Networks and Artificial Intelligence", Plenum Press,
New York (1994), pp. 167-176

PARALLEL IMPLEMENTATION OF NEURAL ASSOCIATIVE MEMORIES ON RISC PROCESSORS

U. Rückert, S. Rüping and E. Naroska

University of Dortmund
Dept. of Electrical Engineering
P.O. Box 500500
W-4600 Dortmund 50
Germany

INTRODUCTION

The implementation of neural networks by means of multiprocessor architectures is a promising compromise between flexible modelling - the system is still program controlled - and a complete implementation by means of special-purpose VLSI chips. The use of general purpose microprocessors has further advantages, e.g. they are relative cheap compared to ASICs of low or medium quantity, they are currently available, and they utilize the highest integration level of state of the art VLSI technologies.

Neural network simulation mostly uses vector-matrix or matrix-matrix operations (e.g. see Garth 1990 and Ramacher 1991). The input/output variables and the weights are usually of type 'real'. Therefore the most compute-intensive parts of simulation programs are floating point multiplications, and processors with a high floating point performance are required for fast neural net simulations. DSPs (Digital Signal Processors) and RISCs (Reduced Instruction Set Computers) are often used for this application (Treleaven 1989).

A special type of neural networks are neural associative memories (NAMs) based on distributed storage of information with binary inputs and outputs. Many different models of NAMs have been discussed and analysed in literature (Willshaw 1960, Palm 1980, Kohonen 1984) and it turns out that the majority of algorithms do not require the weight precision offered by floating point calculation. Integer arithmetic and a dynamic range for the weights (synapses) of 1 to 16 bit seems to be sufficient for NAM applications without noticeable reduction of functionality. Hence, NAM implementations impose different requirements on computational hardware as for neural network implementation in general.

In this paper we report on a case study addressing the problem of implementing NAMs on general purpose microprocessors. After a short introduction to the architecture of NAMs we will give an efficient algorithm for their parallel implementation on single microprocessors. This algorithm was mapped to well known standard microprocessors (MC88xxx, ARM2, Sparc, RS/6000, MC68xxx and Intel 286/486) which will be compared in respect to the number of connections per second (CPS) in the recall phase of the NAM. The paper concludes with a discussion of the main results of this case study.

ASSOCIATIVE MEMORY WITH NEURAL ARCHITECTURE

The structure of a neural associative memory is shown in Figure 1. The basic operation of NAMs (Kohonen 1984) are pattern mapping (heteroassociative recall) and pattern completion (autoassociative recall). In addition NAMs have the capability of fault tolerance as described in Palm (1982), Rückert and Surmann (1991), and Kohonen (1984).

