The Simulation of Systolic Array Implementation Schemes for Hopfield Neural Nets

Jacek Mazurkiewicz[1]
Institute of Engineering Cybernetics
Technical University of Wroc³aw
Poland

***Abstract*- The paper describes the simulation of systolic array schemes for Hopield nets. The implementation presented is based on completely digital circuits. Input data is passed through the neurones in a time shared basis, weights are stored in digital shift registers and no separate threshold detectors are used. The simulation was realised using EASE/VHDL ver. 2.2 and V-System/Windows ver. 4.**

***Resumo*- Este artigo descreve a simulação de uma arquitectura sistólica para redes neuronais de Hopfield. A implementação é puramente digital. As entradas são passadas sequencialmente através dos neurónios, enquanto os pesos estão armazenadas em registos de deslocamento e não são utilizados detectores de limiar. A simulação foi realizada com EASE/VHDL versão 2.2 e V-System/Windows versão 4.**

I. SYSTOLIC ARRAY IMPLEMENTATION OF HOPFIELD NEURAL NETWORKS

The binary Hopfield net has a single layer of processing elements, which are fully interconnected - each neurone is connected to every other unit. Each interconnection has an associated weight. We let T_{ji} denote the weight to unit *j* from unit *i*. In Hopfield network, the weights T_{ij} and T_{ji} have the same value. Mathematical analysis has shown that when this equality is true, the network is able to converge. The inputs are assumed to take only two values: 1 and -1. The network has *N* nodes containing hard-limiting nonlinearities. The output of node *i* is fed back to node *j* via connection weight T_{ij} .

A. Architecture for training

The training is realised in accordance with the Hebbian learning algorithm. The training patterns are presented one by one in a fixed time interval. During this interval, each input datum is communicated to its neighbour *N* times.

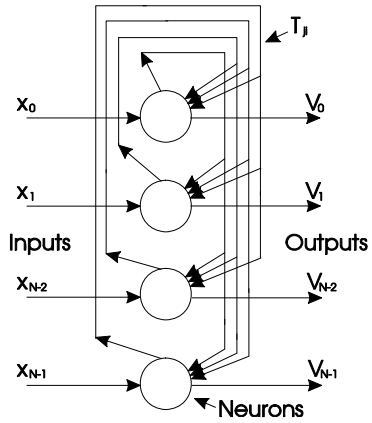

Fig. 1. Hopfield Neural Network

For the systolic implementation of the training algorithm using digital circuits, the input data are assumed to be binary values: 1 and 0 instead of bipolar 1 and -1. Table 1 shows the required changes of weights for the bipolar and binary inputs respectively.

Table 1.

x_i	x_j	ΔT_{ji}
-1	-1	+1
-1	+1	-1
+1	-1	-1
+1	+1	+1

Update of weights for bipolar inputs

x_i	x_j	$\overline{x_i \oplus x_j}$	ΔT_{ji}
0	0	1	+1
0	1	0	-1
1	0	0	-1
1	1	1	+1

Update of weights for binary inputs

[1] Jacek Mazurkiewicz stayed at the Departamento de Electrónica e Telecomunicações da Universidade de Aveiro from 14 June to 13 July 1995 within TEMPUS Project S_JEP 07648-94. The work presented was realised during this visit.

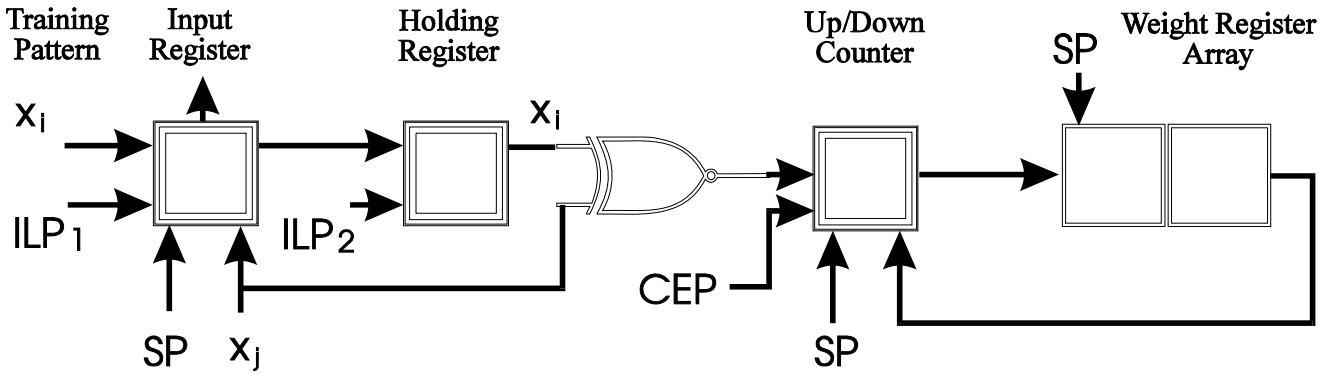

Fig. 2. The implementation of the single neurone training algorithm

The update of weights depends on the values of x_i and x_j as shown below:

$$T_{ji}(k) = \begin{cases} T_{ji}(k-1)+1 & if \quad \overline{x_i \oplus x_j} = 1 \\ T_{ji}(k-1)-1 & if \quad \overline{x_i \oplus x_j} = 0 \end{cases} \qquad (1)$$

where:
$i = (j+k) \bmod N \quad for \quad 0 \le j \le N-1 \quad and \quad 1 \le k \le M$
$T_{ji} = T_{ji}(M)$ where M is the number of patterns to be stored.
The weights are initialised as:

$$T_{ji}(0) = 0 \quad for \quad all \quad j \quad and \quad i.$$

Each weight is modified M times. Fig. 2 shows the implementation of single neurone.

The activation values x_i are moving on the main ring whereas the weights T_{ji} are rotated in the array of shift registers through the up/down counter. At the up/down counter, the weight is incremented or decremented by 1 and transferred to the register at the top of the array. The training patterns are loaded into the input register when ILP_1 is present.

