FAST NEURAL NETWORK IMPLEMENTATION

*Miroslav Skrbek**

Abstract: This article is focused on implementation of neural networks in hardware. We give an overview of so-called shift-add neural arithmetics, which provides a complete set of functions suitable for fast perceptron and RBF neuron implementation. The functions use linear approximation to reach sufficient simplicity. Since the linear approximation produces imprecise results, precision analysis and tests in the learning process are included. They show a very small negative influence of the linear approximation on the neuron behavior.

Furthermore we show the gate-level implementation of all functions provided by the shift-add arithmetics. Only adders and barrel shifters are necessary to accomplish all complex functions like multiplication, square, square root, logarithm, exponent and non-linear activation function of a neuron. All functions are optimized to be performed in a very short time (a few nanoseconds).

We made the first implementation of the shift-add arithmetics on FPGAs. The tests were proceeded on the ECX card. The results are presented in this article. Further implementation, which is also presented in this article, was an on-chip design. Propagation delays and the necessary chip area are discussed in the last section.

Key words: *Neural network, implementation, neurochip, FPGA, on-chip design*

Received: April 19, 1999
Revised and accepted: September 15, 1999

1. Introduction

Personal computers as an implementation platform for neural networks are very popular at present. Many of neural net simulation tools are available (Neural Works, Brain Maker, etc). They provide a very flexible platform to implement neural nets varying in type and size. Since personal computers mostly contain one processor, the time spent by neural calculations grows with the size of the network. This dependence can be suppressed by using parallelism. However, special hardware is necessary to implement many neural units working in parallel.

*Miroslav Skrbek
Department of Computer Science and Engineering, Czech Technical University in Prague, Karlovo nám. 13, 121 35 Prague 2, Czech Republic, E-mail: skrbek@fel.cvut.cz

Since 1990, dedicated neural integrated circuits (neurochips) have been commercially available. These chips are optimized to perform neural calculations with the aim to speed up not only the recall a phase, but also the learning phase. Neurochips can be arranged into a large neural system to provide sufficient power. They are suitable for time-critical or embedded applications.

There are many ways how to design a fast neural processing element for a neurochip. Most of them are based on various technologies (analog, digital or mixed). Fast neural units must be very simple to reach a sufficiently small propagation delay. On the other hand, it must keep most of the required functionality. This article is just focused on problems of neural chips and the design of their internal structure.

Each neurochip itself contains a number of independent neural units. One can find that most operations in neural networks are addition, multiplication, nonlinear function (sigmoid-like) and square function. Sometimes the square root function is necessary if the Euclidean measure is applied. Besides addition, the other functions can be considered as complex because their implementation is very time- and chip-area consuming.

Neurochips use various simplifications to implement these functions. Most of them are based on reducing the input and output ranges of neurons. Binary inputs and outputs are an example. Binary inputs do not need multipliers and the sigmoida-like activation function of a neuron is a simple step function. For no binary inputs and outputs, the multiplier and continuous activation functions are very complex and slow. Especially, if many fast processing units are implemented on a chip.

2. Shift-Add Arithmetics

This section gives an overview of neural arithmetics based on linearly approximated functions. Fundamental stones of the shift-add arithmetic are 2^x, $\log_2 x$, sigmoid-like, gauss-like functions. All these functions are based on the shift operation in combination with the linear approximation.

2.1 Simple functions

The linearly approximated 2^x function can be defined by the expression

$$\mathrm{EXP}_2(x) = 2^{\mathrm{int}(x)}(1 + \mathrm{frac}(x)). \tag{1}$$

The **shift operation** calculates 2^n, where $n \in N$. In the range of $\langle 2^n, 2^{n+1})$, the linear approximation is used. Because $n = \mathrm{int}(x)$, we separate x to its integral and fractional ($\mathrm{frac}(x)$) parts. The integral part of x assesses the number of shift-left operations for the $(1 + \mathrm{frac}(x))$ expression.

The linearly approximated $\log_2 x$ function can be expressed by

$$\mathrm{LOG}_2(x) = \mathrm{int}\,(\log_2(x)) + \frac{x}{2^{\mathrm{int}(\log_2(x))}} - 1. \tag{2}$$

This equation is valid for x from the range of $\langle 1, \infty)$. For easy implementation, zero as a result is generated for x in the range of $\langle 0, 1)$. The integral part of the

result, $\mathrm{int}(\log_2 x)$, is equal to the left-most one in the bit-grid of x. The fractional part of the result is composed of all bits of the bit-grid of x that follow the left-most one (left to right direction).

The sigmoid or tanh functions are often used an activation function of a neuron. Both functions are based on e^x which is very complex to calculate. Instead, we introduce a **new function** based on the power of two. This function is defined by $s(x) = \mathrm{sgn}(x)\left(1 - \frac{1}{2^{|x|}}\right)$. Its linearly approximated version is

$$S(x) = \mathrm{sgn}(x)\left(\frac{1}{2^{\mathrm{int}(|x|)}}\left(\frac{\mathrm{frac}(|x|)}{2} - 1\right) + 1\right). \tag{3}$$

The range of this function is $(-1, 1)$, like tanh; however, this is much closer to the sigmoid function adopted to the range of $(-1, 1)$. Only the shift operation is necessary for calculation of this function. The first step is to prepare a fractional part of x by shifting x once to the right (inserting zero from the left). The second step is to shift this value by $\mathrm{int}(x)$ to the right. Ones must be inserted from the left in this case.

