Applied Soft Computing 29 (2015) 233–244

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

# Implementation of the RBF neural chip with the back-propagation algorithm for on-line learning

J.S. Kim[a], S. Jung[b,*]

[a] Samsung Electronics, Republic of Korea
[b] Intelligent Systems and Emotional Engineering (ISEE) Laboratory, Department of Mechatronics Engineering, Chungnam National University, Republic of Korea

ARTICLE INFO

*Article history:*
Received 18 March 2013
Received in revised form 14 August 2014
Accepted 15 December 2014
Available online 6 January 2015

*Keywords:*
Floating-point processor
RBF neural network
Back-propagation algorithm
FPGA
Nonlinear classifier

ABSTRACT

This article presents the hardware implementation of the floating-point processor (FPP) to develop the radial basis function (RBF) neural network for the general purpose of pattern recognition and nonlinear control. The floating-point processor is designed on a field programmable gate array (FPGA) chip to execute nonlinear functions required in the parallel calculation of the back-propagation algorithm. Internal weights of the RBF network are updated by the online learning back-propagation algorithm. The on-line learning process of the RBF chip is compared numerically with the results of the RBF neural network learning process written in the MATLAB program. The performance of the designed RBF neural chip is tested for the real-time pattern classification of the XOR logic. Performances are evaluated by comparing results from the MATLAB through extensive experimental studies.

## 1. Introduction

Intelligent electronics that process intelligent information are rapidly getting attraction of researchers in engineering and information fields. Not only software programming, but also intelligent hardware for intelligent information processing is increasingly demanded due to the development of hardware technology.

The word of 'intelligence' has been frequently used in the area of machine learning, soft computing, or artificial intelligence [1,2]. Nowadays, the concept of intelligence is used in almost every research fields including signal processing, control, mechatronics, robots, materials, transportation and even construction. Typically, in the areas of dealing with physical systems such as dynamical systems, intelligent electronics are severely demanded to improve the system performance.

A neural network mimicking a human brain is one of powerful and popular intelligent tools since it has learning capability, adaptation capability, and generalization capability. An ultimate goal of using neural network is to use those capabilities to improve the performance in various aspects.

Specially, in the area of dynamical systems, neural network has been used as nonlinear controllers for robot manipulators to minimize the tracking errors [3–6]. With the help of hardware technology, neural network and fuzzy logic controllers are designed on a DSP chip along with FPGA for controlling nonlinear dynamical systems [7,8]. Multilayered neural networks are implemented with particle swarm optimization algorithms on FPGA for identifying dynamical systems [9]. Fuzzy inference modules are also implemented on FPGA [10,11]. Interval type-2 fuzzy systems for a real plant are implemented on FPGA [12] and particle swarm optimization of interval type-2 fuzzy systems is implemented on FPGA [13].

Recently, a lot of attempts to design neural network hardware on a field programmable gate array (FPGA) chip as an intelligent electronics have been made [14–17]. In many cases of developing the neural network hardware, the forward propagation of neural processing is only designed on an FPGA chip for an offline learning and online control scheme although the complete neural network processing requires two stages: forward and backward propagation [18–21]. FPGA implementations on stochastic and adaptive control applications are used [22,23]. Designing the backward propagation process is relatively difficult since a lot of calculations are involved to update internal weights [24–26]. FPGA hardware is developed to implement on-line learning for shape recognition [27]. Multi-layered perceptron network (MLP) has been developed on FPGA with fixed point data notation and its performance is measured by hardware specifications such as the number of multiply–add operation and the number of weight updates per second [28]. The neural estimators were designed by high level programming language and downloaded on NI C-RIO using Labview software for speed estimation of two-mass drive system

* Corresponding author. Tel.: +82 42 821 6876; fax: +82 42 823 4919.
*E-mail address:* jungs@cnu.ac.kr (S. Jung).

http://dx.doi.org/10.1016/j.asoc.2014.12.018
1568-4946/© 2014 Elsevier B.V. All rights reserved.

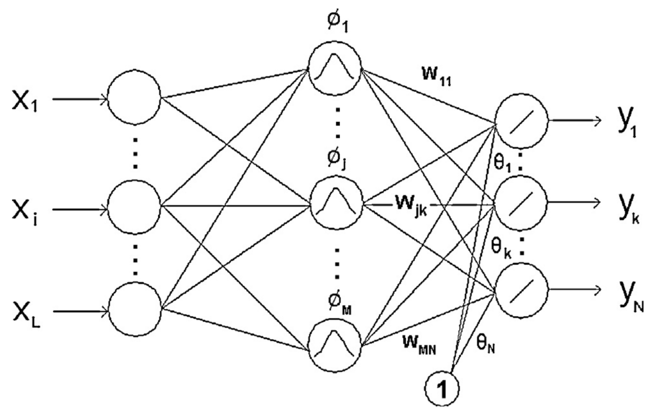


**Fig. 1.** The RBF network structure.

[29]. RBF neural network has been implemented in vision system for real-time face tracking and identity verification. Since the facial identification is focused, the detailed hardware implementation of RBF neural network was not elaborated [30]. Genetic algorithms are implemented on FPGA [31]. Design and implementation of a RBF neural network on FPGA with the detailed back-propagation algorithm for an on-line classifier or controller, of which learning can be done in on-line fashion, are relatively rare in the literature.

In this paper, therefore, a hardware design and implementation of the radial basis function (RBF) neural network (NN) by designing the floating-point processor (FPP) using the hardware description language (HDL) is presented. Due to its nonlinear characteristics of the network, it is very difficult to implement the neural hardware with integer-based operations to satisfy the acceptable accuracy. To calculate nonlinear functions required in the neural network processing such as sigmoid functions or exponential functions, floating-point operations are required.