The loaded data is transferred into the holding register with the control signal ILP_2. The weight value in the lowest register of the array is loaded into the up/down counter using control pulse, *SP*.
The count enable pulse (*CEP)* makes the counter count up or down depending on the value of $\overline{x_i \oplus x_j}$. The modified weight is pushed downwards into the top register of the synaptic weight array using *SP*. At the same time *SP* is applied for shifting the input data clockwise once.

This cycle is repeated N times and all the N weights are modified. The next training pattern is applied and weights are again changed accordingly.
This procedure is repeated M times - M is the number of patterns - and training of the network is completed.

B. Architecture for computation.

The computation of Hopfield neural network considers the following data:

- N^2 synaptic weights T_{ij} which contain the relationship between the neurones. These weights are obtained after training the network;
- the binary input vector X_i , *i=0,1,...,N-1* describes the initial activation values of the neurone;
- the partial sums which represent the system evolution and become, after the threshold function, the new activation values of the neurones.

The input register is initialised with the input vector and each weight register is initialised with the synaptic weights produced during the learning phase.

Then the products between each component of the input vector and the corresponding synaptic weights are computed. The result is transferred to the second stage where each cell computes a sum of these products.

Following this step, a shift command is sent to the synaptic registers and to the input register. A new product is evaluated again. After N steps, the second stage contains values of the new output for the single node. The result is delivered to the evaluation part which applies a threshold function.

$$Net_j(k) = Net_j(k-1) + T_{ji}x_j \quad where \ i = (j+k) \bmod N \qquad (2)$$

$$Net_j = Net_j(N) \quad and \ initial \ value: \quad Net_j(0) = 0 \qquad (3)$$

The answer of each neurone is described by the following equation:

$$V_j = \begin{cases} 1 & if \quad Net_j \ge 0 \\ 0 & if \quad Net_j < 0 \end{cases} \qquad (4)$$

$$for \quad 0 \le i, j \le N-1 \quad and \quad 1 \le k \le N$$

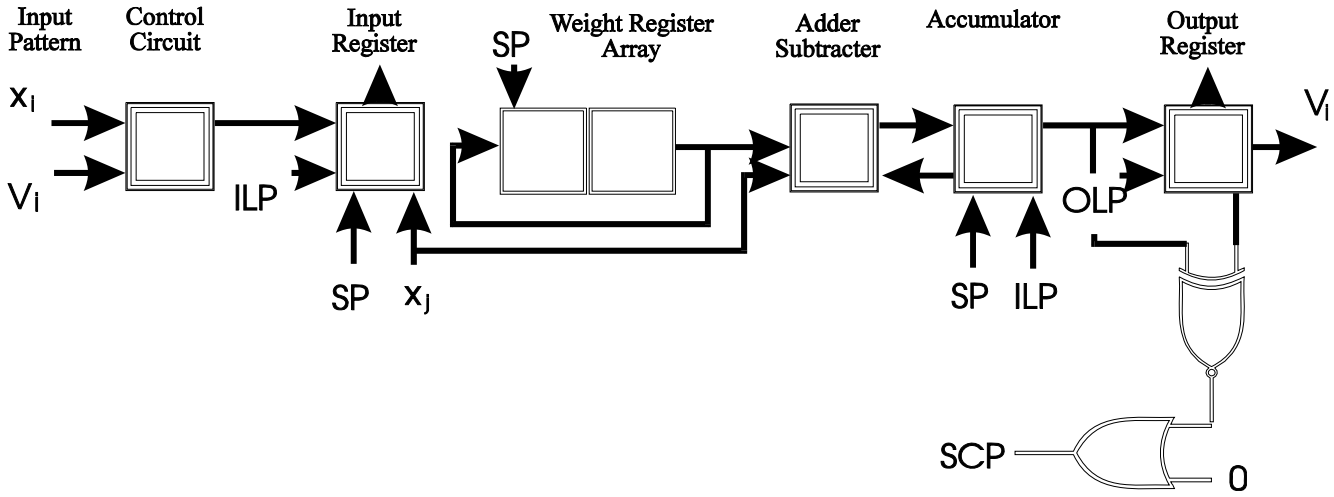

Fig. 3. The implementation of the single neurone for computation algorithm

The architecture for computation consists of a projection of the T_{ij} components in *N* different shift registers for each neurone.

The weight applied to the *j*-neurone along with the input element x_i in the *k* recursive step is T_{ji}. The input vector is loaded into the input register when the input load pulse *ILP* is present.

The contents of the input register and the weight register array are shifted once by the shift pulse *SP*, after one partial computation is over.

The computational element consists of an adder/subtracter which is controlled by the input x_i and an accumulator which is used for storing the partial sum *PS*.

This computational element receives the synaptic weights one by one and, at each time, applies the value of the corresponding input and accumulates the result.

There is no need of multiplier since the input x_i is binary. The content of the lowest register in the weight register array is added or subtracted with the content of the *PS* register - accumulator - depending on the value of x_i .

The partial sum *PS* in the accumulator, which is cleared at the beginning of the computation with *ILP*, is obtained as follows:

$$PS_j(k) = \begin{cases} PS_j(k-1) + T_{ji} & if \quad x_i = 1 \\ PS_j(k-1) - T_{ji} & if \quad x_i = 0 \end{cases} \qquad (5)$$

where:

$$i = (j+k) \bmod N, \quad PS_j = PS_j(N), \quad PS_j(0) = 0$$

Then the shift pulse *SP* is applied to the input register and to the ring array of synaptic weight registers.

The same *SP* controls the accumulator to receive the partial sum from the adder/subtracter. After *N-1* such cycles, one computation is over and the result - Net_j - will be available in the accumulator.

The *N*-stage change of the accumulator is ignored, since it is due to the synaptic weight w_{ii} . The hard limiter thresholding has to be done with the Net_j output.