RBF neural networks mostly use the gauss-like activation function. Instead of this, we define a new function $g(x) = 2^{-|x|}$. The linearly approximated version of $g(x)$ can be defined by the expression

$$G(x) = \frac{1}{2^{\mathrm{int}(|x|)}}\left(1 - \frac{\mathrm{frac}(|x|)}{2}\right). \tag{4}$$

Both $S(x)$ and $G(x)$ functions are shown in Fig. 1. As can be seen, the appearance of $S(x)$ and $G(x)$ in the shape of the sigmoid-like and gauss-like functions is preserved. The very small error caused by linear approximation is obvious.

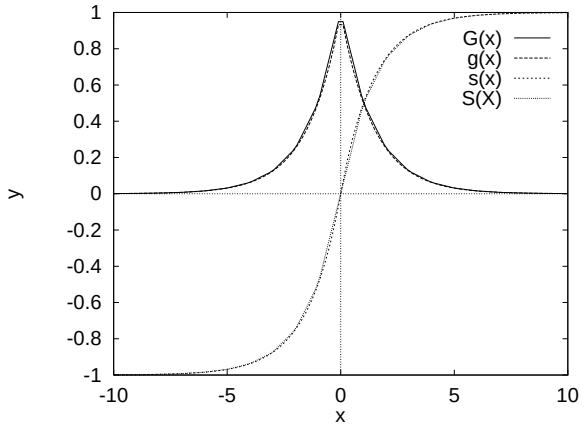

Fig. 1 *$S(x)$ and $G(x)$ functions.*

2.2 Complex functions

Complex functions are a composition of simple functions. Some complex functions require additional add operation to accomplish them. Therefore, we are talking

about shift-add architecture. The complex functions are multiplication, square, and square root functions.

Multiplication is defined by

$$x * y = \mathrm{sgn}(x)\mathrm{sgn}(y)\frac{\mathrm{EXP}_2(\mathrm{LOG}_2(2^n|x|) + \mathrm{LOG}_2(2^n|y|)}{2^{2n}}, \tag{5}$$

where n is the width of the bit grid on which calculations are performed. The arguments, x, y, and the result are in the range of $(-1, 1)$. It is useful because it ensures that the product never exceeds one. The sign of the result is evaluated separately.

The square function is defined for x in the range of $(-1, 1)$ by

$$\mathrm{Sqr}(x) = \frac{\mathrm{EXP}_2(2\mathrm{LOG}(2^n|x|))}{2^{2n}}.$$

The root square function is defined for $0 < x < 1$ by

$$\mathrm{Sqrt}(x) = \frac{\mathrm{EXP}_2\left(\frac{1}{2}\mathrm{LOG}(2^n x)\right)}{2^{\frac{n}{2}}}.$$

3. Precision Considerations

The linearly approximated functions induce an error to the result. The basic question is whether the error can influence the neuron behavior. Important information is the maximum error, which is generated by individual functions. Tab. I summarizes the errors calculated analytically. The table also presents the results of statistical evaluation. The statistical characteristic has been calculated under assumption that arguments of a given function are independent random variables with uniform distribution. They are useful if we want to use the weights of a learned neural net for a net using shift-add arithmetics.

Function	$\epsilon_{\max}$	$\mu(\epsilon)$	$\sigma^2(\epsilon)$
$\mathrm{EXP}_2(x)$	0.086	n/a	n/a
$\mathrm{LOG}_2(x)$	-0.086	n/a	n/a
$x * y$	∓ 0.0625	0	$2.268\ 10^{-4}$
$S(x)$	∓ 0.043	0	$1.64\ 10^{-4}$
$G(x)$	-0.043	n/a	n/a
$\mathrm{Sqr}(x)$	0.043	n/a	n/a

Tab. I *Precision of linearly-approximated functions.*

Note: The maximum error of the $\mathrm{EXP}_2(x)$ function is scaled to the maximum output range due to the other error compatibility. Variance of $S(x)$ is calculated for the x range of $(-8, 8)$.

The table shows that the maximum error of all significant functions is less than 6.5%. This very small error indicates good functionality. If we use weights of

the learned neural network for the network based on shift-add arithmetics, the missclassification is only 5% greater.

The Fig. 2 shows the influence of imprecise operations on the decision boundaries. The left figure is the decision boundary of the neuron of the perceptron type. Precise calculation would produce straight lines in this case. The figure on the right is the decision boundary of the RBF neuron. Precise calculations would produce smooth circles in this case. The linear approximation produces a systematic error, no noise-like error. The experimental results show that a learning algorithm can easily suppress these errors.

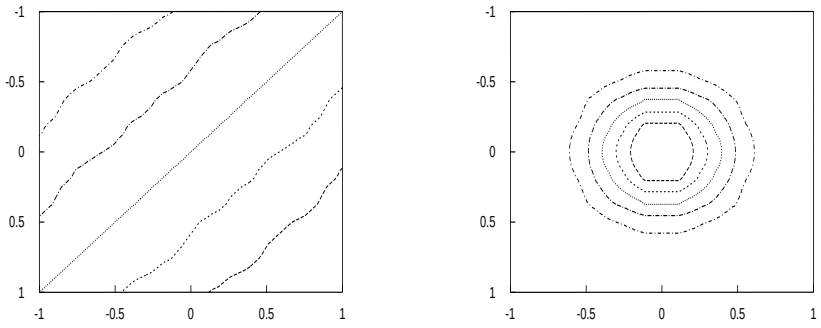

Fig. 2 *Decision boundary of a neuron of a perceptron type (on the left) and the RBF type (on the right).*

4. Software Models

First the proposed architecture was simulated by software models. The shift-add processing element model was written in the assembler and encapsulated in the PASCAL procedures. These procedures were used to build the feed-forward neural network which was learned by the standard Back propagation (BP) learning algorithm. All calculations used in the learning rules of the BP algorithm were performed as floating-point operations. Each weight had both floating-point and integer representations. The delta-rule updates (BP) affected the floating-point weights, whereas the recall phase used the integer weights. Weight convertions (floating-point to integer) were done at the end of each iteration.