Designing the floating-point processor allows us to implement nonlinear functions. Thus, the exponential function can be designed as the 32 bit single-precision floating-point format. In addition, to update weights in the network, the back-propagation algorithm can also be designed with the HDL and implemented in the FPGA hardware. Most operations are performed in the floating-point based arithmetic unit and accomplished sequentially by the order of instructions stored in a ROM.

Finally, the RBF neural network hardware is implemented by writing the assembly codes running on the floating-point processor. The RBF network is tested on the FPGA for nonlinear classifications. Extensive experiments are conducted by comparing possible calculation deviation errors due to the functional approximation between the designed chip and the MATLAB to evaluate the feasibility of using the neural network chip.

## 2. Radial basis function network

### 2.1. RBF network structure

The radial basis function (RBF) network is one of neural network structures that learn by measuring the Euclidean distance of data. The RBF network is simple in the structure that there are one nonlinear hidden layer and a linear output layer as shown in Fig. 1. There are no weights between the input layer and the hidden layer. The linearity of the output layer becomes an advantage of analyzing the stability of the closed loop system in association with feedback controllers. This simple structure turns out to be advantages of RBF network over the multilayered perceptron (MLP) network for dynamical systems [32].

Each neuron at the hidden layer has a Gaussian function described as

$$\phi_j(x) = \exp\left(-\frac{\sum_{i=1}^{L}(x_i-\mu_j)^2}{2\sigma_j^2}\right) \tag{1}$$

where $x_i$ is the $i$th input element, $\mu_j$ is the mean value of the $j$th hidden unit, $\sigma_j$ is the covariance of the $j$th hidden unit, and $L$ is the number of input elements.

The output layer is linear so that the $k$th output is an affine function that can be obtained as

$$y_k = \sum_{j=1}^{M}\phi_j w_{jk} + \theta_k \tag{2}$$

where $\phi_j$ is the $j$th output of the hidden layer in (1) and $w_{jk}$ is the weight between the $j$th hidden unit and the $k$th output, $\theta_k$ is the bias weight of the $k$th output, and $M$ is the number of the hidden units.

### 2.2. Back-propagation learning algorithm

To train the network shown in Fig. 1, the back-propagation algorithm is derived to update weights between the hidden layer and the output layer, and nonlinear function parameters of the hidden units. The output error for the $k$th output is defined as

$$e_k = yd_k - y_k \tag{3}$$

where $yd_k$ is the desired output value. The output error is propagated backward to adjust internal weight values, which is known as the back-propagation algorithm.

Then, the objective function is defined to minimize the output errors as

$$E = \frac{1}{2}\sum_{k=1}^{N} e_k^2 \tag{4}$$

where $N$ is the number of the output units.

The back-propagation algorithm searches the minimum value by calculating the gradient of Eq. (4) with respect to the weight. The gradient of Eq. (4) is calculated as

$$\Delta w = -\eta\frac{\partial E}{\partial w} \tag{5}$$

where $w$ can be weights or Gaussian parameters and $\eta$ is the learning rate. Substituting (4) into (5) and applying the chain rule yields

$$\Delta w = -\eta\frac{(1/2)\partial(e_1^2+\cdots+e_k^2+\cdots e_N^2)}{\partial e_k}\frac{\partial e_k}{\partial w} = -\eta e_k\frac{\partial e_k}{\partial w} \tag{6}$$

Therefore, the function $\partial e_k/\partial w$ needs to be obtained for each internal variable, $(w_{jk}, \theta_k, \mu_j, \sigma_j) \in w$ of the RBF network. One typical example for the weight $w_{jk}$, we have

$$\begin{aligned}\Delta w_{jk} &= -\eta_w e_k\frac{\partial e_k}{\partial w_{jk}} = -\eta_w e_k\frac{\partial(yd_k-y_k)}{\partial y_k}\frac{\partial y_k}{\partial w_{jk}} = \eta_w e_k\frac{\partial y_k}{\partial w_{jk}} \\ &= \eta_w e_k \phi_j\end{aligned} \tag{7}$$

where $\eta_w$ is the learning rate.

