))}{\partial w^2} s_t \qquad (7)$$

Taking into consideration the conditions from equation (4), an expectation for the quantity in equation (7) is:

$$E(\Delta w_t^i) = \frac{\partial J(w_t)}{\partial w_t^i} \qquad (8)$$

This means that the modifying quantity is an estimate value of the first differential coefficient:

$$\Delta w_t^i \approx \frac{\partial J_p(w_t)}{\partial (w_t^i)} \qquad (9)$$

In conclusion, in order to obtain estimators of the first differential coefficients of the error function with respect to all weights in the network, the simultaneous perturbation learning rule requires only two forward operations through the ANN [7, 19, 30, 37].

3.3. Advantages and Disadvantages of using the Weight Simultaneous Perturbation as Learning Rule

The simplicity of an ANN implementation based on the weight simultaneous perturbation optimization method derives from the following considerations:

- The weight simultaneous perturbation method can estimate the gradient using only two values of the error function [7, 19, 30, 37].
- This learning rule requires only forward operations of networks and does not have to calculate the backpropagation error, thus only one circuit realizing the learning mechanism is used to update all the weights, even if the number of weights is large [7, 19, 30, 37].
- The method can be implemented with ordinary logical elements used in common digital circuits [7, 19, 30, 37].

The disadvantages of the weight simultaneous perturbation algorithm are:

- Limited precision weights, which has to be in the interval $[0, 1]$.
- Not efficient for neural networks with a large number of neurons [19].

3.4. Learning Mode Implementations

Any hardware implementation of a neural network needs to optimize the trade-off between three main constraints: high accuracy, compactness, and processing speed [22].

High accuracy leads to high resource consumption and the hardware implementation does not present a very fast computational speed [18].

Therefore, hardware implementations of neural networks typically use a representation of the network parameters with a limited accuracy. There are three methods to deal with the limited accuracy problem and they are divided according to three different training modes for the neural network hardware implementation [18]:

3.4.1. Off-Chip Learning

In this case, the learning process is not accomplished in the hardware implementation, but instead it takes place on a high-precision computer. The learning process results, e.g. the weights, are then quantized and downloaded onto the chip at the end of iterations, thus only the forward pass in the recall phase is performed on-chip [18].

3.4.2. Chip-In-The-Loop Learning

In the chip-in-the-loop learning implementation, the neural network hardware is also only used in the forward propagation and the weights are calculated on a computer, but the updated weights are downloaded onto the chip after each training iteration. In order to increase the accuracy of the system, special functions that transform the weights into discrete values are used [18].

3.4.3. On-Chip Learning

In the on-chip learning implementation, the training process of the neural network is accomplished entirely on-chip, without the use of an extra computer. Since the learning is done only on-chip, the advantage of this technique is that the neural network can be continuously trained, but the drawback that results is that the weight values are represented with limited precision [18, 31].

Chapter 4 presents a two-input neuron digital implementation with the ability of on-chip learning based on the weight simultaneous perturbation method. The inputs, weights, and outputs are described by pulse density signals. The basic computing operations needed in this implementation are based on stochastic logic. The implementation is made on an Altera Field-Programmable Gate Arrays (FPGA).

Chapter 4

Digital Hardware Implementation of a Neuron

As previously stated, the property of biological neural networks of being massively parallel in structure is not met in artificial neural networks (ANN) that are implemented as a software program on an ordinary digital computer. To deal with this drawback, a hardware implementation of neural networks must be used.

Any ANN hardware implementation must satisfy four basic criteria:

- to allow building systems with large numbers of artificial neurons;
- to permit the neurons of the net to make large number of connections to other neurons;
- the weights of the neurons must be changeable such that the ANN can learn and adapt;
- the ANN needs to be implemented in a small package with a manageable dissipated amount of power [13].

At the same time the hardware implementation of an ANN has to optimize three main constraints: accuracy, space and processing speed [22]. Although the analog implementation of the ANN is characterized by small area and low speed, the accuracy of the network

components are limited. On the other hand, digital implementations using hardware elements such as Field-Programmable Gate Arrays (FPGAs) offer a better optimization of the three previous constraints.

With this in mind this thesis describes a digital implementation of an artificial neuron using Field-Programmable Gate Arrays (FPGAs).

The key advantages of a digital implementation using FPGAs are [10]:

- provides a reprogrammable hardware resource;
- can be personalized to specific ANN configurations;
- reduced hardware development cycle;
- low cost.

I chose the Altera's University Program Development Board (UP2) that actually contains two CPLD development boards in one. The Altera MAX 7128S series chip on the left side of the board has a gate count of over 2,500, while the FLEX10K70 series chip on the right of the board has over 70,000 gates [39]. Since a digital implementation needs a great deal of resources, this thesis uses the FLEX chip (EPF10K70RC240-4) [39].

The entire implementation uses the MegaWizard Plug-In Manager megafunctions in the QuartusII software version 51sp1 from Altera installed on a PC that has a 1.8 GHz Pentium 4 microprocessor, 512 MB RAM and is running Windows XP operating system.

Megafunctions are standard performance-optimized functions for Altera devices that have the following advantages [40, 41]:

- provide efficient logic synthesis and device implementation;
- offer improved performance compared to basic logic elements;
- automate the coding process saving design time.

The overall configuration of one artificial neuron is shown in figure 4.1. Basically, the artificial neuron consists of one or more Weight Units and a Neuron Unit. In figure 4.1, the neuron is connected to a Learning Unit and a Command and Control Unit, which are needed

for its proper functioning. Hence, an ANN implementation can have numerous weight and neuron units but only one learning unit and one command and control block. The weight unit calculates the product of the input signals and weight value. The neuron unit calculates the thresholding stochastic weight sum of inputs. The learning unit calculates the quantity of modification for all weights and sends it to all weight units. This quantity has the same value for all the weights.

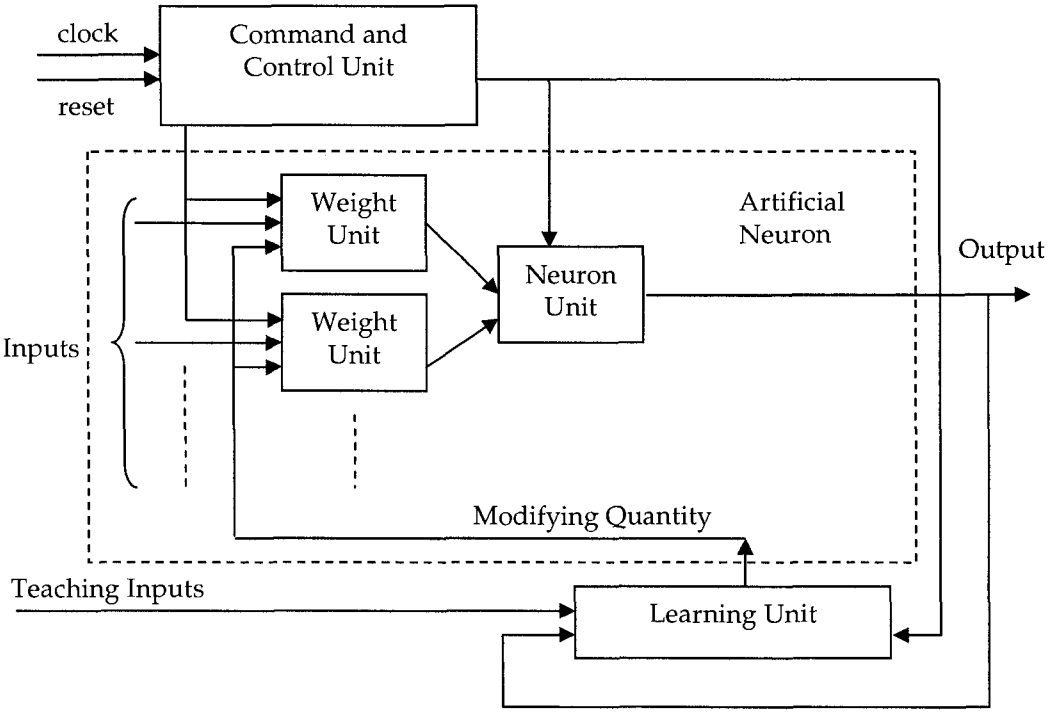


Figure 4.1. One artificial neuron and the units needed for its functioning

The simulation of each unit of the artificial neuron was executed with the Quartus II Simulator, which can perform two types of simulations: functional and timing.

The functional simulation verifies the logical operation of the design without taking into consideration the timing delays in the FPGA. This simulation is performed using only register transfer level (RTL) code. In contrast, timing simulation verifies the operation of the

design taking into consideration timing information. This simulation is performed using the post place-and-route netlist.

Figure 4.2 (part a) and figure 4.3 (part b) show the implementation of one artificial neuron with two weights, whose block diagram is presented in figure 4.1. The entire design has been implemented on Altera's University Program Development Board (UP2). As can be seen, we need just one unit for *commands and controls (inst1)*, one block for *learning (inst6)*, one unit for *neuron (inst)*, and two units for *weight (inst2*, and *inst3)*. The four blocks *inst10*, *inst12*, *inst13*, and *inst14* are not mandatory, as they implement the negative or positive bias of the neuron.

The next sections will present the schematic diagram, the functionality and the simulation results of each unit of this implementation.

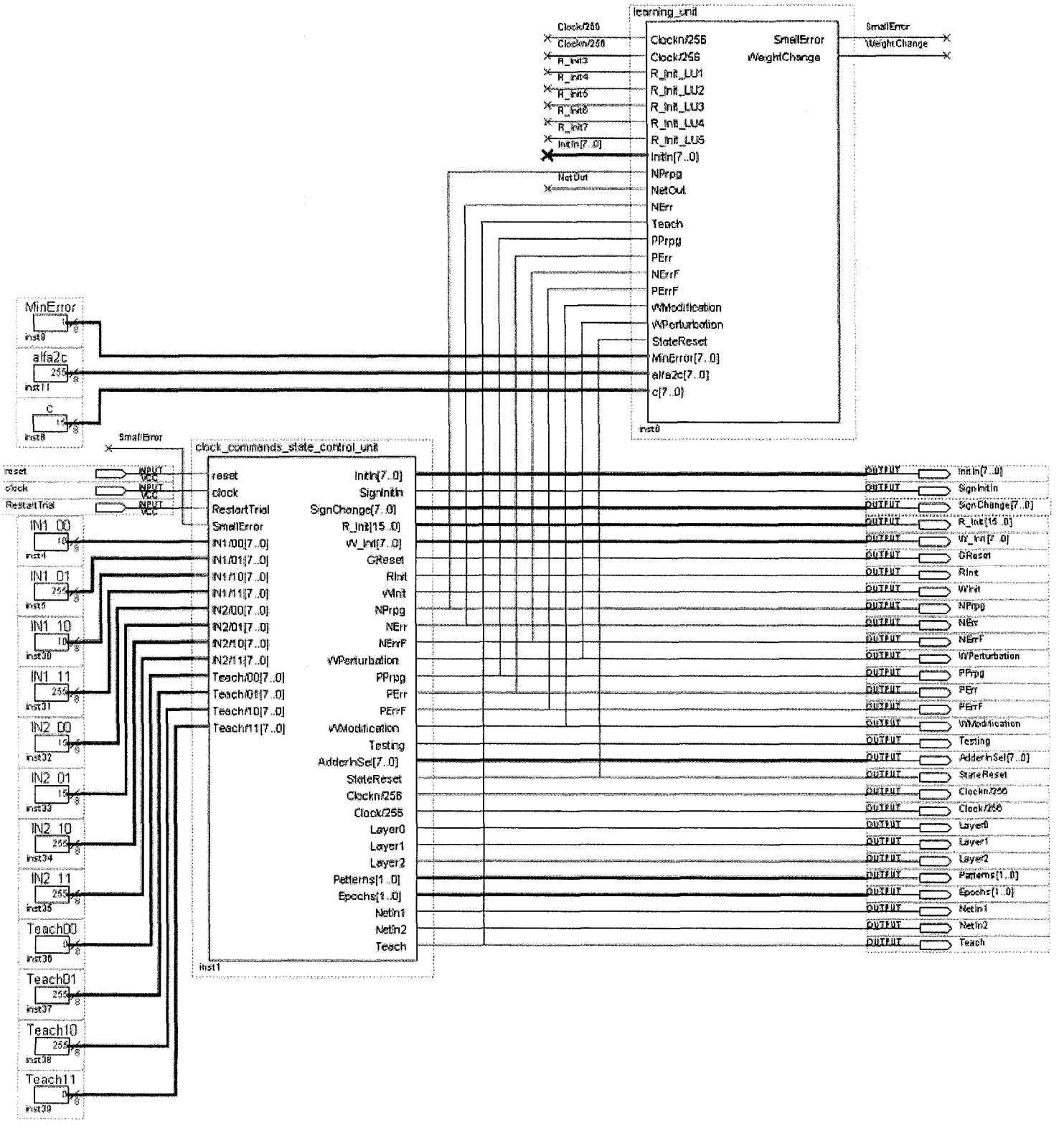


Figure 4.2. The two-input neuron - schematic diagram (part a)

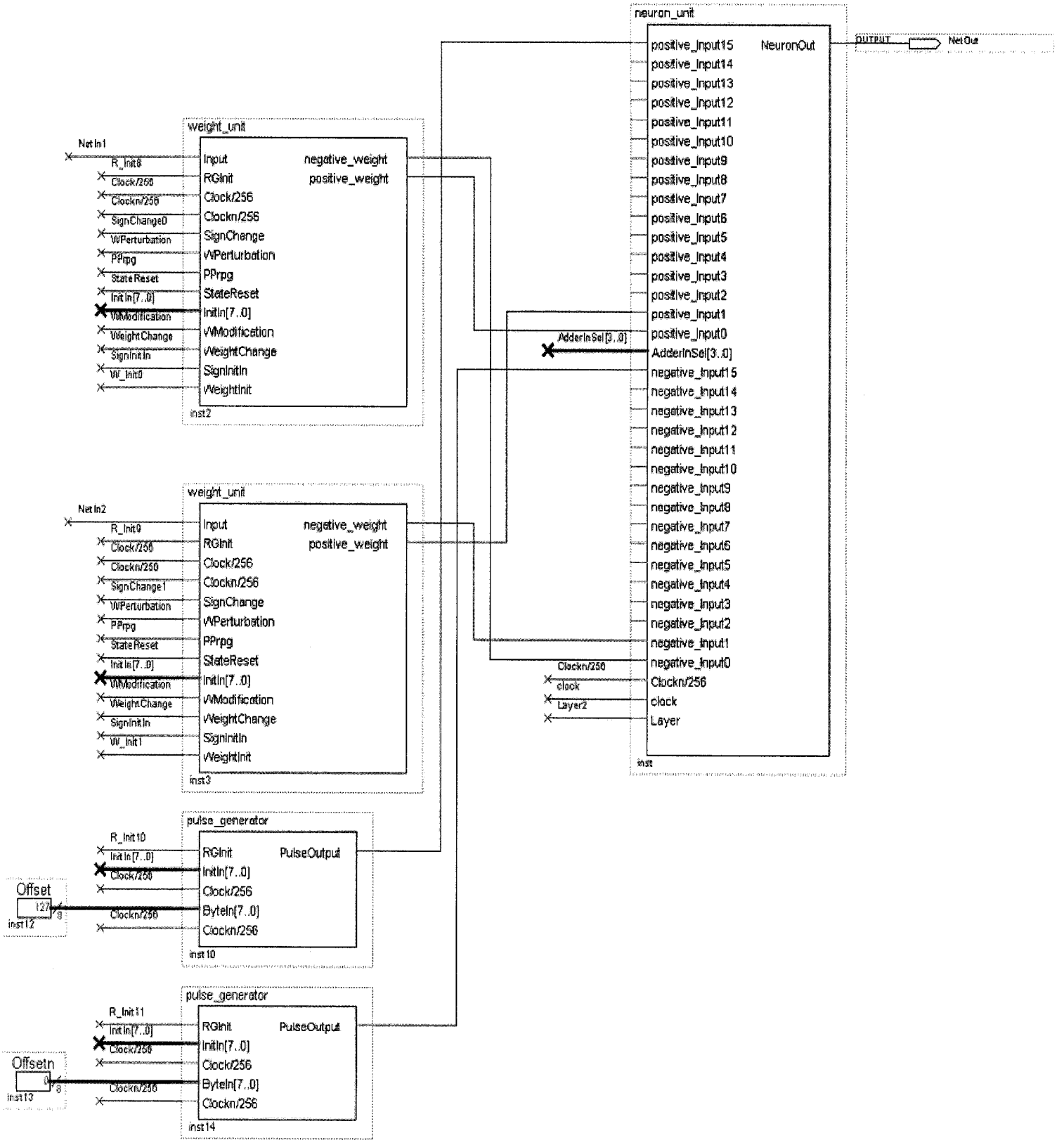


Figure 4.3. The two-input neuron - schematic diagram (part b)

4.1. The Clock Unit

Regardless of the number of neurons and the number of weights of a neural network, the *clock_unit* is the only timing block of the neural network implementation. Its purpose is to feed the entire neuron implementation with clock signals. The symbol of the *clock_ unit* is shown in figure 4.4.

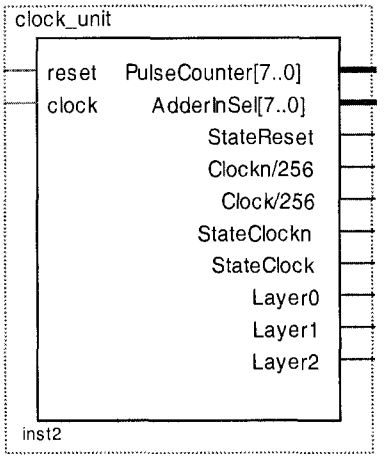


Figure 4.4. The clock_unit symbol

Figure 4.5 shows the clock signals that need to be implemented. The *clock* and *reset* signals are the inputs to the unit. After every 256 pulses of *clock*, one pulse of the *Clockn/256* is obtained. It is proposed that one state to last for 255 pulses of the *Clockn/256* signal.

Two more signals are defined: *StateReset* and *StateClock*, which will indicate the beginning and the end of a state.

Finally, we need some other signals which match the number of layers in which the neurons are organized. For example, for an ANN with 3 layers, we need three signals: *Layer0*, *Layer1*, and *Layer2*.

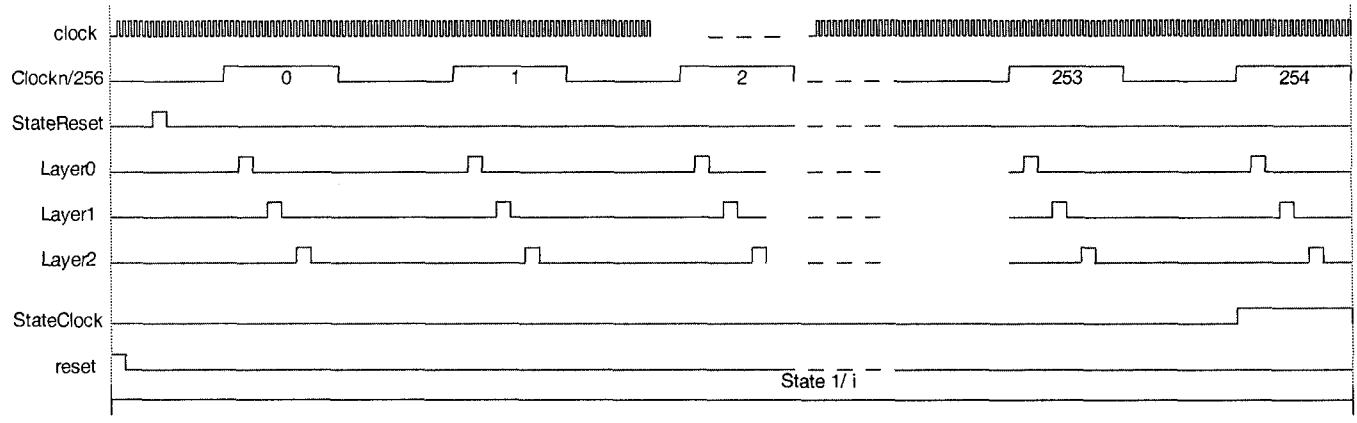


Figure 4.5. The clock timing diagrams to be implemented

4.1.1. Description and Functionality

The clock_unit is shown in figure 4.7 and consists of:

- *clock_counter* - is a binary up counter, with an 8-bit wide output *q[7..0]* and a carry-out *cout* port. It has two inputs: *clock* and synchronous clear *sclr*.
- *clock_counter255* – is an 8-bit up direction counter *q[7..0]*. The modulus of the counter is set to 255. It has synchronous clear *sclr* and carry-in *cin* input ports.
- *decode_8to1* – is an 8-line-to-1-line decoder that has 8 inputs and 1 active-high output. The data input is treated as an unsigned binary encoded number. The *decode_8to1* decodes only the state 0.
- *decode_8to254* – is an 8-line-to-1-line decoder that has 8 inputs and 1 active-high output. This decoder outputs one active-high pulse when *clock_counter255* reaches binary 254. The data input is treated as an unsigned binary encoded number.
- *decode_3to4* – is a 3-line-to-4-line decoder that has four active-high outputs, used with an output latency of 1 clock cycle. It also has three data inputs and one enable input. The enable input shows that all outputs are low when not active. The data

input is treated as an unsigned binary encoded number. The *decode_3to4* decodes only state 1, state 4, state 5 and state 6.

The clock_unit has only two input signals: *reset* and *clock.*

- *reset* is an input that clears the registers at power on. This reset signal can be obtained with a simple circuit that consists of a capacitor (C1) and a resistor (R10) shown in figure 4.6.

  At power on, the capacitor C1 is discharged, thus the inverter gate has an active-high output (*reset* condition). Then the capacitor C1 begins charging and after the R10*C1 time-out the voltage reaches the logic 1 level that is applied to the inverter gate, which generates a 0 logic at its output (*reset* is inactive).

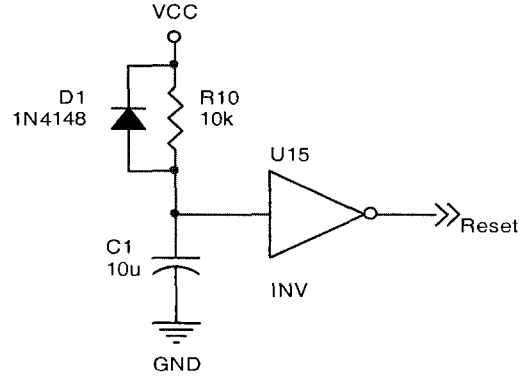


Figure 4.6. The additional reset circuit

- *clock* is a positive-edge-triggered signal that is the general clock of the entire neuron network implementation. It is an external signal.

The output signals of the clock unit are:

- *PulseCounter* is an 8-bit bus *q[7..0]* that counts from 0 to 254.
- *AdderInSel* is an 8-bit bus *q[7..0]* that counts from 0 to 255.
- *StateReset* is a positive pulse that appears in each state cycle on the first active low level of *Clockn/256.*

- *Clockn/256* is identical to *AdderInSel7*, which is the most significant bit of the 8-bit bus *AdderInSel[7..0]*.
- *Clock/256* is $\overline{Clockn/256}$.
- *StateClock*, is the 255th positive pulse of *Clockn/256* and it is obtained by decoding the value 11111110 of *PulseCounter[7..0]*.
- *StateClockn* is $\overline{StateClock}$.
- *Layer0 i*s a short positive pulse that appears in each state cycle on the first active high level of *Clockn/256*.
- *Layer1 i*s a short positive pulse that appears in each state cycle on the first active high level of *Clockn/256*, but after *Layer0*.
- *Layer2 i*s a short positive pulse that appears in each state cycle on the first active high level of *Clockn/256*, but after *Layer1*.

On the negative edge of the *reset* signal, *clock counter* starts to count up. When it reaches the maximum number of the pulses (256), the next counter *clock_counter255* starts counting. This second counter counts up only to 255 pulses. The output *PulseCounter[7..0]* of *clock_counter255*, is decoded by *decode_8to1* and *decode_8to254* in order to obtain *StateReset* and *StateClock*. The purpose of the AND gate *inst4* is to narrow the width of the *StateReset* pulse so that it equals the width of the *AdderInSel4* pulse.

On the positive edge of the clock of the D flip-flop *inst1* and when *AdderInSel7* is high, the output of the D flip-flop (*StateClock*) goes to zero.

On the positive level of the *AdderInSel4* signal, *decode_3to4* is controlled by the selection signal *AdderInSel[7..5]* to decode only the states 1, 4, 5, and 6. In this way, on each high level pulse of the *Clockn/256* signal, we obtain four successive pulses of width equal to the positive level of the *AdderInSel4*: *StateReset*, *Layer0*, *Layer1*, and *Layer2*.

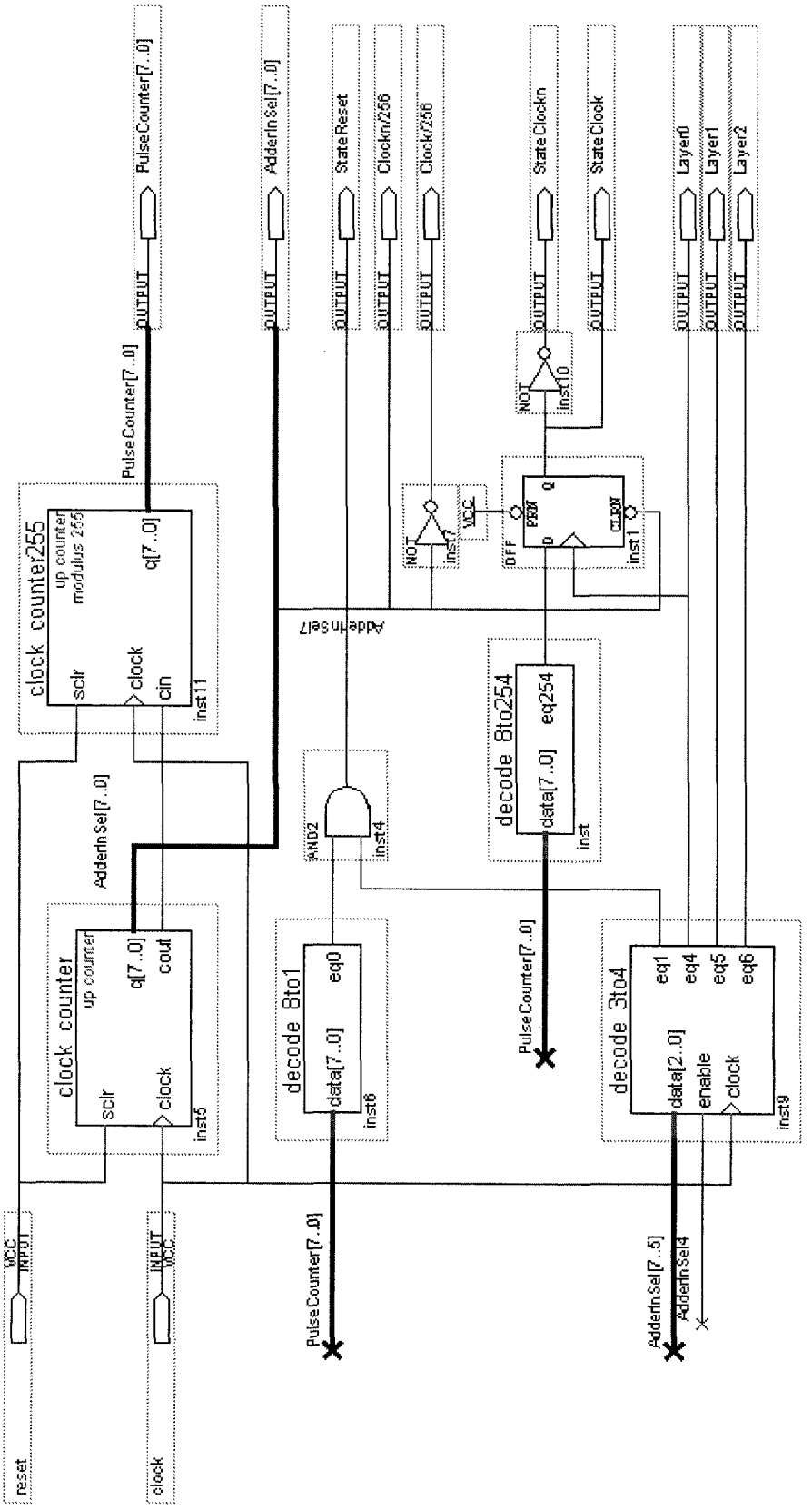


Figure 4.7. The clock_unit – schematic diagram

4.1.2. The Simulation Results

Prior to running a simulation, we must create a vector source file, Vector Waveform File (.vwf), which is the source of simulation input vectors. This file is used by the Quartus II Simulator to generate the output signals that a programmed device would produce under the same input conditions.

The Vector Waveform File contains the following:

- Maximum frequency f_{max} = 20 MHz
- End time of simulations = 300 ms
- *reset* after 6 *clock*'s pulses
- The *clock* signal frequency f = 5 MHz, which means T = 200 ns
- One state = 255 * (256 *clock* pulses) = 255 * (200 ns x 256) = 255 * 51200 ns = 13.1 ms

The simulation has been made on the ALTERA FLEX 10K family EFP10K70RC240_4 FPGA device.

The simulation lasted 2 minutes and 17 seconds. The implementation of this unit uses 35 total logic elements and needs 26 total input/output pins. The simulated waveforms are shown in detail in figure 4.8 and 4.9.

By comparing these figures with the signals in figure 4.5 we can see that the simulated output signals are the same as those to be implemented.