A number of neural benchmarks, which are commonly used, were applied for the shift-add neural net testing. The XOR problem, parity problem, digit recognition, an inside-outside test, sonar signal recognition[1] were used.

Fig. 3 shows the learning curves for the XOR problem. The curve, BP, describes the learning process error of the neural net without shift-add processing elements. The curve, AP, was obtained for the net by processing elements using the activation function according to Eq. (3). The multiplication based on the shift-add arithmetics was not applied for this net. The last curve, NP, shows the learning curve of the net with shift-add processing elements being fully implemented.

[1] The data set used by Gorman and Sejnowski in their study of the clasification of sonar signals by neural nets. It is a benchmark provided by Carnegie Mellon University School of Computer Science.

The experiments show that the BP and NP curves differ due to the shape differences between the sigmoida and the $S(x)$ function. The higher slope of $S(x)$ function (for $x = 0$) speeds up the learning process. On the contrary, the very closed AP and NP curves show a very small negative influence of the linearly approximated multiplication function on the net behavior.

Furthermore, generalization was tested as a very important neural net property. The result of the generalization test of the XOR problem is shown in Fig. 4. Since the network has two inputs and one output, an approximation of the input space can be shown as a grey-scale picture. The pixel coordinates in the picture relates to the neural net inputs and the pixel color relates to the neuron output. The white lines show the decision boundaries of hidden neurons.

If a sufficiently small learning error is reached, then solutions corresponding to the BP and NP learning curves are very close. Fig. 4 shows the surface approximating the XOR function and its contours. The contours of the surface show that the surface is not seriously corrupted.

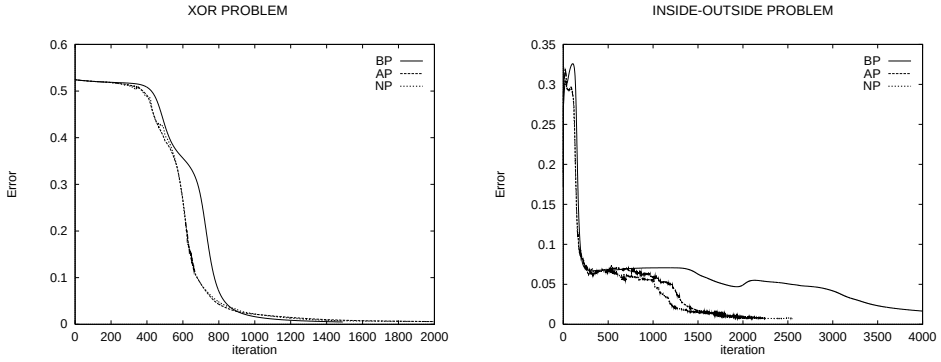

Fig. 3 *Learning error of XOR problem and inside-outside test.*

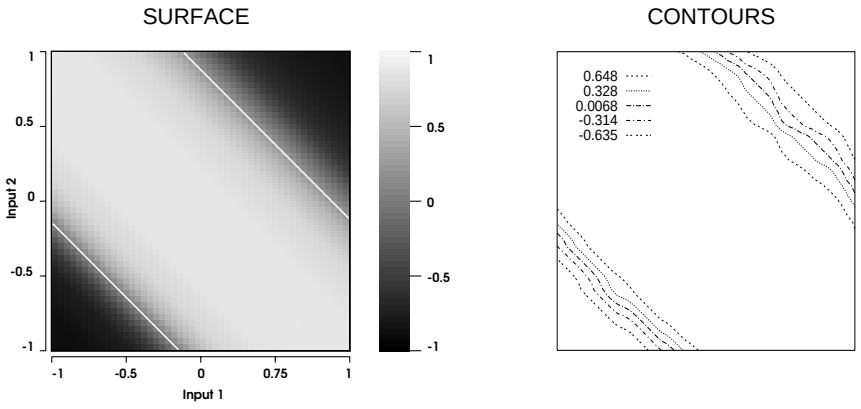

Fig. 4 *XOR problem – generalization test.*

A more complicated benchmark is the inside-outside problem. It also needs a two-input neural network with one output and one hidden layer. The training set contains pairs: coordinates of a point in the input space, and the desired output value expressing the membership of the point to a given set of points in the input space. We used a set in the shape of two clusters located on the diagonal of the input space. The lower cluster is smaller than the upper one. The generalization can be shown in the same way as it was for the XOR problem.

The results are shown in Fig. 5. The related learning curves are shown in Fig. 3 (right side). Fig. 5 shows that the shift-add neural network provides very good approximation. Even the approximation of NP is better than BP in this case because the learning process of BP was stopped at a higher learning error. The faster convergence is obvious from Fig. 3.