The accumulator content is a 2's complemented binary number whose most significant bit is the sign bit. If the sign bit is inverted using a binary inverter, we get the required V_j output.

After *N-1* partial computations the sign bit of the accumulator is loaded into the output register.

This is controlled by the output load pulse *OLP*, which is appeared after *N-1* shift operations by *SP*. V_j obtained after one computation is fed back to the input register and computations are repeated until convergence.

The convergence is reached when *SCP* - stop computation pulse - becomes 0.

A computation phase consists of the following steps:

- each input x_i is cycling from one neurone to its neighbour;
- each x_i going through a processing element is multiplied with T_{ji} and the result is involved to the local partial sum;
- the threshold function is applied;
- this procedure is repeated until convergence.

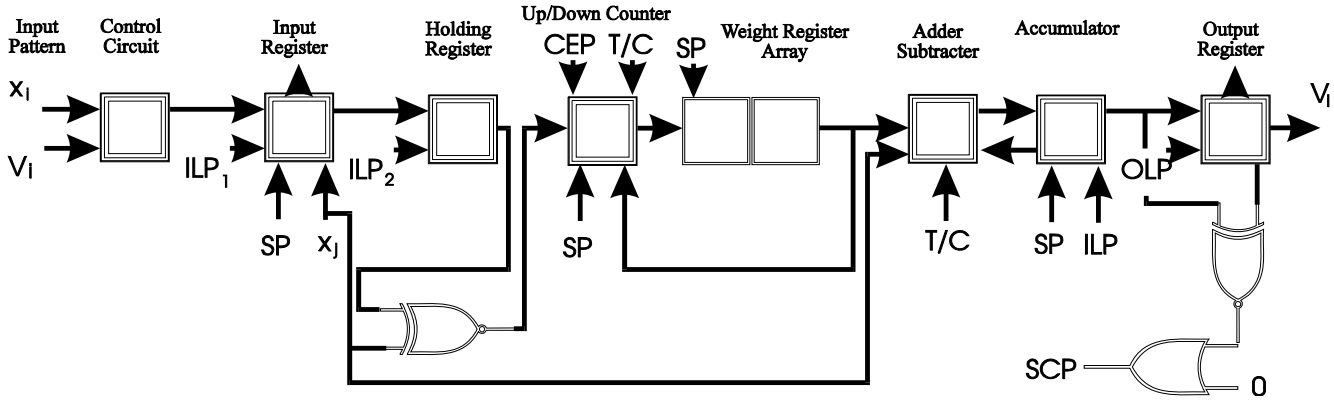

Fig. 4. The implementation both for training and computation.

II. GENERALISED SYSTEM BOTH FOR TRAINING AND COMPUTATION

The previous chapter described two architectures for implementation the training and computation part of the Hopfield neural net behaviour.

The construction of physical system based on these foundations requires a solution to combine two parts of the Hopfield algorithm. It is necessary to add one control signal *T/C* to drive the right phase. When *T/C* is 1, the network receives the exemplar patterns for training and functions as explained earlier.

When *T/C* is 0, the network receives the error pattern and computes the output till it converges to a stable state. While the network is doing the recognition process, the data in the weight register array bypass the up/down counter.

During training weights go through the up/down counter and update their values according to the equations shown before.

The previous chapter described how the single neurone is implemented and how it works. The whole Hopfield net is constructed as fully interconnected slabs like this one. Each slab is connected to another one using only two signals. The first signal connects input registers to satisfy the exclusive-or function to drive up/down counter during training part of the algorithm and to deliver the second datum for adder/subtracter when the net realises computation. The second signal generated during computation with the output of the single slab is necessary to produce *SCP* (Stop Computation Pulse). It seems that the structure can be in very easy way extended to any number of neurones, but it isn't true. We can notice at the Fig. 2. - related to training part - and at the Fig. 3. - related to computation part - a lot of different signals which are necessary for normal work.

These signals have to be delivered to each slab. First of all - this is a systolic structure - so each slab has to be supported by *SP* (Shift Pulse) line. This line drives weight register array, input register and up/down counter in each cell. There are two signals to shift the component of the input pattern between **Input Register** and **Holding Register**: ILP_1 and ILP_2. The **Output Register** is controlled by the **Output Load Pulse** *OLP*, which is appeared after *N-1* shift operations by *SP*. Finally it is necessary to pick *CEP* line to choose the direction of counting at the up/down counter and *T/C* line to distinguish training part and computation.

III. PROJECT

The project of the architecture presented above was realised using EASE/VHDL ver. 2.2. system. The conception of the simple neurone for training procedure and the simple neurone for ordinary work was translated into VHDL using this system. The results achieved were the inputs for the simulation package.

A. Introduction to the system

The system allows to create the hierarchical structure of the project. The top of this tree is the whole architecture which is treated as a black-box supported by all necessary inputs and outputs. The VHDL code is generated always for this level of implementation. Then the user starts his elaboration by creating the lower branches of whole structure. During each downstairs step system creates the proper inputs and outputs for the actual point of elaboration. The user can define the simple object of each level, except the top, as: a state machine, object which is defined by VHDL procedure written by the user or as a complex part, which is the root for the lower branch. So as the base leaves of the tree we can notice only two kinds of elements.

The creation of each part of the structure is very similar to the composition of picture using CorelDraw for example or to elaboration of schematic file using OrCad. Very sophisticated and easy available by buttons menu makes every change or edition of each object as easy as possible.

The user during composition of new state machine defines all necessary states first. Then he creates the

transitions among the states and combines the activation conditions to them. The conditions can be declared as clock dependent or asynchronous to drive different kinds of reset signals to states. The next step allows to write actions for each state to drive different output signals. Writing these conditions for transitions and actions requires the VHDL syntax among input and output signals. Of course it's necessary to provide clock signal for state machine, which drives the moments of state exchange.