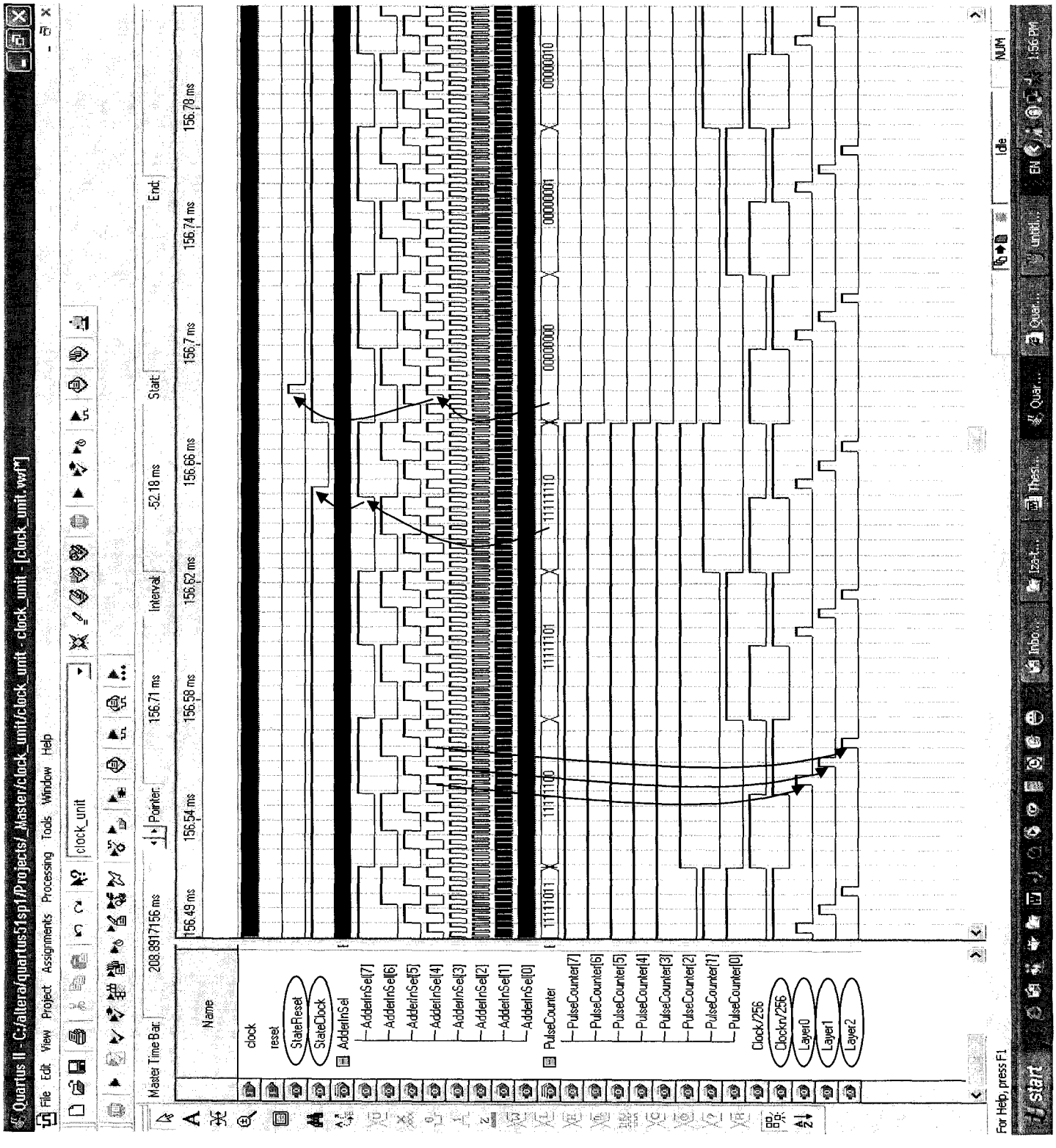


Figure 4.8. The clock_unit – wave diagram 1

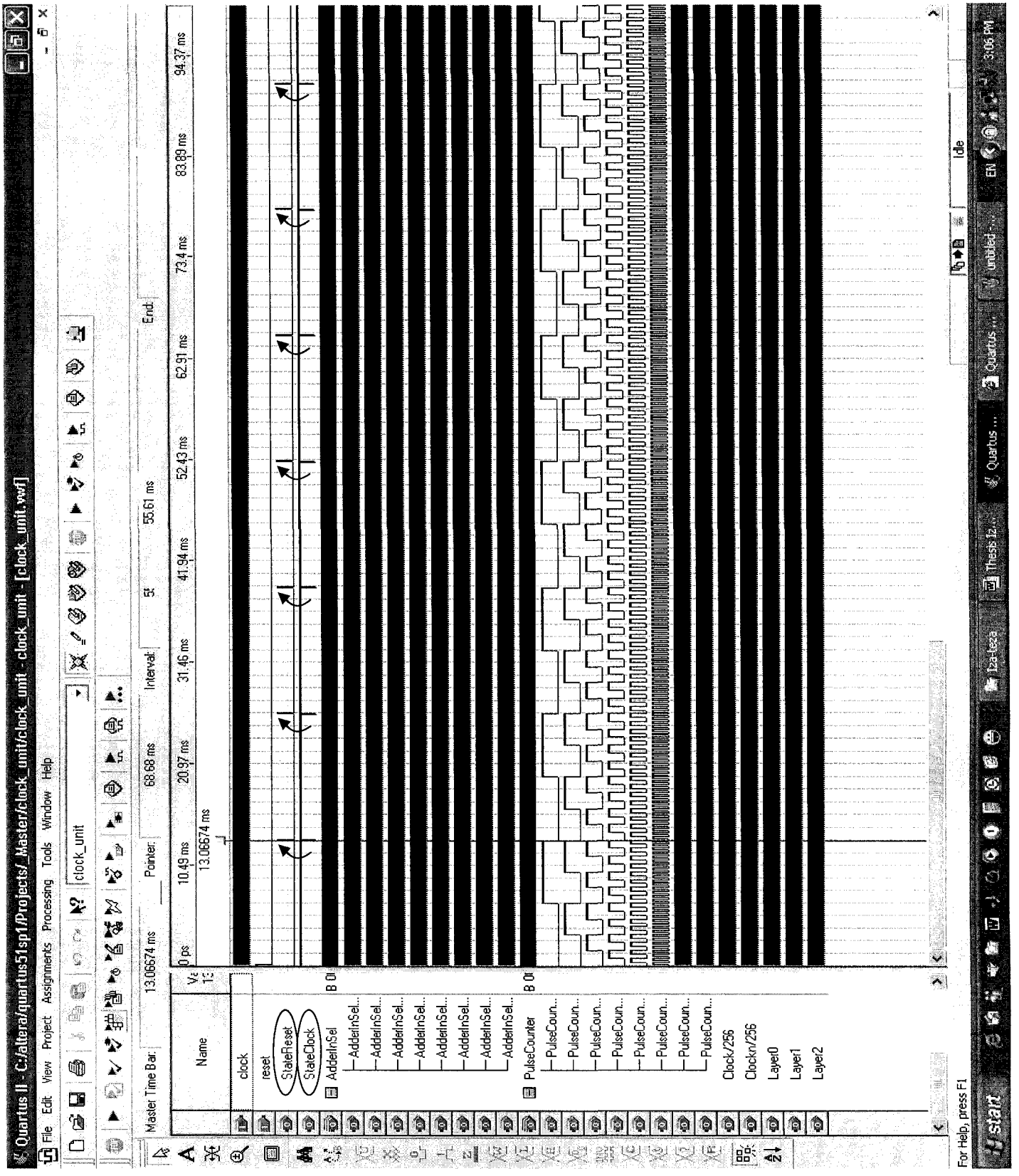


Figure 4.9. The clock_unit – wave diagram 2

4.2. The State Machine Unit

The objective of the *state_machine unit* is to implement a state machine that uses control inputs as well as the clock to direct their operation. The outputs of the state machine will be function of the present state and synchronous to the system clock. The symbol of the *state_machine* unit is shown in figure 4.10.

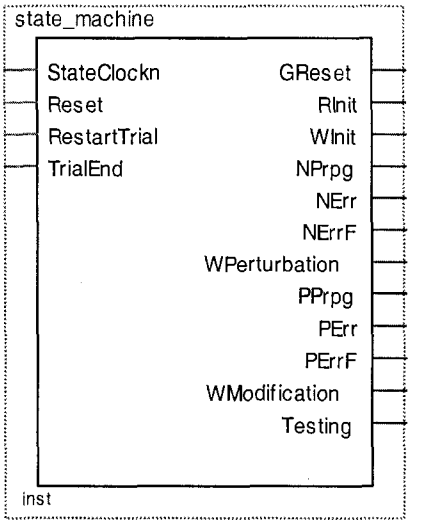


Figure 4.10. The state_machine unit symbol

The following definitions are introduced:

- a *pattern* is when one set of data inputs are propagated forward trough the consecutive layers of the ANN in order to obtain an output.
- an *epoch* is a period of training process involving ANN weight updates using training error from all patterns of training data just one time.
- a *trial* consists of many epochs with the same α (the learning coefficient) and c (perturbation).

This thesis implements a synchronous state machine whose outputs are signals named after states. An assignment is defined for every state:

0. In this state a general reset for all the registers of the implementation is generated.

1. In this state the random generators are initialised with random numbers.
2. In this state the weights are initialised to an initial random number value.
3. This state is for the normal propagation throughout the ANN.
4. In this state the error between the number of pulses of the output of the ANN and the corresponding teaching signal $J(\omega)$ is calculated.
5. In this state the result obtained in the previous state is transformed in an equal number of pulses, named error function calculation.
6. In this state the weights are perturbed simultaneously with the perturbation c, which is constant, but with different signs.
7. This state is for the perturbed propagation throughout the ANN.
8. In this state the error between the number of pulses of the output with perturbation of the ANN and the corresponding teaching signal $J(\omega+cs)$ is calculated.
9. In this state the result obtained in the previous state is transformed in an equal number of pulses, named error perturbed function calculation, and if the error < 0.01 or the number of epochs $> 2^{16}$ the trail ends and goes to the testing procedure (state 11).
10. In this state the weight's modification (either through modification or perturbation) is calculated.
11. During this state the neural network can be tested in order to see if it has learnt properly. The state machine remains in this state until an exterior *RestartTrial* signal is applied. Then the state machine goes to state 2.

Figure 4.9 shows the timing diagram of the state machine.

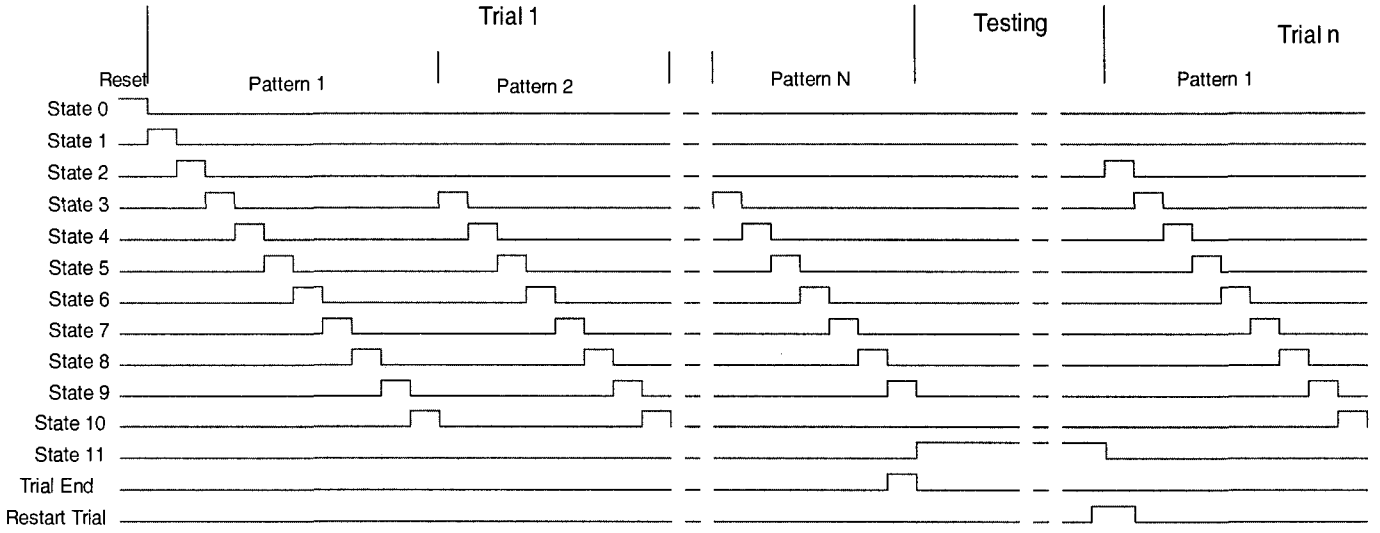


Figure 4.11. The state_machine timing diagrams to be implemented

4.2.1. Description and Functionality

After the states were previously defined, we obtained the number of flip-flops that is equal to the number of the state variables in the state machine shown in figure 4.12. The state machine has 12 states. The duration of one state was specifically defined in section 4.1.

The names of the states of the *state_machine unit* and its output signals are:

State 0 = General Reset, which gives the output *GReset*

State 1 = Random Generators Initialization, which gives the output *RInit*

State 2 = Weight Initialization, which gives the output *WInit*

State 3 = Normal Forward Network Propagation, which gives the output *NPrpg*

State 4 = Normal Output Error Calculation (testing signal), which gives the output *NErr*

State 5 = Normal Error Function Calculation, which gives the output *NErrF*

State 6 = Weight Perturbation, which gives the output *WPerturbation*

State 7 = Perturbed Forward Network Propagation, *which* gives the output *PPrpg*

State 8 =Perturbed Output Error Calculation (teaching signal), which gives the output *PErr*

State 9 = Perturbed Error Function Calculation, which gives the output *PErrF*

State 10 = Weight Modifications, which gives the output *WModification*

State 11 = Testing, which gives the output *Testing*

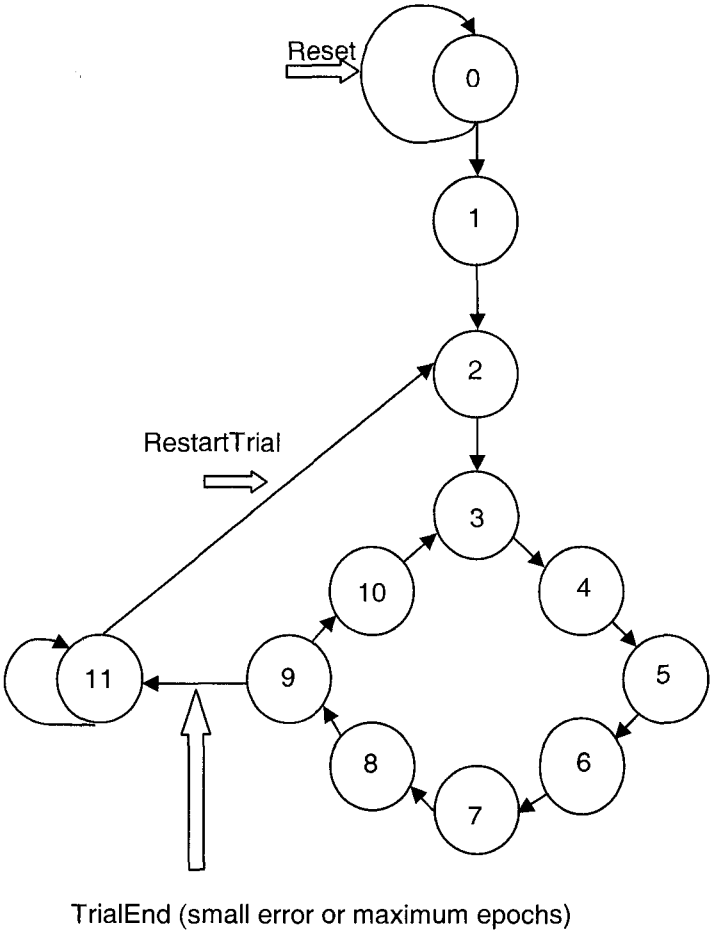


Figure 4.12. The state_machine - state diagram

The functionality of the *state_machine* is very simple. First, if the machine is in the start *State 0* and *reset* is logic *1*, then there is no transition. When *reset* falls down in logic *0*, then when the first *StateClock* goes *to 1* there is a transition to *State 1*. Moreover, if the reset is kept in

0, there is another transition to *State 2* and so on, until *State 9* is reached. Now, there are two branches that can follow:

- If the *TrialEnd* signal is *1*, which means the right values for weights were found, then there is a transition to *State 11*. Then, if the *RestartTrial* signal is *0* the state machine remains in *State 11*, else it goes to *State 2* to start the cycle again.
- If the *TrialEnd* signal is *0*, which means the right values for weights were not found, then there is a transition to *State 10*, and so on.
- Finally, the state machine stops in *State 11* when the error < 0.01 or the number of the epochs > 2^{16} .

The input signals of the *state_machine unit* are:

- *StateClockn* – this signal comes from the *clock-unit* and is the general clock.
- *Reset* – this is the *reset* signal that comes from the additional reset circuit shown in figure 4.6.
- *RestartTrial* – this signal is the output of the following additional reset circuit (Figure 4.13) that was implemented on the Altera board :

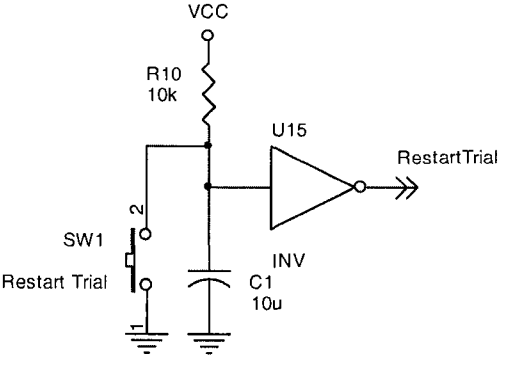


Figure 4.13. The additional RestartTrial circuit

When SW1 is pushed, a positive pulse is generated resulting in the *RestartTrial* signal.

- *TrialEnd* - this signal comes from the *controls_unit* and shows when the learning process has stopped.

The *state_machine* unit consists of the *neuron_states* block and the *state_dec* block, which are shown in figure 4.14.

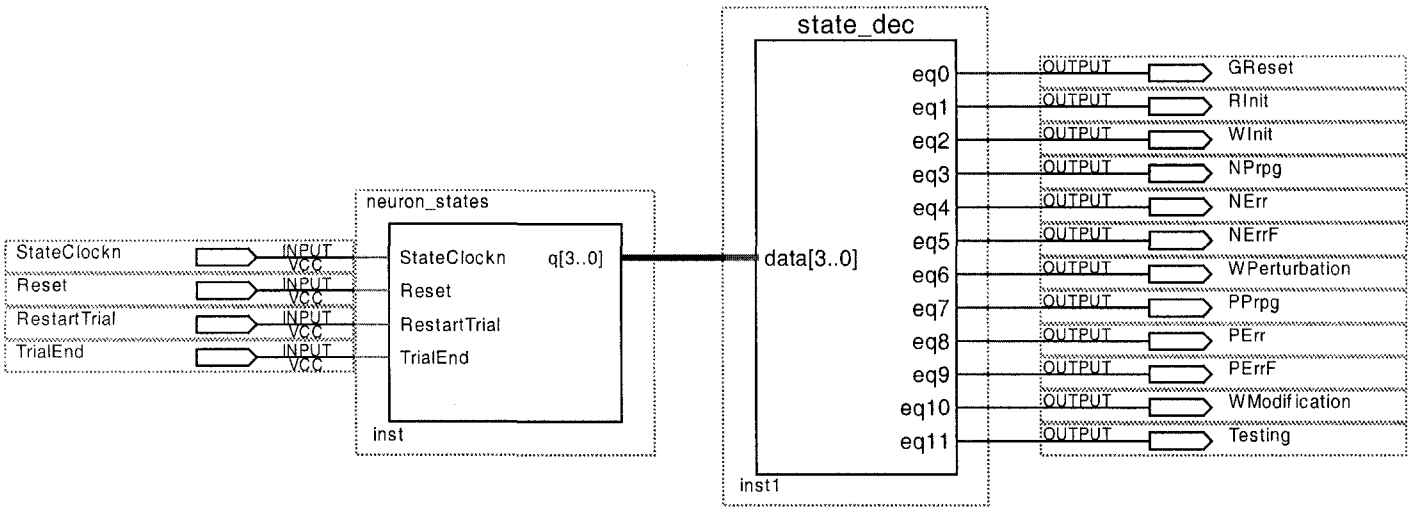


Figure 4.14. The state_machine - schematic diagram

- *neuron_states* block – is a state machine that generates at its output a 4-bit state code. It was defined in VHDL within a CASE statement from the QuartusII software version51sp1. The code is attached in appendix A.
- *state_dec* block – consists of a 4-line-to-12-line decoder that has active-HIGH outputs. Only the first 12 states are decoded.

4.2.2. The Simulation Results

Prior to running a simulation, we must create a vector source file, Vector Waveform File (.vwf), which is the source of simulation input vectors. This file is used by the Quartus II Simulator to generate the output signals that a programmed device would produce under the same conditions.

The Vector Waveform File contains the following:

- Maximum frequency f_{max} = 20 MHz
- End time of simulations = 300 ms

- *reset* after 6 *clock*'s pulses
- The *clock* signal frequency f = 5 MHz, which means T = 200 ns
- One state = 255 * (256 *clock* pulses) = 255 * (200 ns x 256) = 255 * 51200 ns = 13.1 ms

The simulation has been made on the ALTERA FLEX 10K family EFP10K70RC240_4 FPGA device.

The simulation lasted 2 minutes and 10 seconds. The implementation of this unit uses 29 total logic elements and needs 16 total input/output pins. The simulated waveforms are shown in detail in figure 4.15, 4.16 and 4.17.

As seen from the above mentioned figures, the *state_machine* has a correct functionality.

However, we can see some short glitches of 2.1 ns, but they do not disturb the functionality of the *state_machine* because they do not appear on the active positive edge of the clock. Such being the case, the glitches were not eliminated.

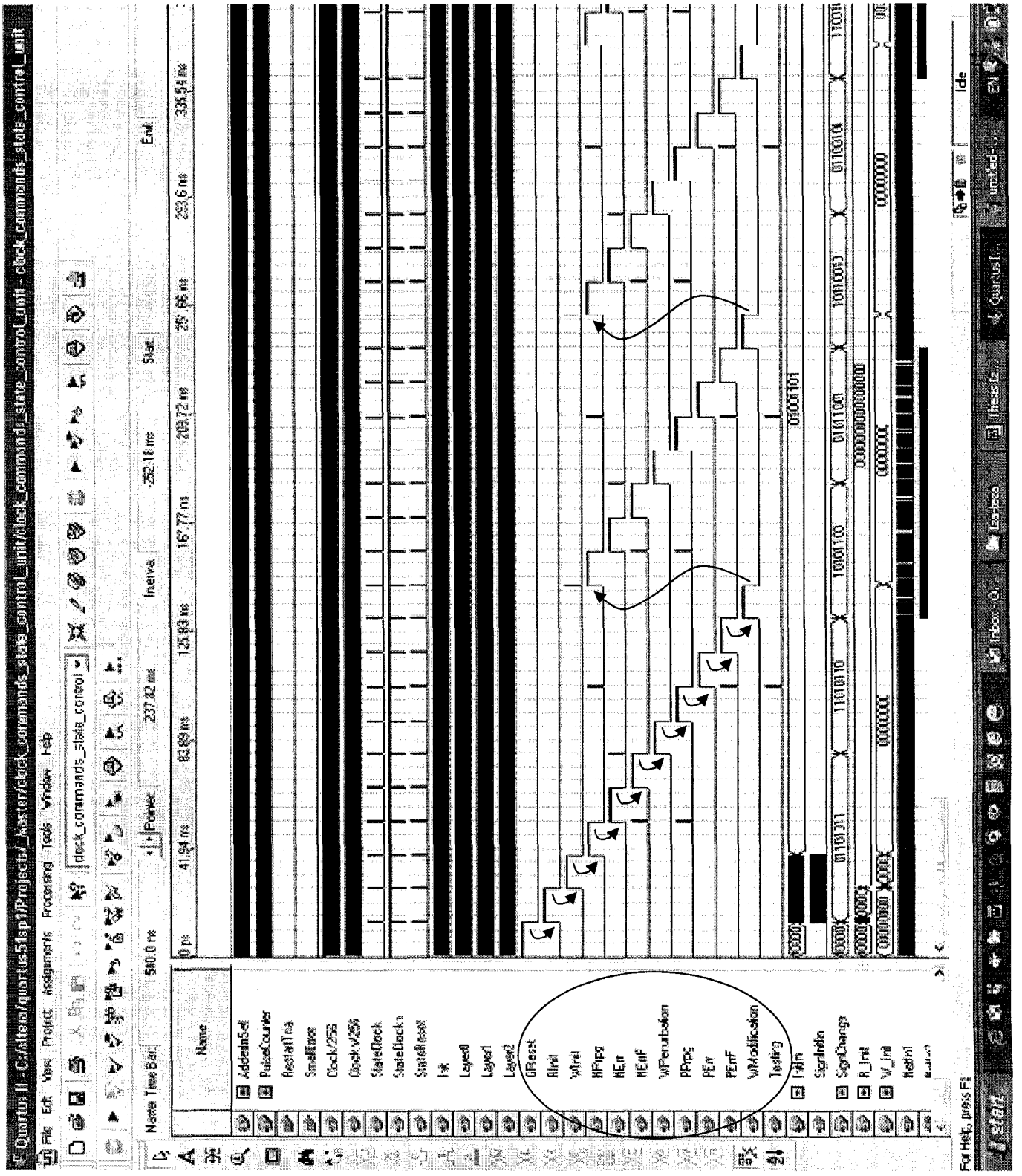


Figure 4.15. The state_machine unit – wave diagram 1

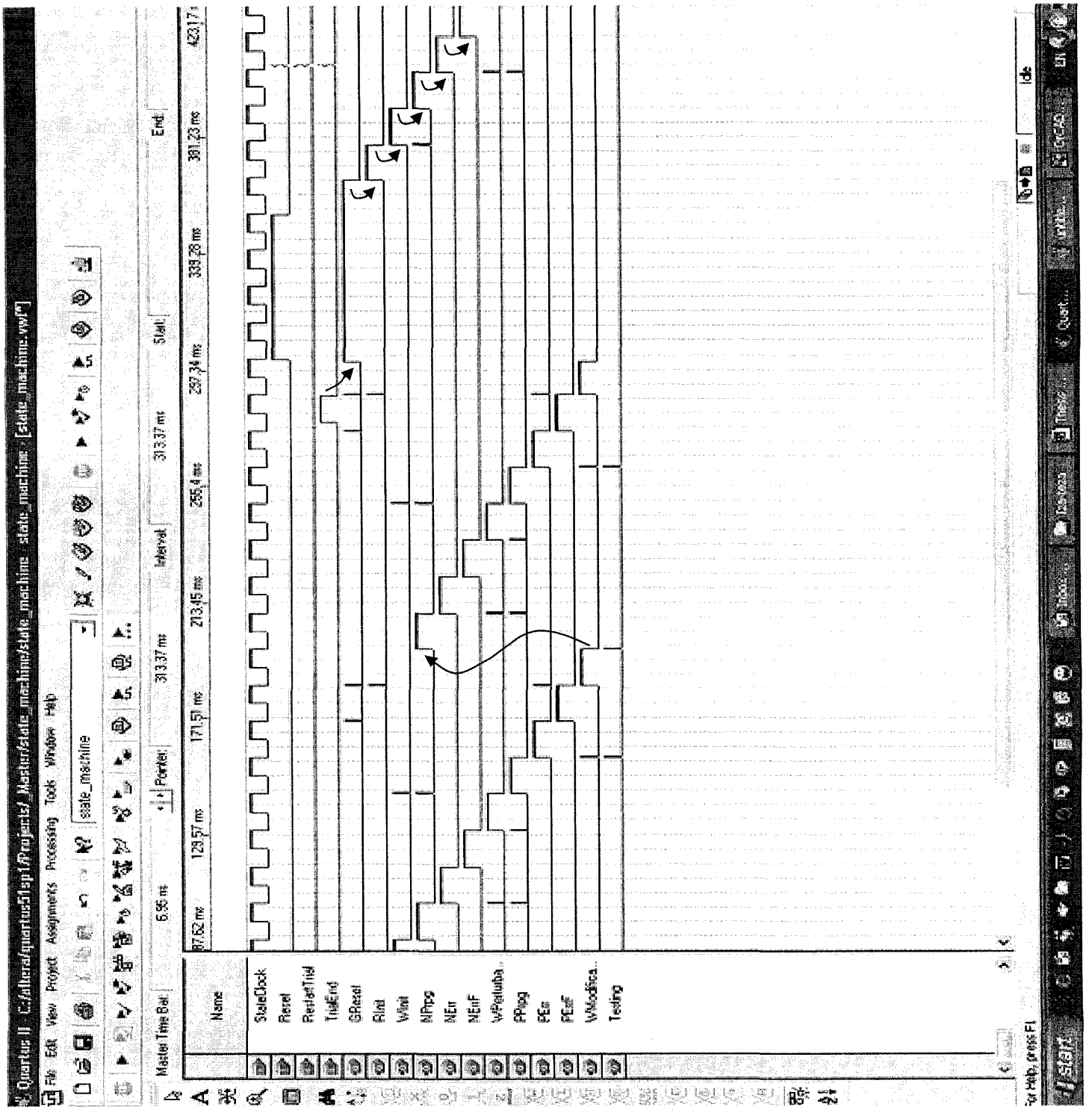


Figure 4.16. The state_machine unit – wave diagram 2

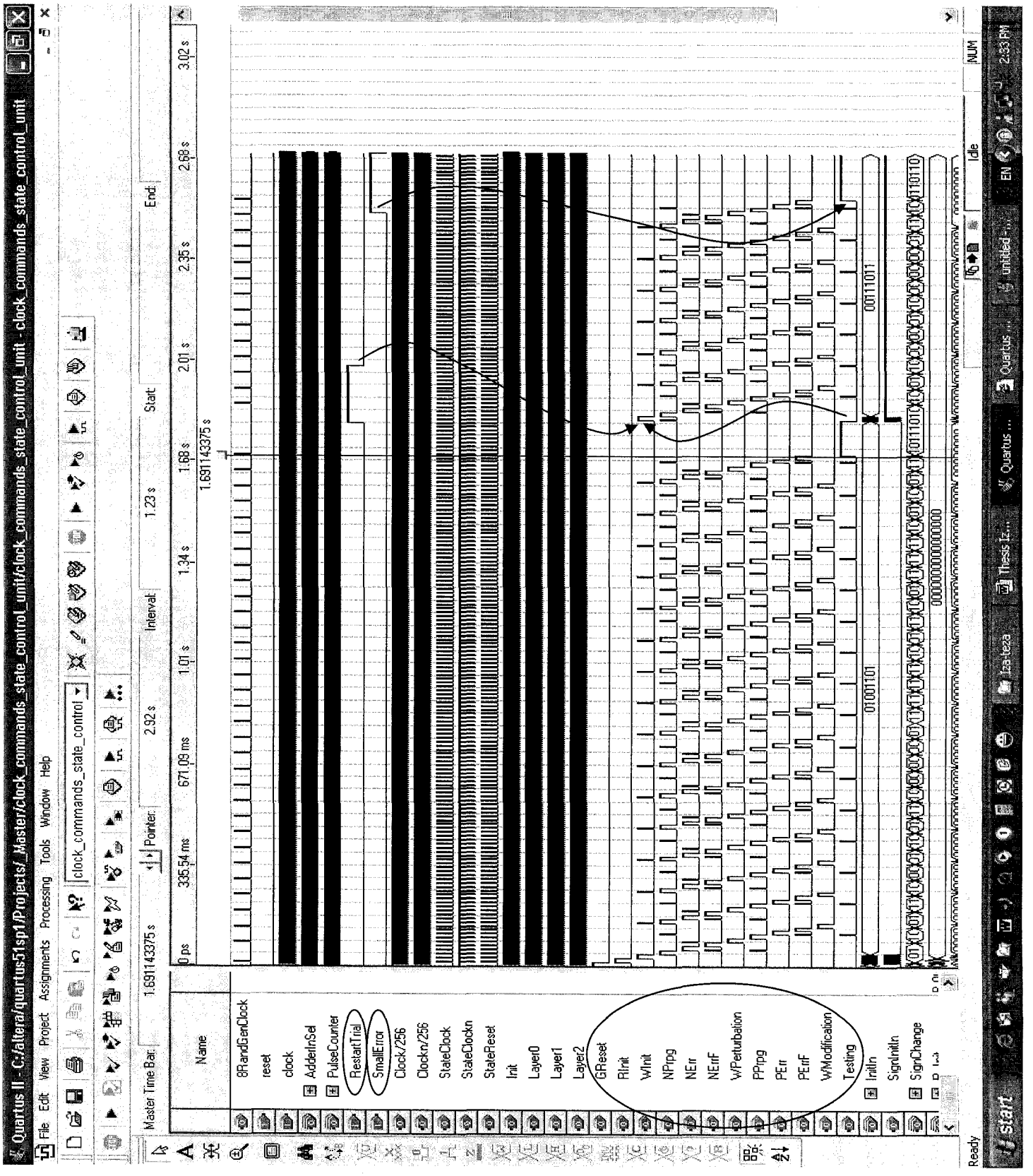


Figure 4.17. The state_machine unit – wave diagram 3

4.3. The Command Unit

The symbol of the *commands_ unit* is shown in figure 4.18.

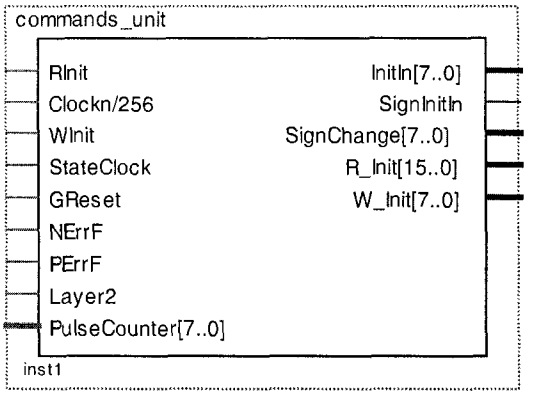


Figure 4.18. The commands_unit symbol

The purpose of this block is to provide successive command signals for initializing the *pulse_generator* units and the *weight_counter* units. At the same time, *commands_ unit* will generate the initial random number needed to initialize the random generators in order to obtain a good source of pseudorandom sequence. Equally important, the *commands_unit* will randomly generate the signs for perturbations and for the modified quantities from the learning unit.

4.3.1. Description and Functionality

The *commands_unit* contains one 8-bit random generator block, one 9-bit random generator block, two registers and a couple of basic digital gates to make the logic operation as it can be seen in figure 4.19.

- *random9_generator* – is a block that will be described in section 4.3.2 of this thesis and generates an 8-bit random signal named *InitIn* and a 1-bit random signal named *SignInitIn*. *InitIn* is used for a random initialization of all the 8 random

generators of the neural network, whereas *SignInitIn* is used for the initialization of the sign of the weights.