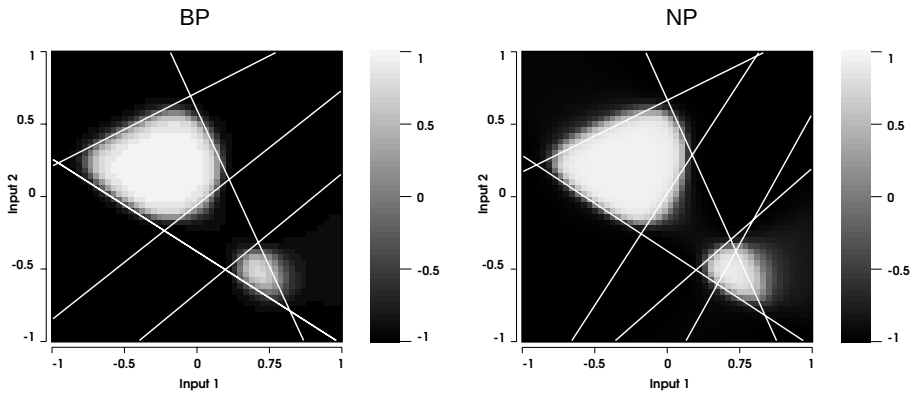

Fig. 5 *Generalization test of the inside-outside problem.*

5. Implementation in Hardware

In this section, we show the implementation of functions defined in Sect. 2. Most functions are based only on the shift operation that can be implemented very easily in hardware. Simplicity of all functions enable high operational speed. Implemented functions are arranged to functional blocks. Each block accepts or generates a value in the fixed point format. The following formats are recognized: *fracval (FV)*, *logval (LV)* and *sgmval (SV)*. The FV value represents numbers in the range of $(-1, 1)$, the LV value is the logarithm of FV multiplied by 2^{16}. The SV is a number in the range of $(-16, 16)$. All formats use a direct code to express negative numbers. The implementation presented uses sixteen-bit-wide data paths, but it may be scaled for any meaningful width.

5.1 Basic building blocks

The $\mathrm{EXP}_2(x)$ function is implemented by the EXP block (see Fig. 6). This block accepts a *logval* value which is the logarithm of the sixteen bit *fracval* number. The output of this block generates the *fracval* value that can be directly accepted by

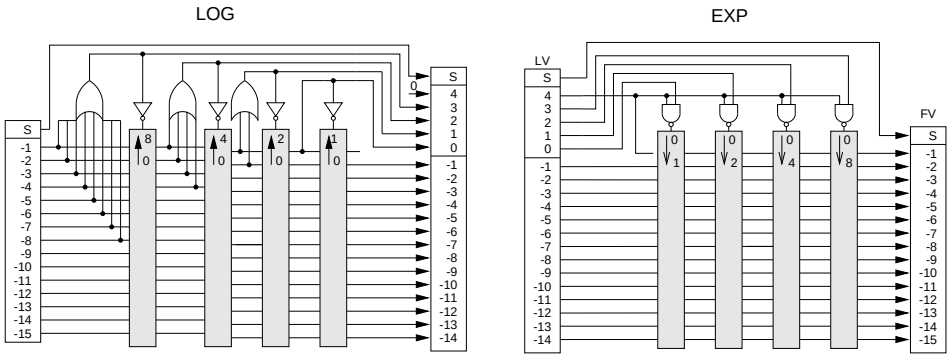

Fig. 6 *LOG and EXP block implementation.*

the dot-product accumulator. This block is dedicated for multiplication; therefore it includes also division by 2^{2n} (see. Eq. (5)). The sign is passed from the input to the output without changes.

The EXP block is a barrel shifter which is slightly modified to force the zero output for small numbers being out of the 16-bit precision.

The $\mathrm{LOG}_2(x)$ function is implemented by the LOG block (see Fig. 6). This block consists of a chain of four individually controlled shifters. The first of them shifts the input value by eight if eight msb bits are equal to zero. Otherwise, the input value is passed through without changes. The second shifter does the same but for four msb bits. When the result appears at the output of the fourth shifter, the msb bit, which is not connected, indicates the logical one. This is the one, which is shifted out of the bit grid because it is expressed by the exponent. The other bits of the shifter are passed as fractional parts of the result to the output.

The AF block implements the $S(x)$ function and AFR block implements the $G(x)$ function (see Fig. 7). The structure of both blocks is very similar. Both blocks contain barrel shifters only. The AF block does not change the signum because the sigmoid-like function follows the signum of the dot-product. The $G(x)$ function generates positive outputs only; therefore, the signum of the output is permanently cleared. Both blocks accept a dot product value (SV) in the range of $(-16, +16)$. The *fracval(FV)* value is generated at the output.

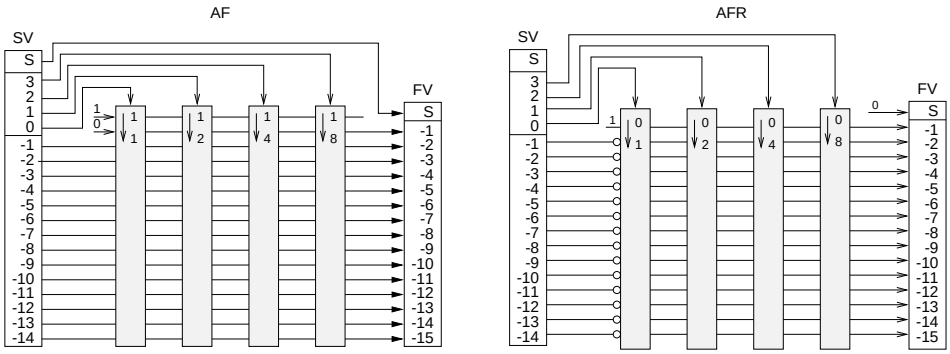

Fig. 7 *The AF and AFR block implementation.*

5.2 Composed blocks

The multiplier, square and square root blocks are considered as complex blocks. Each composed block is a composition of the basic blocks.

Multiplier

The multiplier is based on two LOG blocks and one EXP block. Two input values passed through the LOG blocks are added and sent to the EXP block after that. The signum is calculated separately by the XOR function.