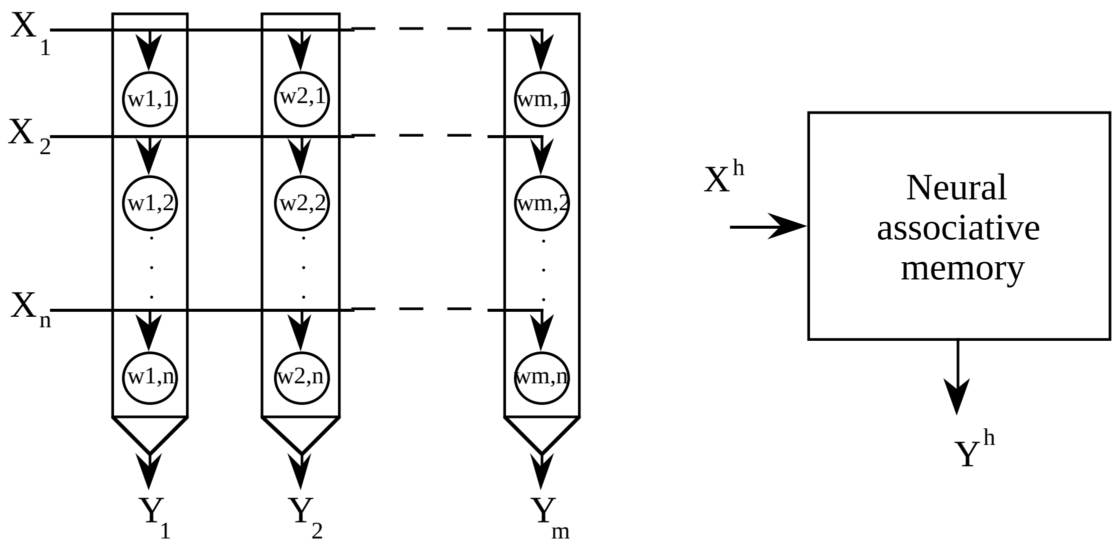

Figure 1 Structure of a neural associative memory

There are two phases in working with NAMs. The first is called learning phase in which the weights ($w_{1,1}$; $w_{1,2}$; ... $w_{m,n}$) are updated according to a learning rule and the specific set of z input-output pairs (X^h, Y^h) which have to be stored (h = 1, 2, ...,z). Each of the weights is responsible for several input-output associations (distributed storage of information).

The second phase is the recall phase, where the input X^h is presented and the memory creates the output Y^h. For each output component y_j an activity a_j is calculated by the form:

$$a_j = \sum_i w_{i,j} \cdot x_i^h$$

The associated binary output vector is obtained by a threshold operation. In general the components of the input vector and the output vector as well as the weights can be of arbitrary type. Because of the computational demands and due to storage efficiency (Palm 1990) we restrict our considerations on NAMs with binary input and output vectors and reduced weight precision of 1 to 16 bit.

Another important fact with regard to NAMs is the coding of the input- and output vectors. The theory shows, that the best storage efficiency can be reached by using sparse coding (Palm 1980, Palm 1982, Meunier et al 1991). For sparsely coded vectors the probability p for a component that takes the value '1' (representing the active state) is rather small, e.g. only l = log(n) (k = log(m)) components of the input (output) vectors are active at any time.

In summary, the dominant characteristics of NAMs in regard to implementation are a regular as well as modular architecture, a reduced weight precision and sparsely coded I/O patterns. The latter effects mainly the communication and dataflow management as will be discussed below.

MULTIPROCESSOR ARCHITECTURE

For applications like information-retrieval or associative knowledge processing (Palm et al 1991), NAMs with several thousand neurons are needed. This requires a high performance and a large size of memory of the used computers. Especially for real time applications, the solution can be a parallel architecture. For NAMs a highly scalable and simple architecture is shown in Figure 2.

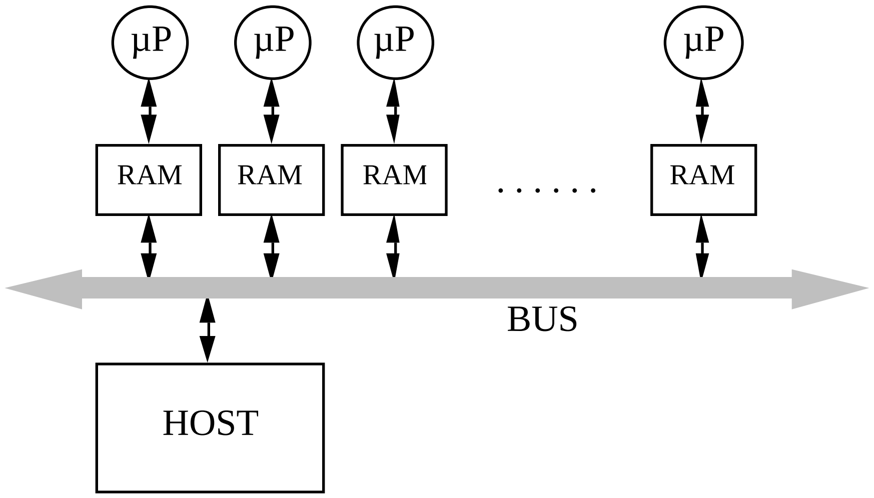

Figure 2 Multiprocessor architecture for NAMs

Several microprocessors with local memory are connected to a common bus. The controlling and management of the system is done by a host computer. For the application of NAMs there is in general no need for communication between the microprocessors. The host schedules the tasks for the processors and broadcasts the I/O-vectors and weights to the different memory parts. During the recall and learning phases all processors can work in parallel.