- *random8_generator* – is a block that will be described in section 4.3.3 of this thesis and creates the 8-bit *SignChange* signal.
- *decode16* – consists of an 8-line-to-16-line decoder that has active-HIGH outputs, with an output latency of 1 clock cycle, and an enable input. When the enable input is LOW then all the outputs are LOW. This decoder has a number of outputs equal to the number of the *Pulse_generator* units. For example, for a 2-input neuron, the number of outputs of *decode16* is 16. The *decode16* decodes all its states.
- *decode8* – consist of an 8-line-to-8-line decoder that has active-HIGH outputs, with an output latency of 1 clock cycle, and an enable input. When the enable input is LOW then all the outputs are LOW. This decoder has a number of outputs equal to the number of the *weight_unit* units. For example, for a 2-input neuron, the number of outputs of *decode8* is 8. The *decode8* decodes all its states.

The input signals of the *commands_unit* are:

- *9binaryconst[8..0]* – the external random-number constant input that initializes the *random9_generator.*
- *RInit* – this signal comes from the *state_machine* unit and indicates the state when the random generators need to be initialized.
- *Clockn/256* – this signal comes from the *clock_unit* and it is the general clock.
- *WInit* - this signal comes from the *state_machine* unit and indicates the state when an initial value of weight and its corresponding sign need to be initialized.
- *StateClock* – this signal comes from the *clock_unit* and it is a small pulse given at the end of each state.
- *GReset* – this signal comes from the *state_ machine* unit.

- *8binaryconst[7..0]* – the external random-number constant input that initializes the *random8_generator.*
- *NErrF* - this signal comes from the *state_machine* unit and indicates the state in which the error function is calculated.
- *PErrF* - this signal comes from the *state_machine* unit and indicates the state in which the perturbed error function is calculated.
- *PulseCounter[7..0]* - this signal comes from the *clock_unit* counting the 255 pulses in one state.
- *Layer2* - this signal comes from the *clock_unit* and is a short positive pulse that appears in each state cycle on the first active high level of *Clockn/256,* but after *Layer1.*

The output signals of the *commands_unit* are:

- *InitIn[7..0]* – is the random number that represents the initial value the *weight_counter* units and the *pulse_generator* units are loaded when ANN begins to learn.
- *SignInitIn* – is the randomly generated signal that indicates the sign of the initial value of the weight.
- *SignChange[7..0]* – each bit of this signal goes to one of the *weight_unit* units to change the sign of the weight. The number of bits of this signal is equal to the number of the weight units. For instance, for 8 weight units we need an 8-bit *SignChange* signal.
- *R_Init[15..0]* - every bit of this signal initializes one of the *pulse_generator* units.
- *W_Init[7..0]* – every bit of this signal initializes one of the *weight_unit* units.

The *random9_generator* provides the initial random numbers to the *random8_generator* units located in the *pulse_generator* units. In addition, the least significant bit of the *random9_generator* output provides the initial random sign for the initial weights. These

random numbers are generated only during the first logic-HIGH of *Clockn/256* from *RInit* state and the logic_HIGH StateClock of WInit state of the *state_machine*. The *random9_generator* will be explained in section 4.3.2.

Only at the beginning of *RInit* state the *decode16* detects the presence of its binary input values. The *decode16* block successively provides the narrow initializing pulses to the *random8_generators* units.

In the same way, the *decode8* block successively provides the narrow initializing pulses to the *weight* units, during the *WInit* state. These pulses have the same width as the pulses of *Layer2* signal, as can be seen in figure 4.19. The address lines of d*ecode16* and *decode8* blocks are provided by *PulseCounter[7..0]*.

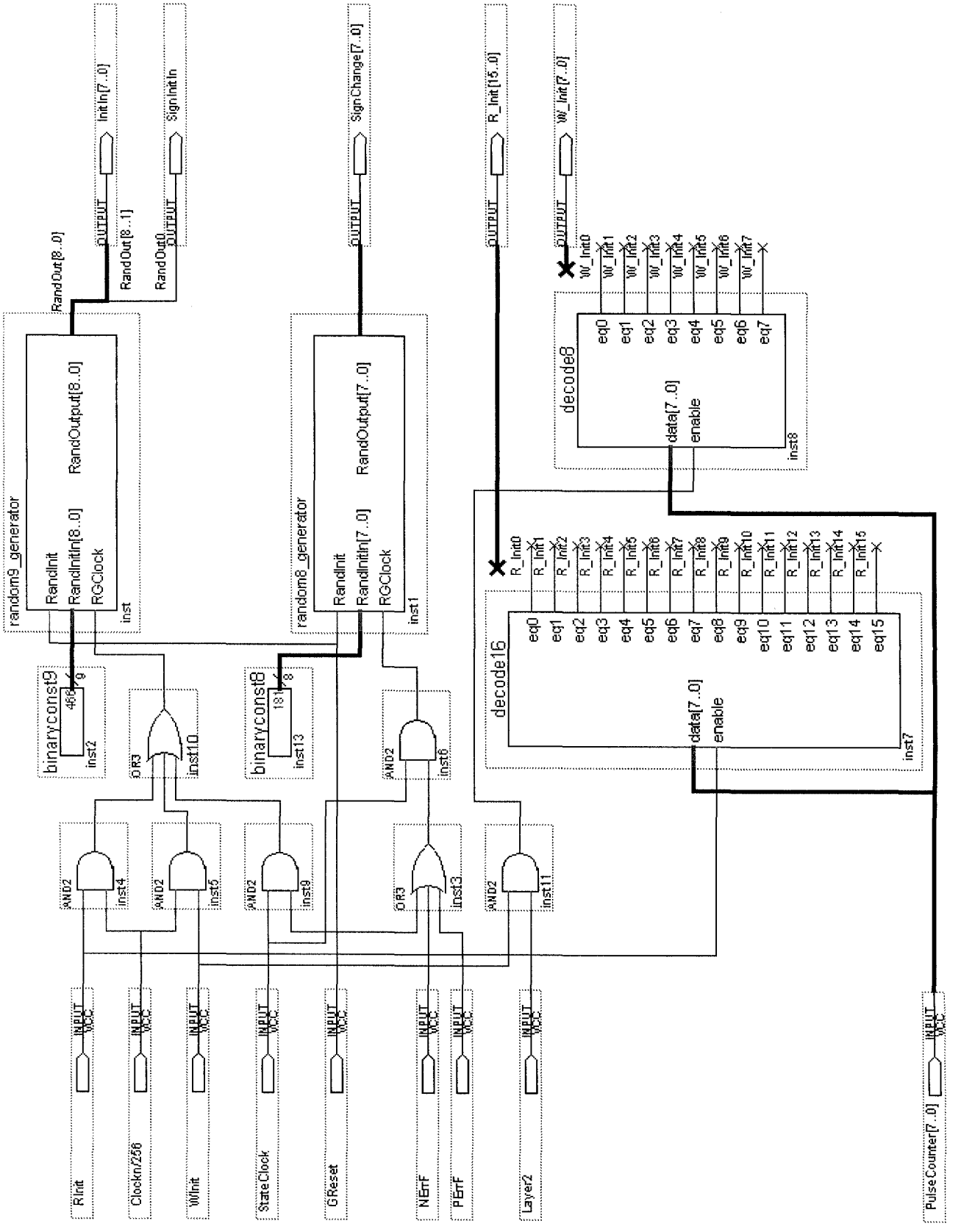


Figure 4.19. The commands_unit - schematic diagram

4.3.2. 8 and 9-bit Random Generator Units

As can be seen in figure 4.19, the schematic diagram of the *commands_ unit* contains two blocks named *random9_generator* and *random8_generator*. This section describes these two blocks. The symbol of the *random9_generator* unit is shown in figure 4.20 and the symbol of

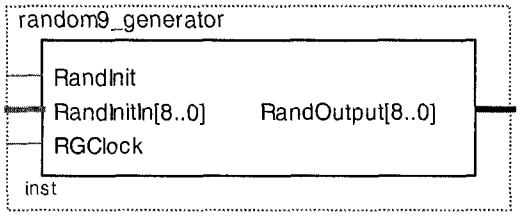


Figure 4.20. The random9_generator symbol

random8_generator unit is shown in figure 4.21. Both of them are sources of pseudorandom

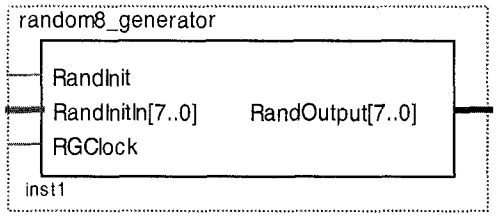


Figure 4.21. The random8_generator symbol

numbers. They can easily be implemented by using standard digital gates. A *Fibonacci* implementation using linear feedback shift registers *(LFSR)* has been used, which consists of a simple shift register in which a binary-weighted modulo-2 sum of the taps is fed back to the input. These standard digital gates never produce totally random numbers, thus the results are actually pseudo-random sequences. However, a very long pseudo-random binary sequence can be considered a random sequence. The table in figure 4.22 shows how to generate an n-bit pseudo-random binary sequence.

For instance, in order to obtain an 8-bit random generator, an 8-bit shift register that shifts left one bit per each clock cycle is needed. From the table in figure 4.22 we can see that for

n = 8, bits 8, 6, 5, and 4 must be XORed together in order to obtain the next shifted input used as the least significant bit in the shift register.

n	XOR From bits
3	3,2
4	4,3
5	5,3
6	6,5
7	7,6
8	8,6,5,4
9	9,5
10	10,7
11	11,9
12	12,6,4,1

Figure 4.22. Primitive Polynomials Modulo 2.
Note: The least significant bit number is "1".
Adapted from [40].

As it can be seen from the table, only a minimal number of XOR gates are needed for the circuit. If the previous value is "x", then the next value in the random sequence is actually "2x" or "2x+1".

In this way, we can build a pseudo-random generator with 2^n-1 non-zero sequences, which are repeating themselves in the same order.

The counter must first be loaded with an initial value, named the seed, which cannot be zero because otherwise the shift register would be stuck at zero. The circuit will always generate the same sequence of numbers for a given initial seed value [40].

In this thesis, the 9-bit pseudo-random generator creates the seed for the 8-bit generators, which means that the 8-bit random generators have different seeds for every trial cycle. Figures 4.23 and 4.24 show the schematic diagrams of *random9_generator* and *random8_generator*, respectively.

All the random generators must be initialized only one time per trial in the *RInit* state of the *state_machine* unit. Also, the weight units must be initialized only in the *WInit* state of the *state_machine* unit.

The functionality of the *random9_generator* is the same as the *random8_generator*.

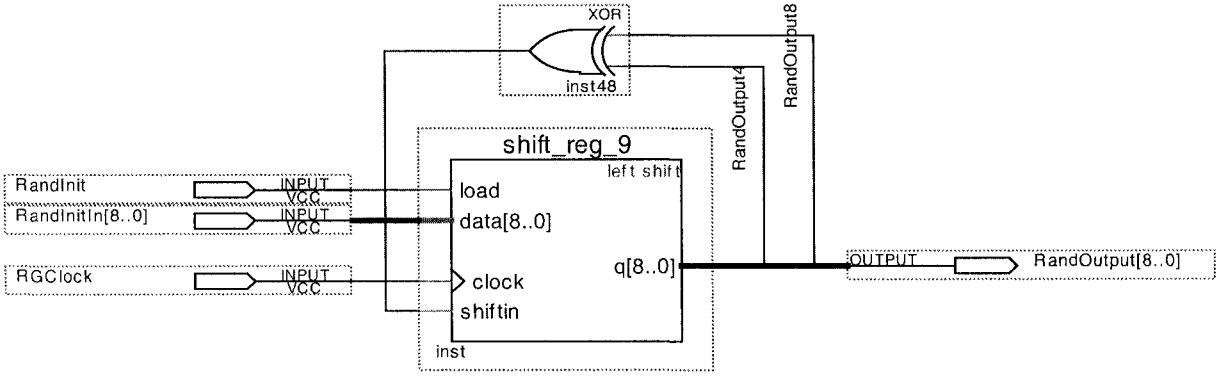


Figure 4.23. The random9_generator - schematic diagram

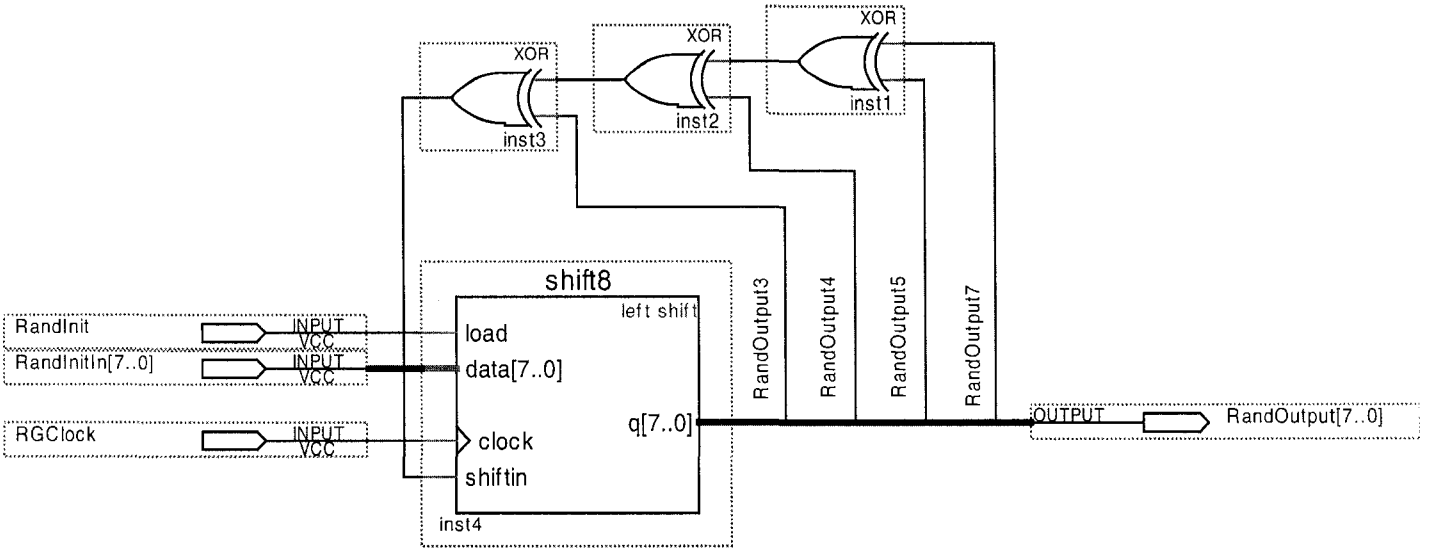


Figure 4.24. The random8_generator – schematic diagram

4.3.3. The Simulation Results

In order to simulate the *commands_unit* we need to use the *clock_unit* and the *state_machine* unit.

The Vector Waveform File contains the following:

- Maximum frequency f_{max} = 20 MHz
- End time of simulations = 300 ms
- *reset* after 3 *clock*'s pulses
- The *clock* signal frequency f = 5 MHz, which means T = 200 ns
- One state = 255 * (256 *clock* pulses) = 255 * (200 ns x 256) = 255 * 51200 ns = 13.1 ms
- *9binaryconst[8..0]* is set to 011010010
- *8binaryconst[7..0]* is set to 01010100

The simulation has been made on the ALTERA FLEX 10K family EFP10K70RC240_4 FPGA device. The simulation lasted 15 minutes. The simulated waveforms are shown in figures 4.25 to 4.30.

Figure 4.25 shows the random outputs of the *random9_generator* seeded with the value of *011010010*. Figure 4.26 shows the random outputs of the *random8_generator* seeded with the value of *01010100*.

Figure 4.27 shows the *R_Init [15..0]* signal generated during the *RInit* state. The glitches that can be seen on the *R_Init waveform* in figure 4.27 do not affect the functionality of the *commands*_unit because they do not appear during the active period of *R_Init*. In the same figure we can see that the *SignIntIn* and *InitIn[7..0]* signals are random numbers.

Figure 4.28 shows the *W_ Init[7..0]* signal generated during the *WInit* state. In the same figure we can see that the *SignIntIn, InitIn[7..0]* and *SignChange[7..0]* signals are random numbers.

Figures 4.29 and 4.30 show the *R_Init* and *W_Init* signals synchronized with all the states of *state_machine*.

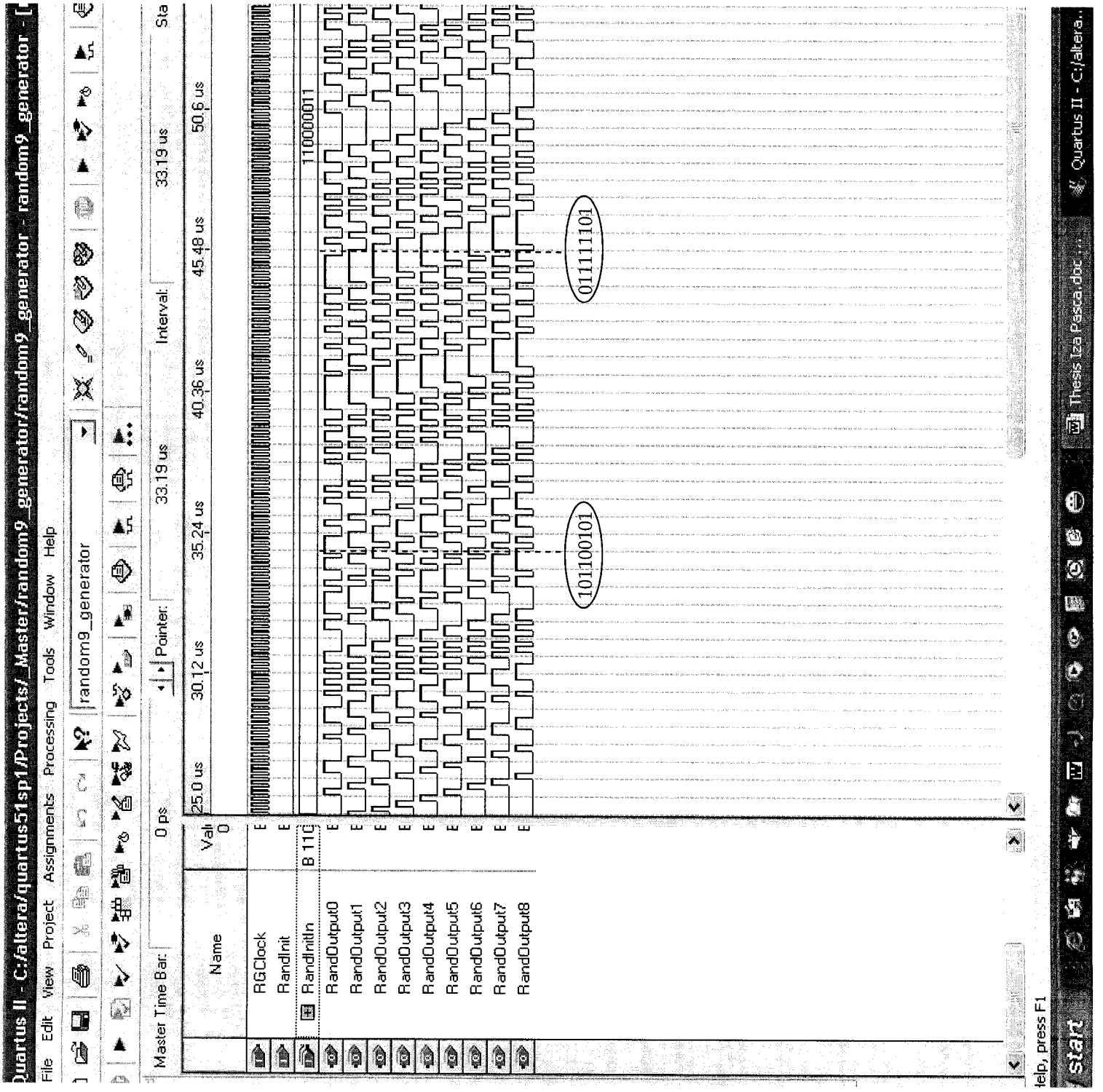


Figure 4.25. The random9_generator – wave diagram

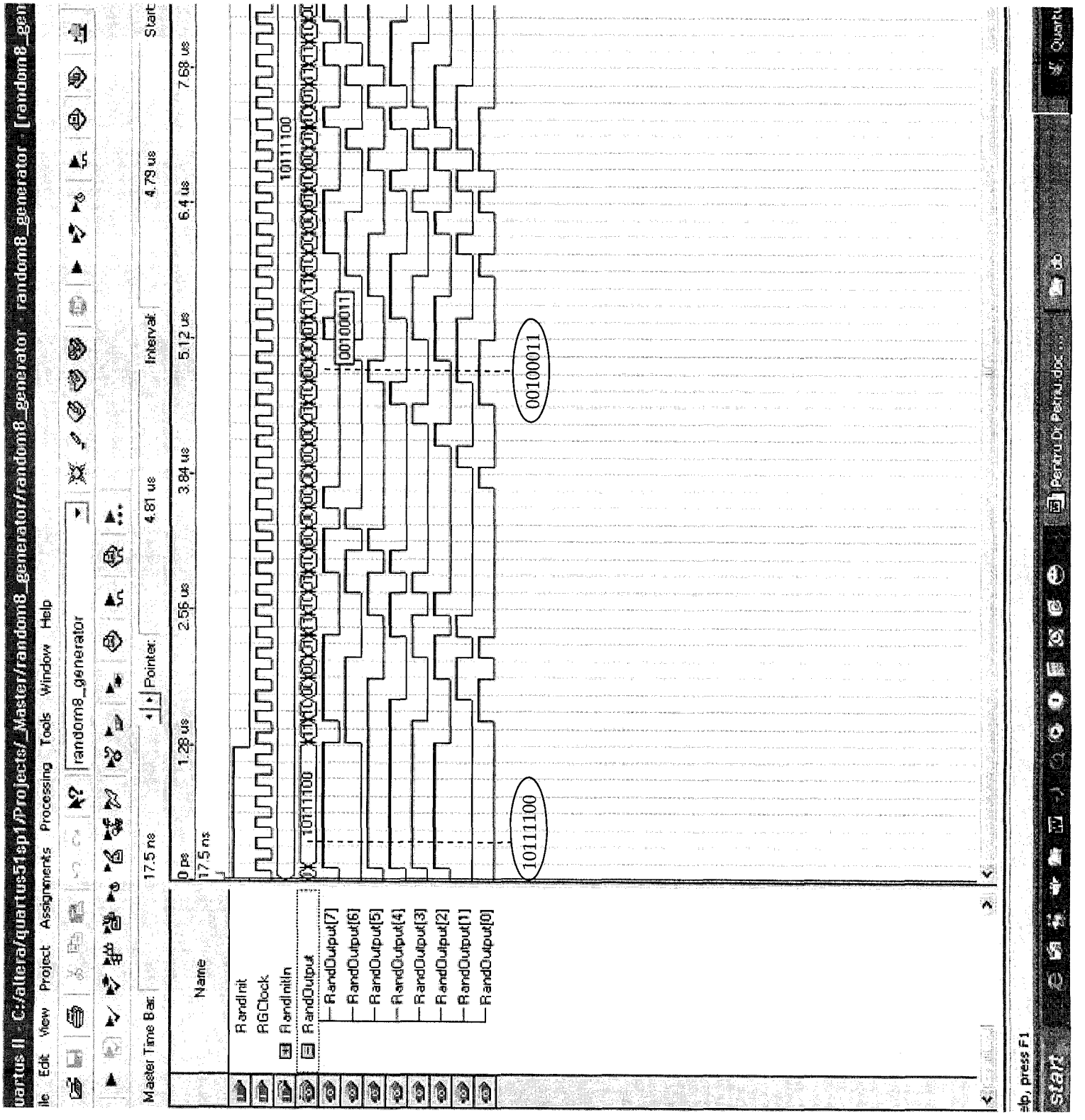


Figure 4.26. The random8_generator – wave diagram

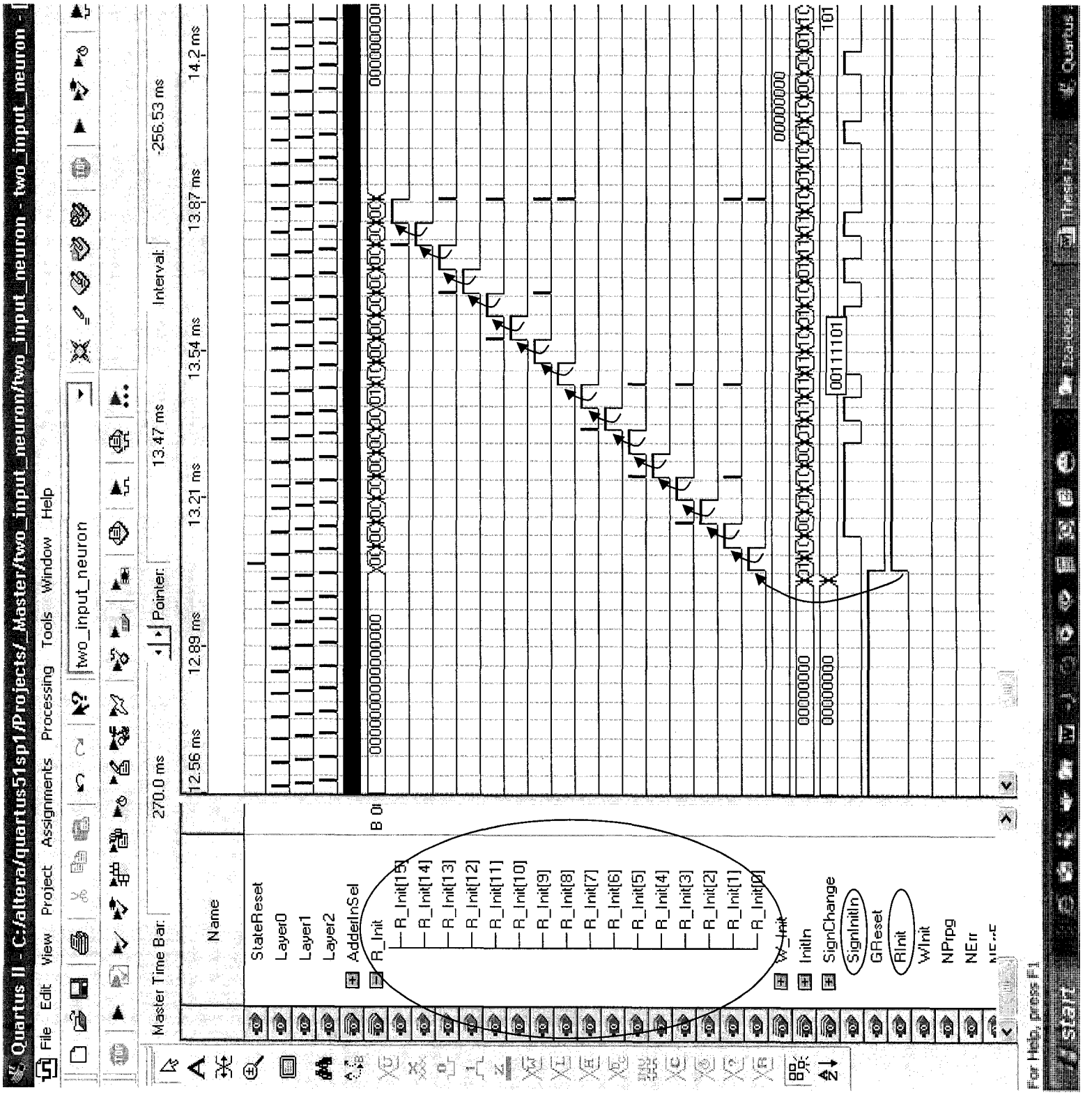


Figure 4.27. The commands_unit - wave diagram (with bit representation of R_Init)

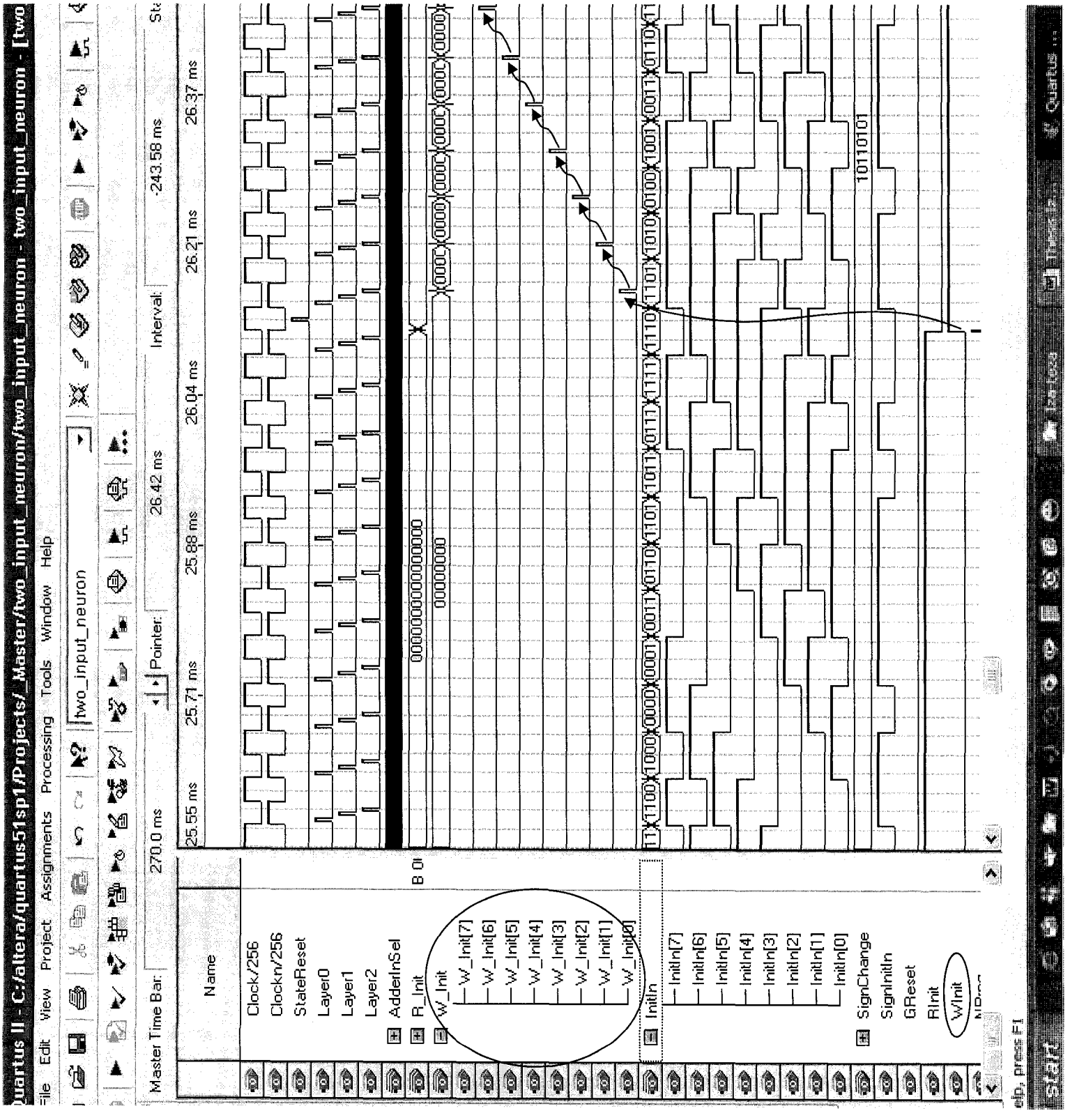


Figure 4.28. The commands_unit – wave diagram (with bit representation of W_Init)