Square function

The square can have the same structure as the multiplier, but the adder is replaced by the shift-to-the-left operation. We also propose an implementation that avoids the barrel shifter of the LOG block. This can be described by the equation

$$\mathrm{Sqr}(x) = (a + 2c) \overset{0}{\ll} (n + x_{n-1})$$

where $a = 2^n$, n is a position of the left-most one, $b = x_{n-1}2^{n-1}$, x_{n-1} is a value of the bit which is the right neighborhood of the left-most one, $c = x - a - b$. The $\overset{0}{\ll}$ symbol represents the shift to the left operation (zeros are inserted from the right). The $\mathrm{Sqr}(x)$ function is evaluated in two steps. The first step is to determine the a, b, c, and $n+x_{n-1}$. The second step is shift to the right by $n+x_{n-1}$. If the operation is performed for numbers in the range of $(0, 1)$, then the negative argument changes the direction of the shift operation. Detection of the left-most one (i.e. $n + x_{n-1}$) and $2c$ calculation can be done by a chain of simple blocks (see Fig. 8).

The $\mathrm{Sqr}(x)$ block receives *fracval(FV)* value (sixteen bit). Bits $2^{-1} \ldots 2^{-9}$ of the input are passed through the chain. After this, the chain calculates $a + 2c$ and sets the XC outputs. Simultaneously, one of SHR outputs is activated. The active the SHR bit asks the barrel shifter for an appropriate number of shifts. The XC bits are completed by $2^{-9} - 2^{-15}$ bits and the complete tuple is passed through the shifter.

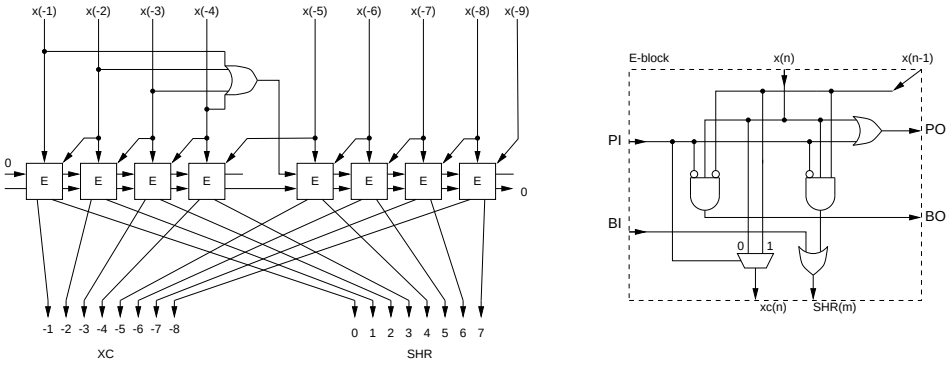

Fig. 8 $\mathrm{Sqr}(x)$ *chain.*

Square root function

Similarly, we can calculate the square root function. It can be realized as a composition of the LOG, shift, and EXP blocks. We can also use an alternative implementation defined as

$$\mathrm{Sqrt}(x) = \begin{cases} (a + d/2) \overset{0}{\gg} (n/2) & \text{if } n \text{ is even} \\ (a + a/2 + d/2) \overset{0}{\gg} (\frac{n-1}{2}) & \text{if } n \text{ is odd} \end{cases}$$

where $a = 2^n$, n is a position of the left most one and $d = x - a$. The internal structure of SQRT block is similar to the SQR block.

5.3 Neural blocks

The blocks defined in Sect. 5.2 correspond to particular functional units of a neural processing element (PE). They create a collection of blocks sufficient for implementation of the processing elements of both perceptron and RBF types. The structure of the perceptron and the RBF processing element are shown in Fig. 9.

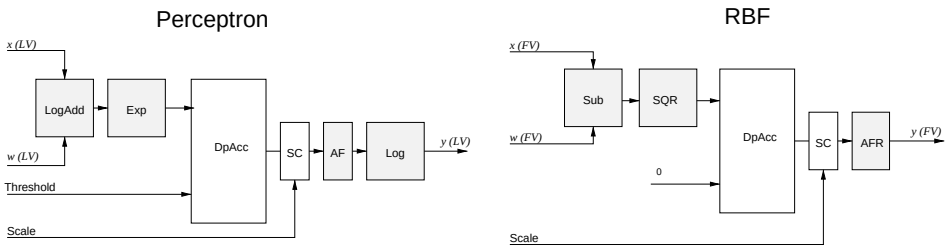

Fig. 9 *Perceptron and RBF processing element structure.*

Perceptron type PE

This processing element accepts the weights and inputs in their logarithm form (*logval, LV*). Each pair of input and weight is added by the LogAdd block and the result is passed through the EXP block to the dot product accumulator (DpAcc). DpAcc accumulates products, $w_i x_i$. After processing all inputs, the dot product value is passed through a chain of post-processing blocks. This chain consists of the scale (SC), activation function (AF) and logarithmer (Log) block. The logarithmer produces the logarithm of the output to ensure compatibility to other processing elements.

The processing rate of this block depends on the adder and barrel shifter propagation delay in the LogAdd, Exp and DpAcc blocks. If we use a fast conditional-sum adder for the LogAdd, the propagation delay can be a few nanosecond. The dot product adder should be implemented as a carry save adder to reach sufficiently short propagation delay for the 24 or the 32 bits wide dot product.

RBF type PE

The RBF processing element is slightly different. Its block diagram is also shown in Fig. 9. The Sub and SQR blocks are used to calculate the Euclidean distance between the input and the weight vectors. The gauss-like activation function, $G(x)$, is implemented by the AFR block. The DpAcc block is the same as the DpAcc block of the perceptron type.