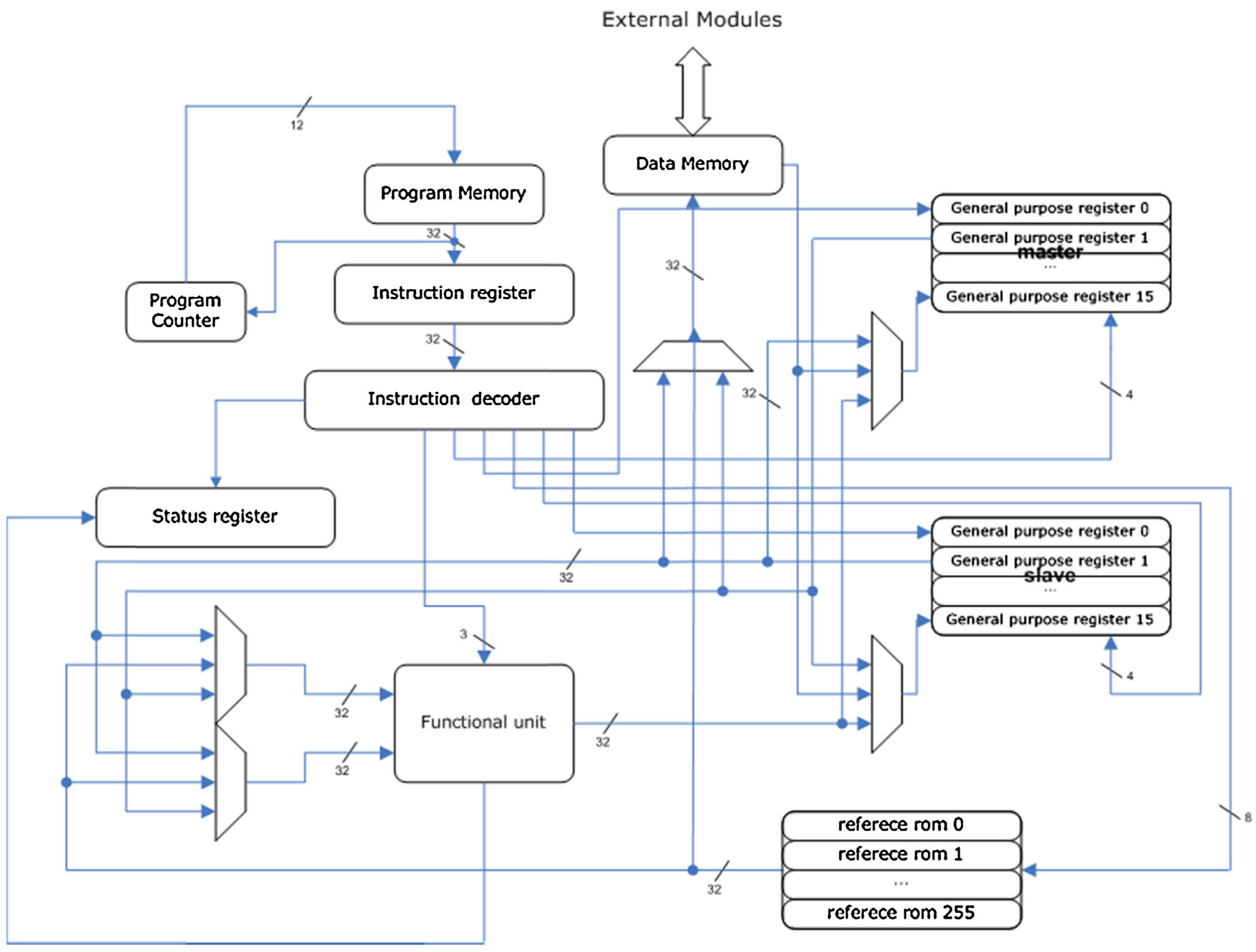


**Fig. 2.** The architecture of the floating-point processor (FPP).

In the same manners, we can have the detailed update equations as

$$\Delta w_{jk} = \eta_w e_k \phi_j,$$

$$\Delta \theta_k = \eta_\theta e_k,$$

$$\Delta \mu_j = \eta_\mu \phi_j \sum_{i=1}^{L} \frac{(x_i - \mu_j)}{\sigma_j^2} \sum_{k=1}^{N} e_k w_{jk}, \tag{8}$$

$$\Delta \sigma_j = \eta_\sigma \phi_j \sum_{i=1}^{L} \frac{(x_i - \mu_j)^2}{\sigma_j^3} \sum_{k=1}^{N} e_k w_{jk},$$

where $\eta_w, \eta_\theta, \eta_\mu, \eta_\sigma$ are learning rates. Finally, all weights are updated recursively.

$$w(t+1) = w(t) + \Delta w \tag{9}$$

## 3. Design of a processor based on floating-point format

### 3.1. Nonlinear function representation

To implement the RBF network hardware, Eqs. (1), (2), (8), and (9) are the main algorithms. Eqs. (1) and (2) are implemented for the forward propagation, (8) and (9) for the back-propagation process. In the forward propagation, the Gaussian function has to be represented in the hardware description language (HDL).

Unfortunately, there is no property of representing nonlinear functions in the HDL. Thus, the numerical approximation procedure of expressing the Gaussian function is required.

The Gaussian function for the real value $x$ is given by

$$f(x) = e^{-((x-\mu)^2/2\sigma^2)} \tag{10}$$

One of approximating methods is to use the Taylor series expansion. For example, the exponential function can be expressed by the Taylor series as follows.

$$e^{-x} = 1 - x + \frac{1}{2!}x^2 - \frac{1}{3!}x^3 + \frac{1}{4!}x^4 - \frac{1}{5!}x^5 + \frac{1}{6!}x^6 - \frac{1}{7!}x^7 + \frac{1}{8!}x^8 + \cdots + \mathrm{HOT} \tag{11}$$

where the high order terms (HOT) are ignored. In the programming with the HDL for an FPGA chip, we use the following format instead of (11) for simplicity.

$$e^{-x} = ((\cdots(\frac{1}{8}x - 1)\frac{1}{7}x + 1)\frac{1}{6}x - 1)\frac{1}{5}x + 1)\frac{1}{4}x - 1)\frac{1}{3}x + 1)\frac{1}{2}x - 1)x + 1 + \cdots + \mathrm{HOT} \tag{12}$$

Truncation of the HOT may yield the approximating error of nonlinear functions. Effects by the approximation process are analyzed and evaluated for possible applications in the later section.