Each processor has to create the output for a specific number of neurons. When the calculations are finished, the host can fetch the results. Because of the sparsely coded I/O patterns the speed up in the recall phase for this multiprocessor architecture is almost linear the more mircoprocessors are added.

IMPLEMENTATION

Distribution of the weight-matrix

The distribution of the weight-matrix depends on the size of the matrix, the precision of the weights and the number of parallel processor units. In this paper we investigate processor types mainly with a databus size of 32 bit. For a weight precision of w_b bit one word of a memory segment contains $32/w_b$ weights. It is very useful to split the matrix into vertical slices, so that each microprocessor (one per slice) can calculate its output locally. The new weights during the learning phase and also the activity during the recall phase can be calculated for each slice without having information about the other slices.

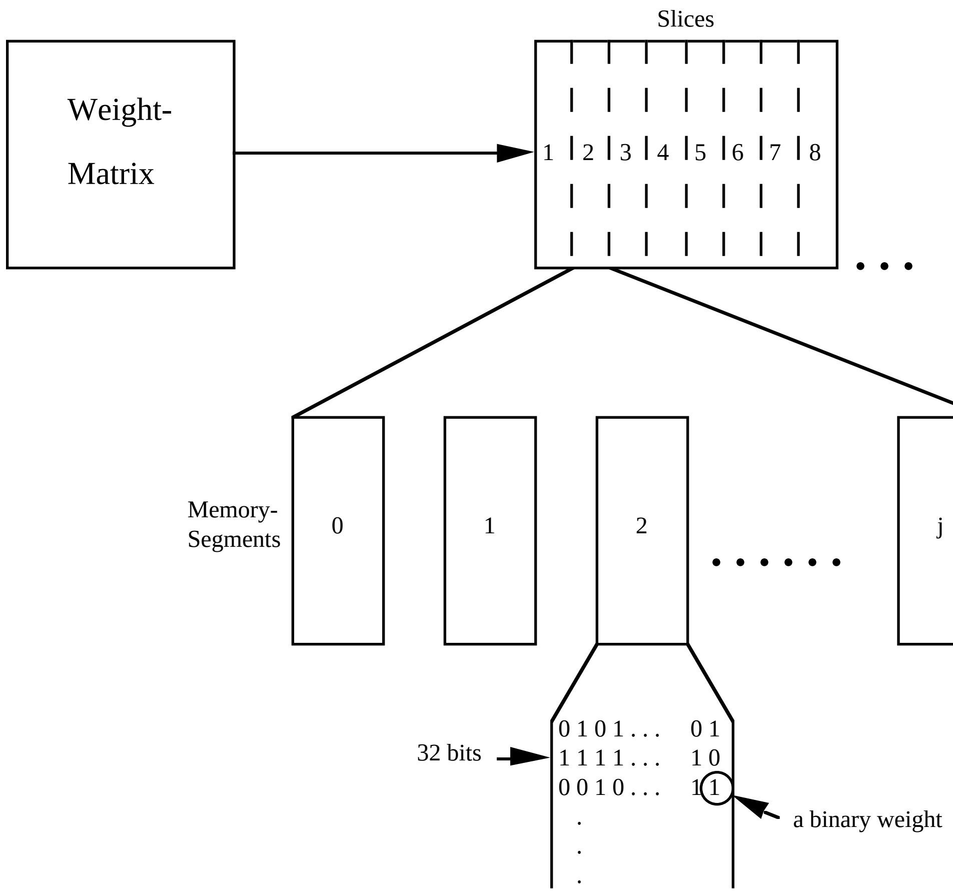

Figure 3 Splitting the weight-matrix for parallel processing

The different processors work on several memory segments as shown in Figure 3. If the system has processors with similar performance, the number of segments per slice should be about the same.

As mentioned before, sparse coding of input and output vectors will result in a relatively simple communication management. Instead of transfering a n-bit vector only l addresses of size log(n) have to be transfered (Palm and Palm 1991, Rückert et al 1992).