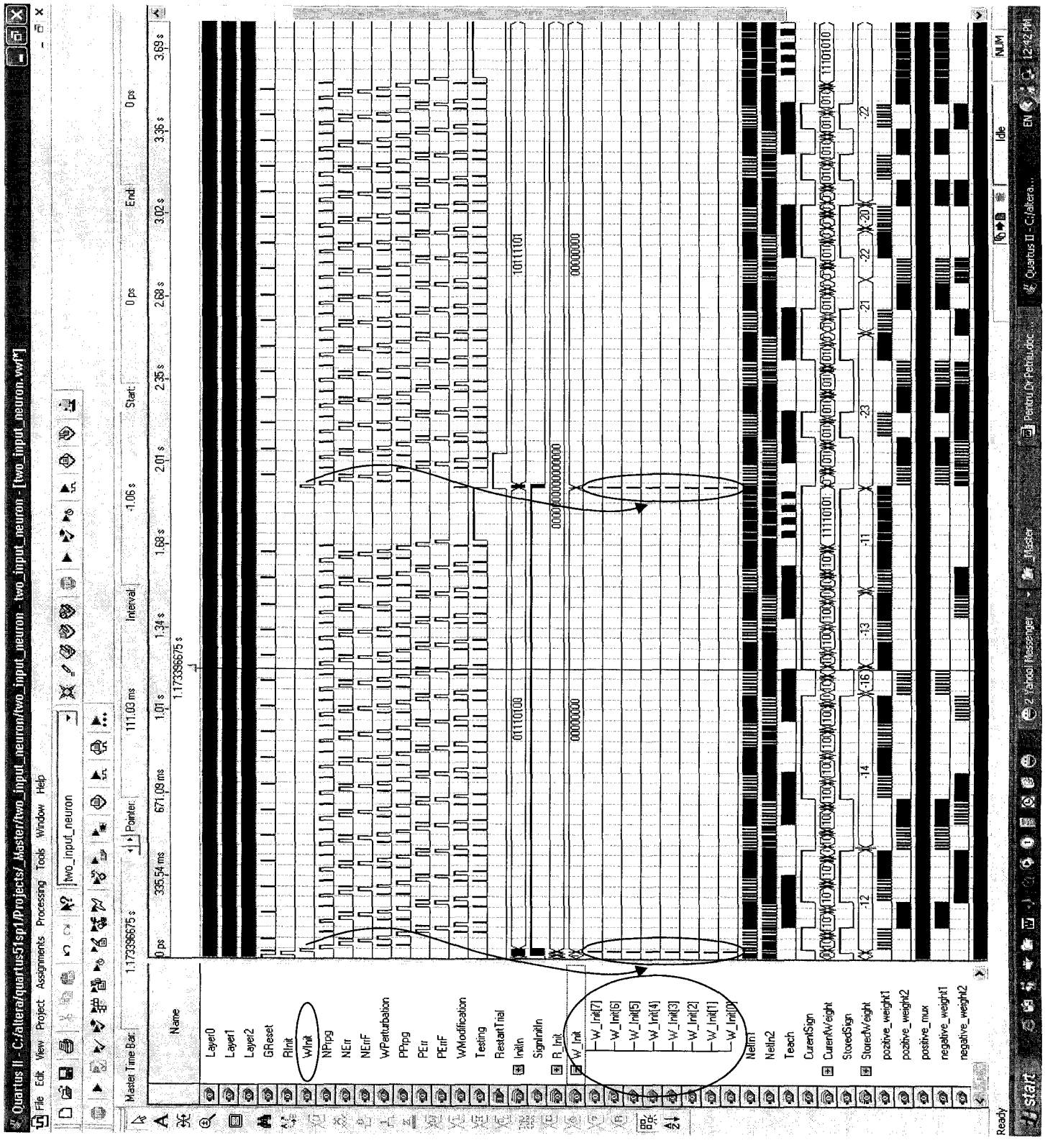


Figure 4.29. The commands_unit – wave diagram (W_Init synchronized with all the states of state_machine)

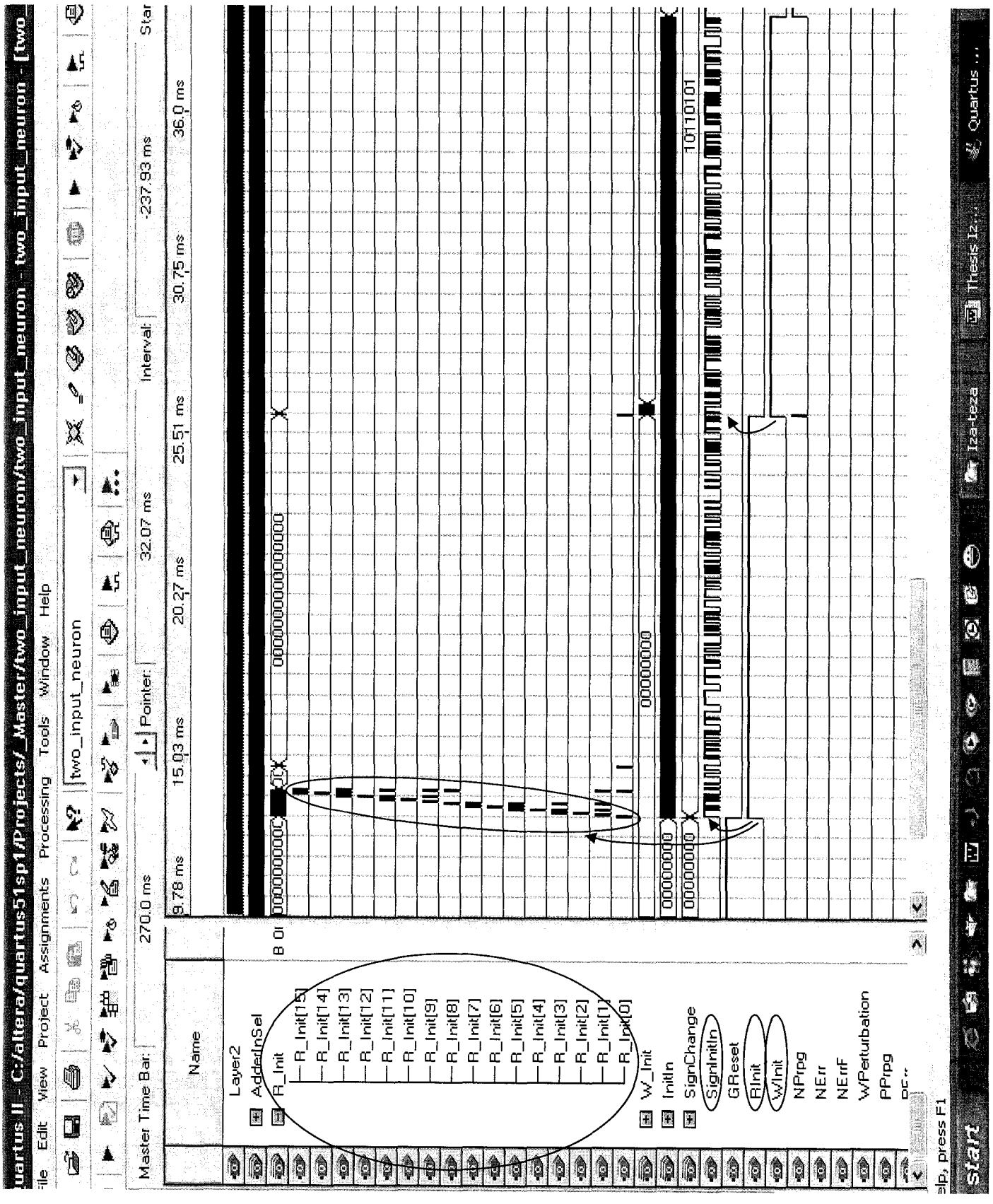


Figure 4.30. The commands_unit –wave diagram (R_Init and SignInitIn synchronized with all the states of state_machine)

4.4. The Control Unit

The *controls_unit* block creates the signals that count up the numbers of the patterns, epochs and trials. Also, this block converts the input and teaching digital signals into pulse signals. For simplicity, we consider a *controls_unit* block for a neuron with two inputs, one teaching signal and four patterns. Figure 4.31 shows the symbol of this *controls_unit.*

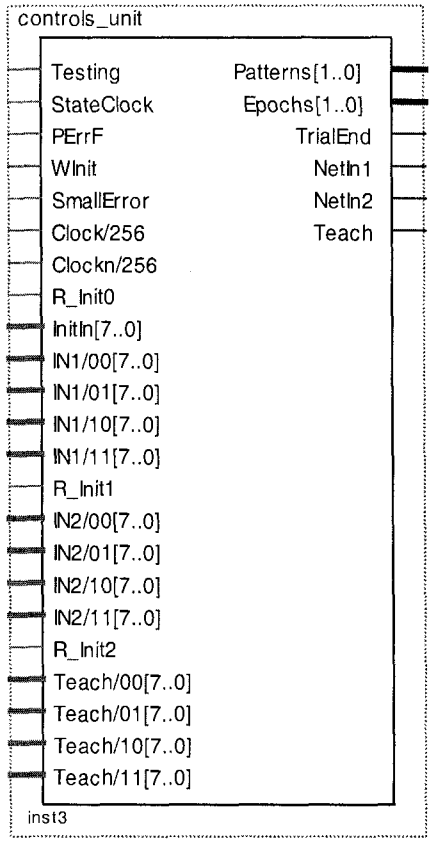


Figure 4.31. The controls_unit symbol

4.4.1. Description and functionality

The *controls_unit* consists of as many signal generator blocks as the number of the input and teaching signals. For example, for a neuron with two inputs and one teaching signal,

the *controls_unit* block must have three *signal_generator* blocks. The *controls_unit* block also contains two counters that count the patterns and the epochs.

The signals generated by *clock_unit, state_machine unit* and *commands_ unit* are not depicted again, but they are used as inputs to the *controls_unit* block. Their description has been already done in their corresponding sections.

Figure 4.32 illustrates the *controls_unit* schematic diagram. It consists of the following blocks:

- *Pattern_counter* – is a binary up counter, with a 2-bit wide output *q[1..0]* and a carry-out port *cout*. It has two inputs: *clock* and synchronous clear *sclr*. The number of bits of the *Pattern_counter* output signal is given by the number of the learning patterns of the ANN. For example, for a 2-input neuron with 4 learning patterns, we need 2 bits. By counting every *StateClock* pulse of the *PErrF* state from the beginning (*Testing* or *WInit* state), this counter counts up the number of the patterns. When the *Pattern_counter* reaches its maximum number, it gives a pulse to the second counter *Epoc_counter.*
- *Epoc_counter* – is a binary up counter, with a 16-bit wide output *q[15..0]* and a carry-in *cin* port. It has two inputs: *clock* and synchronous clear *sclr*. When *Epoc_counter* takes a pulse from the *Pattern_counter* counter it begins to count up the number of the epochs. When it reaches the maximum number of *65636*, it sends out a signal and stops the *state_machine* unit, which means that the learning process was not convergent.
- *signal_generator* – will be described in section 4.4.3.

The input signals of the *controls_unit* are:

- *Testing* - this signal comes from the *state_machine* unit and represents the testing state.

- *WInit* - this signal comes from the *state_machine* unit and represents the weight initialization state.
- *PErrF* - this signal comes from the *state_machine* unit and represents the perturbed error function calculation state.
- *StateClock* - this signal comes from the *clock_unit* and is a small pulse given at the end of each state.
- *Clockn/256* - this signal comes from the *clock_unit* and is the general clock.
- *InitIn [7..0]* - this signal comes from the *commands_unit* and is the random number.
- *R_Init [2..0]* – this signal comes from the *commands_unit.* It is actually formed by taking the first 3 least significant bits of *R_Init [15..0]* and is an initialisation signal.
- *SmallError* - this signal comes from the *learning_unit* and indicates when the error is < 0.01
- *IN1/00, IN1/01, IN1/10, IN1/11* – these are external signals that supply the ANN with inputs. In our case, there are four different values for input1.
- *IN2/00, IN2/01, IN2/10, IN2/11* – these are external signals that supply the ANN with inputs. In our case, there are four different values for input2.
- *Teach/00, Teach /01, Teach/10, Teach /11* - these are external inputs that supply the ANN with teaching signals. In our case, there are four different values for *Teach* input, one for each pattern.

The output signals of *controls_ unit* are:

- *PaternSel[1..0]* – this signal selects the learning patterns. For example, for 4 learning patterns, the *PaternSel* signal needs to have only two bits.
- *Epochs [15..0]* – this signal will go to an additional exterior circuit, in order to capture the number of the epochs on a display.
- *TrialEnd* – this signal stops the *state_machine* unit

- *NetIn1* – represents the *Input1* signal converted to a random pulse representation signal, which has a number of pulses between *0* and *255*.
- *NetIn2* – represents the *Input2* signal converted to a random pulse representation signal, which has a number of pulses between *0* and *255*.
- *Teach* – represents the *Teaching Input* signal converted to a random pulse representation signal, which has a number of pulses between *0* and *255*.

The number of *NetIn and Teach* signals is equal to the number of the inputs and teaching signals of the ANN, respectively. These signals will be described in section 4.4.3 Signal generator.

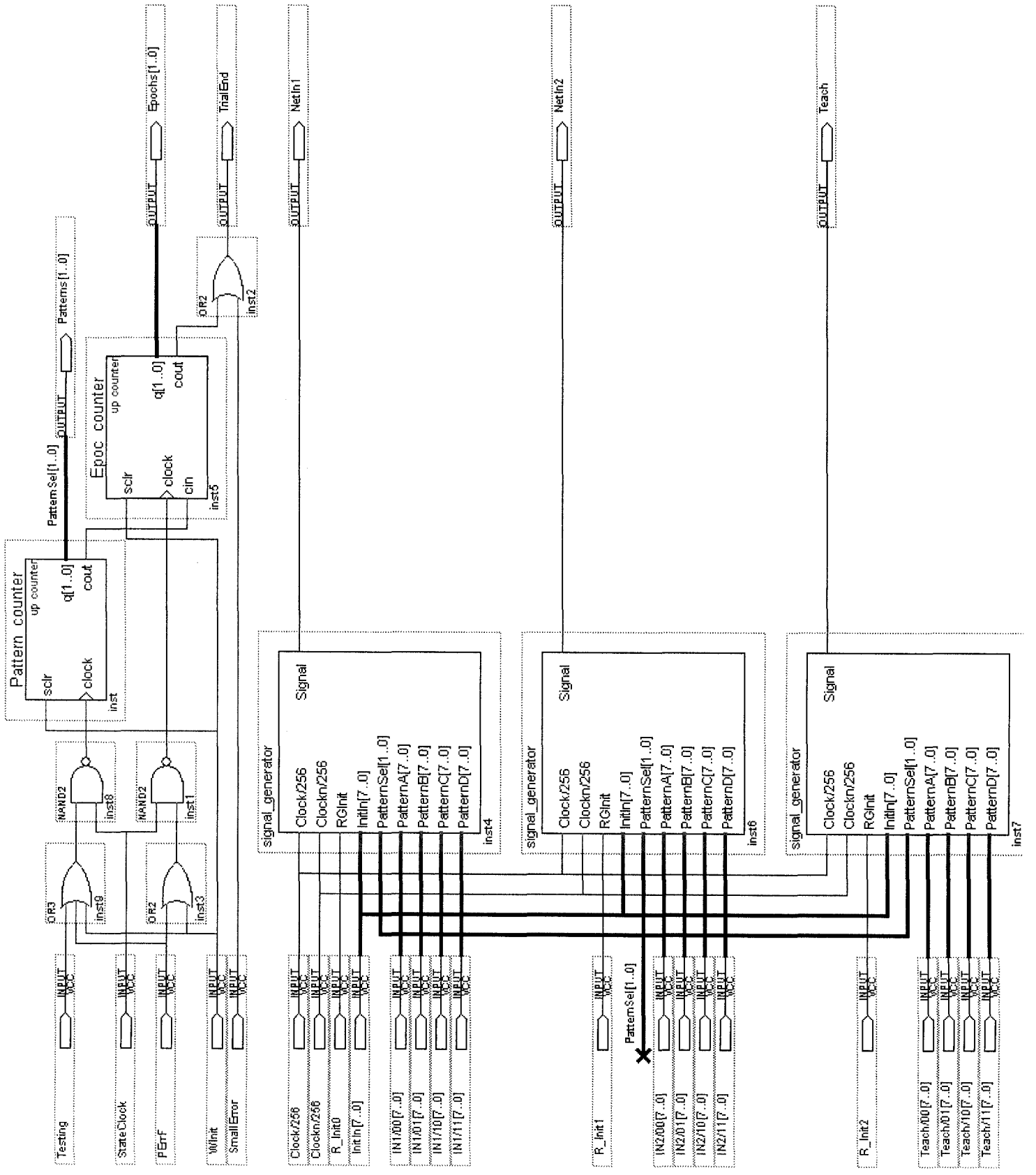


Figure 4.32. The controls_unit - schematic diagram

4.4.2. The Pulse Generator

The *pulse_generator* block converts digital signals into pulse signals. Figure 4.33 shows its symbol.

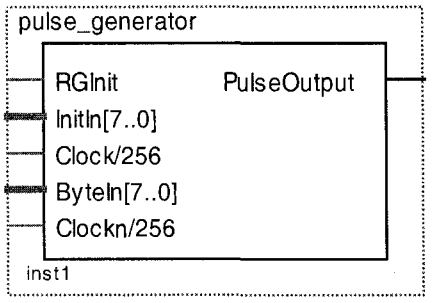


Figure 4.33. The pulse_generator symbol

An 8-bit random-number generator and a comparator have been used to build this block, as it is shown in figure 4.34:

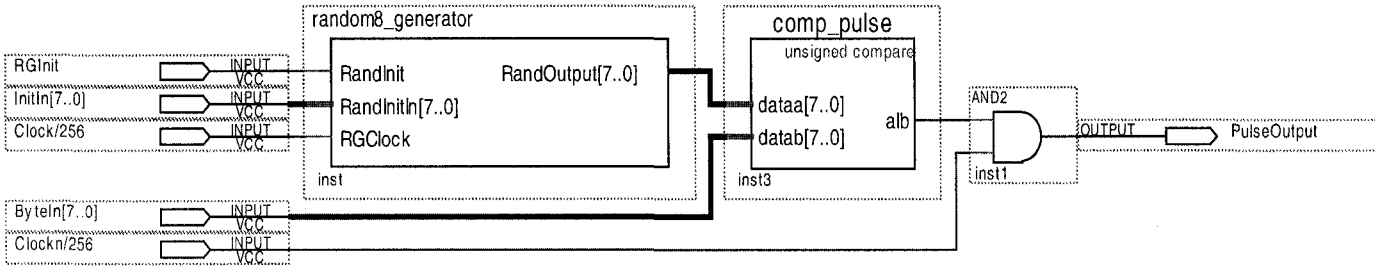


Figure 4.34. The pulse_generator – schematic diagram

- *random8_generator* – this block was described in section 4.3.2.
- *comp_pulse* – this block performs the unsigned comparison $a < b$, where a is the random number from the *random8_generator*, and b is the value to be compared with. When this condition is satisfied, the output of *comp_pulse* goes to 1, otherwise it remains in 0. The result of the comparison is outputted only on the positive level of the *Clockn/256* signal, which is accomplished by the AND2 gate *inst1*.

The *random8_generator* produces random numbers. The *comp_pulse* block compares a given value with a random value generated by the *random8_generator*. If the given value is larger than the random number, then the *pulse_generator* generates a single pulse. If the given value is smaller or equal than the random number, then no pulse is generated. This happens only on the positive level of the *Clockn/256*. This procedure is repeated and new random numbers are generated at each *Clockn/256*. Hence, a large given value results in many pulses, whereas a small given value results in very few pulses. By using this method, the given values are represented by pulse density.

The *pulse_generator* block is used in many other blocks, such as:

- *controls_unit* – located in the *signal_generator* block, where inputs and teaching signals must be converted into a series of pulses.
- *weight_unit* – where the weight values calculated in the *weight_unit* must be converted into a series of pulses.
- *learning_unit* – where the digital error value, c-perturbation and α-learning coefficient must be converted into a series of pulses.

4.4.3. The Signal Generator

The *signal_generator* block converts the input and teaching signals into pulse signals. Figure 4.35 shows the *signal_generator* symbol. Its schematic diagram is shown in figure 4.36.

The *signal_generator* consists of eight multiplexers and one *pulse_generator* block:

- *pattern_mux* – is a 4-to-1 multiplexer, which chose the right signal to be sent to the *pulse_generator* unit. The *PatternSel[1..0]* signal, which comes from the *Pattern_counter* block, selects the pattern_*mux* outputs.
- *pulse_generator* – this block has been described in section 4.4.2.

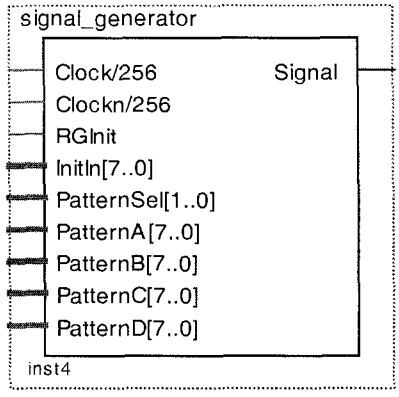


Figure 4.35. The signal_generator symbol

Each input value of the ANN must be written as an 8-bit value [7..0]. For example, a *0* input value will be written as *00000000*, whereas a *1* input value will be written as *11111111*. This convention allows feeding the block with noisy inputs, such as *011011111*. The new 8-bit [7..0] form of the input value is sent to the *pulse_generator* block to be converted into a series of pulses.

In order to represent a digital number as a number of pulses we make the following convention:

- zero logic is represented through no pulses in a time unit
- one logic is represented by 255 pulses in a time unit
- a, a value between *0* and *1*, is represented by (255*a) pulses. For instance, if $a = 0.45$ then it is represented by 255*0.45 = 115 pulses.

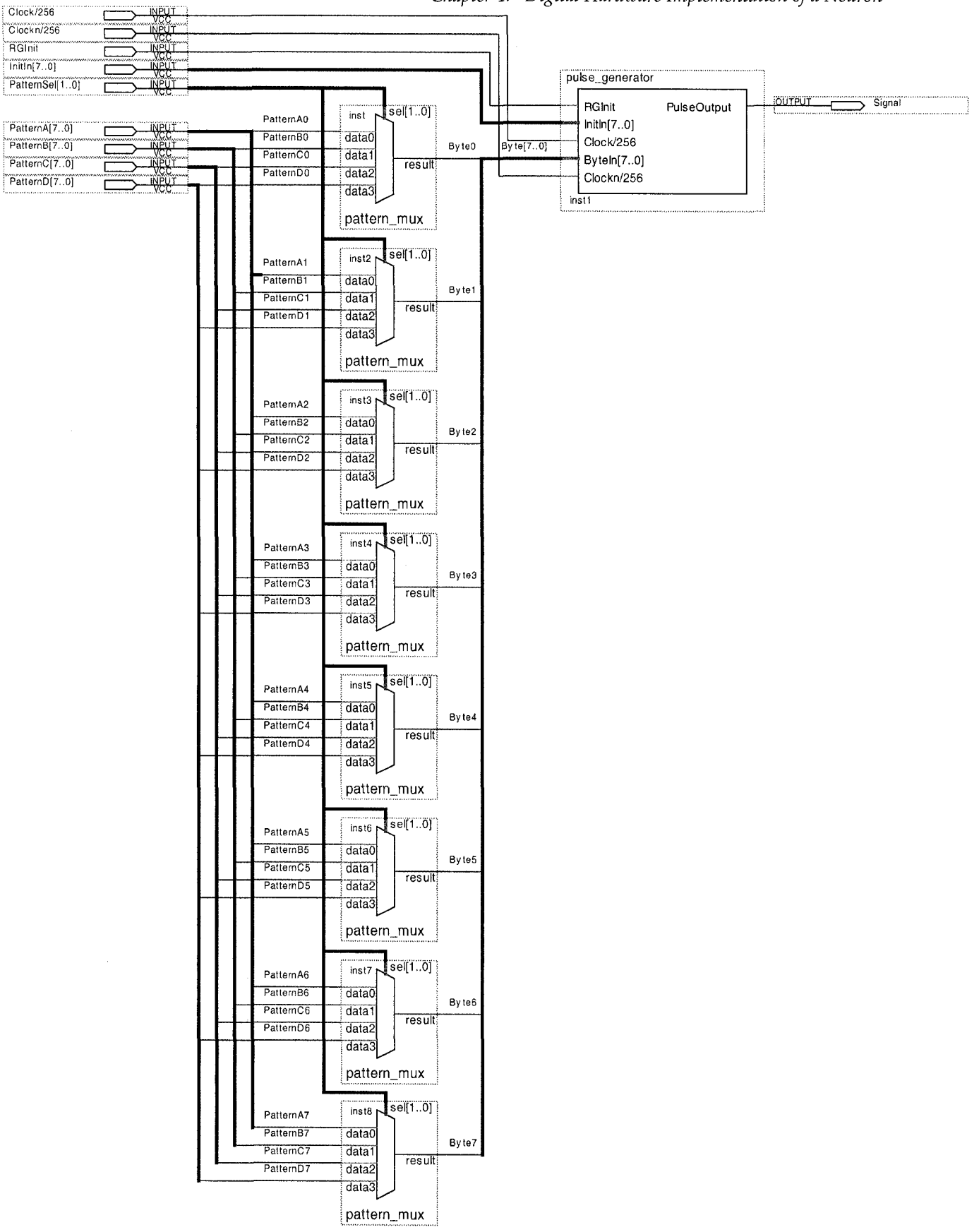


Figure 4.36. The signal_generator – schematic diagram

4.4.4. The Simulation Results

In order to simulate the *controls_unit* we need to use the *clock_unit*, the *state_machine* and the *commands_unit*. In this way we obtained a new block: *clock_commands_state_control_unit*. Figures 4.37 and 4.38 show the symbol and the schematic diagram of the *clock_commands_state_control_unit*, respectively.

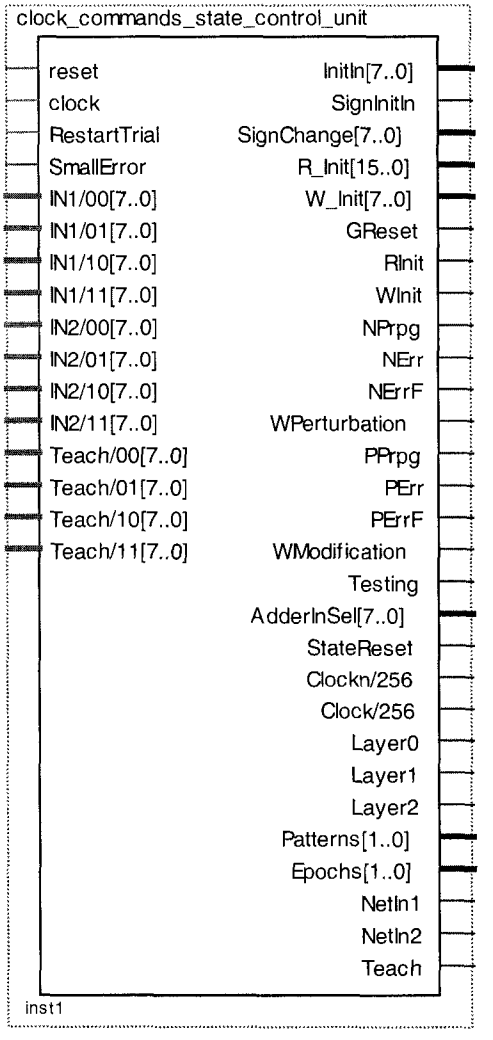


Figure 4.37. The clock_commands_state_control_unit symbol

This block can be kept for another implementation of an ANN, but a few adjustments are required. For an extra layer, *decode_3to4* located in the *clock_unit* must decode an extra state. If we have more weights and inputs, the *decode16* and *decode8* decoders from the *commands_unit* must also increase the number of their outputs. Also, the *signal_generator* from the *controls_unit* must be adapted to the increased number of weights and inputs.

In our case of the two input neuron with 4 patterns, the Vector Waveform File contains the following:

- Maximum frequency f_{max} = 20 MHz
- End time of simulations = 2.7 s
- *reset* after 3 *clock*'s pulses
- The *clock* signal frequency f = 5 MHz, which means T = 200 ns
- One state = 255 * (256 *clock* pulses) = 255 * (200 ns x 256) = 255 * 51200 ns = 13.1 ms
- *9binaryconst[8..0]* is set to 100110101
- *8binaryconst[7..0]* is set to 01101011
- IN1/00, IN1/01, IN1/10, IN1/11 is set to 0,0,1,1
- *IN2/00, IN2/01, IN2/10, IN2/11* is set to 0,1,0,1

In order to obtain a short time of simulation and a correct view of patterns and epochs, 4 patterns for one epoch have been considered. However, the ANN implementation can calculate *65536* epochs.

The simulation has been performed on the ALTERA FLEX 10K family EFP10K70RC240_4 FPGA device. The simulation lasted 15 minutes and 9 seconds. The implementation of this unit uses 191 total logic elements and needs 174 total input/output pins. The simulated waveforms are shown in figures 4.39 and 4.40.

As can be seen in the above mentioned figures, the *controls_unit* has a correct functionality. The patterns and the epochs are correctly counted up. The *Net1*, *Net2* and *Teach* digital signals are converted into pulse signals.

Figure 4.40 shows an example of a digital-to-pulses conversion for the following case: the pattern is *10*, the epoch is *01*, *Net1* is *1*, *Net2* is *0*, and *Teach* is *1*. The conversion resulted in: *Net1* has 255 pulses, *Net2* has 0 pulses, and *Teach* has 255 pulses.