If the user combines the simple object with the VHDL procedure system creates the skeleton of the procedure. It provides a header, footer and the interface of the procedure - the definitions of output and input signals and fix the way of calling the procedure. The user ought to create manually using his favourite text editor the body of the procedure. At the end system creates the special file for the ready-made procedure and links the file with the proper part of project.

When the whole project is ready system checks if the elaborated structure if correct: if all inputs and outputs are driven and if all branches are created. If the results show no errors system generates VHDL code related to the project.

Unfortunately these procedures which are written by the user aren't checked. Also there is no warning if conditions of transitions and actions for state machine include syntax error. It means that the system is not created as a tutorial to VHDL. The user ought to know at least the basis of this language.

The system also doesn't contain the already libraries for the most popular and the most indispensable simple objects used for creation the complex structures. The user can define and store his own libraries.

B. Implementation of single neurone for training

The top level of implementation (Fig. 5.) is a definition of "black-box" supplied with all necessary input and output signals, which takes part of the structure shown at Fig.2.

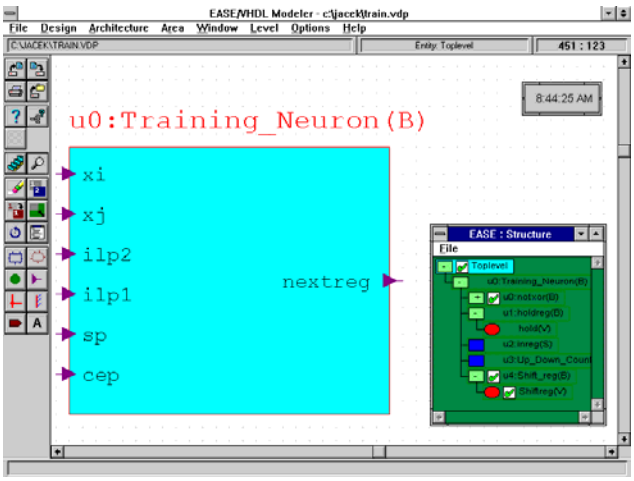

Fig. 5. Neurone for training procedure - top level

The lower level illustrates how the simple parts of whole structure were mirrored into VHDL blocks.

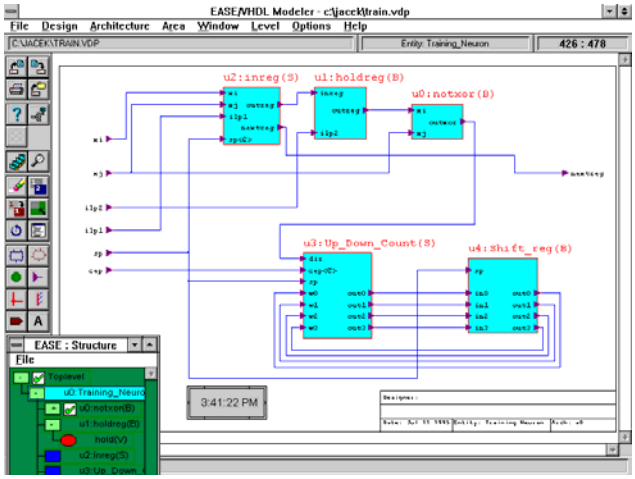

Fig. 6. Neurone for training - lower level

The *Input Register* is a state machine which includes three states: *Load, Zero, One*. The Load state starts and restarts the normal work of the register. It allows to preload the simple component of training pattern - x_i when there is a change at the ILP_1 signal into register. In this state the output signal is provided directly by input x_i . The *Zero* and *One* states correspond to normal work of the register. If x_j - input signal - is 0 the structure is switched by the nearest change of clock signal *SP* into *Zero* state and 0 is generated as an output. If x_j equals 1 we can observe the same reaction but *One* state is the destination of switching and output is driven into logic 1.

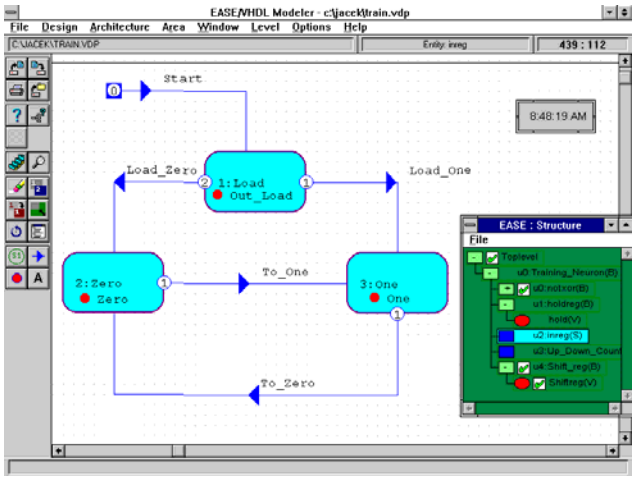

Fig..7. Input Register for training

The *Holding Register* is created as an object with suitable VHDL procedure. When there is a change at the ILP_2 signal it loads itself by the output of *Input Register* and serves this value as its output continually.

The *Negative-Exclusive-Or* Functor is also created as an object described directly by VHDL procedure. It generates the result of *Negative-Exclusive-Or* Function

made at two inputs: x_i and x_j as a signal to drive the direction of counting.