The overflow algorithm

In order to utilize the full performance of a given microprocessor we developed the new so called "overflow algorithm". The main idea of this algorithm is to handle a subset of $32/w_b$ weights contained in each memory word in parallel. Figure 4 shows the structure of the memory words.

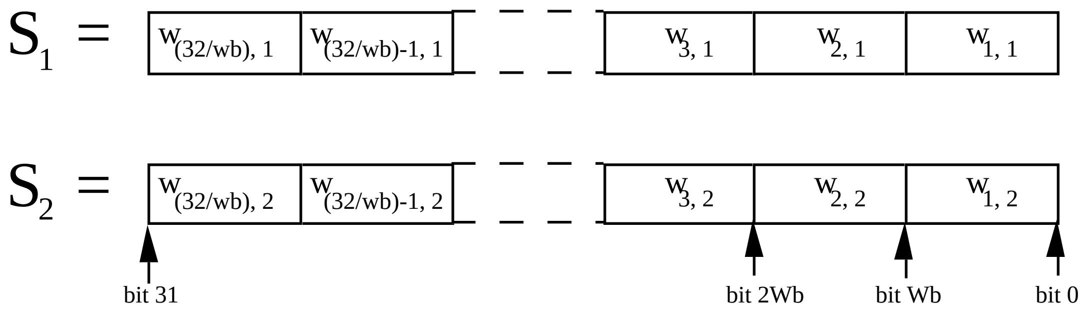

Figure 4 Internal structure of two memory words (S_1, S_2)

The problem is that adding the two words S_1 and S_2 can create an overflow between two adjacent weights. Therefore it is necessary to split S_1 into $S_{1,1}$ and $S_{1,2}$ as well as to split S_2 into $S_{2,1}$ and $S_{2,2}$. This is shown in Figure 5. $S_{1,1}$ contains the weights with an odd index i. The space between the weights is filled with zeros. This extra space can now be used for the overflow bits. $S_{1,2}$ will do the same for the weights with an even index i. Therefore the word must be shifted w_b bits to the right to bring the weights into the correct position. S_2 is handled in the same way. The splitting can be done by using two standard microprocessor commands, "AND" and "SHIFT".

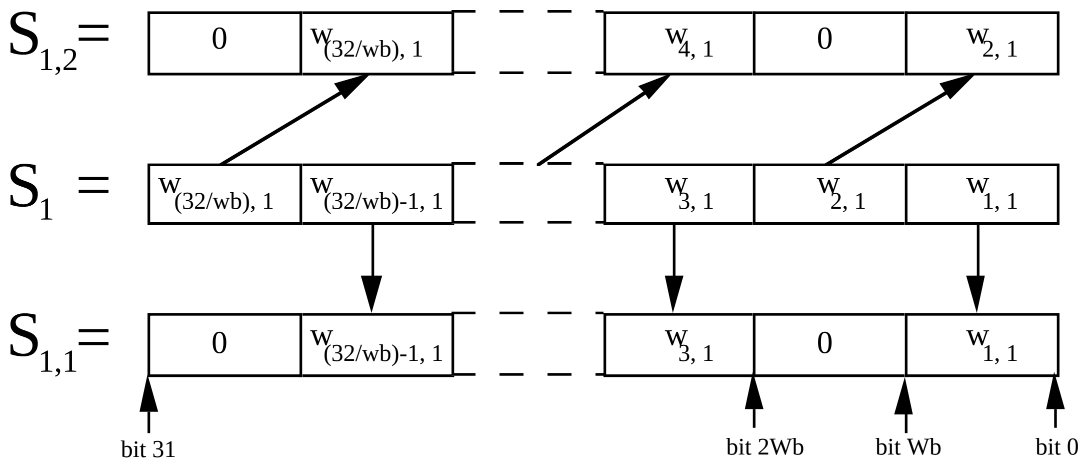

Figure 5 Splitting S_1 into $S_{1,1}$ and $S_{1,2}$

After splitting $S_{1,1}$ and $S_{2,1}$ can be added. A possible overflow will be stored in the space between the weights. A single "ADD" instruction is now able to handle $32/(2\cdot w_b)$ weights in parallel. The result of $S_{1,1} + S_{2,1}$ is shown in Figure 6.