### 3.2. Architecture of the processor

The floating-point based processor is designed. The processor can perform arithmetic calculations such as addition, subtraction, multiplication, and division. The architecture of the processor

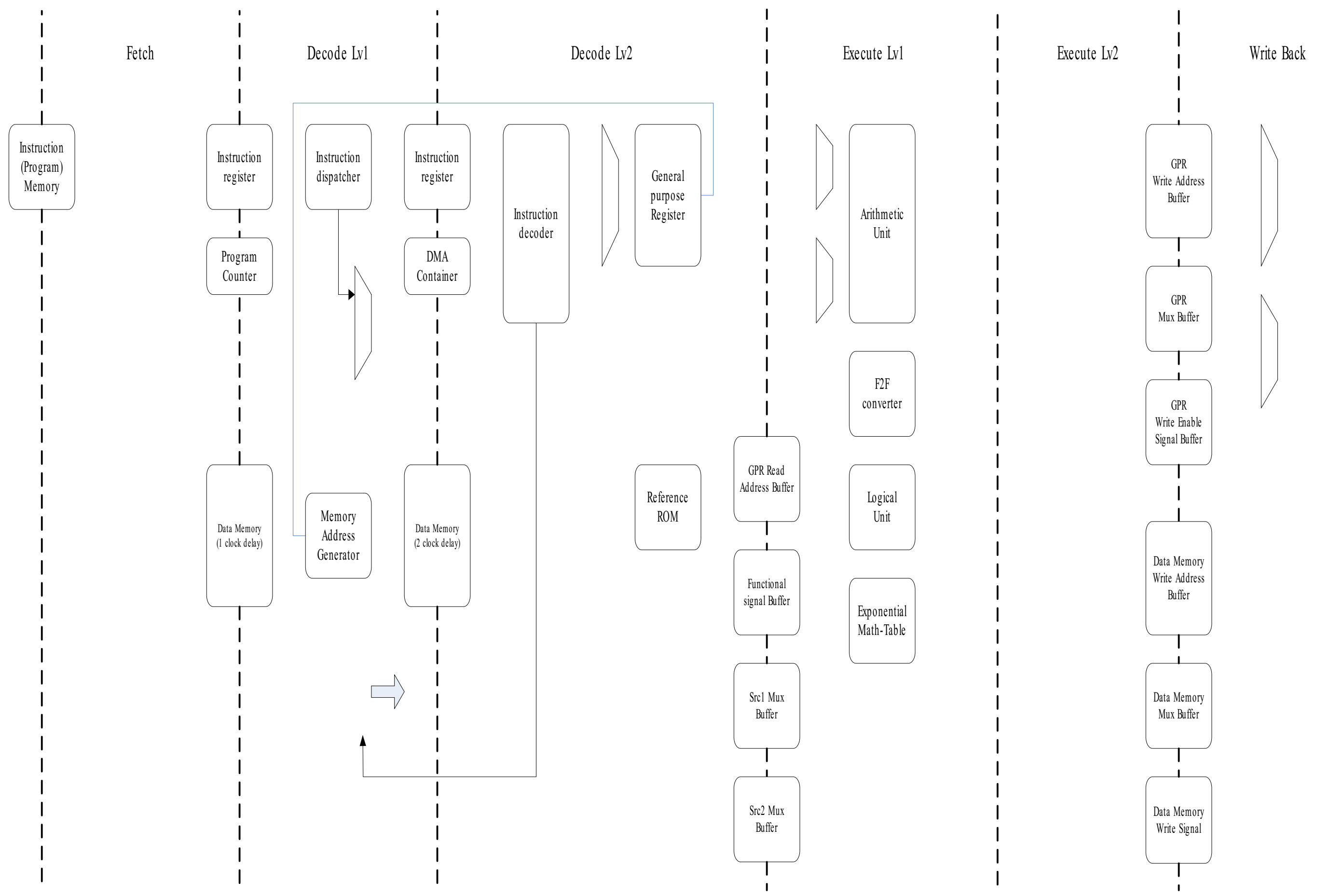


**Fig. 3.** The pipelining architecture of FPP.

follows the Harvard reduced instruction set computer (RISC) architecture. The overall structure of the processor is described in Fig. 2.

The processor has the basic 4 cycles of fetch, decode, execute, and write-back steps to execute an instruction. Each unit has the function as follows.

- Program memory: it is a ROM that stores instruction codes. Its word length is 32 bit long and 12 bit address can access 4096 instructions. Thus, the size of the program memory is 16 K bytes.
- Data memory: it is a RAM that stores data. Its word length is 32 bit long and the size is 16 K bytes.
- Program counter: it contains the address of the next instruction in the program memory. It refers to the status register to execute Jump or conditional branches.
- Instruction register: it stores the instruction code from the program memory for one cycle for decoding.
- Instruction decoder: a 32 bit instruction code is separated into an op-code and operands and then decoded into control signals.
- Status register: it states conditions of the functional unit after each calculation.
- Register group: it has the master group and the slave group. It can be used as a source for operand 1 and operand 2 for the functional unit. It has 32 registers and the register size is 16 bit.
- Reference ROM: it contains frequently used constants for calculations such as $\pi$, a constant for exponential function, and coefficients for the Taylor series expansion of sinusoidal functions and other nonlinear functions.
- Functional unit: it performs the arithmetic calculation based on a 32 bit single precision format such as addition, subtraction, multiplication, and division. It also performs logical operation as well. It has the converter for changing the floating-point format to the fixed-point format or vice versa.

**Table I**
Special cases of data format.

| Class | Exponent | Fraction |
|---|---|---|
| Zeros | 0 | 0 |
| De-normalized numbers | 0 | Non zero |
| Normalized numbers | 1–254 | Any |
| Infinities | 255 | 0 |
| NaN (not a number) | 255 | Non zero |

### 3.3. Instruction set

Instruction sets such as LD, MOV, ADD, SUB, MUL, DIV are defined as op code. The instruction set is 32 bit long composed of 7 bit op code, 24 bit operand as shown in Fig. 3. Assembly codes by the sequence of instructions can represent neural network algorithms.