Both types of the processing element use the SC block. This block multiplies the dot-product by a value which determines the scale factor of weights. This is necessary because the weights are in the range of $(-1, 1)$, which is insufficient for most neural applications. The multiplier is not necessary because the set of powers of two sufficiently covers the usual scale factors.

6. ECX Card

The first implementation of the shift-add arithmetics was verified on a card called the ECX card. The ECX card was designed and produced in the shape of an ISA card for the PC compatible computers. The block diagram is shown in Fig. 10.

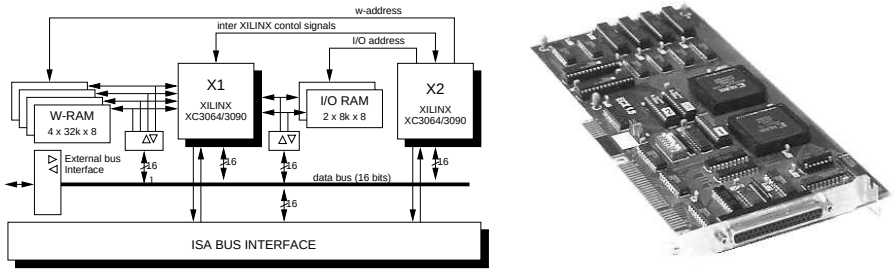

Fig. 10 *ECX block diagram and the board picture.*

The card contains two XILINX FPGAs of the XC3000 family. The first FPGA (X1) acts as a neural processing element and the second FPGA (X2) acts as a controller. The other circuits create an environment in which XILINX FPGAs behave. This environment consits of

- weight memory,
- input/output data memory,
- ISA bus interface,
- external data interface.

The weight memory is a fast static RAM of 32K×32 bits. All 32 bits are passed to X1 at once. Thus the interface ensures sufficient data throughput between the processing elements and the weight memory. The weight memory is mapped into the host computer memory address space, so that the weights can be directly loaded or modified by the host. The weight download and the modification are limited to sixteen-bit wide transactions.

The input/output memory is a fast static RAM of 8K words. Each word is only 16-bit wide. This memory uses separated data bus, so the weights and inputs can be sent to the X1 chip at once. No time multiplexing is required. The input/output memory is also mapped to the host computer memory address space.

The X2 chip controls weight and input/output memory addressing. Its next task is to synchronize access to both memories with host computer to avoid access conflicts. Synchronization proceeds by interrupt that inform the host computer that the access into the memories is allowed.

The ISA bus interface ensures all data transfers between the host computer and the card. This interface establishes mapping of the card memories into the memory address space of the host computer. Further, it provides I/O port for downloading the configuration data into FPGAs. This interface also provides signals to create four control registers as a part of the X2 chip design. These registers are visible in the host I/O space. A definition of these registers is not strictly given, it only depends on X2 chip configuration. We use these registers to control the neural net evaluation.

The external bus interface is provided for the direct connection with an application by a simple asynchronous protocol. This interface consists of the bidirectional bus drivers on the data path and control signals for handshaking. The protocol can be customized by the X2 chip configuration. The external bus enables to bypass the host computer with the aim to speed up the data transfer between the application and the card.

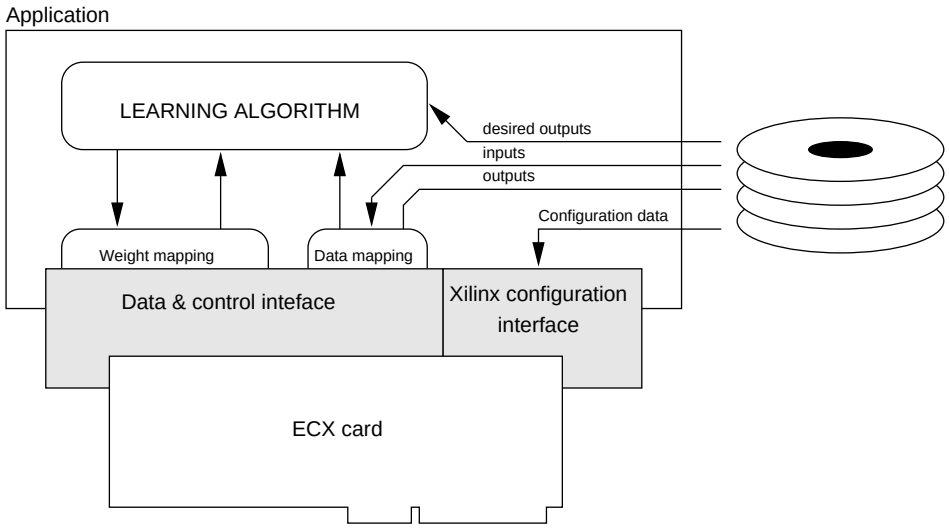

Fig. 11 *ECX software structure.*

6.1 Host software

The ECX card is controlled by host software. The host software consists of low level (driver like software) and application software. The low level software provides the

FPGA download function and the application interface. FPGA configuration files are read by the ECX software and serialized to a bit stream that is downloaded to X1 and X2 chips. This operation is performed at power up or always when reconfiguration of FPGAs is required.

The application interface establishes the data connection between the application and the card memories. Low level software must control the memory mapping because three pages of the card memories are mapped into the single 64kB window of the host memory address space.

The application software for the ECX card provides learning algorithm, learning data access, storing results, user interfaces and data presentation. The application software varies according to a neural-net paradigm being tested.