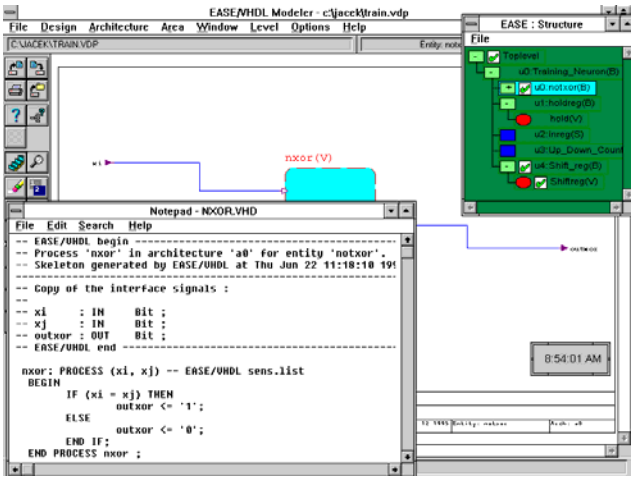

Fig. 8. Negative-Exclusive-Or function for training

The *Up/Down Counter* is a 4-bit binary counter with parallel input enable and the possibility to change the direction of counting. It is implemented as a state machine of 17 states. 16 states are necessary to drive normal work of such counter and the last one is responsible for parallel input. The transitions among the base 16 states depend on the required direction and change the current state to next or previous when the change of logic state at the *SP* - clock signal is noticed. Each state drives the four output lines of counter to transmit the actual counted number: 0..15. The change of states is driven by *CEP* - *Count Enable Pulse*, which ought to be created as clock shape. The *Load* state is created to load the value of weight transmitted by *Weight Register Array* by four lines.

This state based on the loaded value makes the counter start counting from the proper state. This load is synchronised by *SP* - clock signal with the work of *Weight Register*. So the load into counter isn't completely asynchronous, but it doesn't depend on the *CEP* signal. The transition from the *Load* state to the one of the 16 counter states is driven by the general clock signal *SP*: After the single change of state of the counter the updated value of weight is transferred to the top of *Weight Register Array*.

The *Weight Register Array* is created as an object described directly by the VHDL procedure. It loads into the first cell of an array the updated value of weight transmitted from counter by four lines in parallel way when the change at clock signal *SP* is noticed.

Simultaneously it shifts the previous weights to the next cells and drive the output lines by the proper at the moment weight which is the input for the counter described above. The whole array is constructed by four cells, each cell can store four bits of information.

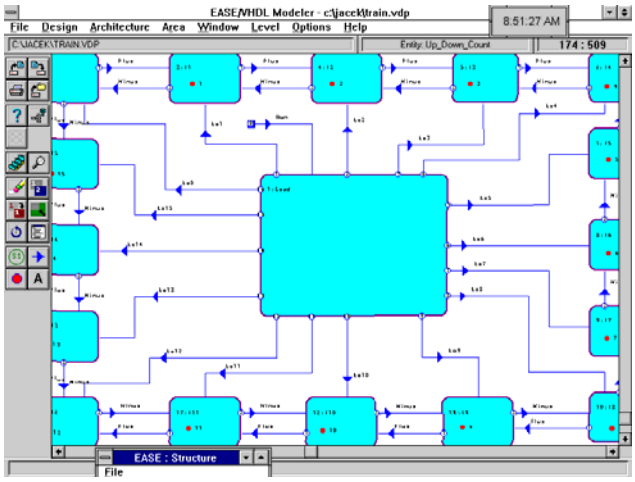

Fig. 9. Up/Down Counter for training procedure

C. Implementation of single neurone for computation

The top level of implementation (Fig. 11.) is a definition of "black-box" supplied with all necessary input and output signals, which takes part of the structure shown at Fig.3.

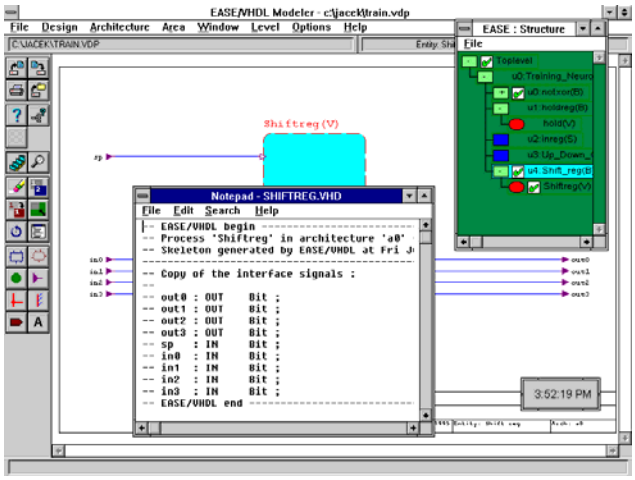

Fig. 10. Weight Register Array

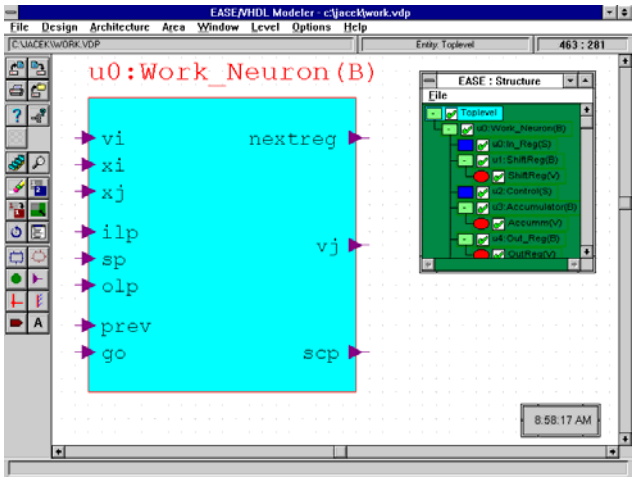

Fig. 11. Neurone for computation algorithm

The lower level illustrates how the simple parts of whole structure were mirrored into VHDL blocks.