Figure 4.38. The clock_state_commands_controls – schematic diagram

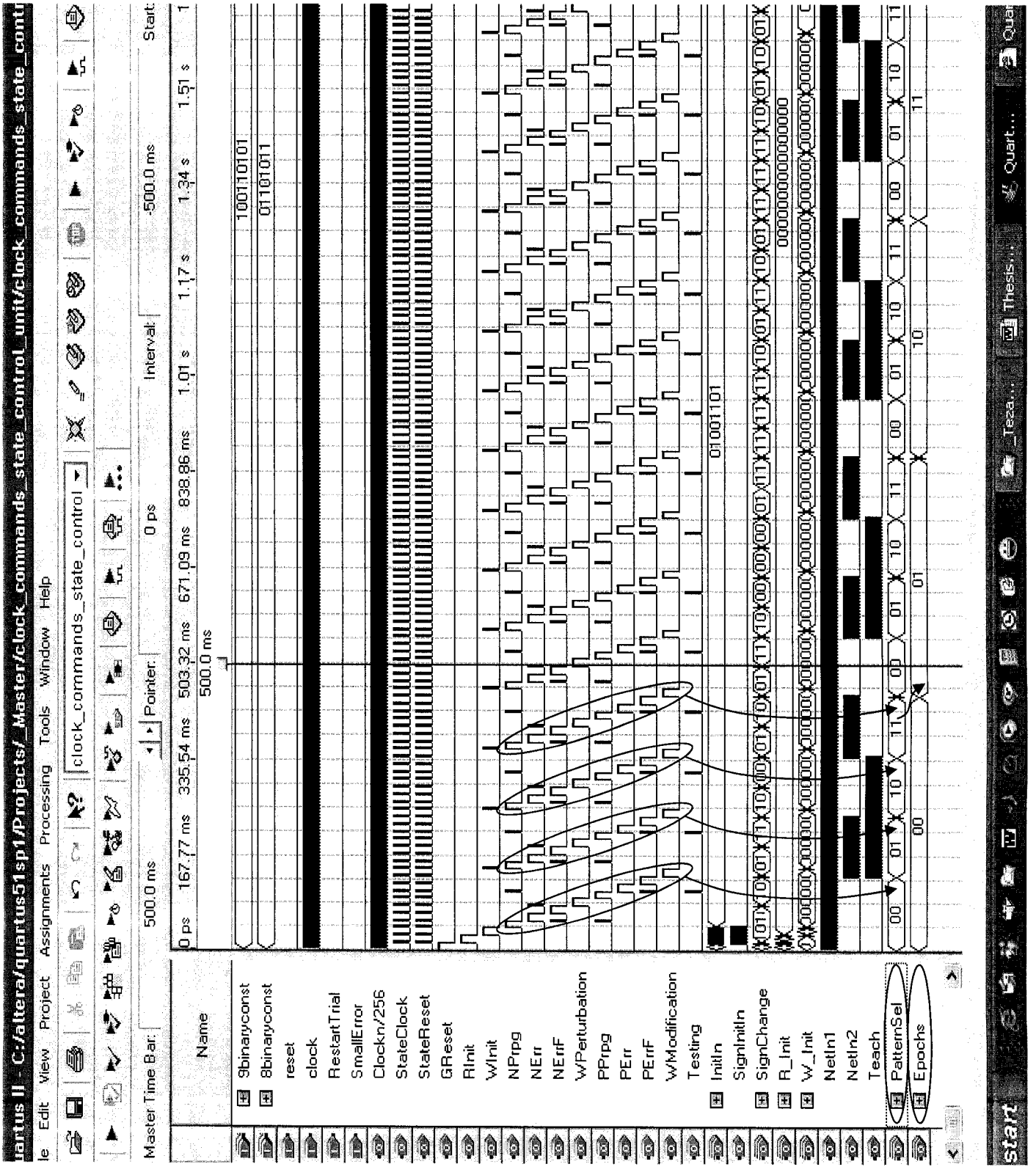


Figure 4.39. The controls_unit – wave diagram (patterns and epochs)

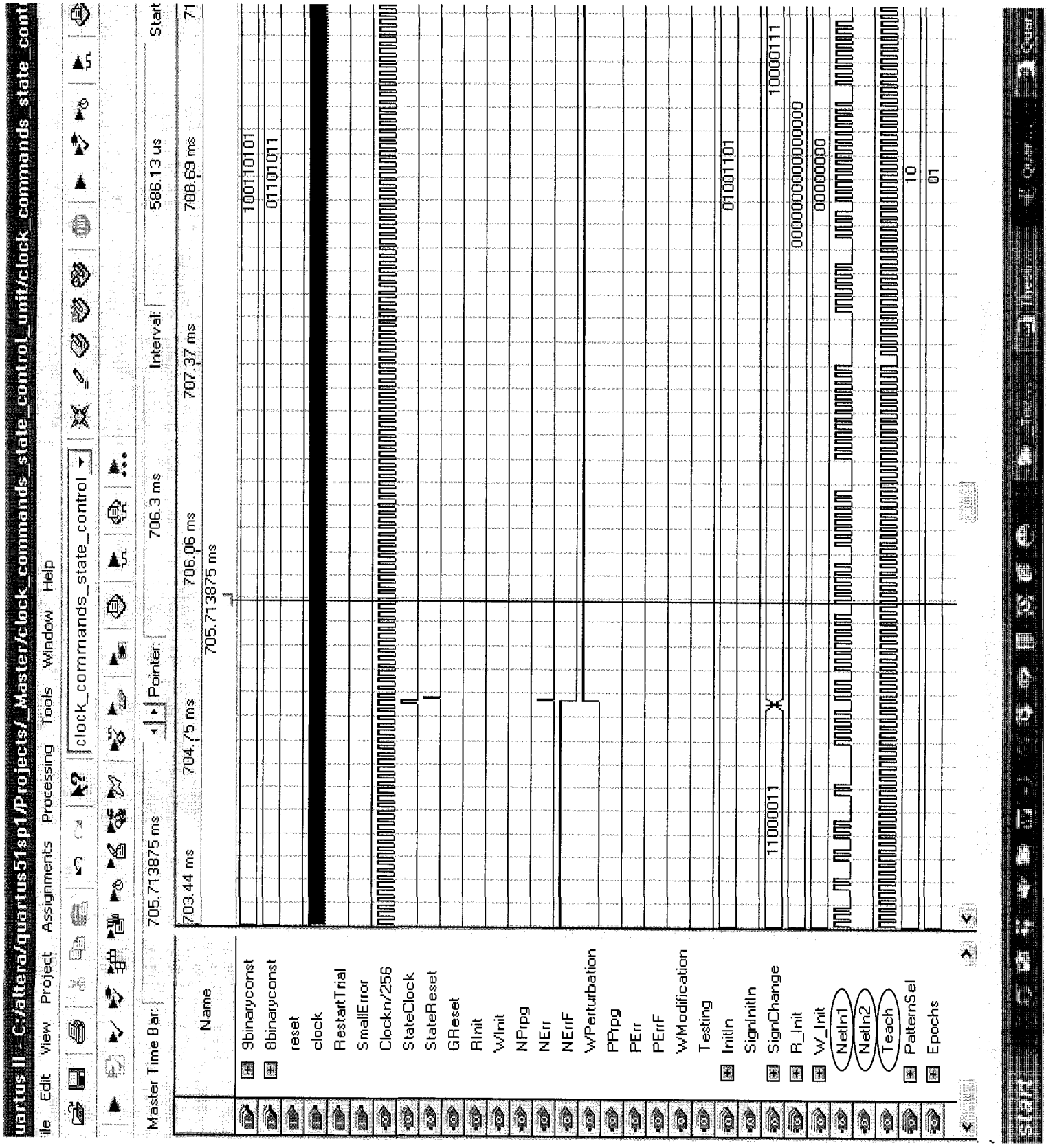


Figure 4.40. The controls_unit – wave diagram:(Net1, Net2, Teach)

4.5. The Weight Unit

As will be described, the *weight_unit* block is implemented as a distinct unit in order to reach the highest level of parallelism, which is a very important feature of a neural network.

The *weight_unit* symbol is shown in figure 4.41. This unit calculates the product of the input signals and the weight values. Moreover, the *weight_unit* updates the weight values.

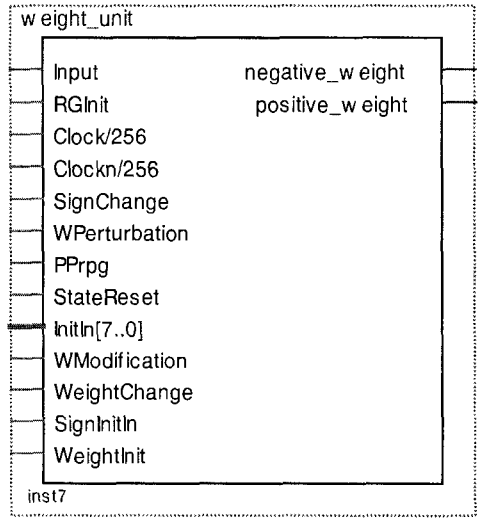


Figure 4.41. The weight_unit symbol

4.5.1. Description and functionality

Figure 4.42 illustrates the *weigh_unit* schematic diagram. The *weight_unit* consists of two counters, one *pulse_generator* block, two D flip-flops, three 1-to-2 demultiplexers and some basic gates:

- *weight-count* - is a plain binary up/down counter, which counts up if the up/down input port is set to *1* and counts down if up/down is set to *0*. It has an 8-bit output *q[7..0]*. Also, the *weight-count* counter has two other input ports: a count enable port and an asynchronous load port. The *weight-count* counters update the weight values based on the quantity received from the *learning_unit* and carries out

addition or subtraction of the perturbation. At the same time, the *weight-count* counters store the weight values.

- *pulse_generator* – this block converts the digital weight values delivered by counters into pulse signals. The *pulse_generator* block was described in section 4.4.2.
- *1to2Demux* – is a 1-line-to-2-line demultiplexer, which has a selection input. This type of demultiplexer is used in the following three ways:
  - *(inst19)* – this demultiplexer selects where the output of the *weight_unit* unit will be sent. If the sign of the result is positive, the result of the *weight_unit* unit will be sent to the positive side of the *neuron_unit*, whereas if the sign is negative, the output is sent to the negative side.
  - *(inst 20)* - this demultiplexer selects whether the D flip-flop (*inst*) is set or reset in the *PPrpg* state of the *state_machine* unit.
  - *(inst 21))* - this demultiplexer selects whether the D flip-flop (*inst1*) is set or reset in the *W_Init* state of the *state_machine* unit.

The *weight_count* counter *(inst14)* stores an initial value of a weight, which is a random value *InitIn[7..0]*, whereas the D flip-flop *(inst1)* store its corresponding sign. Then the initial value and its sign are sent to the *weight_count* counter *(inst10)* and the D flip-flop *(inst)*, respectively. During the *PPrpg* state, the *weight_countI* counter *(inst10)* sends the initial weight value to the *pulse_ generator* block, which will converted it to pulses. Next, the demultiplexer *(inst19)* separates the negative and the positive outputs.

In the *WPerturbation* state, the *weight_count* counter *(inst10)* adds or subtracts the perturbation *c* (that comes from the *learning_unit* through the *WeightChange* signal, depending on the perturbation's sign) to its saved weight value. Hence, one role of the *weight-unit* is to add a perturbation to the present weight value. This operation is simultaneously done for all

the weights in each *weight_count* counter *(inst10)* of each *weight_unit*. The perturbation c is constant.

Next, in the *WModification* state, the modifying quantity from the *learning_unit* is sent to the *weight_count* counter *(inst14)*. This quantity is common to all weights. The sign of the quantity is sent to the D flip-flop *(inst1)*, which decides whether counting up or down should be performed. The sign is generated by the *commands_unit*. These operations modify the weights according to:

$$w_{t+1} = w_t - \alpha \Delta w_t$$

The multiplication between input and weight is calculated by an AND gate *(inst17)*. All the products are computed at once in a single state cycle, i.e. 255 pulses.

The inputs of the *weight_unit* are:

- *Input* – this signal comes from the *controls_unit* and has a pulse form.
- *RGInit* – this signal comes from the *commands_unit* and is an initialisation signal. It is actually one of the *R_Init[15..0]* bit.
- *Clock/256* – this signal comes from the *clock_unit* and is the inverted general clock.
- *Clockn/256* – this signal comes from the *clock_unit* and is the general clock.
- *SignChange* [7..0] – this signal comes from the *commands_unit* and each bit of this signal goes to one of the *weight_unit* units to change the sign of the weight.
- *WPerturbation* – this signal comes from the *state_machine* unit
- *StateReset* – this signal comes from the *clock_unit* and is a positive pulse that appears in each state cycle on the first active low level of *Clockn/256*.
- *WModification* – this signal comes from the *state_machine* unit and represents the weight modification state
- *WeightChange* - this signal comes from the *learning_unit* and represents a perturbation or a modified quantity.

- *InitIn [7..0]* – this signal comes from the *commands_unit* and is the random number that represents the initial value the *weight_counter* unit is loaded when ANN begins to learn.
- *W_Init* – this signal comes from the *commands_unit* and is an initialisation signal.
- *SignInitIn* – this signal comes from the *commands_unit* and is the randomly generated signal that indicates the sign of the initial value of the weight.
- *PPrpg* – this signal comes from the *state_machine* unit and represents the perturbed forward network propagation state.

The outputs of the *weight_unit* are *negative_weight* and *positive_weight*.

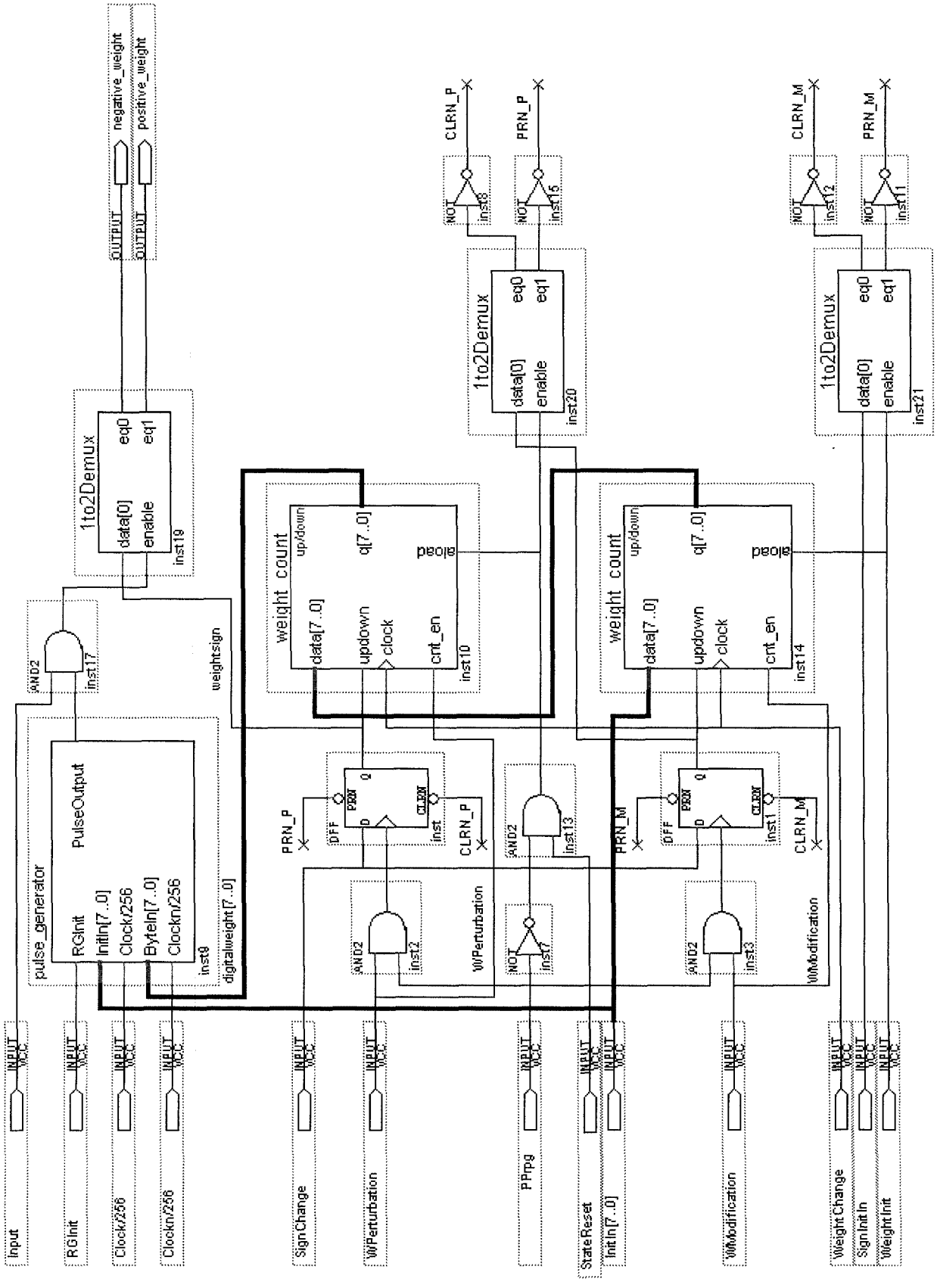


Figure 4.42. The weight_unit – schematic diagram

4.5.2. The Simulation Results

In order to simulate the *weight_unit* we need to use the *clock_commands_state_control_unit* (figure 4.37) from the previous section. The Vector Waveform File contains the following parameters:

- Maximum frequency f_{max} = 20 MHz
- End time of simulations = 2.7 s
- *reset* after 3 *clock*'s pulses
- The *clock* signal frequency f = 5 MHz, which means T = 200 ns
- One state = 255 * (256 *clock* pulses) = 255 * (200 ns x 256) = 255 * 51200 ns = 13.1 ms
- *9binaryconst[8..0]* is set to 100110101
- *8binaryconst[7..0]* is set to 01101011
- *IN1/00, IN1/01, IN1/10, IN1/11* is set to 0,0,1,1
- *IN2/00, IN2/01, IN2/10, IN2/11* is set to 0,1,0,1

In order to obtain a short time of simulation and a correct view of patterns and epochs, 4 patterns for one epoch have been considered.

The simulation has been performed on the ALTERA FLEX 10K family EFP10K70RC240_4 FPGA device.

The simulation lasted 17 minutes and 35 seconds. The implementation of this unit uses 191 total logic elements and needs 174 total input/output pins. The simulated waveforms are shown in figure 4.43.

As can be seen in the above mentioned figure, the *weight_unit* has a correct functionality. The *negative_weight* and *positive_weight* signals have the proper number of pulses and they appear in the *NPrpg* or *PPrpg* state of the *state_machine*.

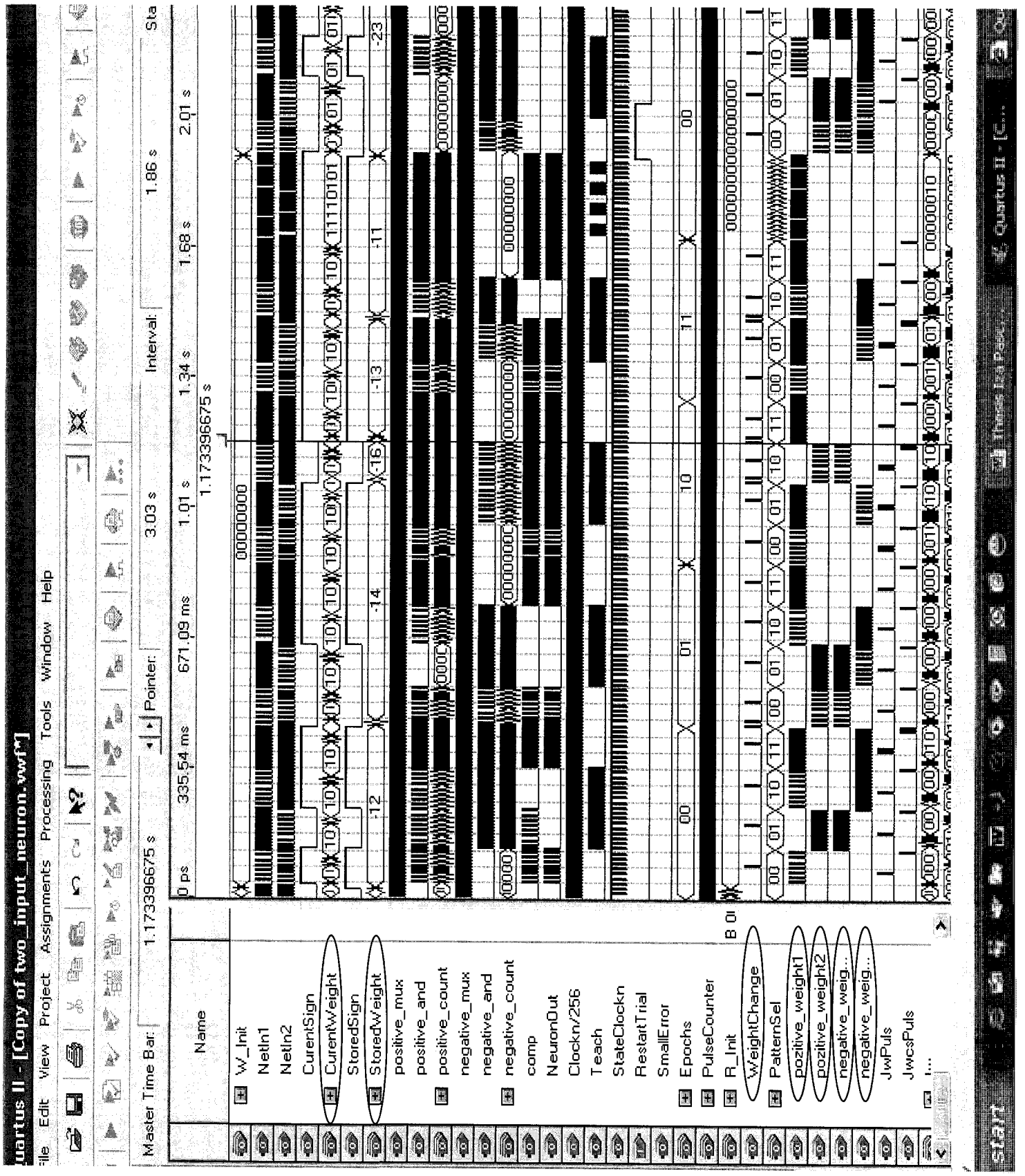


Figure 4.43. The weigth_unit – wave diagram

4.6. The Neuron Unit

This block implements an activation function by thresholding a stochastic sum. In other words, the *neuron_unit* calculates the weighted sum of its inputs. The *neuron_unit* symbol is shown in figure 4.44.

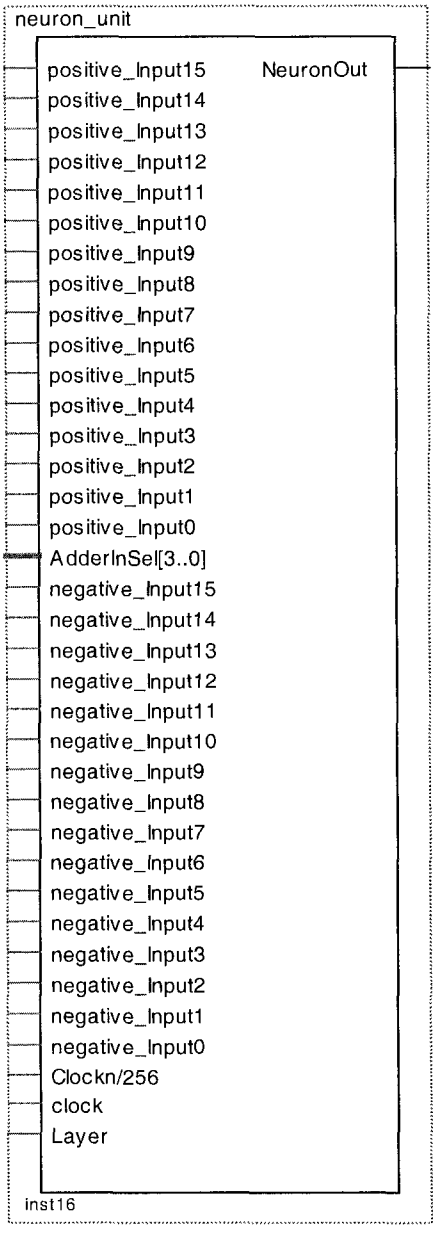


Figure 4.44. The neuron_unit symbol

4.6.1. Description and Functionality

Figure 4.45 shows the *neuron_unit* schematic diagram, which consists of two multiplexers, two counters and a comparator:

- *Mux16_1* - is a 16-to-1 multiplexer, whose outputs are selected by *AdderInSel[3..0]* from the *clock_unit.*
- *ncounter* - this counter counts the number of pulses given by the weight units. The *inst3* counter counts the number of positive inputs, whereas the *inst4* counter counts the number of negative inputs.
- *comparator_neuron* – is an unsigned 8-bit comparator that generates a positive pulse if input a (the total number of positive weight pulses) is greater than input b (the total number of negative weight pulses).

Thus, the neuron unit generates a single positive pulse if the number of positive pulses is larger than the number of negative pulses. If a weighted input sum of a neuron is very large, then the maximum number of pulses per unit time is limited. If the weighted input sum of a neuron is less than the lowest limit of the output, then no pulse in generated. Between these two extreme situations, the number of the output pulses is equal to the weighted sum of inputs.

The above described input–output behavior of the neuron unit is characterized by a piecewise-linear function whose amplification factor is 1. This function is determined by the saturation of pulse density. In conclusion, the system uses a linear function with a restriction applied [14].

The inputs of the *neuron_unit* are:

- *Layern* - this signal comes from the *clock_unit* and corresponds to the layer in which the neuron is located. For example, if we have a neuron in layer 1, then the name of this signal will be *Layer1*.

- *clock* – is a positive-edge-triggered clock that is the general clock of the entire ANN implementation and is externally generated. This signal comes from the *clock_unit.*
- *positive_weight* - this signal comes from the *weight_unit,* and represents the number of positive pulses of the weight signal.
- *negative_weight* - this signal comes from the *weight_unit,* and represents the negative of positive pulses of the weight signal.
- *Clockn/256* – this signal comes from the *clock_unit* and is the general clock.
- *AdderInSel[3..0]* – this signal comes from the *clock_unit* and is actually formed from the first four least significant bits of *AdderInSel[7..0]* signal and are to address the *Mux16_1* multiplexer.

The output of the *neuron_unit* is the *NeuronOut* signal.

As it can be seen from the *neuron_unit* schematic diagram (figure 4.43), the clock for the *neuron_unit* is much faster then *Clockn/256.* This allows for all the inputs of the *Mux16_1* multiplexers *(inst and inst1)* to be added by the *ncounter* counters *(inst3 and inst4).*

The *ncounter* counts up all the inputs of the *Mux16_1* multiplexers *(inst and inst)* only during the specific layer pulse.

To cope with the glitches that appear at the outputs of the *Mux16_1* multiplexers *(inst and inst1),* two *clock*-ed AND gates *(inst8* and *inst2)* are needed, which clean the signal applied to the *ncounter* counters *(inst3 and inst4).* The glitches that appear on the *NeuronOut* output signal of the *neuron_unit* are eliminated by using the D-flip-flop *inst6.*

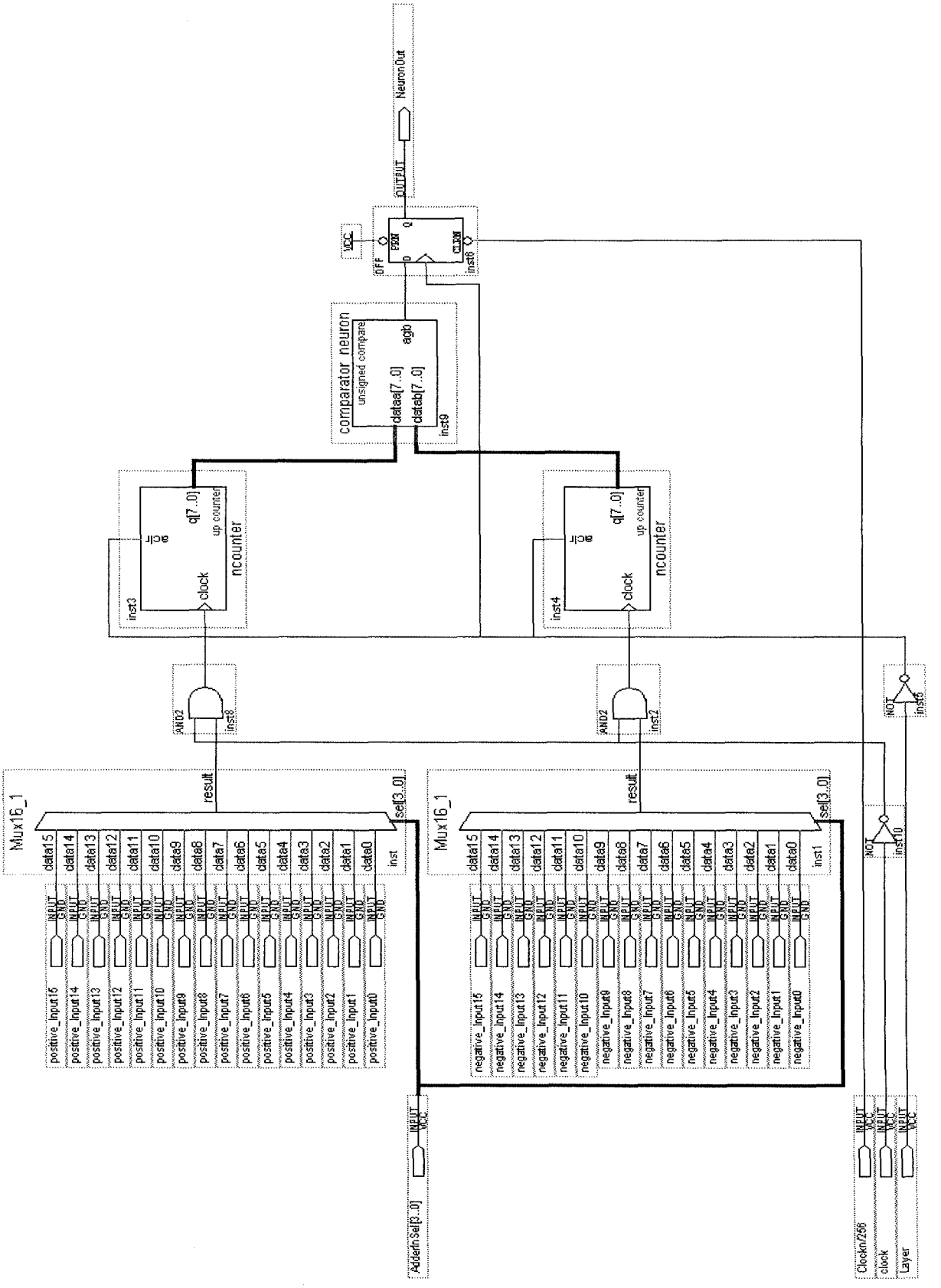


Figure 4.45. The neuron_unit – schematic diagram

4.6.2. The Simulation Results

In order to simulate the *neuron_unit* we need to use the *clock_commands_state_control_unit* (figure 4.37) and two *weight_unit* units. The Vector Waveform File contains the following parameters:

- Maximum frequency f_{max} = 20 MHz
- End time of simulations = 2.7 s
- *reset* after 3 *clock*'s pulses
- The *clock* signal frequency f = 5 MHz, which means T = 200 ns
- One state = 255 * (256 *clock* pulses) = 255 * (200 ns x 256) = 255 * 51200 ns = 13.1 ms
- *9binaryconst[8..0]* is set to 100110101
- *8binaryconst[7..0]* is set to 01101011
- *IN1/00, IN1/01, IN1/10, IN1/11* is set to 0,0,1,1
- *IN2/00, IN2/01, IN2/10, IN2/11* is set to 0,1,0,1

Again, in order to obtain a short time of simulation and a correct view of patterns and epochs, 4 patterns for one epoch have been considered.

The simulation has been performed on the ALTERA FLEX 10K family EFP10K70RC240_4 FPGA device. The simulation lasted 21 minutes and 20 seconds. The implementation of this unit uses 191 total logic elements and needs 174 total input/output pins. The simulated waveforms are shown in figure 4.46.

The *NeuronOut* signal has the proper number of pulses and it appears in the *NPrpg* or *PPrpg* state of the *state_machine*. The *positive_count, positive_mux, positive_and, negative_count, negative_mux,* and *negative_and* signals are shown in order to understand the correct functionality of the *neuron_unit* unit.

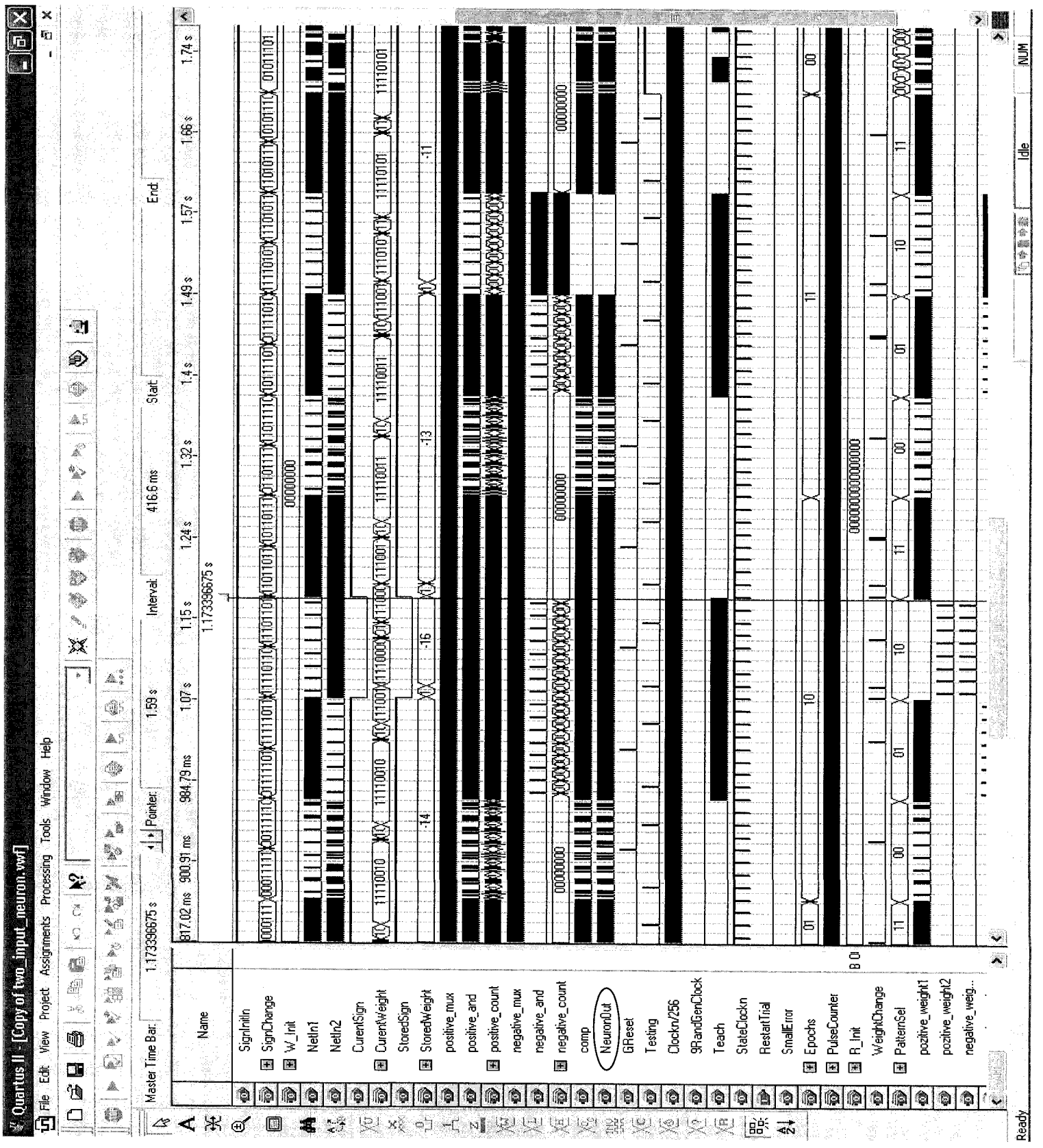


Figure 4.46. The neuron_unit - wave diagram

4.7. The Learning Unit

The *learning_unit* calculates the basic modifying quantity using simultaneous perturbation:

$$\Delta w_t^{\ i} = s_t^{\ i} \frac{J(w_t + cs_t) - J(w_t)}{c}$$

This quantity is common for all weights and is sent to all *weight_unit* units. The learning rule described by this formula requires only forward operations of the ANN.