To save the time for the instruction, instruction is executed by the pipelining process of fetch, decode, execute and write-back as shown in Fig. 3.

### 3.4. Numerical operations

The data is a 32 bit single floating-point based on IEEE format. Table I shows the special cases of number representations.

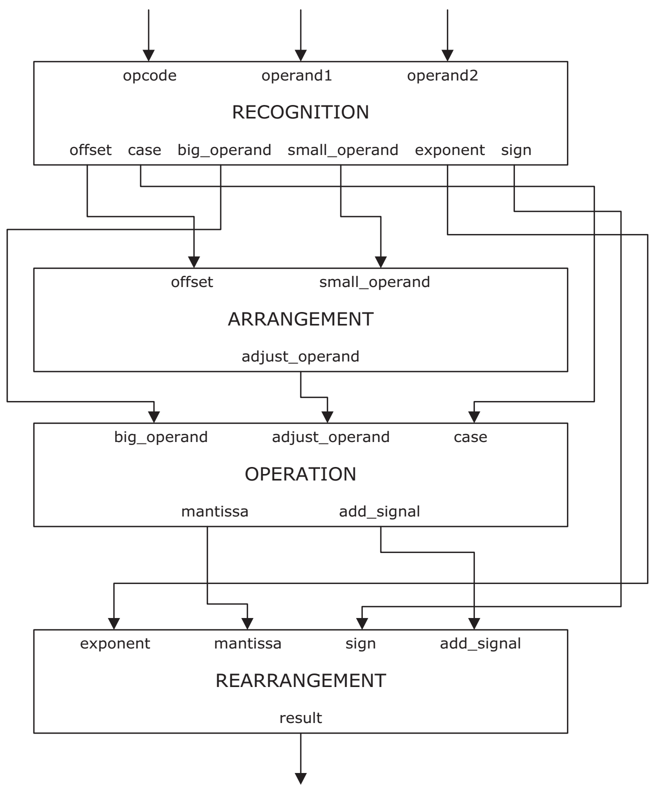


**Fig. 4.** Addition and subtraction process.

### 3.4.1. *Addition/subtraction*

Addition/subtraction takes 4 steps of recognition, arrangement, operation, and rearrangement as shown in Fig. 4.

The recognition module decides the case depending upon conditions of operand, exponent, and mantissa as listed in Table II. The arrangement module adjusts the number of bits and operation module executes addition/subtraction. The rearrangement module modifies the exponent and sends it out. All of processes are completed within one clock cycle.

**Table II**
Cases of addition/subtraction.

| Case | Case check bit select [3.0] | Comment |
|---|---|---|
| 1 | 0001 | Sign of each operand is same.<br>Values of exponent and mantissa are different. |
| 2 | 0010 | Sign of each operand is same.<br>Value of exponent is same.<br>Value of mantissa is different. |
| 3 | 0011 | Sign of each operand is same.<br>Values of exponent and mantissa are same. |
| 4 | 1001 | Sign of each operand is different.<br>Exponent of operand 1 is greater. |
| 5 | 1010 | Sign of each operand is different.<br>Exponent of operand 2 is greater. |
| 6–1 | 1011 | Sign of each operand is different.<br>Value of exponent is same.<br>Exponent of operand 1 is greater. |
| 6–2 | 1100 | Sign of each operand is different.<br>Value of exponent is same.<br>Exponent of operand 1 is greater. |
| 6–3 | 1101 | Sign of each operand is different.<br>Value of exponent is same.<br>Value of mantissa is same. |

### 3.4.2. *Multiplication*

The multiplication module is relatively simple. Multiplication can be done by an exponent part and a mantissa part separately. Signs are determined by XOR operation, exponents by addition, and mantissa by regular multiplication. Conditions such as zero, overflow, underflow, de-normalized error are also checked during operation. Fig. 5 shows the multiplication block module.

### 3.4.3. *Division*

In the same way of multiplication, each part is calculated separately as in Fig. 6. The exponent part can be modified with respect to the result of mantissa operation. This can be adjusted by considering several cases of operands. The flow of the mantissa division process is shown in Fig. 7. Two mantissa values are compared under different cases.

### 3.4.4. *Format converter*

The data format needs to be modified when different format data such as integer-based data to process data internally are obtained from the external devices. Fig. 8 shows the format converter that converts integer to single precision and single precision to integer.

## 4. Radial basis function network design

### 4.1. *Process of designing RBF network*

Using the floating-point processor as a core unit, the RBF network can be designed. The flow chart of processing the RBF network is shown in Fig. 9. The procedure can be divided into three parts, initialization, forward calculation, and backward calculation. This iterative procedure can be repeated until the error converges to satisfy the specified tolerance.

### 4.2. *Programming RBF network*

The block diagram of the RBF network design is shown in Fig. 10 and the detailed design by HDL is shown in Fig. 11. It consists of the floating-point processor, memories, a boot module, and a status controller.

The process of the RBF network can be programmed by the instruction designed in the floating-point processor. The assembly codes are written for calculating the forward process and the backward process of the RBF network. Fig. 12(a) shows the program for the forward process of calculating Eq. (2) and the outputs are calculated from the inputs via the hidden layer. Each operation of calculating $\phi_j$ is shown in Fig. 12(a). Then the output $y_k$ is calculated and compared with the desired value to yield the error. Next is the back-propagation algorithm. The error is used for the backward calculation. Four different parameters are updated, weights between the hidden and the output layer, the bias weight of the output layer, the mean value of the hidden units, and the covariance values of the hidden units. The assembly codes for updating weights are presented in Fig. 12(b).