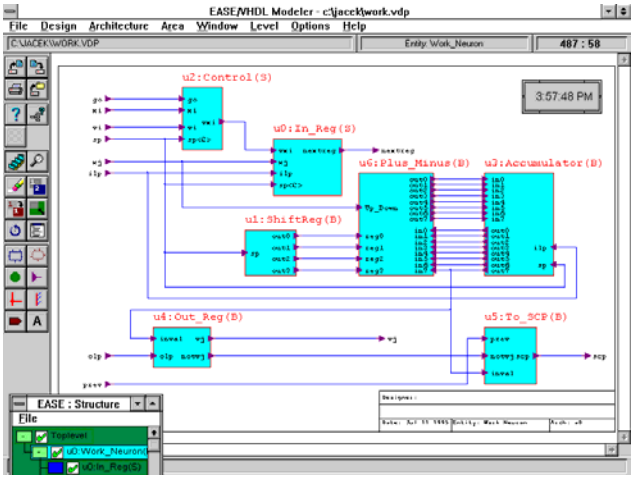

Fig. 12 Neurone for computation - lower level

The *Control Circuit* is a state machine which includes three states: *Load*, *Zero*, *One*. The *Load* state starts and restarts the normal work of the structure. It allows to preload the simple component of input pattern - x_i when there is a change at the *GO* signal into object. In this state the output signal is provided directly by input x_i . The *Zero* and *One* states correspond to normal work of the *Control Circuit*. If v_i - input signal - is 0 the structure is switched by the nearest change of clock signal *SP* into *Zero* state and 0 is generated as an output. If v_i equals 1 we can observe the same reaction but *One* state is the destination of switching and output is driven into logic 1. The v_i input signal is driven by the output generated by this neurone at the previous moment. This feedback is necessary to achieve a convergence.

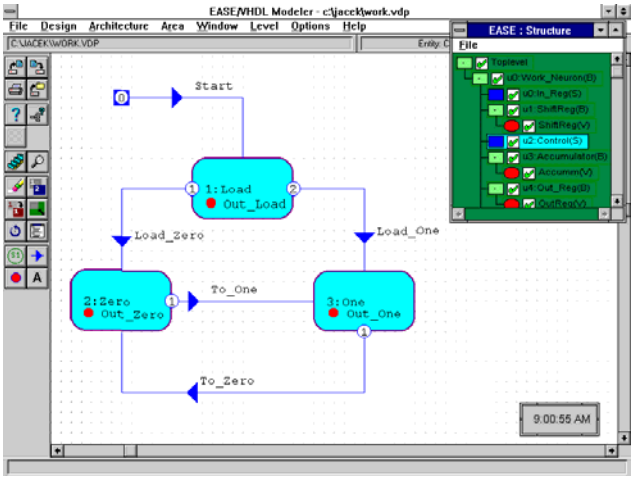

Fig. 13. Control Circuit for computation algorithm

The *Input Register* is implemented exactly in the same way as Input Register for the structure for the training algorithm. The only differences are the names of input and output signals.

The *Weight Register Array* is implemented using the same VHDL syntax as the *Weight Register Array* necessary for training algorithm. This structure works in the same way, but the top cell of array is loaded by the value from the lowest cell, because the task of the *Weight Register Array* is to serve the weights produced during training.

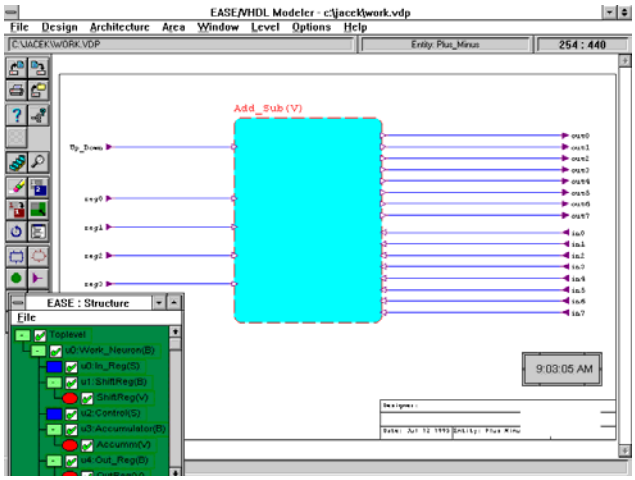

Fig. 14. Adder/Subtracter for computation algorithm

The *Adder/Subtracter* is the 4 bit adder/subtracter but the first argument of each function is stored using 8 bits. The result of each operation also requires 8 bits. This solution allows to expand the range of available values. The *Adder/Subtracter* works using U2 code for data. The device was created as an object directly described by VHDL procedure. The construction of 8-bit adder/subtracter is based on the idea of cascade n-bit adder. The partial sums are calculated using 1-bit adders or half-adders and the carry signals are transmitted to the next module and are used as the next arguments. The 8-bit subtracter compares the bits of arguments at the same position and drives the carry signals, modifies the values at proper positions of arguments and then subratcts them. The device has to work using U2 code - because the final value of net just before of the calculation of neurone's answer can be positive or negative value. In case of this *Adder/Subtracter* follows the most significant and just previous carry signals to check if the result of operation is valid. *SP* - clock signal loads previous partial sum from accumulator and returns the modified value. The value of single component x_j is the switch: add or subtract. At the beginning of the work the accumulator is set to zero by *ILP* signal.

The *Output Register* is implemented also as a device with direct VHDL description. The only task of it is to load the value of MSB bit from the value stored in the accumulator and to serve as an output of the structure after translation by *NOT* functor and without any changes as an argument for block which generates *SCP* signal. The new value to *Output Register* is loaded when the change at the *OLP* line is noticed.

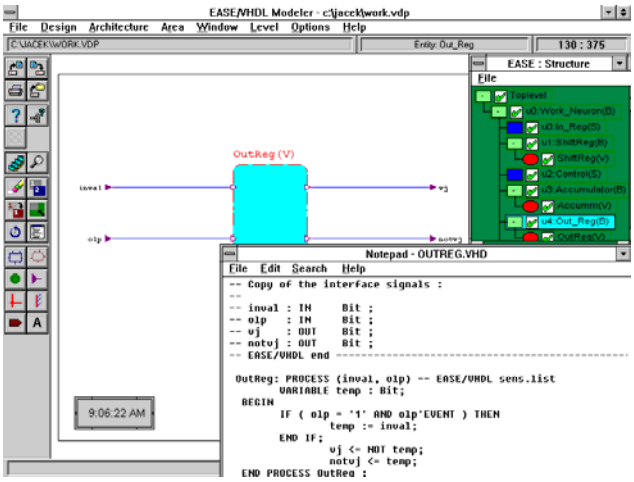

Fig. 15. Output Register for computation algorithm

To_SCP is the object to generate *Stop Computation Pulse* signal. This part which is also described as the VHDL direct procedure contains two functors: *Or* driven by *Exclusive-Or*. The first argument for this circuit is the signal from *Output Register* and the second is *SCP* signal produced by previous neurone.