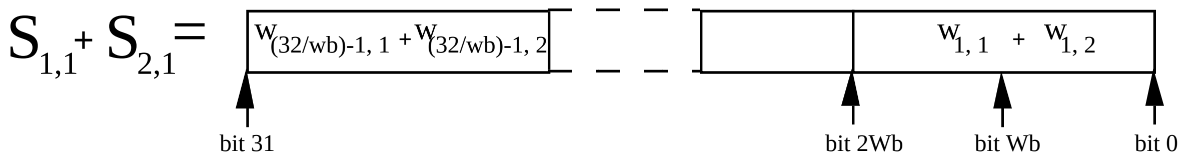

Figure 6 Adding $S_{1,1}$ and $S_{2,1}$

The number of additions which are possible without getting overflow problems is given by:

$$\# \text{add} = \frac{2^{2 \cdot w_b} - 1}{2^{w_b} - 1}$$

When this number is reached, splitting can be done again and adding the weights can proceed.

Assuming $w_b = 4$ bit and a word length of 32 bit the microprocessor has to isolate and add 8 weights serially. Using the overflow algorithm, the microprocessor operates on 4 weights in parallel and the cycle "isolate and add" has to be done only twice. In general a single microprocessor is able to handle about

$$c\# = \frac{w_{proc}}{2 \cdot w_b}$$

columns (neurons) of a NAM in parallel. The parameter w_{proc} is the width of the data bus of the processor (at present mainly 32 bit). In other words standard microprocessors are capable of emulating about c# processors with a smaller data path w_b in parallel. Obviously, for binary weights the highest parallelism is achieved.

COMPARISON OF DIFFERENT GENERAL PURPOSE PROCESSORS

Description of the measurements

For testing the performance of the different processors a complete association of a 1024x1024 bit matrix is done. The input vector has 20 active components ($l = 20$) and the output vector contains $k = 1024/(w_b \times 32)$ active bits. That means, if w_b is equal to 1 the output vector has a dimension of 1024 and contains 32 active components. If w_b is equal to 4 the dimension is 256 and the number of active components is 8. This condition guarantees that the storage requirement for NAMs with different weight precision (w_b = 1, 2, 4, 8, 16) is the same.

All vectors are stored as a list of addresses which describe the positions of the active bits. It is important to mention that the output vectors are also stored in this way, because the transfomation from the bit form to the address form takes some time and is included in our time measurements.

Table 1 shows a list of all measured (and estimated) times for the different processors expressed as MCPS (Million Connections Per Second). There are three types of values summarized in this table. They are marked by the letters A, C and E. The letter A means that the program is written in Assembler, C means that the program is written in C-language and E means that the MCPS value was estimated by the clock rate, the necessary commands and the number of clock cycles per command.

Obviously, the processors investigated in this case study are not a complete set of all general purpose microprocessors. The listed processors were available at our institute.

Comparing the different results it is important to remind the way how the values were measured. A C-program is usually not as optimized as an assembler program. To investigate this, Table 2 shows a comparison between the resulting MCPS for a C-program and for an assembler program.

Table 1 Results for different weight precisions

	$w_b = 1$	$w_b = 2$	$w_b = 4$	$w_b = 8$	$w_b = 16$	
ARM2	7,9 (A)	5,2 (A)	2,9 (A)	1,6 (A)	0,9 (A)	MCPS
MC68000	2,0 (A)	1,7 (E)	1,0 (E)	0,5 (E)	0,4 (E)	MCPS
80286	2,1 (A)	2,2 (E)	1,2 (E)	0,6 (E)	-	MCPS
MC68030	20,6 (A)	17,8 (A)	8,4 (A)	5,0 (A)	2,7 (A)	MCPS
MC88100	27,8 (A)	18,8 (E)	10,0 (E)	5,7 (E)	2,9 (E)	MCPS
i486-50	34,1 (C)	29,3 (C)	13,5 (C)	7,5 (C)	4,1 (C)	MCPS
RS/6000	18,0 (C)	15,8 (C)	8,4 (C)	4,9 (C)	2,8 (C)	MCPS
Sparc2	14,5 (C)	13,8 (C)	6,6 (C)	3,8 (C)	2,0 (C)	MCPS