### 4.3. *FPGA programming of RBF network*

The hardware design of the RBF network is described in Fig. 11. It is composed of a ROM for booting, a ROM for storing instructions, a data memory, a core processor, a control unit, and a communication module. The communication module sends resultant data to the external device after calculation by other modules.

- A: Boot_ROM: initialize and load parameters of RBF network such as learning rates, weights to the data memory.

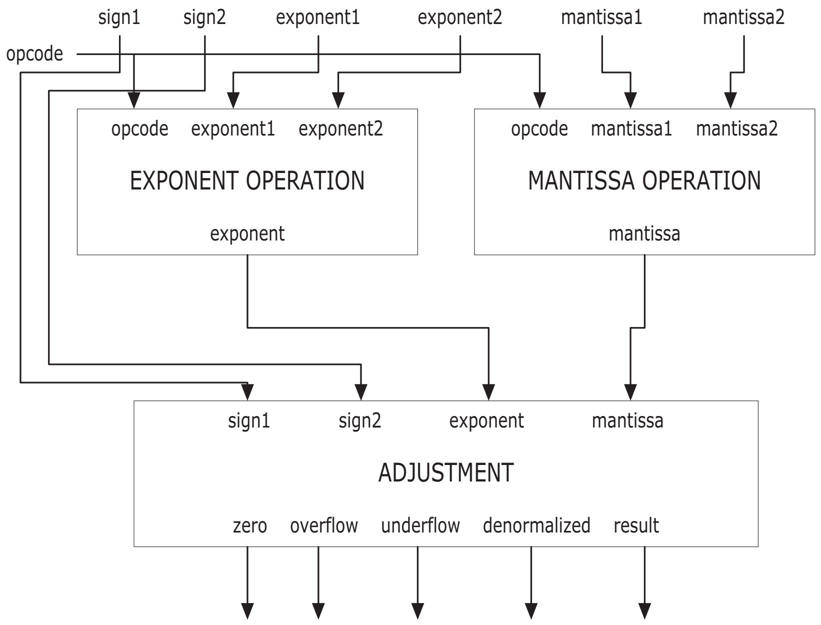


**Fig. 5.** Multiplication process.

- B: Instruction_ROM: holds instructions for calculating the RBF network. It has a size of 32 bit 16 K byte words.
- C: data memory: stores resultants after calculating the RBF network. Parameters such as $\mu$ or $\sigma$ are stored. It has a 16 K byte memory.
- D: core module: calculates actual processes of the RBF network after 4 step pipelined structure depending upon instructions.
- E: control module: once the calculation of the RBF network is done, the value of the objective function is sent to the PC through serial communication. It controls where to send the data by specified address.
- Others: multiplexers: synchronize the data or signals if necessary.

## 5. XOR logic classification experiment

### 5.1. Experimental setup

The performance of the designed neural chip is tested on Cyclone II EP2C70F672C8 FPGA board, which has 300,000 gates. The classification of the XOR logic is performed by the neural chip. The XOR logic is known to be the prototype example of the nonlinear classification example determining the performance of neural networks. The RBF network designed as in Fig. 11 is embedded into this FPGA chip.

Initial weights and input patterns are given in Table III. Although the number of RBF units and good initial center values can be easily