IV. SIMULATION

The simulation of architecture of simple neurone for training and simple neurone for computation described above was realised using V-System/Windows ver. 4. As the inputs for simulator were used VHDL files generated by EASE/VHDL system. The main goal was to check if these two architectures work in accordance with the foundations.

A. Introduction to the system

The system is an integrated device, which allows to simulate and check the VHDL project. The first programme included is VHDL compiler with the mechanism to find any kind of errors and warnings. The user obtains very detailed information about the kind and position where something is wrong in project.

Then if the compilation has no errors the user can start the simulation. It's possible to simulate the whole structure or simpler parts and single components. The user's decision about the kind of simulation makes the system to open the proper input lines to give a chance to drive them and also to open the proper outputs to check if the results are correct.

The behaviour of the system the user can observe in many different ways. First of all system generates the wave forms of all input, output and interior signals at proper time scale.

The user can follow all changes of signals using special cursors, rescale the wave forms. The same information is presented by textual version as a list of signals and the values related to them.

At the same time system shows the state of variables defined in objects, actual position of simulation in whole structure, the active processes and the current position at VHDL source file.

The user can change his requirements related to the simulation at every moment by every active console. There is no problem to force the inputs at the moment the user chooses.

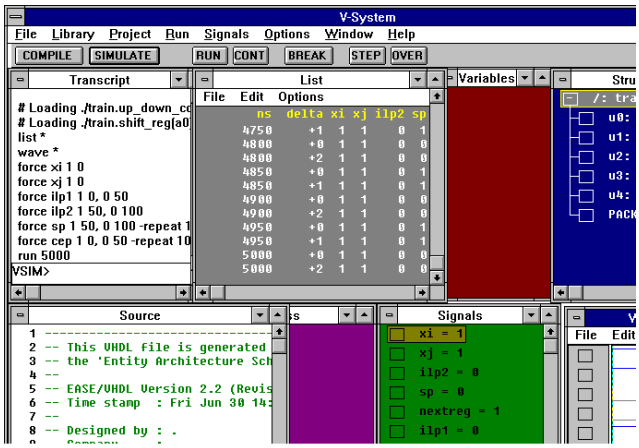

Fig. 16. Main screen of V-System/Windows

The system also allows to trace the realisation of VHDL programme, it's possible to drive the programme step by step, to pass by the procedures, to fix the breakpoints.

In general the whole system is a very good and precisely created device for simulation even very sophisticated VHDL projects. The only fault is to small screen of typical monitor used with PC computers. It's very hard to organise it in efficient and clear enough way.

B. Simulation of single neurone for training

The simulation of the architecture of single neurone for training was realised in two stages. First it was necessary to check if all simple objects work correctly. Then all inputs of whole structure were forced and the observation of behaviour and collaboration of objects was realised.

The two first components of input pattern were force to 1. The period of *SP* - clock signal was 100 ns, *CEP* - the second clock, which drives the counter worked with the same frequency as the base clock, which drives all state machines in the structure, but had the alternative phase.

The input values x_i and x_j were loaded to *Input Register* and *Holding Register* during first 50 ns and during the second 50 ns by proper forcing ILP_1 and ILP_2 signals.

The line Net_3 is the output of *Input Register* and the input of *Holding Register*, line Net_0 is the output of *Holding Register* and the first argument of *Not-Exclusive-Or* functor. Line Net_8 drives the direction of counting and the state of this line is produced by *Not-Exclusive-Or* functor.

At lines Net_13-Net_10 is possible to observe the output of counter. It's easy to notice that the counter counts up. These signals are the inputs for the *Weight Register Array*.

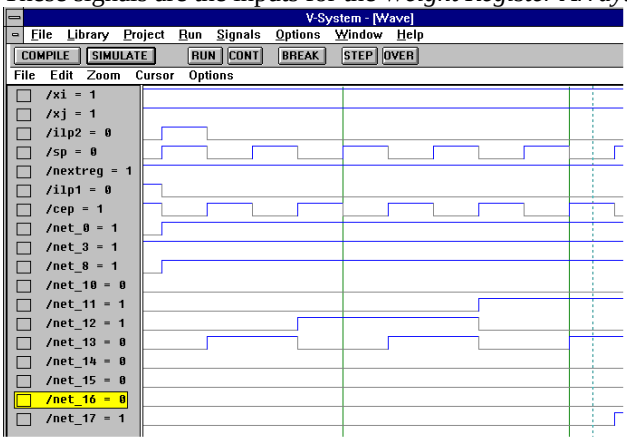

Fig. 17. Simulation of single neurone during training procedure

The states of lines Net_17-Net_14 present the same values as at the output of the *Counter*, but there are four cycles of clock signal of delay between the output of the *Weight Register* and the output of the *Counter*. This shows how the four cells *Shift Register* works.

C. Simulation of single neurone for computation

The simulation of the architecture of single neurone for computation was also realised in two stages. First it was necessary to check if all simple objects work correctly.

Then all inputs of whole structure were forced and the observation of behaviour and collaboration of objects was realised. The first components of input pattern were force to 1. The period of *SP* - clock signal was 100 ns.

This base clock drives all state machines in the structure. The input values x_i and x_j were loaded to *Input Registers* and *Control Circuits* during first 50 ns by proper forcing *ILP* signal.

The same *ILP* signal sets the initial value of accumulator to zero. The line Net_8 allows to observe the output of *Control Circuit* and the input signal to *Input Register*.