The symbol of the *learning_unit* is shown in figure.4.47.

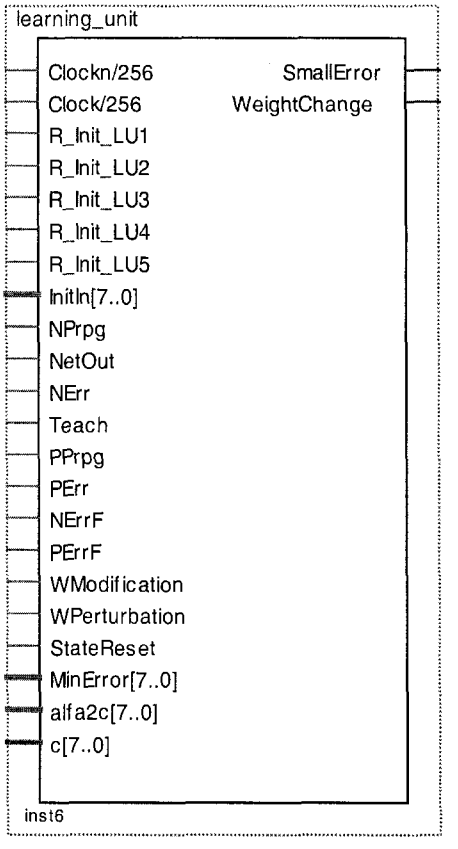


Figure 4.47. The learning_unit symbol

4.7.1. Description and Functionality

The *learning_unit* block consists of two *error_unit* blocks, five *pulse_generator* blocks, one counter and one comparator, as it can be seen in figure 4.48.

- *error_unit (inst19, inst20)* – this block will be described in section 4.7.2.

  - inst19 - calculates the normal error

  - *inst20 – calculates the* perturbed error

- *pulse_generator (inst17, inst21, inst18, inst24, inst25)* – this block has been described in section 4.4.2.

  - *inst17* – transforms the normal error into pulses

  - *inst21* – transforms the perturbed error into pulses

  - *inst18* – transforms the difference *J(w+cs)-J(w)* into pulses

  - *inst24* - transforms *a/2c* into pulses

  - *inst25* – transforms *c* into pulses

- *learning_counter (inst22)* – is a plain binary up/down counter, which counts up if the up/down input is set to *1* and counts down if the up/down input is set to *0*. The output of the counter is an 8-bit bus *q[7..0]*. The *error_counter* has a count enable input port and an asynchronous load input port. The counter calculates the difference (in number of pulses) between the following two error functions: *J(w)* and *J(w+cs)*.

- *learning_comp (inst23)* – is an unsigned 8-bit comparator that generates a positive pulse if input *a* (*MinError [7..0]*) is greater than input *b* (the absolute value of *J(w+cs) - J(w)*). This comparator sends a positive pulse to the *controls_unit* to stop the learning process because the difference between the two error functions *J(w)* and *J(w+cs)* is less than 0.01.

- *Abs_value (inst3)* - is one of the arithmetic megafunctions provided by the Quartus II software, function that computes the absolute value of *J(w+cs) - J(w)*..

The inputs of the *learning_unit* block are:

- *Clock/256* – this signal comes from the *clock_unit* and is the inverted general clock.
- *Clockn/256* – this signal comes from the *clock_unit* and is the general clock.
- *R_Init_Lu1* – this signal comes from the *commands_unit* and is an initialisation signal. It is actually bit 3 of *R_Init[15..0]*.
- *R_Init_Lu2* – this signal comes from the *commands_unit* and is an initialisation signal. It is actually bit 4 of *R_Init[15..0]*.
- *R_Init_Lu3* – this signal comes from the *commands_unit* and is an initialisation signal. It is actually bit 5 of *R_Init[15..0]*.
- *R_Init_Lu4* – this signal comes from the *commands_unit* and is an initialisation signal. It is actually bit 6 of *R_Init[15..0]*.
- *R_Init_Lu5* – this signal comes from the *commands_unit* and is an initialisation signal. It is actually bit 7 of *R_Init[15..0]*.
- *InitIn[7..0]* – this signal comes from the *commands_unit* and is the random number that represents the initial value the *pulse_generator* unit is loaded when ANN begins to learn.
- *StateReset* – this signal comes from the *clock_unit* and is a positive pulse that appears in each state cycle on the first active low level of *Clockn/256*.
- *NPrpg* – this signal comes from the *state_machine* unit and represents the Normal Forward Network Propagation state.
- *NetOut* – this signal comes from the *neuron_unit* and is the output of the neuron.
- *NErr* – this signal comes from the *state_machine* unit and represents the Normal Output Error Calculation state.
- *Teach* – this signal comes from the *controls_unit* and represents the *Teaching Input* signal converted to a random pulse representation signal, which has a number of pulses between *0* and *255*.

- *PPrpg* – this signal comes from the *state_machine* unit and represents the Perturbed Forward Network Propagation state.
- *PErr* – this signal comes from the *state_machine* unit and represents the Perturbed Output Error Calculation state.
- *NErrF* – this signal comes from the *state_machine* unit and represents the Normal Error Function Calculation state.
- *PErrF* – this signal comes from the *state_machine* unit and represents the Perturbed Error Function Calculation state.
- *WModification* – this signal comes from the *state_machine* unit and represents the Weight Modifications state.
- *WPerturbation* – this signal comes from the *state_machine* unit and represents the Weight Perturbation state.
- *MinError [7..0]* – this is an external constant (*0.01*) translated in an 8-bit format
- *Alfa2c[7..0]* – this is an external constant (*a/2c*) translated in an 8-bit format
- c [7..0] – this is an external constant (the perturbation*c*) translated in an 8-bit format

The outputs of the *learning_unit* unit are the *SmallError* and the *WeightChange* signals.

At the end of the *NPrpg* state, the *error_unit (int19)* computes the error *J(w)* between the output of the neuron and its teaching signal. Next, in the *NErrF* state, the *pulse_generator (inst 17)* sends the number of the pulses (corresponding to the error *J(w)* received from the *error_unit (int19))* to the *learning_counter.*

In a similar way, at the end of the *PPrpg* state, the *error_unit (int20)* computes the perturbed error *J(w+cs)* and in the *PErrF* state, the *pulse_generator (inst 21)* forwards the number of the pulses (corresponding to the perturbed error *J(w+cs))* to the *learning_counter* .

The *learning_counter* is designed such that the error *J(w)* is counted down, and then the perturbed error *J(w+cs)* is counted up. At the end of the *PErrf* state we obtain the difference between the perturbed error and the normal error.

The learning coefficient α and the perturbation c are constants during a trial. The value of these parameters is transformed into a number of pulses by the *pulse_generator* blocks *(inst24* and *inst25)*.

Also, the difference between *J(w+cs)* and *J(w)* is transformed into pulses through the *pulse_generator (inst18)*. By multiplying this difference by $\alpha/2c$, the *learning_unit* computes the basic modifying quantity, which is then sent to all the *weight_unit* units.

During the *WPerturbation* or *WModification* states, a perturbation or a modified quantity, respectively, which are obtained from the *learning_unit*, will be sent to the *weight_unit* units through the *WeightChange* signal.

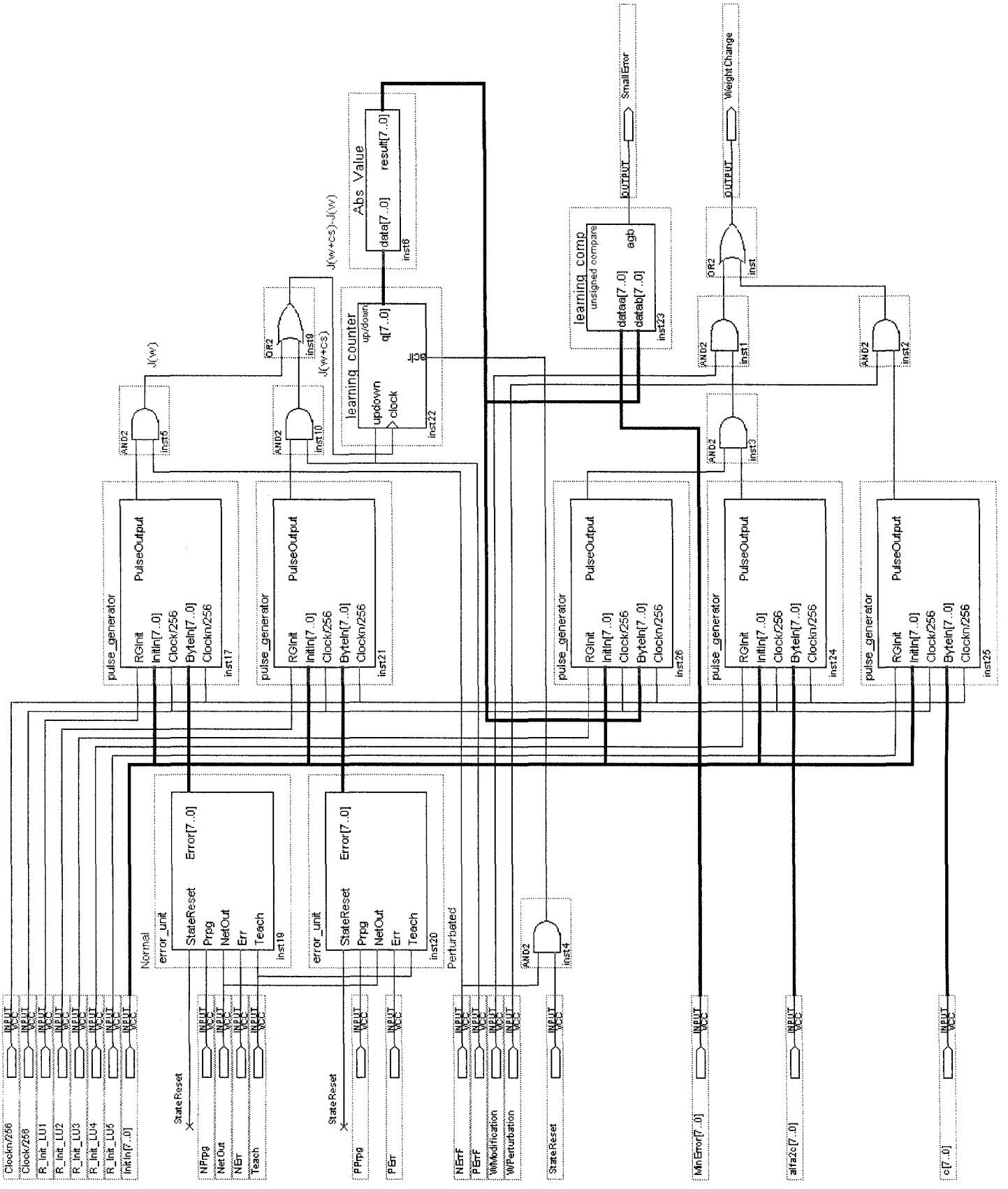


Figure 4.48. The learning_unit - schematic diagram

4.7.2. The Error Unit

Figure 4.49 shows the symbol of the *error_unit*.

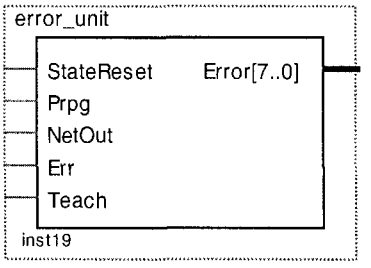


Figure 4.49. The error_unit symbol

The *error_unit* block calculates the absolute difference (in number of pulses) between the output of the ANN and the corresponding teaching signal.

$$J(w) = |o - d|$$

By counting up for the output pulses and counting down for the teaching pulses, the error_unit block provides the error.

Figure 4.50 shows the *error_unit* schematic diagram.

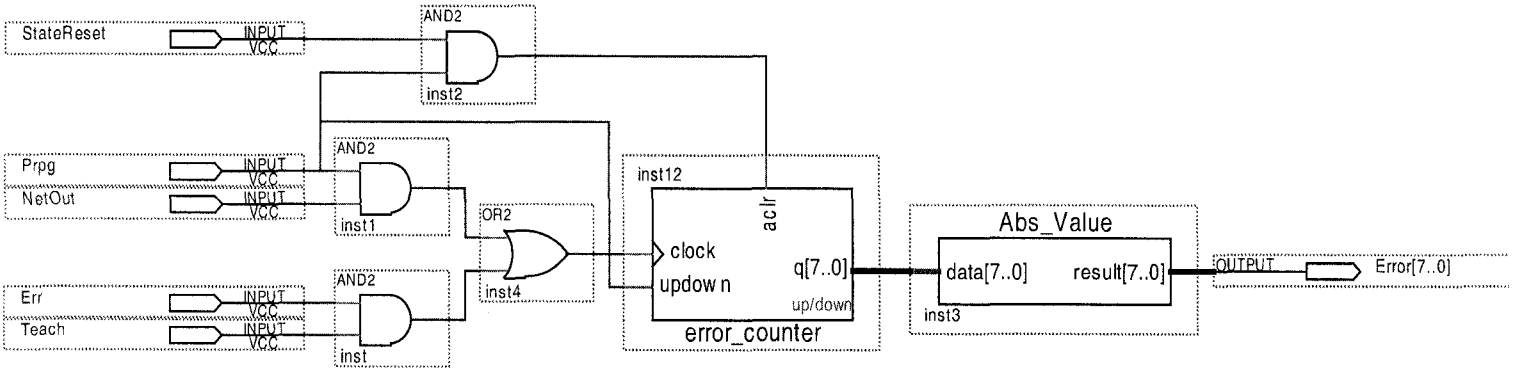


Figure 4.50. The error_unit – schematic diagram

The *error_*unit consists of a counter, a few gates and a block that computes the absolute value of the error.

- *error_counter* – is a plain binary up/down counter, which counts up if up/down is set to 1 and counts down if up/down is set to 0. The output of the counter is an 8-bit bus *q[7..0]*. The *error_counter* has a count enable input port and an asynchronous load input port. The counter calculates the difference (in number of pulses) between the output of the ANN and the corresponding teaching signal.
- *Abs_value (inst3)* - is one of the arithmetic megafunctions provided by the Quartus II software, function that computes the absolute value of $J(w) = |o - d|$.

After perceiving the *StateReset* pulse in *Prpg* state, the *error_counter* counts up the *Netout* pulses, which come from the *neuron_unit*. Next, after the *error_counter* perceives the *StateReset* pulse in *PErr* state, the *error_counter* counts down the *Teach* pulses, which arrive from the *controls_unit*. Finally, the *Abs_value* block calculates the absolute value of the error: $J(w) = |o - d|$

This block is used for both normal error and perturbed error calculations in the *NPrpg* and *PPrpg* states of the *state_machine* unit.

4.7.3. The Simulation Results

In order to simulate the *learning_unit* we need to use the *clock_commands_state_control_unit* (figure 4.37), two *weight_unit* units and one *neuron_unit* unit. This way we obtain the new schematic diagram for a two-input neuron with the ability to learn, as it can be seen in figures 4.2 and 4.3. The Vector Waveform File contains the following parameters:

- Maximum frequency f_{max} = 20 MHz
- End time of simulations = 2.7 s

- *reset* after 3 *clock*'s pulses
- The *clock* signal frequency f = 5 MHz, which means T = 200 ns
- One state = 255 * (256 *clock* pulses) = 255 * (200 ns x 256) = 255 * 51200 ns = 13.1 ms
- *9binaryconst[8..0]* is set to 100110101
- *8binaryconst[7..0]* is set to 01101011
- *IN1/00, IN1/01, IN1/10, IN1/11* is set to 0,0,1,1
- *IN2/00, IN2/01, IN2/10, IN2/11* is set to 0,1,0,1

In this case too, in order to obtain a short time of simulation and a correct view of patterns and epochs, 4 patterns for one epoch have been considered.

The simulation has been performed on the ALTERA FLEX 10K family EFP10K70RC240_4 FPGA device. The simulation lasted 25 minutes and 49 seconds. The implementation of this unit uses 191 total logic elements and needs 174 total input/output pins. The simulated waveforms are shown in figure 4.51.

As can be seen, the *learning_unit* has a correct functionality. The *SmallError* and *WeightChange* signals appear in the desired state of the *state_machine.* Also, the *Error* calculation is accurate and comes out in the *NErrF* and *PErrF* states of the *state_machine.*

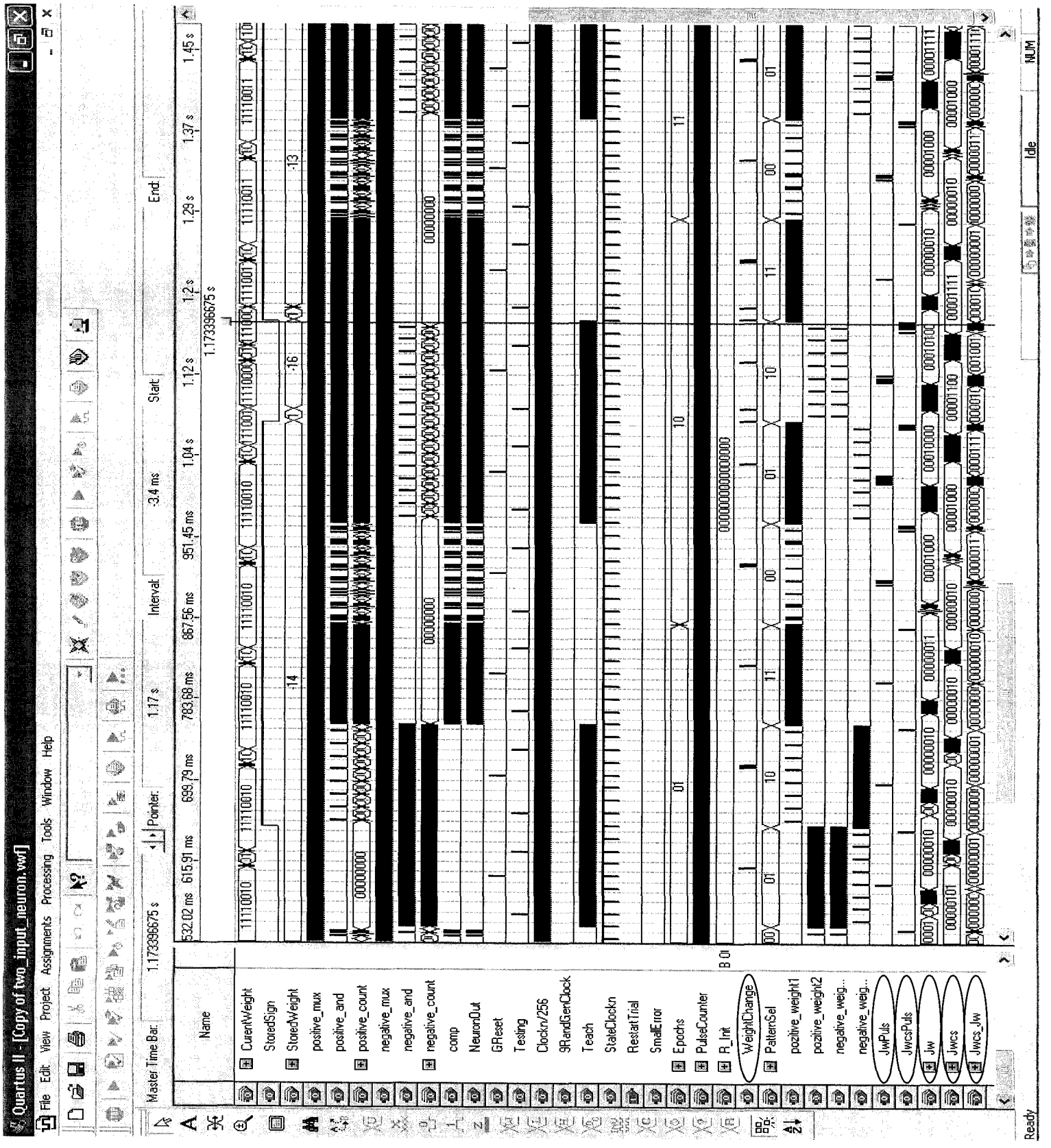


Figure 4.51. The learning_unit – wave diagram

Chapter 5

Applications

This chapter contains two examples that I designed using the hardware neuron implementation. First, an implementation of an analogical function y = 1 - x is presented. Then, an implementation of the digital function XOR is shown.

5.1. The Implementation of the Analogical Function y = 1 - x

Figures 5.1, 5.2, 5.3 and 5.4 show the schematic diagram of the implementation of the artificial neural network that performs the function y = 1 – x.

This specific ANN has three layers, one input, one output and four neurons in the hidden layer. In addition to the weight and neuron blocks, there is another block for commands and controls, named *clock_commands_state_control_big* unit. This block is designed to supply 22 *R_Init* signals needed for initializing the pulse generators, and 9 *W_Init* signals needed to set the weight units. The four blocks *inst10, inst12, inst13* and *inst14 (*form *Layer2)* are needed for implementing a positive and a negative bias. We select 4 points as inputs: 1.0, 0.666, 0.333, and 0.0. According to the rule described in section 4.4.3, these signals have the following corresponding number of pulses: 255 pulses, 0.666 x 255 = 170 pulses, 0.333 x 255 =

85 pulses, and 0 = 0 pulses, respectively. The teaching signals for these inputs are: 0, 0.333, 0.666, 1 and the number of the pulses are: 0 pulses, 85 pulses, 170 pulses and 255 pulses, respectively. Thus, this implementation has only four patterns. The learning coefficient is set to $a = 0.024$ and the perturbation to $C = 0.098$ (corresponding to 25 pulses). The resulting ratio is $a/c = 0.25$ (corresponding to 64 pulses).

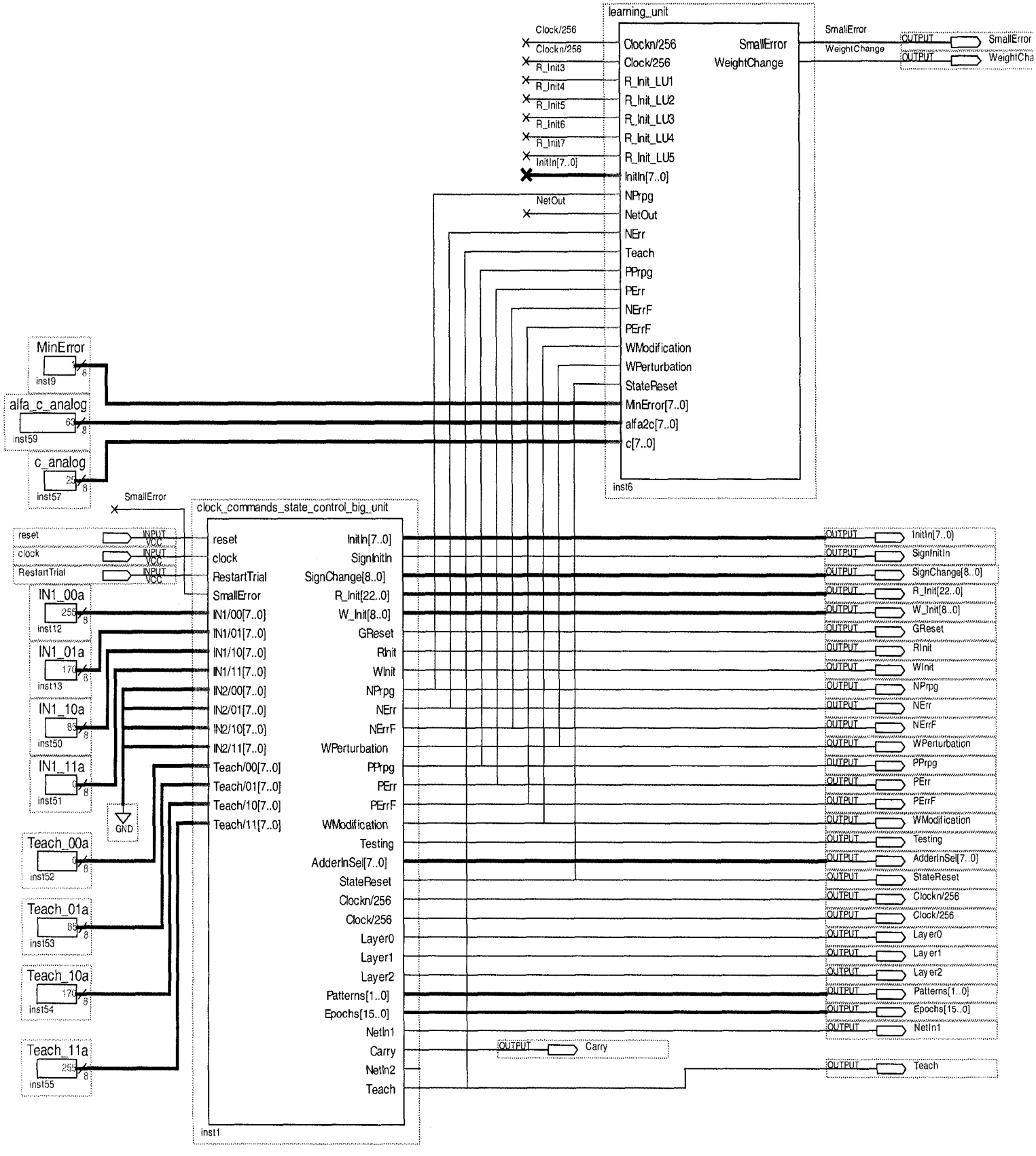


Figure 5.1. The function y = 1 - x implementation - schematic diagram (part a)

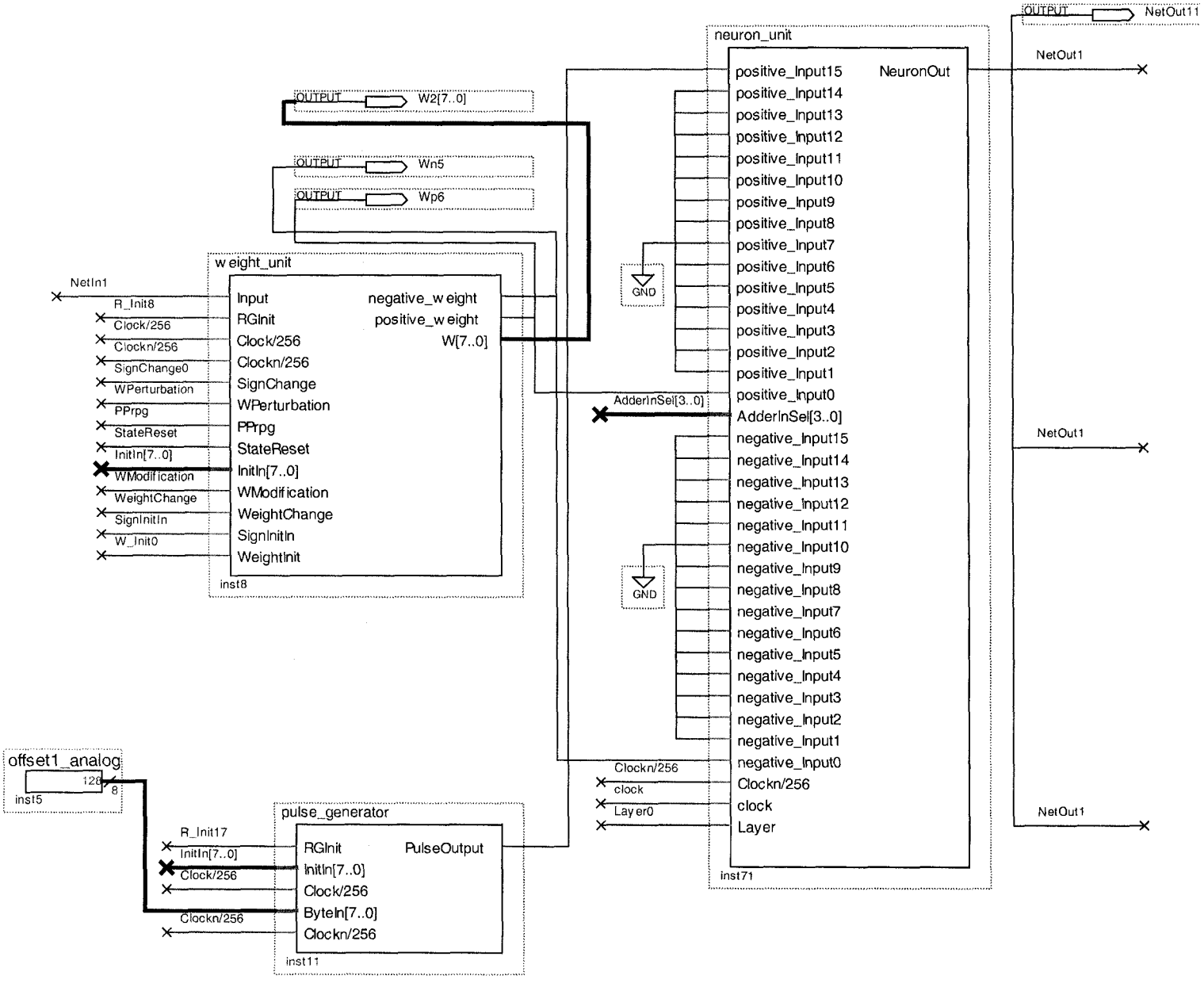


Figure 5.2. The function y = 1 - x implementation - schematic diagram (Layer0 - part b)

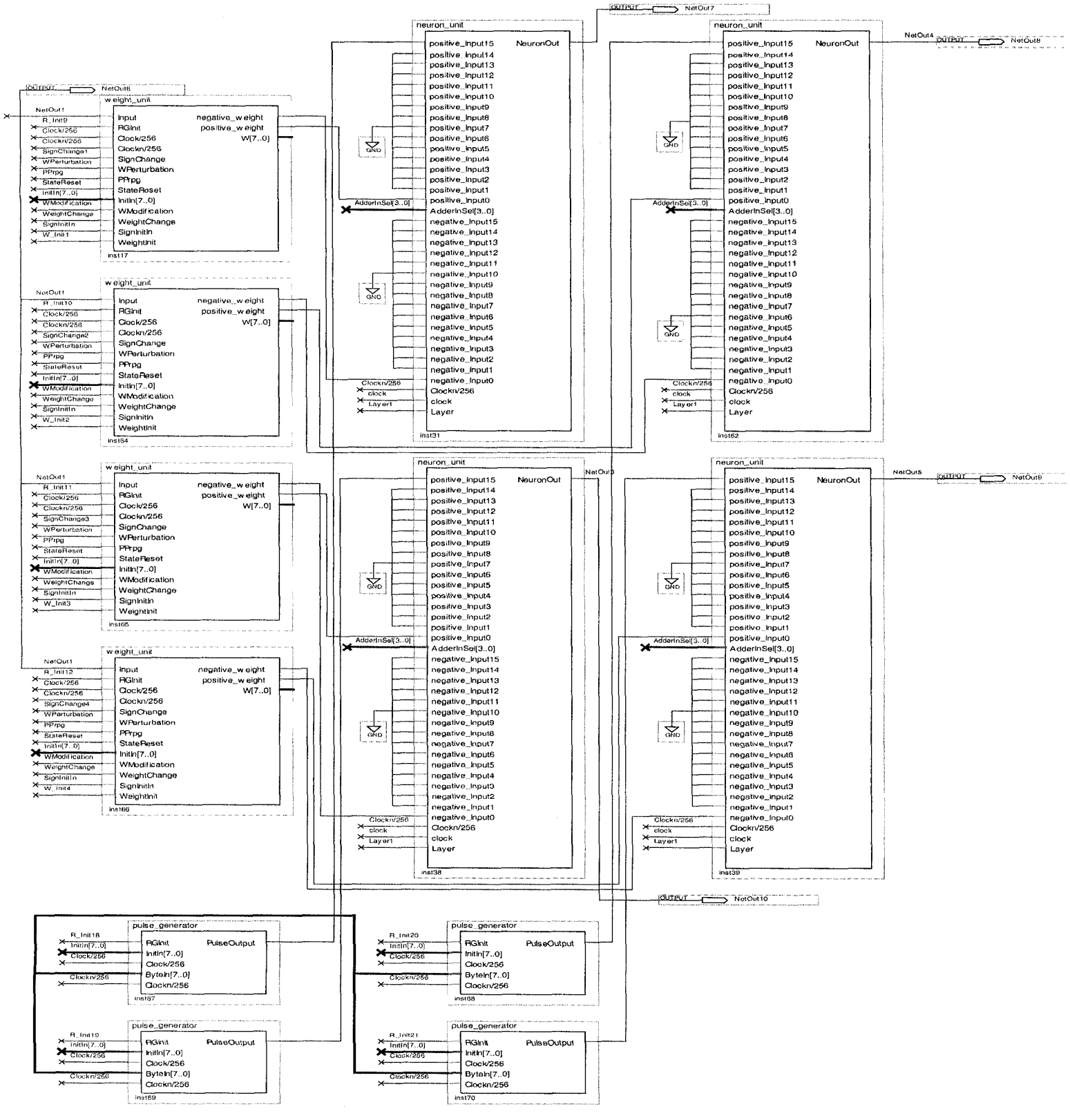


Figure 5.3. The function y = 1 - x implementation - schematic diagram (Layer1 - part c)