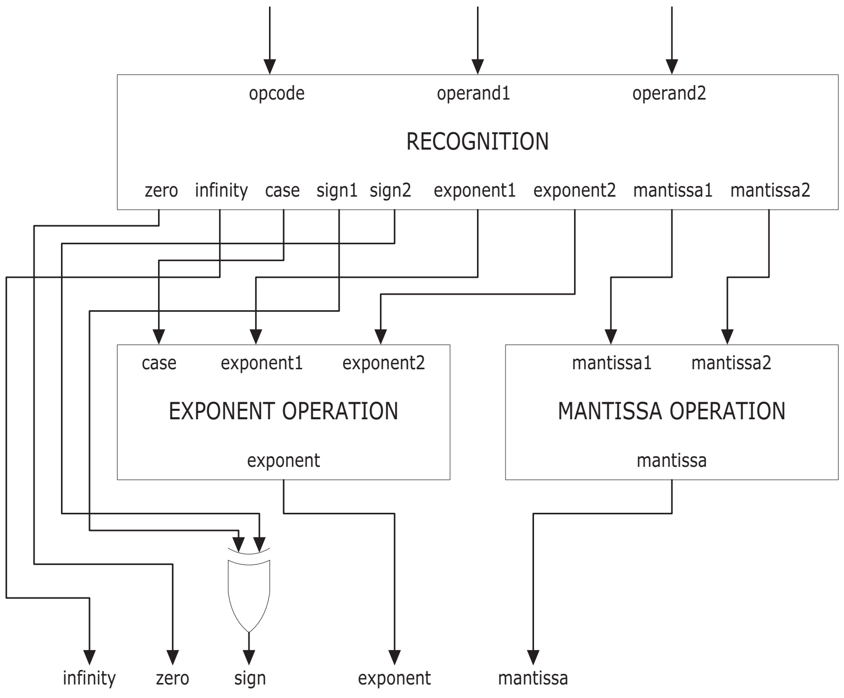


**Fig. 6.** Division process.

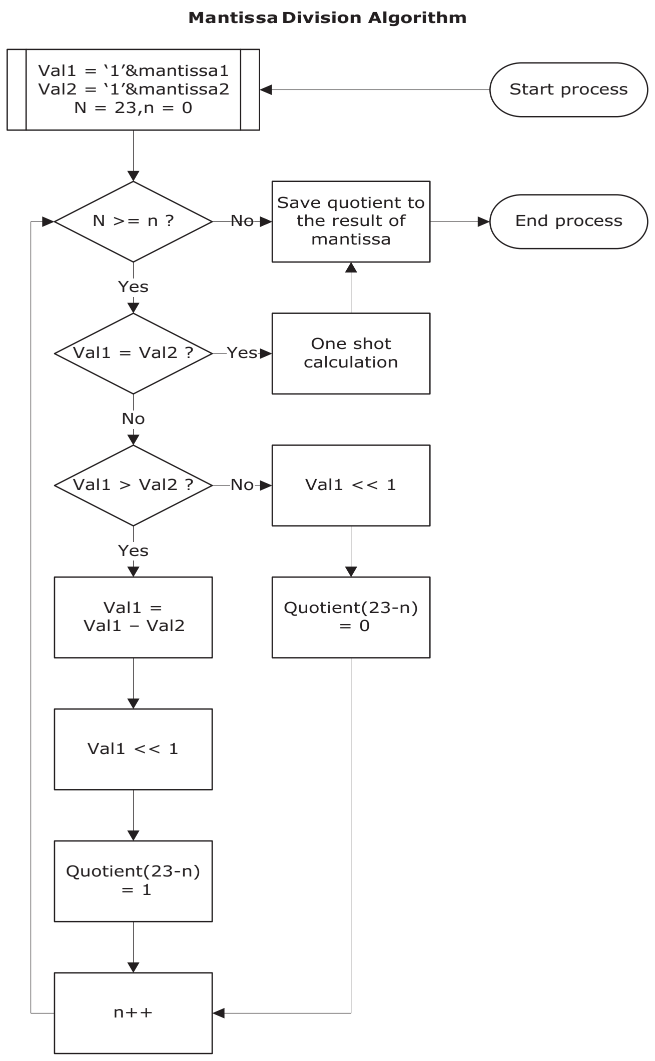


**Fig. 7.** Mantissa division process.

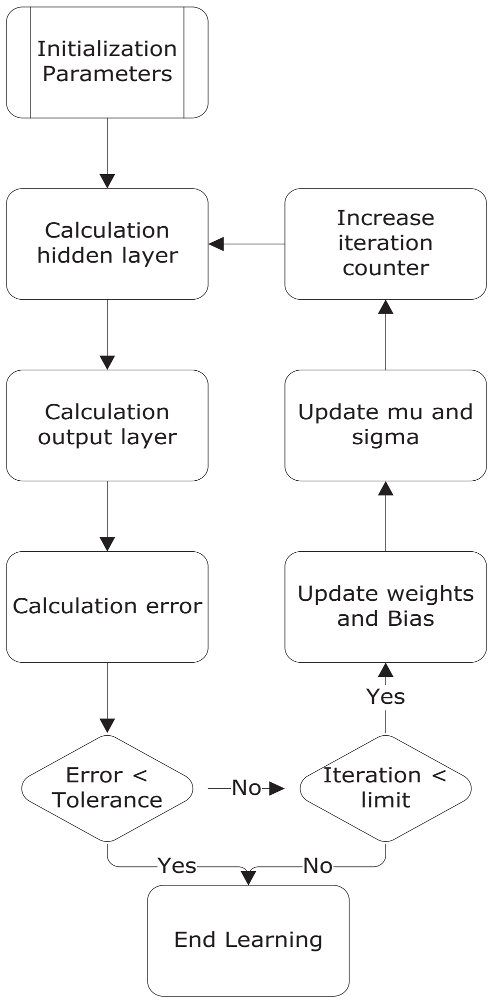


**Fig. 9.** Flow chart of RBF network process.

found, here nominal values are used. To examine the numerical accuracy of the neural chip, the same initial conditions are specified for the neural chip and the MATLAB simulation. The 50th order of the Taylor series approximation is used.

### 5.2. Classification results

Learning takes about 650 iterations until the error converges as shown in Fig. 13. Fig. 13 shows the comparison plots of the error convergence between the neural chip and the MATLAB simulation. Under the same conditions, the RBF network is tested for the neural chip and the MATLAB. Fig. 13 shows the similarity of the error convergence with a small deviation when the error starts converging at around the 650th iteration. The difference is minimal and can be improved by allowing the high order of the Taylor series since