6.2 Neural net implementation on the ECX card

Neural net implementation on the ECX card consists of a neural unit and a controller. The neural unit implements all neural calculations while the controller maps the weights, inputs and outputs to the neural unit according to the required topology.

The neural unit can contain one or more neural processing elements working in parallel. The number of the processing elements is constrained by the X1 chip capacity and the bandwidth of the weight memory bus. Two 16-bit wide weights or four 8-bit wide weights can all be passed at once.

The controller is implemented in the X2 chip. The X2 chip addresses weight, input and output memories and ensures data transfers. The structure of the controller depends on network topology. Even more topologies can be supported. Since neural nets usually have more neurons than the processing elements available, time-sharing technique is used. The physically implemented processing element are applied for different parts of the network. This can be easily realized by various memory addressing strategies.

The layered, feed-forwarded topology was implemented for our experiments. The controller contains the layer, neuron, input and output counters. The memory addressing is derived from the content of these counters. The number of neurons per layer is variable and controlled by the control registers.

Tab. II summarizes the delay and complexity of the implemented perceptron-like PE. The first part of the table summarizes the building blocks of the PE. The second part summarizes the PE implementation. The delay of PE is separated to weighting, i.e. the sum of $w_i x_i$ products, and activation function evaluation. The processing rate is given by the delay of weighting. Our results correspond to 3.5 MCPS (connections-per-second) processing rate.

The latest version of the ECX card has been modified for the XILINX XC4010 FPGAs (XC4000 series). This version can implement more complex processing elements and operate faster. BP16, BP8, BP8D, RBF16, RBF8, RBF8D processing elements were tested on a modified card. BP or RBF in the name denotes the processing element type (perceptron or RBF). The number 16 or 8 is weight and input/output data width. D denotes two neurons working in parallel. FPGA capacity usage for all processing elements is shown in Tab. III.

Function	Delay	CLBs
2^x (16 bits)	58	33
$\log_2 x$ (16 bits)	83	35
multiplication(add-exp, 16 bits)	148	47
$\sum_i a_i$ (24 bits)	160	56
activation function f, 16 bits)	49	30
PE(perceptron)	N/A	198
PE ($\varphi_t = \varphi_{t-1} + w_i x_i$)	295	N/A
PE ($f(\varphi)$)	169	N/A

Tab. II *PE's components implemented on XC3090.*

	Max	BP8	BP8D	BP16	RBF8	RBF8D	RBF16
CLB	400	239	368	365	211	332	308
I/O cells	61	57	57	57	57	57	57
F&G generators	800	246	461	426	201	371	338
H generators	400	37	58	43	23	30	41
Flip-flops	800	130	175	160	124	165	154
CLB fast carry logic	400	18	36	28	14	28	24

Tab. III *Processing elements on XC4010EPC84-4.*

The neural network implemented on the ECX card was tested under the same software as it was used in software tests. The recall phase of the network was replaced by the interface to the ECX card, the other parts including learning stayed unchanged. The XOR problem, the inside-outside test, and the sonar test were used to verify the card functionality. Fig. 12 shows the learning error curves of hardware implementing the eight-bit and sixteen-bit version of the processing element. The eight-bit version has lower weight resolution; therefore it needs a longer learning period complicated task as XOR.

The recall delay of the ECX card was measured on the neural net with 60 inputs, 10 outputs and 140 hidden neurons. Average delay per input vector was calculated after processing 500 vectors. The results are summarized in Tab. IV. The table also contains the recall delay of the software model (BP) running on PC with K5 processor at 166MHz.

Network	BP8D	BP8	BP16	BP (SW model)
60/140/10	1.76	2.52	2.52	27.56

Tab. IV *ECX card delay [ms] vs. the software model delay [ms].*

The ECX delays presented in Tab. IV include also certain overhead caused by data transfers and setup of the ECX card before starting the network evaluation.

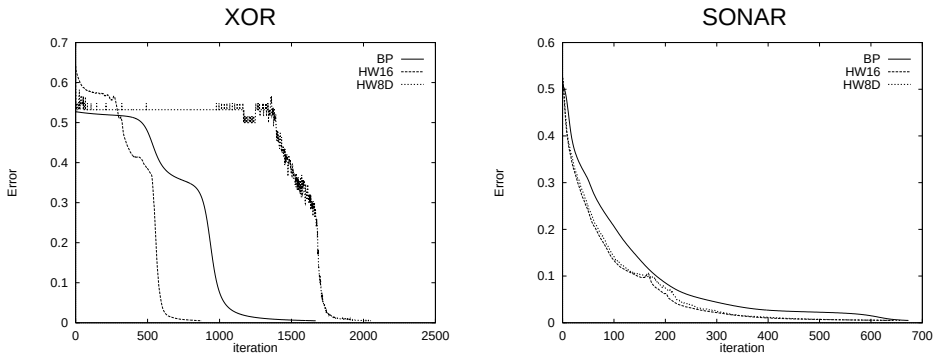

Fig. 12 *Learning curves measured for ECX card.*

7. On Chip Implementation

The fastest implementation we did was on-chip implementation. The processing elements introduced in Sect. 5.3 were designed on 0.7 μm Standard Cell Technology by the CADENCE design system with ES2 standard cell libraries. All function blocks were designed and verified by simulation. For each block a chip layout has been created to measure the required chip area and the propagation delay. The results are summarized in Tab. V. The table contains a list of functional blocks. The first group of the blocks includes all basic and complex blocks. This group contains two LogAdder blocks, a faster and a slower versions. The slower version is a carry-ripple adder and the faster version is a conditional-sum adder. The second group shows various multiplier configurations. For comparison purposes a precise 16×16 multiplier was also added to this group.