A : Assembler   C : C programming language   E : estimated

Table 2 Comparison of C and Assembler programs

	Assembler	C	
ARM2	7,9	2,1	MCPS
MC68030	20,6	9,3	MCPS
MC88000	27,8	10,2	MCPS

It seems to be a fact that a program written in assembler is about 2 to 4 times faster than a C-program for this special application of NAMs.

On the other hand it is important which compiler is used. Table 3 shows a comparison of two different C-compilers on a i486-50MHz computer. In this example the program produced by the second compiler is nearly two times faster than the other.

Table 3 Comparison of two C-compilers on a i486-50MHz computer

	N = 1	N = 2	N = 4	N = 8	N = 16	
Zortec	20,9	16,8	8,83	4,92	2,67	MCPS
Gnu	34,1	29,3	13,5	7,53	4,13	MCPS

DISCUSSION

Comparing the different processors can not be done simply by comparing the MCPS values. As shown in Table 2 and Table 3 it is very important to remind the type of computer language and also the special product that is used. For this application of NAMs some compilers do a much better job than others, which does not mean that they are much better products because they surely have their disadvantages for other applications.

The first 5 processors listed in Table 1 are measured using an assembler program. Therefore the simulation times are almost minimized. The last 3 processors are measured by running a C-program, so the listed performances are expected to increase for running assembler programs.

It is an interesting result that the i486-50MHz processor is about 2 times faster than the Sparc processor (40 MHz). A possible explanation is the usage of different C-compilers. With the Zortec compiler the i486-50MHz has a performance of 20,9 MCPS which is about the same as the RS/6000 and a little faster than the Sparc. It might be possible that another compiler on the SUN workstation could increase the performance. Other explanations might be the different sizes of cache memory or the influence of the operating system (Unix, MS-DOS, OS/2, ...) on the performance which is difficult to estimate. Nevertheless the i486-50MHz seems to be a powerful processor for this special type of application. The instructions that are used for the overflow algorithm are very simple (ADD, AND, SHIFT) and are executed by the i486-50MHz in a few clock cycles. This could explain the fact that this special CISC processor is as fast as the RISC processors.

Comparing the Motorola microprocessor families MC68k (CISC) and MC88k (RISC), the RISC architecture is the better choice for this special application, as might be expected. With a clockrate of 16 MHz the MC88100 is about 1.3 times faster than the MC68030 with a clockrate of 50 MHz ($w_b = 1$, Tab. 1). However CISC processors as the 68k and i486 are mass products and hence cheaper than RISC processors, in general.

A comparison of different generations of the same microprocessor family (68xxx, 80x86) shows a considerable speed up of 10 or more. This should be considered while developing application specific integrated circuits (ASICs), especially in respect to the expected cost/performance ratio of the complete system.

Another interesting result is that the gain in performance from $w_b = 2$ to $w_b = 1$ is not as high as might be expected (a factor of 2). The reason for that is the time needed for the transformation of the output vectors from bit to address form. The largest gain results for the MC88100. This is mainly because the processor has appropriate instructions for this transformation.

Nevertheless, in this paper it is not our intension to find out the best microprocessor for simulation of NAMs. This seems to be impossible at the moment because a general benchmark is missing for that work. Such a benchmark should provide the same presuppositions for all processors. There are a lot of dependencies which influence the performance and which could not be simply estimated (compiler, cache memory, operating system, ...). At present we are working on a specification of an appropriate benchmark for NAMs based on distributed storage and sparsely coded I/O patterns.