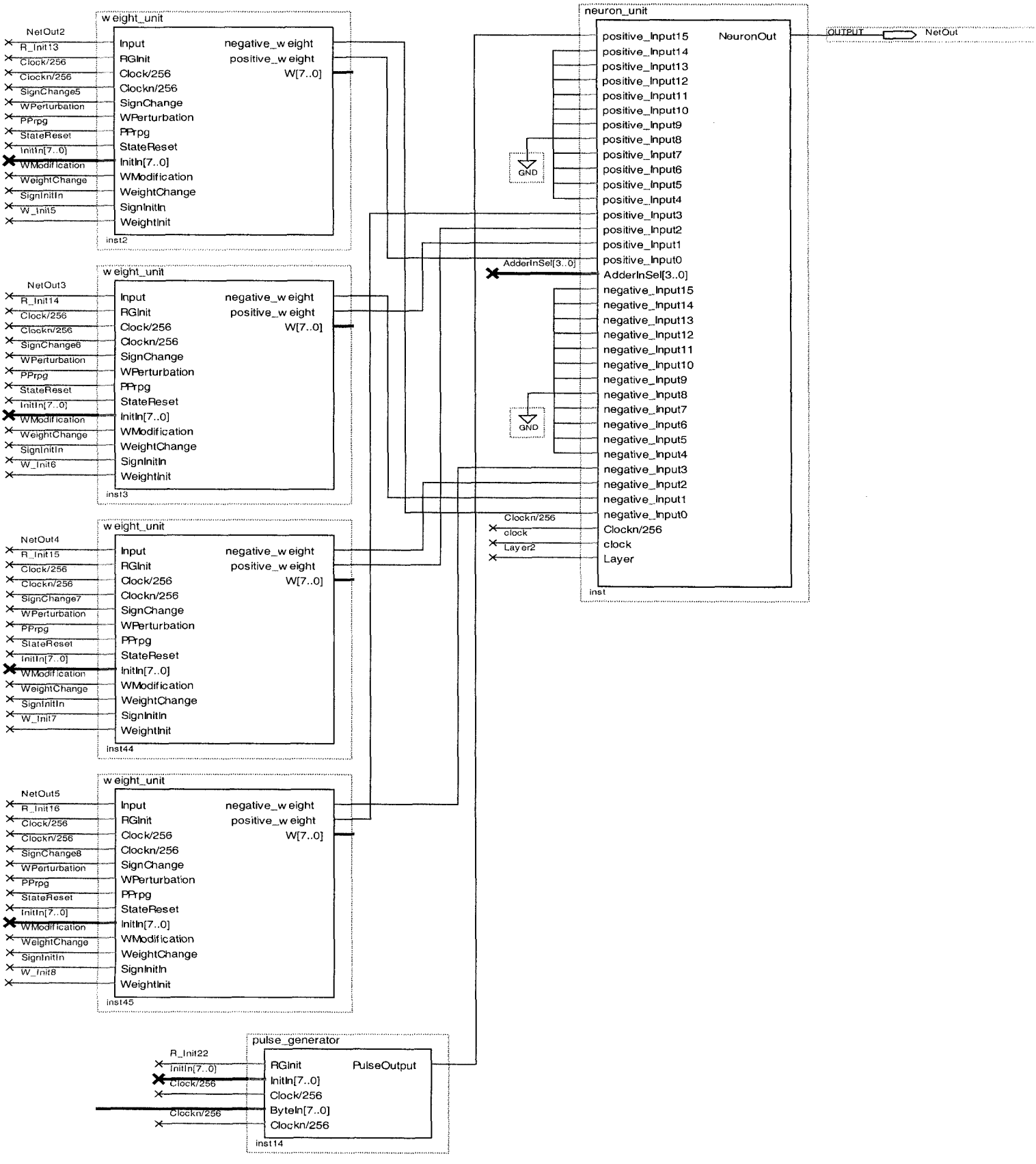


Figure 5.4. The function y = 1 - x implementation - schematic diagram (Layer2 - part d)

5.1.1. The Simulation Results

The Vector Waveform File contains the following parameters:

- End time of simulations = 10 s
- *reset* after 3 *clock*'s pulses
- The *clock* signal frequency f = 10MHz, which means T = 100 ns
- One state = 255 * (256 *clock* pulses) = 255 * (100 ns x 256) = 255 * 25600 ns = 6.528 ms
- *9binaryconst[8..0]* is set to 100110101
- *8binaryconst[7..0]* is set to 01101011
- *MinError* is set to 000000001
- *Offset* = 128

The simulation has been performed on the ALTERA SRATIX II family EP2S15F672I4 FPGA device. The simulation lasted 10 hours and 26 minutes. The implementation of this unit uses 533 registers. The simulated waveforms are shown in figures 5.5, 5.6, and 5.7.

Figure 5.5 shows the first 50 epochs of the simulation of the y = 1 - x function.

Figure 5.6 shows a detail of the simulation of the y = 1 - x function.

Figure 5.7 shows the simulation for an improperly chosen learning coefficient ($a/c = 127$ pulses), which makes the error function reach a local minim and triggers the network to pass in the testing state.

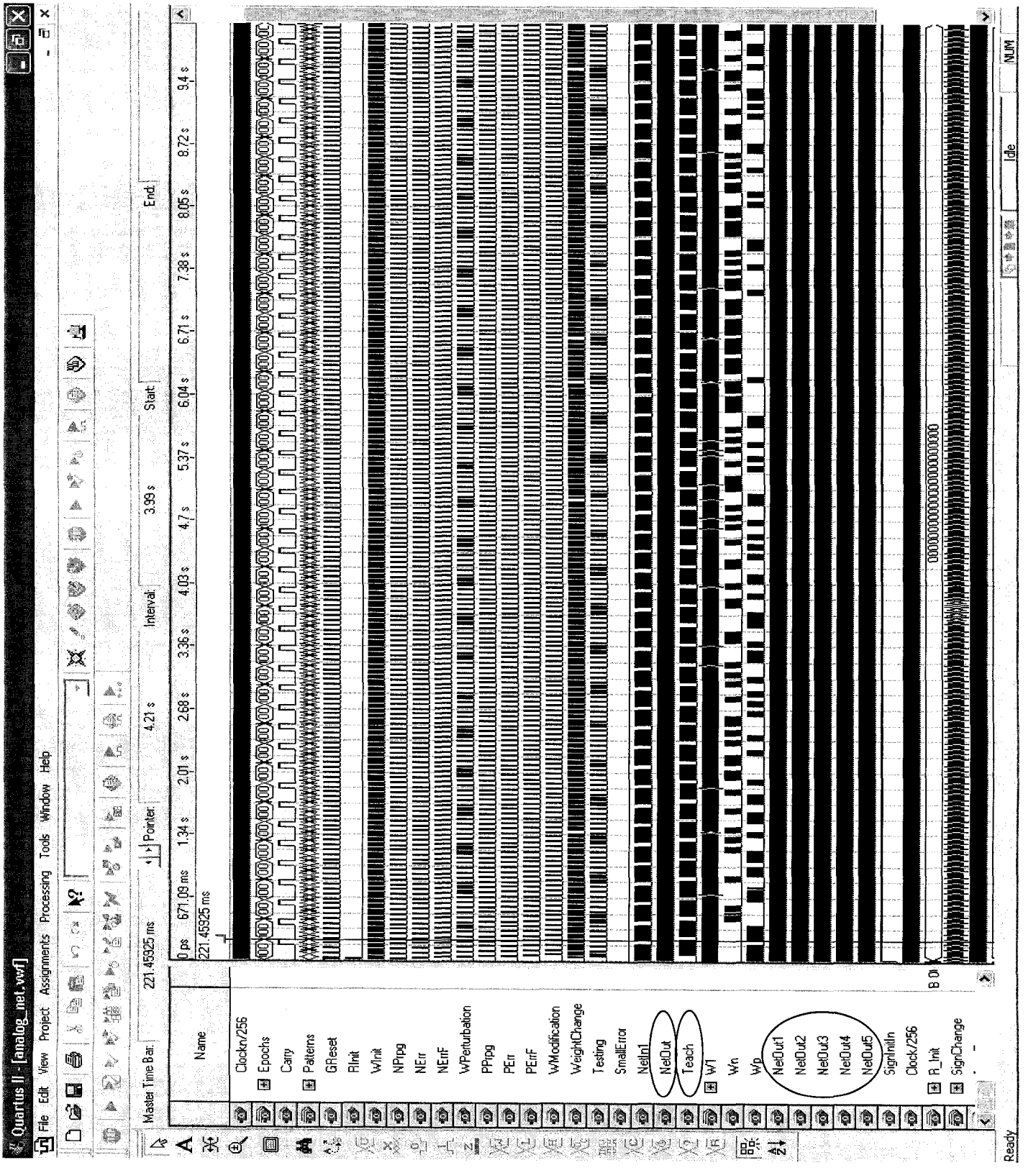


Figure 5.5 The function y = 1 - x implementation – wave diagram

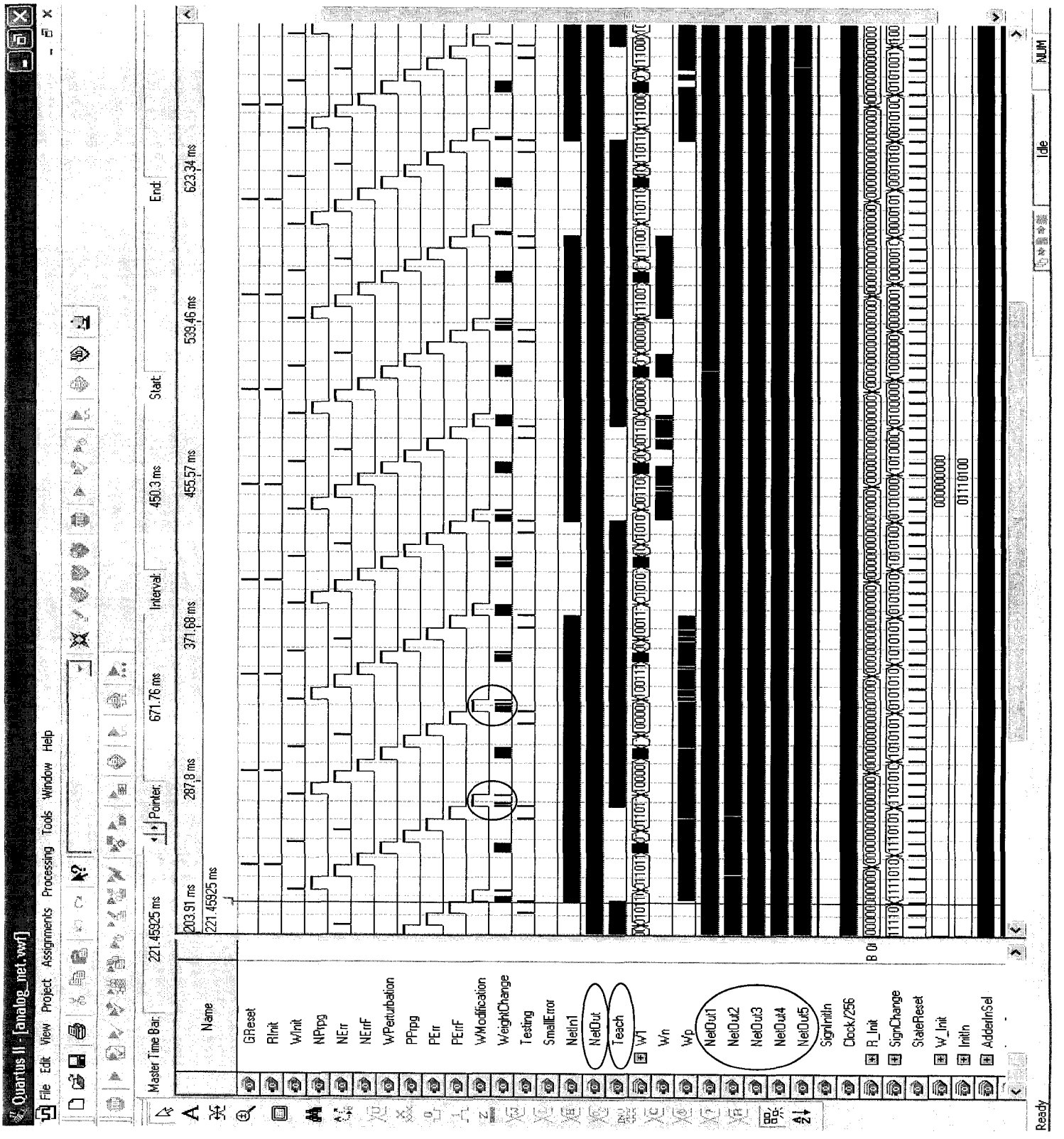


Figure 5.6 The function y = 1 - x implementation – wave diagram (detail)

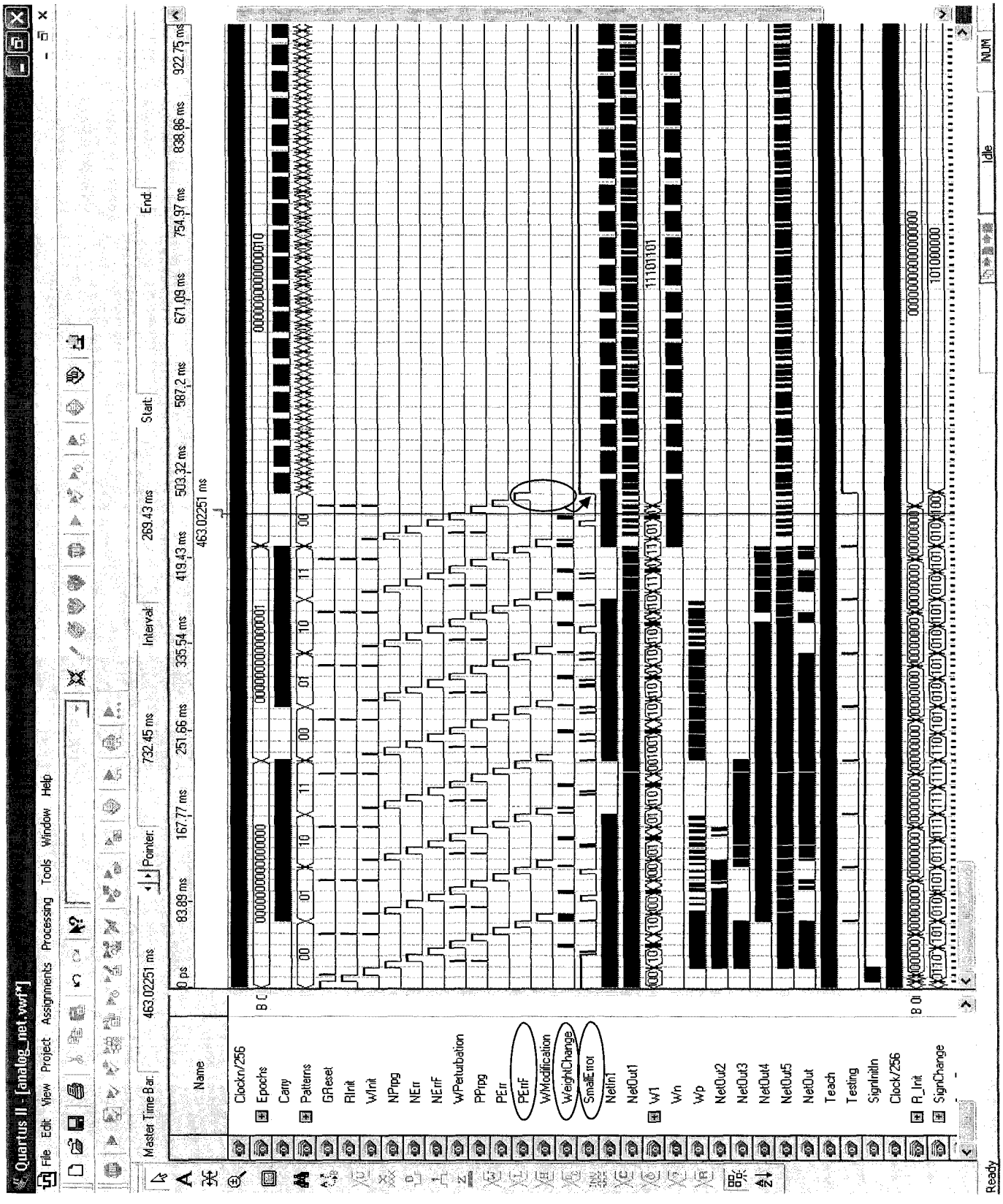


Figure 5.7 The function y = 1 - x implementation with great learning coefficient – wave diagram

5.2. The Implementation of the Digital Function XOR

This section presents a hardware implementation of a neural network that is capable to learn the digital function XOR. The inputs, teaching signals and patterns used in the network are shown in figure 5.8.

XOR	I_1	I_2	Teaching signal
Pattern1	0	0	0
Pattern2	0	1	1
Pattern3	1	0	1
Pattern4	1	1	0

Figure 5.8. The truth table for the two-input XOR gate

The hardware implementation of the XOR function consists of 3 layers, 2 inputs, 1 output, and 2 neurons in the hidden layer. This means that we need 8 weight units and 16 pulse generator units, thus the *W_Init* signal will have 8 bits, whereas the *R_Init* signal will have 16 bits. The learning coefficient is set to $a = 0.024$ and the perturbation to $C = 0.098$ (corresponding to 25 pulses). The resulting ratio is $a/C = 0.25$ (corresponding to 64 pulses). The input, output, and teaching signals are represented by the same rule described in section 4.4.3, in which 0 logic is transformed into 0 pulses and 1 logic to 255 pulses.

Figure 5.9 and 5.10 illustrate the schematic diagram of the XOR artificial network hardware implementation.

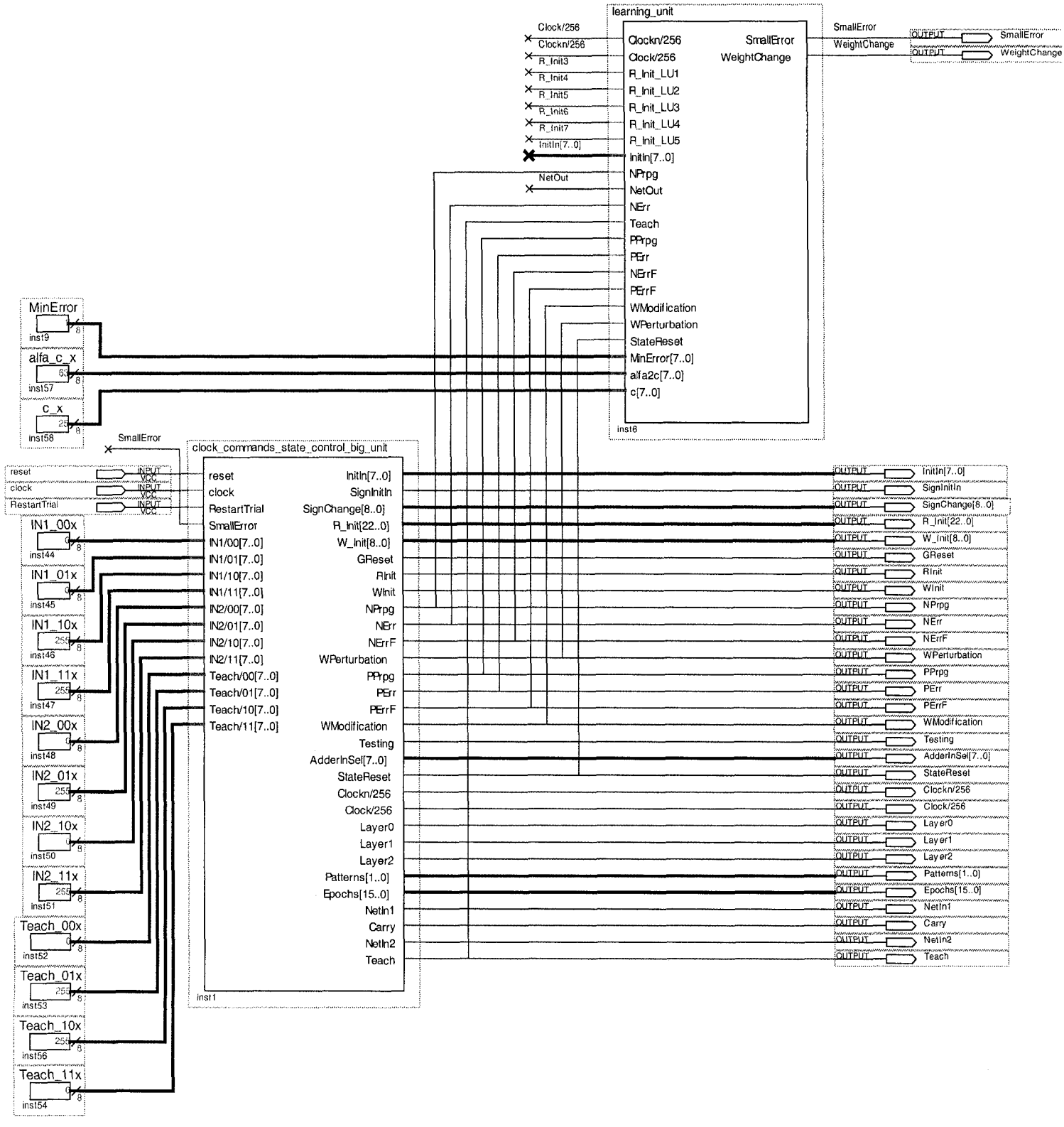


Figure 5.9. The two-input XOR - schematic diagram (part a)

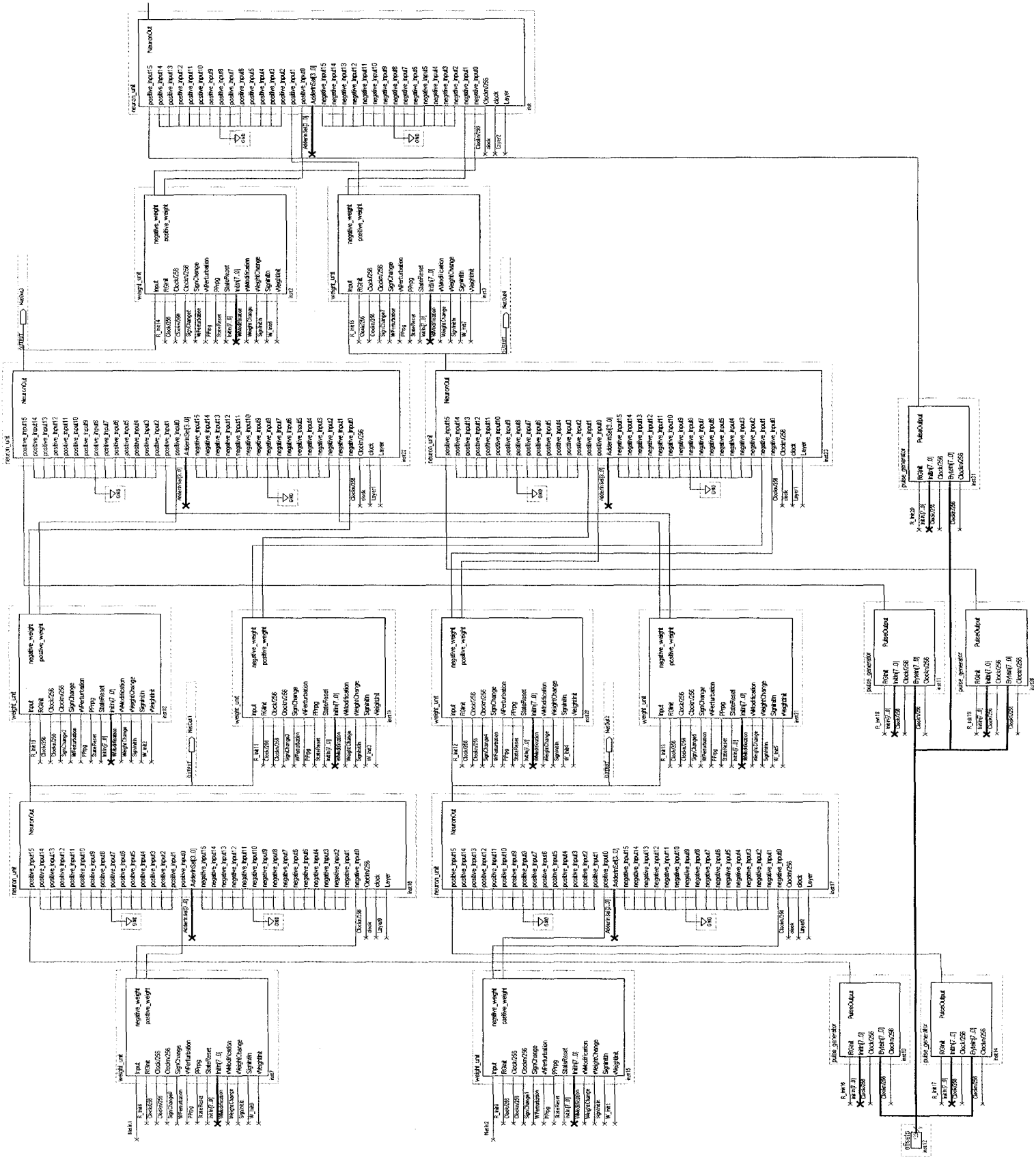


Figure 5.10. The two-input XOR - schematic diagram (part b)

5.2.1. The Simulation Results

The Vector Waveform File contains the following parameters:

- End time of simulations = 10 s
- *reset* after 3 *clock*'s pulses
- The *clock* signal frequency f = 10MHz, which means T = 100 ns
- One state = 255 * (256 *clock* pulses) = 255 * (100 ns x 256) = 255 * 25600 ns = 6.52 ms
- *9binaryconst[8..0]* is set to 100110101
- *8binaryconst[7..0]* is set to 01101011
- *MinError* is set to 000000001
- *Offset* = 128

The simulation has been performed on the ALTERA SRATIX II family EP2S15F672I4 FPGA device. The simulation lasted 10 hours and 46 minutes. The implementation of this unit uses 490 registers. The simulated waveforms are shown in the figures: 5.11, 5.12, and 5.13.

Figure 5.11 shows the waveforms of the first 45 epochs of the simulation of the XOR function.

Figure 5.12 shows how the negative and positive weights of the first neuron of the implementation are updated after every pattern. The same figure shows in detail how the following signals are changed: the weight modification (WModification), the outputs of the neurons (Output1, Output2, Output3, Output4), the output of the network (NetOut), and the sign change (SignChange), which is needed in the calculation of the weight and error modification.