Lines Net_27-Net_30 describe the output value - the weight - from the *Weight Register Array*. The values you observe are the examples of weights fixed only for simulation in permanent way.

The *Weight Array Register* serves the values step by step without any modifications, because for this stage of work the weights are constants used by *Adder/Subtracter*.

Lines Net_17-Net_24 represent the output from the *Adder/Subtracter*, which is transmitted as a new value of partial sum.

It's easy to notice that the output from *Accumulator* - lines Net_25, Net_26, Net_14, Net_13, Net_12, Net_11, Net_0, Net_16 bring the same value which was taken from *Adder/Subtracter*.

The only difference is the delay of one clock cycle. The accumulator is loaded by zero at the begin of work and then it serves the actual value of the product evaluated during computation.

The *OLP* signal is forced as clock shape to give a chance to observe how *SCP* signal is produced by the last part of structure - line Net_5.

The same signals we can notice during the analysis of the more complex structure which is a combination of four single neurones.

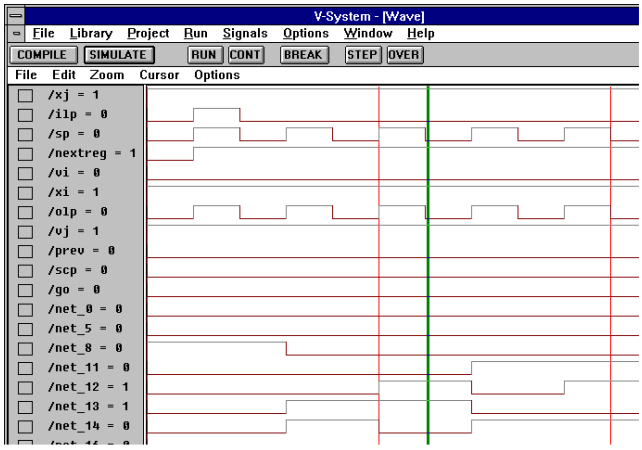

Fig. 18. Simulation of single neurone during computation (1)

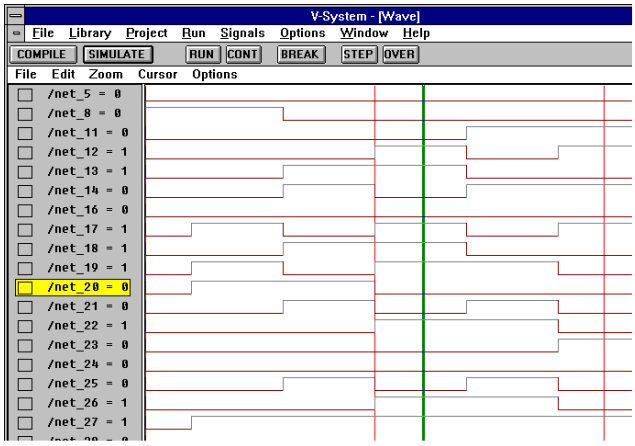

Fig. 19. Simulation of single neurone during computation (2)

V. CONCLUSION

Systolic arrays are the simplest regular and modular structures with only local short interconnections.

They are a good compromise between time consuming sequential realisation and silicon area consuming parallel architectures. In terms of silicon area required, a fully parallel implementation is very expensive.

If *N* is the number of neurones required and *S* is the area of an individual physical cell, we need an area of $O(N^2S)$ for a parallel implementation and we have 1 cycle time for the evaluation.

On the other hand, sequential algorithm requires a silicon area of $O(S)$, but the time of computation will be $O(N^2)$.

A mean solution is provided by the systolic architectures with a silicon area $O(NS)$ and a time of computation $O(N)$.

If we are going to implement *N* neurones connected as Hopfield network we have to use *2N* input registers, *N* accumulators to store the partial sums, *N* output registers to store the results, *N* multi-bit adders/subtracters and *N* up/down counters.

In addition, it is necessary to install *N* weight register arrays. Each such device is composed of *N* cells and each cell is able to store single weight with standardised precision.

The maximum speed of work of the structure depends on the technology of all functors and components were made and the speed is inversely proportional to the number of neurones.

The systolic array implementation scheme for Hopfield neural network employs completely digital circuits. The network requires *N* clock cycles for updating its weights for each input pattern.

The trained weights are stored in an array of shift registers for each neurone.

The systolic architecture for the computation of Hopfield net initialises the input register with the input vector and communicates each element of the input vector to its nearest neighbour with each clock pulse.

On the other hand there are a lot of limitations which are the serious problems if we want to extend this architecture to larger networks.

VI. ACKNOWLEDGEMENTS

The author would like to say thank the Departamento de Electrónica e Telecomunicações da Universidade de Aveiro, especially to Prof. António de Brito Ferrari for the possibility to realise the one month academic visit from 14 June to 13 July 1995 within TEMPUS Project S_JEP 07648-94. The work presented was realised during this stay.

REFERENCES

[1] K. V. ASARI, C. ESWARAN: *Systolic Array Implementation of Artificial Neural Networks*, Indian Institute of Technology, Madras 600 036, India, 1992

[2] J. N. HWANG, S. Y. KUNG: *Parallel Algorithms/Architectures for Neural Networks*, Journal of VLSI Signal Processing 1, pp. 221-251, Kluwer Academic Publishers, Boston, 1989

[3] R. TADEUSIEWICZ: *Sieci neuronowe*, Akademicka Oficyna Wydawnicza RM, Warszawa, 1993

[4] EASE/VHDL Version 2.2 Installation Guide, User's Guide, Reference Guide, Document Version EASE-IG-7, TRANSLOGIC BV, Ede, the Netherlands, November 1994

[5] EASE/VHDL & EASE/VHDL Modeler Tutorial, Document Version EASE-T-1, TRANSLOGIC BV, Ede, the Netherlands, January 1995

[6] V-System/Windows User's Manual, *VHDL Simulation for PCs Running Windows & Windows NT Version 4*, Model Technology Inc., Beaverton, USA, November 1994