Overall the speeds available from general purpose microprocessors are sufficient for a range of NAM applications. For example, giving a 4096x4096 NAM with binary weights ($w_b = 1$) in which about 320000 patterns ($l = 12$, $k = 3$) can be stored with low error probability (Rückert 1991), a single i486-50MHz microprocessor (34 MCPS, Tab. 2) performs about 700 associations per second. The associations per second are given by:

$$\frac{\text{Assoc}}{\text{sec}} = \frac{\text{MCPS}}{l \cdot n}$$

It should be possible to increase the performance by working on assembler level. If even higher speed is required a multiprocessor architecture (Fig. 2) could be used. Last but not least we have to be aware of the fact that the third RISC processor generation offers a 64 bit data path, an internal clock rate above 100 MHz and a superpipelined and superscalar architecture (Weiss 1992).

CONCLUSION

The reported case study on simulation of NAMs on the basis of general purpose microprocessors is interesting in three aspects at least.

Firstly, adapting the implementation of NAMs to the characteristics of the hardware gave us an efficient algorithm with which it is possible to handle about $c\# = w_{proc}/2w_b$ neurons in parallel by a single processor. For a low weight precision ($w_b = 1, 2$) a speed up factor of about 10 is available.

Secondly, in order to achieve peak performance low level work on assembler program level is rewarded. Our assembler programs are about 2 to 4 times faster than C-programs for this special application of NAMs.

Thirdly, the speeds available from general purpose microprocessors are sufficient for a rage of NAM applications. Hence, it seems to be advisable to prove for each application of NAMs, whether a high cost ASIC solution is necessary or a low cost solution with general purpose microprocessors would meet the requirements.

ACKNOWLEDGEMENT

This work has been partly supported by the german ministry of research and technology BMFT, contract number 01-IN 103 B/O.

REFERENCES

Garth, S., "Simulators for neural networks", in Advanced Neural Computers, R. Eckmiller (ed), Elsevier Science Publishers, 1990, pp 177-183.

Kohonen, T., "Selforganisation and Associative Memory", Springer Verlag, Berlin, 1984.

Meunier, C., Yanai, H.F., Amari, S.I., "Sparsely coded associative memories: capacity and dynamical properties", Network, November, 1991, pp 469-487

Palm, G., "On Associative Memory", Biol. Cybern. 36, 1980, pp 19-31.

Palm, G., "Neural Assemblies", Springer Verlag, Berlin, 1982.

Palm, G., Local Learning Rules and Sparse Coding in Neural Networks, in Advanced Neural Computers, R. Eckmiller (ed.), North-Holland, 1990.

Palm, G., Rückert, U., Ultsch, A., "Wissenverarbeitung in Neuronaler Architektur", Tagungsband zum 4. Int. GI-Kongress: Wissensbasierte Systeme - Verteilte kuenstliche Intelligenz, Muenchen, 1991.

Palm, G., Palm, M., Parallel Associative Networks: The PAN-System and the Bacchus-Chip, in Proc. of the 2nd Int. Conf. Microelectronics for Neural Networks, U. Ramacher, U. Rückert, J.A. Nossek (eds.), Kyrill&Method Verlag, München 1991.

Ramacher, U., "Guide lines to VLSI design of neural nets", in VLSI Design of Neural Networks, U. Ramacher, U. Rückert, Kluwer Academic, 1991, pp 1-17.

Rückert, U., "VLSI Design of an associative memory based on distributed storage of information", in VLSI Design of Neural Networks, U. Ramacher, U. Rückert, Kluwer Academic, 1991, pp 1-17.

Rückert, U., Surmann, H., "Toleranz of binary associative memory towards stuck-at-faults", in Proceedings of the International Conference on Artficial Neuronal Networks (ICANN-91), Helsinki, 1991, pp 1195-

Rückert, U., Funke, A., Pintaske, C., "Acceleratorboard for Neural Associative Memories", to be published in Microelectronics for Neural Networks, Neurocomputing, Vol. 4, No. 6, 1992

Treleaven, P.C., "Neurocomputers", International Journal of Neurocomputing, Issue 1, 1989, pp 4-31

Weiss, R., "Third-generation RISC processors", EDN March 30, 1992, pp 97-108

Willshaw, D.J., Bunemann, O.P., Longuett-Higgins, H.C., Non-holografic associative memory, Nature 222, 1969, pp 960-.