For each block, we present the required chip area, the typical and the maximum propagation delay. The delay for the whole shift-add processing element is split into two delays. One is for product calculation (from input to DpAdder), and the other is for the activation function evaluation (from DpAdder to output).

The standard implementation of the processing element requires about 2.5 mm^2. Most of the chip area is covered by the multiplier and the RAM implementing activation function. The processing rate is given by the sum of the multiplier delay and the DpAcc block delay. This delay is about 37 ns. The activation function delay is about 14 ns (RAM propagation delay).

A chip area of about 1.4 mm^2 is required for the shift-add neural processing elements. The delay affecting the processing rate is 18 ns. The shift-add activation function evaluation is about 4.3 ns.

From these results we can see that the shift add architecture provides a faster and simpler processing element than any other commonly used. The results presented here strongly depend on the technology used. More progressive technologies (0.35 μm for example) will allow to design faster shift-add blocks than we present. However, if we compare the standard and the shift-add implementation on a given technology, the advantages of the shift-add arithmetics are obvious.

Block	Ar	Dt	Dm
LogAdd(slow)	0.083	7.2	14.2
LogAdd (fast)	0.234	3.8	7.4
EXP	0.106	2.3	4.7
LOG	0.250	3.3	6.5
AF	0.081	2.1	4.2
AFR	0.086	2.1	4.3
SC	0.083	2.2	4.4
DP Adder (24 bit)	0.472	3.9	6.8
SQR	0.104	4.4	8.6
SQRT	0.117	4.8	9.4
MUL (16×16)	0.946	n/a	30.2
MUL(2× Log-Add(s)-Exp)	0.689	12.8	25.4
MUL(2× Log-Add(f)-Exp)	0.840	9.5	18.6
MUL(Add-Exp) fast	0.345	6.1	12.1
MUL(Add-Exp) slow	0.193	9.5	18.9
SH-A proc. elem. (PER)	1.44	9.3	18.0
		11.7	23.3
SH-A proc. elem. (RBF)	1.29	16.4	32.4
		4.4	8.9

Tab. V *Summary of delays and chip areas for proposed neural blocks. Ar is a chip area in (mm^2), Dt is a typical delay per block and Dt is the maximum delay per block.*

8. Conclusion

The shift-add arithmetics is the background for a fast neural implementation. The shift-add arithmetics provides a number of simple functions required by perceptron and RBF neurons. The great advantage is that these functions can be easily implemented in hardware. Most functions are based on the barrel shifter principle.

Since the functions use linear approximation, they do not produce precise results. Software tests show that errors can be easily suppressed by the learning algorithm. By comparing the various models learned under the back propagation algorithm, we found a very small negative influence of the imprecise multiplication. On the other hand, the power of two based activation functions speed up the learning process.

First implementation of the shift-add architecture was tested on the XILINX FPGA based card. The tests show the functionality of the shift-add architecture in hardware. The nets implemented on the card were much faster than the software implementation.

Another implementation was made on a chip. The chip had not been fabricated yet, but a chip design provided a lot of valuable results. The simulation provided an area necessary to implement the chip and propagation delays of all functional blocks. From the result we can see that it is possible to obtain faster and area-saving neural blocks. The results reached strongly depend on the technology and

the design, so a better result can be obtained for optimized technology libraries. Better results can also be obtained on technologies with an optimized multiplexor because our blocks contain more than ninety percent of multiplexors.

Acknowledgment

The author of this article thanks all institutions and people who participated in this project. Namely to the Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague (Czech Republic) and the Faculty of Applied Physics TU Delft and the Faculty of Technical Mathematics and Informatics TU Delft (The Netherlands). The author also thanks all his students who participated in this project. Namely to Jaroslav Maurenc, Jan Hvozdovič, Karel Sršeň and Jakub Adamczyk.

This research was supported by the internal grant of the Czech Technical University in Prague, no.3096374/1997.

References

[1] Marchesi M., Orlandi G., Piazza F.: Fast neural networks without multipliers. IEEE Transaction on Neural Networks, **4**, 1, January 1993, 53-62.

[1] Elmasry M.I.: VLSI artificial neural networks engineering. Kluwer Academic Publishers, 1994, ISBN 0-7923-9493-3.

[2] Jabri M., Coggins R., Flower B.: Adaptive analog VLSI neural systems. Chapman & Hall, London 1996. ISBN 0-412-61630-0.

[3] Maurenc J.: Neurochip architecture. Final thesis, Czech Technical University in Prague, Prague, 1995.

[4] Hvozdovič J.: Neuronová síť na FPGA XILINX. Final thesis, Czech Technical University in Prague, Prague, 1995.

[5] Sršeň K.: Shift-add neuron na standardních buňkách. Final thesis, Czech Technical University in Prague, Prague, 1998.

[6] Adamczyk J.: Neuronový akcelerátor na X4010. Final thesis. Czech Technical University in Prague, Prague, 1999.

[7] Skrbek M., Šnorek M.: An architecture for an efficient implementation of neural networks. Proceedings of the ESM'95, M. Šnorek and M. Sujansky and A. Verbraeck (eds.), Prague, Czech Republic, 1995, 785-789.

[8] Šnorek M., Skrbek M.: Artificial neural network accelerators. Elektrotechnik und Infomationstechnik, Neuronale Netze, 7-8, 1995, 329-333.

[9] Skrbek M.: Experimental card for shift-add neural architecture verification. Workshop'96, CTU Prague and TU Brno, Prague, 1996, 295-296.

[10] Skrbek M.: New neurochip architecture. Ph.D. thesis. Czech Technical University in Prague, Prague, 1999.