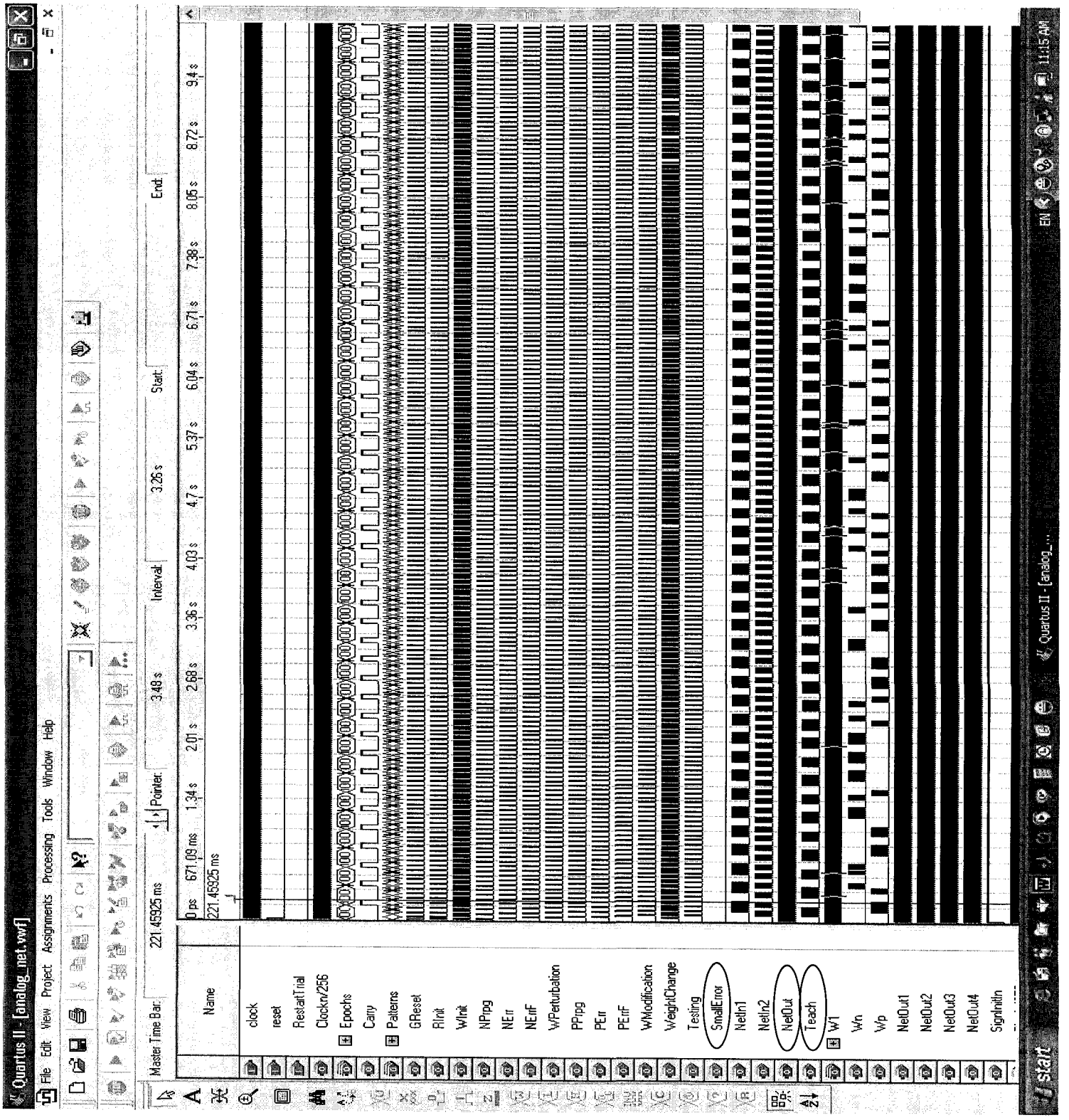


Figure 5.11 The two-input XOR – wave diagram

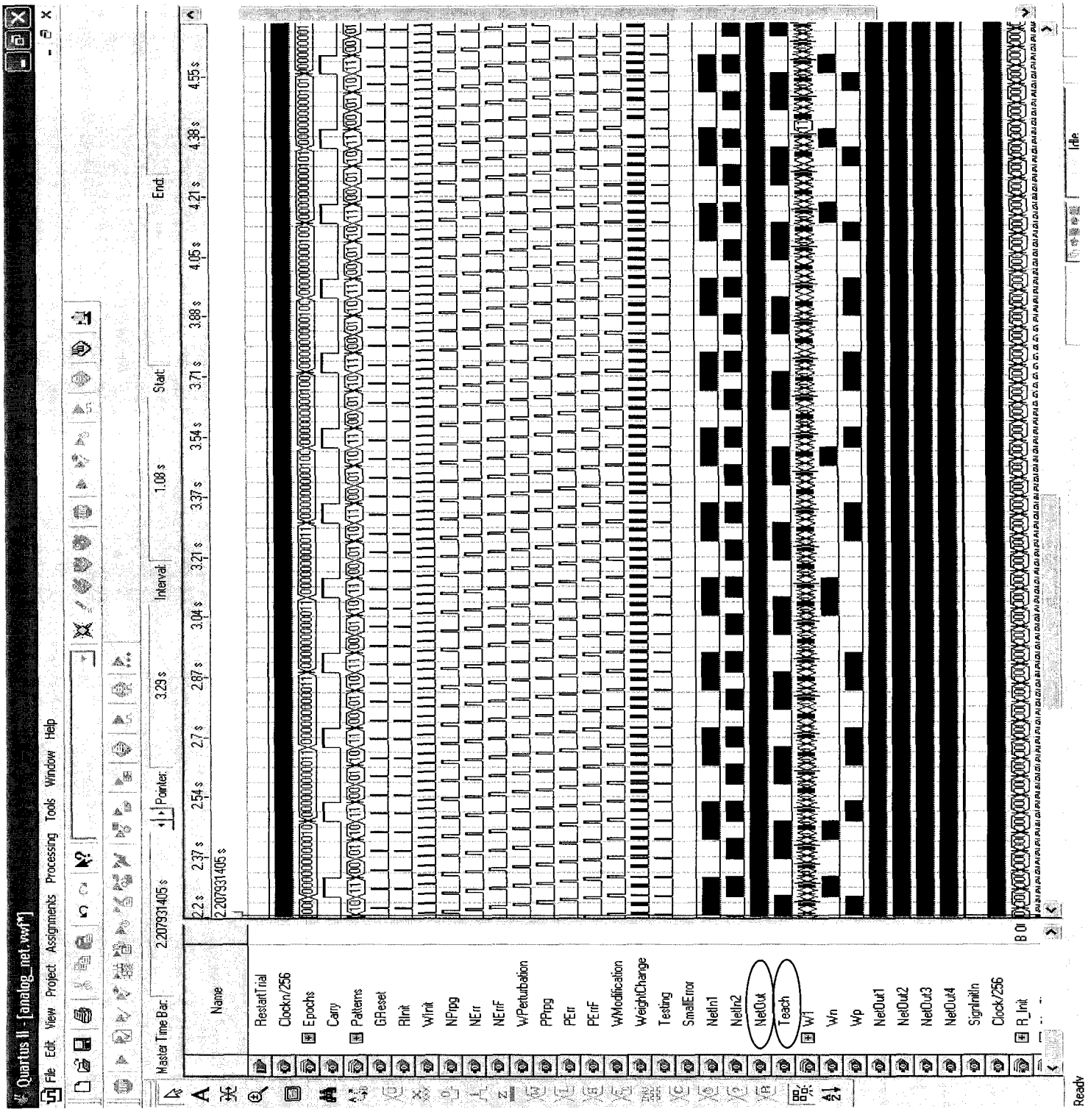


Figure 5.12 The two-input XOR – wave diagram (detail)

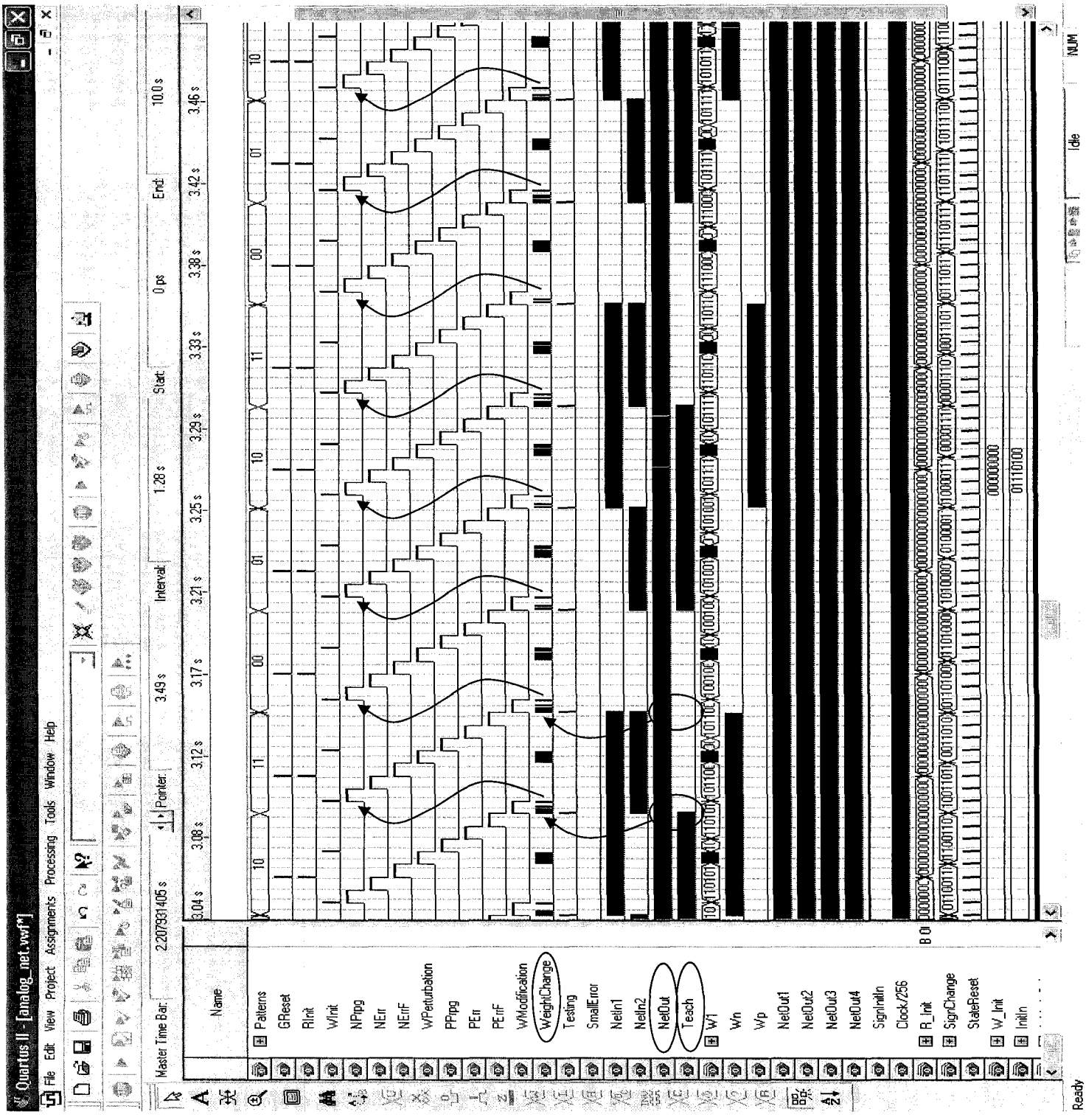


Figure 5.13 The two-input XOR – wave diagram (detail)

Chapter 6

Conclusions

6.1. Summary of the Research

The main contribution of this thesis is the study and development of a gate-level prototyping environment for modular hardware ANN architectures using an internal stochastic pulse data representation. We have used QuartusII 5.1sp1 web edition integrated development environment to design, compile and simulate the developed neuron circuits and test neural network examples before their FPGA implementation.

The starting point of this work was the understanding of the design and functionality of different types of artificial neural network hardware implementations.

We chose to work with the QuartusII software version 51sp1 from Altera installed on a PC that has a 1.8 GHz microprocessor, 512 MB RAM and is running Windows XP operating system. MegaWizard Plug-In Manager megafunctions from the QuartusII software version 51sp1 were used to design every unit of the implementation.

A two-input neuron with learning ability was designed from scratch. The learning method implemented is the on-chip simultaneous perturbation learning rule. The inputs, weights, and output are described by pulse density signals. The basic computing operations

needed in this implementation are based on stochastic logic. The implementation is made on an Altera Field-Programmable Gate Arrays (FPGA).

Basically, the artificial neuron consists of one or more Weight Units and a Neuron Unit. Also the neuron hardware implementation has a Learning Unit and a Command and Control Unit, which are needed for its proper functioning. Hence, an ANN implementation can have numerous weight and neuron units but only one learning unit and one command and control block. The weight unit calculates the product between the input signals and the weight values. The neuron unit calculates the thresholding stochastic weight sum of inputs. The learning unit calculates the quantity of modification for all weights and sends it to all weight units. This quantity has the same value for all the weights. The entire behaviour of the implementation is given by a state machine with 12 states. In order to obtain estimators of the first differential coefficients of the error function with respect to all weights in the network, the simultaneous perturbation learning rule requires only two forward operations through the ANN: first without perturbing the weights and second with perturbing weights. The code for the *state_machine* unit was written in Hardware Description Language (HDL) – version IEEE standard 1164 – within a CASE statement in the QuartusII software.

A great deal of work has been done in order to obtain accurate signals of each unit of the implementation. Each unit of the implementation has been simulated. The simulation has been made on both the ALTERA FLEX 10K and STRATIX families FPGA devices.

This implementation is suitable to an ANN that has 3 layers and only one neuron in the output layer. The limit of weight units are given by the maximum number of bits of the initialization registers.

Finally, two applications using the artificial neuron implementation were designed: an analog function and a digital function. The simulation has been performed on the ALTERA STRATIX II family FPGA device.

6.2. Future Work

Further developments could include the following:

- Increase the number of outputs of the ANN in order to cope with the main limit of this implementation: a single output. This could imply the adding of a new state to the state machine unit and designing a circuit to add the error of each neuron.
- Increase the maximum number of the layers, inputs, weights and neurons available to the implementation by increasing the number of the registers for initialization.
- The transmission speed can also be increased by choosing a newly available Altera FPGA.
- Increase the representation of the signals from one-bit representation to two-bit representation in order to obtain much more precise results.

References

[1] J.J. Hopfield, "Artificial neural networks", *Circuits and Devices Magazine, IEEE*, Vol. 4, No. 5, pp. 3-10, September 1988.

[2] D.E. Rumelhart, J.L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol.1, Cambridge, MA, pp. 10-20, 1986.

[3] M. T. Hangan, H. B. Demuth, M. Beale, *Neural Network Design*, Boston, MA, Thomson Publishing Inc., 1996.

[4] Q. J. Zhang, K. C. Gupta, V. K. Devabhaktuni, "Artificial Neural Networks for RF and Microwave Design: From Theory to Practice", *IEEE Transactions on Microwave Theory and Techniques*, Vol. 51, No. 4, pp. 1339-1350, April 2003.

[5] M. Caudill, C. Butler, *Naturally intelligent systems*, Cambridge, Massachusetts, London: The MIT Press, 1990.

[6] W. Maass, C. M. Bishop, *Pulsed Neural Networks*, Cambridge, Mass.MIT. Press, pp. 90-348, 1999.

[7] Y. Maeda, T. Tada, "FPGA Implementation of a Pulse neural network with Learning Ability Using Simultaneous Perturbation", *IEEE Transactions on Neural Networks*, Vol. 14, No. 3, pp. 688-695, May 2003.

[8] N. Nedjah, L. Mourelle, "FPGA-Based Hardware architecture for Neural Networks: Binary Radix vs. Stochastic", *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design SBCCI, 2003.*

[9] N. Nedjah, L. Mourelle, "Stochastic Reconfigurable Hardware for Neural Networks", *Proceedings of the Euromicro Symposium on Digital Systems Design, 2003.*

[10] S. L. Bade, B. L, Hutchings, "FPGA Based Stochastic Neural Networks Implementation", *Proceedings of the IEEE Workshop of FPGA for Custom Computing Machines*, pp. 189-198, 1994.

[11] E.A. Vittoz, "Analog VLSI implementation of neural networks", *Handbook of Neural Computation*, Eds. E. Fiesler, R. Beale, IOP Publishing Ltd and Oxford University Press, 1997, E.1.3.

[12] V. Beiu, "Digital integrated circuit implementations", *Handbook of Neural Computation*, Eds. E. Fiesler, R. Beale, IOP Publishing Ltd and Oxford University Press, 1997, E.1.4.

[13] T. S. Axelrod, "Neural Networks Hardware Implementations", *Handbook of Neural Computation*, Eds. E. Fiesler, R. Beale, IOP Publishing Ltd and Oxford University Press, 1997, E.1.1:1.

[14] I. Saxena, P.G.Horan, "Optical Implementations", *Handbook of Neural Computation*, Eds. E. Fiesler, R. Beale, IOP Publishing Ltd and Oxford University Press, 1997, E.1.5:1-E1.5:4.

[15] L. Zhao, *Random Pulse Artificial Neural Network Architecture*, M.A.Sc. Thesis, University of Ottawa, School of Information Technology and Engineering, 1998.

[16] A. F. Murray, D. D. Corso, L. Tarassenko, "Pulse Stream VLSI Neural Networks Mixing Analog and Digital Techniques" *IEEE Transactions on Neural Networks*, Vol. 2, No. 2, pp. 193-204, March 1991.

[17] E. M. Petriu, L. Zhao, R. D. Sunil, V. Z. Groza, A. Cornell, "Instrumentation Applications of Multibit Random Data Representation", *IEEE Transactions on Instrumentation and Measurement*, Vol. 52, No. 1, pp. 175-181, 2003.

[18] P.D. Moerland, E. Fiesler, "Neural network adaptations to hardware implementations", *Handbook of Neural Computation*, Eds. E. Fiesler, R. Beale, IOP Publishing Ltd and Oxford University Press, 1997, E.1.2.

[19] Y. Maeda, A. Nakazawa, Y. Kanata, "Hardware Implementation of a Pulse density Neural Network using Simultaneous Perturbation Learning Rule", *Analog Integrated Circuits and Signal Processing*, Vol. 18, No. 2, pp. 1-10, 1999.

[20] B. Widrow, M. A. Lehr, "30 years of adaptive neural networks: perceptron, Madaline, and backpropagation", *Proceedings of the IEEE*, Vol. 78, No. 9, pp. 1415-1442, September 1990.

[21] M. Jabri, B. Flower, "Weight Perturbation: An Optimal Architecture and Learning Technique for Analog VLSI Feedfordward and Recurrent Multilayer Networks", *IEEE Transactions on Neural Networks*, Vol. 3, No. 1, pp. 154-157, 1992.

[22] P. D. Moerland, E. Fiesler, "Hardware Friendly Learning Algorithms for Neural Networks: an overview", *Proceedings of Fifth International Conference on Microelectronics for neural networks*, pp. 117-124, February 1996.

[23] H. Hikawa, "Frequency-Based Multilayer Neural Network with On Chip Learning and Enhanced Neuron Characteristics", *IEEE Transactions on Neural Networks*, Vol. 10, No. 3, pp. 545-553, May 1999.

[24] S.T. Ribeiro, "Random-pulse machines", *IEEE Transactions on Electronic Computers*, vol. EC-16, No. 3, pp. 261-276, 1967.

[25] B. Flower, M. Jabri, "Summed Weight Neuron Perturbation: An *O(N)* Improvement over Weight Perturbation" *Advances in Neural Information Processing Systems (NIPS92)*, Vol. 5, pp. 212-219, Morgan Kaufmann, San Mateo CA, 1993.

[26] J. Alspector, R. Meir, B. Yuhas, A. Jayakumar, "A Parallel Gradient Descent Method for Learning in Analog VLSI Neural Networks", *Advances in Neural Information Processing Systems (NIPS92)*, Vol. 5, pp. 836-844, San Mateo CA, 1993.

[27] S.L. Toral, J M. Quero, L. G. Franquelo, "Stochastic Pulse Coded Arithmetic", *IEEE International Symposium on Circuits and Systems, ISCAS 2000,* May 28-31, Geneva, Switzerland, 2000.

[28] M. Martincigh, A. Abramo, "A New Architecture for Digital Stochastic Pulse-Mode Neurons Based on the Voting Circuit", *IEEE Transactions On Neural Networks,* Vol. 16, No. 6, November, 2005.

[29] Y. Kondo, Y. Sawada, "Functional Abilities of Stochastic Logic Neural Networks", *IEEE Transactions on Neural Networks,* Vol. 3, No. 3, May, 1992.

[30] Y. Maeda. T. Tada, "FPGA Implementation of a Pulse Density Neural Network Using Simultaneous Perturbation", *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks,* IJCNN 2000, Vol. 3, pp. 296-301, July 2000.

[31] H. Hikawa, "Pulse Mode Multilayer Neural Network with Floating Point Operation and On Chip Learning", *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks,* IJCNN 2000, Vol. 2, pp. 24-27, July 2000.

[32] H. Hikawa, "Digital Pulse Mode Neural Network with Simple Synapse Multiplier", *The 2001 IEEE International Symposium on Circuits and Systems,* ISCAS 2001, Vol. 3, pp. 569-572, May 2001.

[33] H. Hikawa, "A New Digital Pulse- Mode Neuron with Adjustable Activation Function", *IEEE Transactions on Neural Networks,* Vol. 14, No. 1, January 2003.

[34] H. Hikawa, "A Digital Hardware Pulse-Mode Neuron With Piecewise Linear Activation Function", *IEEE Transactions on Neural Networks,* Vol. 14, No. 5, September, 2003.

[35] H. Hikawa, "Hardware Pulse Mode Neural Network with Piecewise Linear Activation Function Neurons", *The 2002 IEEE International Symposium on Circuits and Systems,* ISCAS 2002, Vol. 2, pp. II-524 – II-527, May 2002.

[36] H. Hikawa, "A Multilayer Neural Network with Pulse Position Modulation", *Systems and Computers in Japan,* Vol. 34, No.13, 2003.

[37] Y. Maeda, H. Hirano, Y. Kanata, "A Learning Rule of Neural Networks via Simultaneous Perturbation and its Hardware Implementation", *Neural Networks,* Pergamon, Vol. 8, No. 2, pp. 251-259, 1995.

[38] E.M. Petriu, K. Watanabe, T. Yeap, "Applications of Random-Pulse Machine Concept to Neural Network Design," *IEEE Transactions on Instrumentation and Measurement,* Vol. 45, No. 2, pp.665-669, 1996.

[39] J. O. Hamble, M. D. Furman, *Rapid Prototyping of Digital Systems,* Kluwer Academic Publishers, Boston/Dordrecht/London, 2001, pp. 235-236.

[40] R. K. Dueck, *Digital Design with CPLD Applications and VHDL,* 2nd edition, Thomson Delmar Learning, 2005.

[41] ***, QUARTUS Help: Altera Corporations., Available: https://www.altera.com/support/software/download/altera_design/quartus_we/dnl-quartus_we.jsp.

Appendix A

The VHDL Code of the Neuron State Machine

```
LIBRARY ieee;
USE ieee.std_logic_1164.All;

ENTITY neuron_states IS
      PORT  (
            StateClockn              : IN  STD_LOGIC;
            Reset                    : IN  STD_LOGIC;
            RestartTrial, TrialEnd   : IN  STD_LOGIC;
            q                        : OUT STD_LOGIC_Vector(3 downto
0));
END neuron_states;

ARCHITECTURE a OF neuron_states IS

      TYPE STATE_TYPE IS (s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10,
s11);
      SIGNAL state: STATE_TYPE;
BEGIN
      PROCESS (StateClockn, Reset)
      BEGIN
            IF Reset = '1' THEN
                  state <= s0;
                  ELSIF StateClockn'EVENT AND StateClockn = '1' THEN
                  CASE state IS
                        WHEN s0 =>
                        IF Reset='1'
                              THEN
                                    state <= s0;
```

```
                        Else
                              state <= s1;
                        END IF;

                  WHEN s1 =>
                              state <= s2;

                  WHEN s2 =>
                              state <= s3;

                  WHEN s3 =>
                              state <= s4;

                  WHEN s4 =>
                              state <= s5;

                  WHEN s5 =>
                              state <= s6;

                  WHEN s6 =>
                              state <= s7;

                  WHEN s7 =>
                              state <= s8;

                  WHEN s8 =>
                              state <= s9;

                  WHEN s9 =>
                  IF TrialEnd='1'
                        THEN
                              state <= s11;
                        Else
                              state<=s10;
                        END IF;

                  WHEN s10 =>
                              state <= s3;


                  WHEN s11 =>
                  IF RestartTrial='1'
                        THEN
                              state <= s2;
                        Else
                              state<=s11;
                        END IF;

            END CASE;
      END IF;
END PROCESS;

WITH state SELECT
      q     <=    "0000"      WHEN  s0,
                  "0001"      WHEN  s1,
                  "0010"      WHEN  s2,
```

```
                    "0011"      WHEN  s3,
                    "0100"      WHEN  s4,
                    "0101"      WHEN  s5,
                    "0110"      WHEN  s6,
                    "0111"      WHEN  s7,
                    "1000"      WHEN  s8,
                    "1001"      WHEN  s9,
                    "1010"      WHEN  s10,
                    "1011"      WHEN  s11;
END a;
```

Appendix B

The Two-Input Neuron Schematic Diagram

Figures B.1 and B.2 contain the schematic diagram of an artificial neuron implementation with two inputs and two weights.

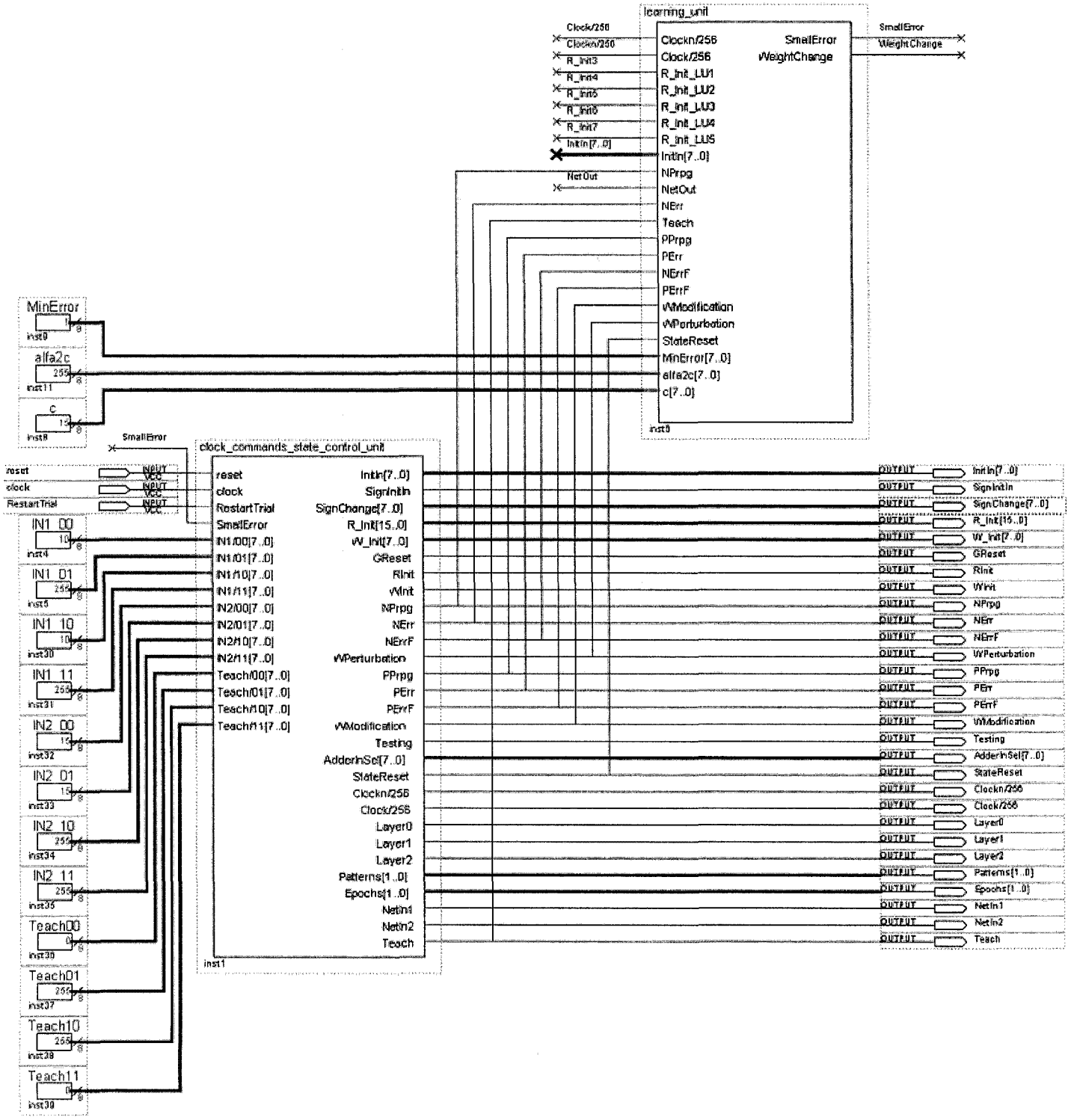


Figure B.1. The two-input neuron - schematic diagram (part a)

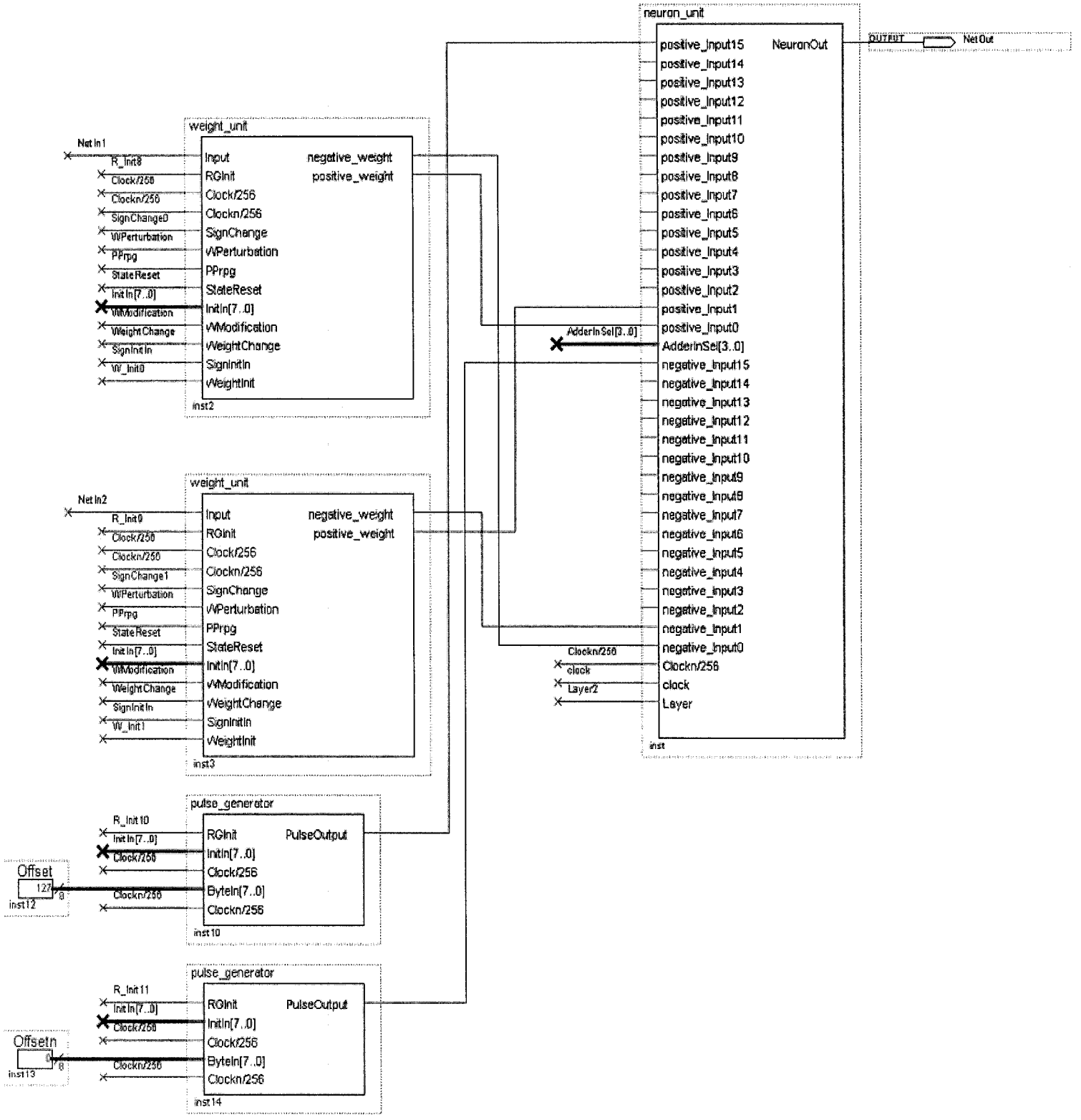


Figure B.2. The two-input neuron - schematic diagram (part b)

Appendix C

The Blocks of the Neuron Implementation

Figures C.1 contains each block with their signals used in the artificial neuron implementation.

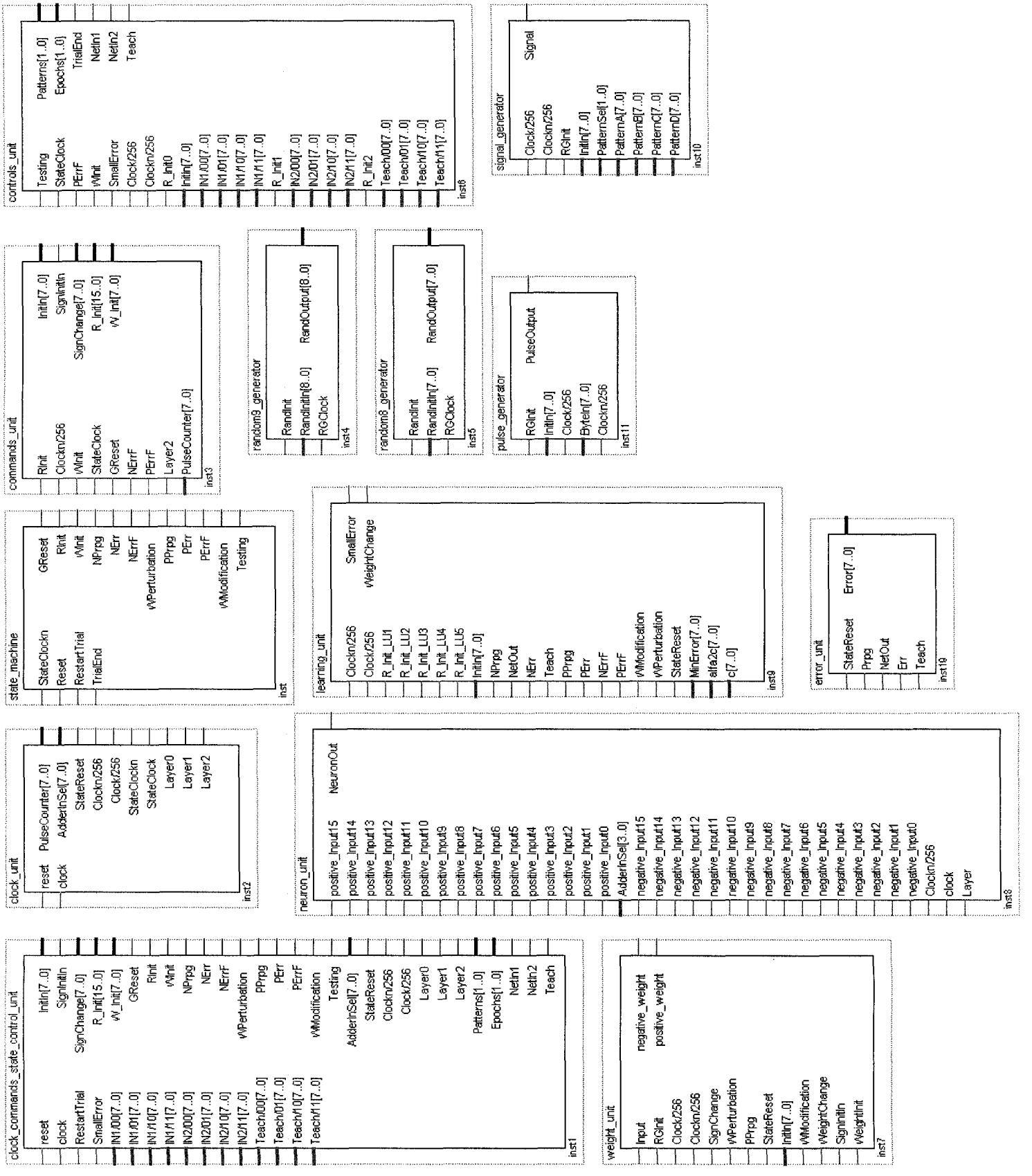


Figure C.1.The constituent blocks of the artificial neuron implementation