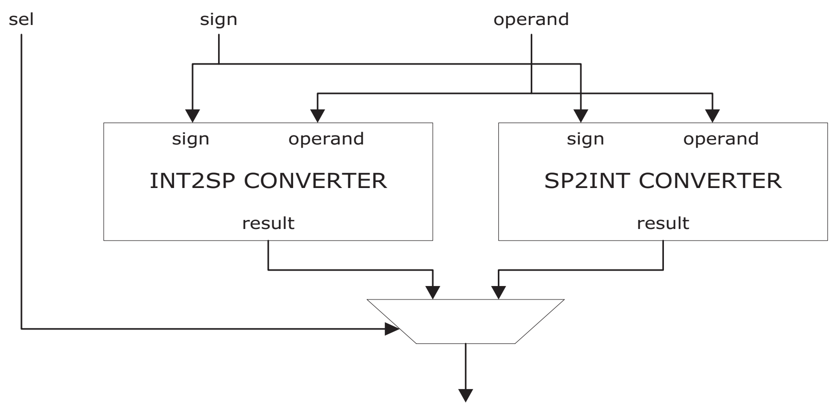


**Fig. 8.** Format converter.

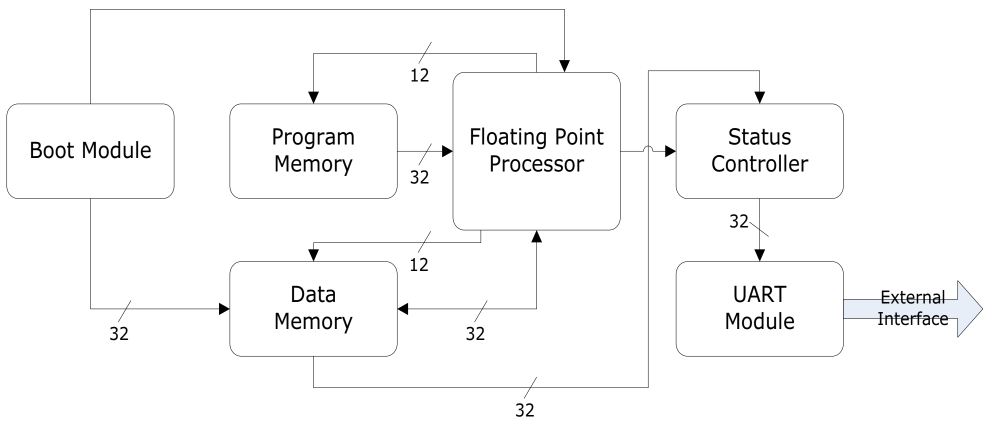


**Fig. 10.** RBF network block diagram.

**Table III**
Initial conditions for the RBF network.

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| X00 | 0.1 | Desired2 | 0.9 |
| X10 | 0.1 | Desired3 | 0.1 |
| X01 | 0.9 | Output weight0, $w_0$ | 0.1 |
| X11 | 0.1 | Output weight1, $w_1$ | 0.1 |
| X02 | 0.1 | Bias, $\theta$ | 1 |
| X12 | 0.9 | $\mu_0$ | 0.5 |
| X03 | 0.9 | $\mu_1$ | 0.5 |
| X13 | 0.9 | $\sigma_0$ | 0.5 |
| Desired0 | 0.1 | $\sigma_1$ | 0.5 |
| Desired1 | 0.9 | Tolerance | 0.001 |

**Table IV**
Final results tested after learning.

| X0 | X1 | Y | Output | Error |
|---|---|---|---|---|
| 0.1 | 0.1 | 0.1 | 0.100996 | −0.000996 |
| 0.1 | 0.9 | 0.9 | 0.898341 | 0.001659 |
| 0.9 | 0.1 | 0.9 | 0.898341 | 0.001659 |
| 0.9 | 0.9 | 0.1 | 0.101117 | −0.001117 |

**Table V**
Final parameter values of the RBF network.

| Parameters | Value |
|---|---|
| Weight0, $w_0$ | −0.8526668 |
| Weight1, $w_1$ | −0.84844798 |
| Bias, $\theta$ | 0.95275021 |
| $\mu_0$ | 0.11095808 |
| $\mu_1$ | 0.90131116 |
| $\sigma_0$ | 0.33398876 |
| $\sigma_1$ | 0.23129223 |

this deviation occurs due to the numerical approximation of the nonlinear functions.

The learning stops when the error satisfies the tolerance specified in Table III. After the learning process, all patterns are tested for the classification. The resultant classification is given in Table IV. We clearly see that learning of the neural chip has been done correctly.

### 5.3. Classification analysis

After learning, the final parameter values are determined and given in Table V.

Based on the data from Table V, the output value of the hidden units with respect to each input pattern is calculated. Table VI summarizes the values.

**Table VI**
Outputs of hidden layer after learning.

| Input pattern | $\varphi_1(x)$ | $\varphi_2(x)$ |
|---|---|---|
| (0.1, 0.1) | 0.99892410 | 0.00000612 |
| (0.1, 0.9) | 0.06134785 | 0.00247532 |
| (0.9, 0.1) | 0.06134785 | 0.00247532 |
| (0.9, 0.9) | 0.00376761 | 0.99996786 |

**Table VII**
RMS errors for different order.

| Parameters ($N$) | RMS error |
|---|---|
| 10 | 0.406756 |
| 20 | 0.001010 |
| 30 | 0.002087 |
| 40 | 0.000035 |
| 50 | 0.000035 |

$$\varphi_1(x) = e^{-(\|x-\mu_0\|^2/2\sigma_0^2)} = e^{-(x-0.11095808/2(0.33398876)^2)},$$
$$\varphi_2(x) = e^{-(\|x-\mu_1\|^2/2\sigma_1^2)} = e^{-(\|x-0.90131116\|^2/2(0.23129223)^2)} \qquad (13)$$

Thus, we can draw the decision boundary for the two classes. Based on Eq. (13), the decision boundary can be drawn on the input space. We see from Fig. 14 that two classes are separated by the hyper plane generated by the RBF neural network.

## 6. Performance evaluation

### 6.1. Effects by approximation order

The effect of the numerical approximation by the Taylor series expansion is investigated. The error convergence of the neural hardware is tested for different orders of the Taylor series, and compared with that calculated by the MATLAB program under same initial conditions. The learning rates $\eta_w = 0.05$, $\eta_\theta = 0.05$, $\eta_\mu = 0.005$, $\eta_\sigma = 0.005$ are used in this case.

When the 10th order approximation is used, the network does not converge. Table VII shows the summary of the convergent error shown above for different orders of the Taylor–Maclaurin series. We see the error decrease as the order increases. Beyond the 40th order, the performance is compatible. So we decide to use the 40th order of the Taylor series expansion for the nonlinear function approximation.

```
LD    mr0    mem0     ; load x0            }
LD    mr1    mem1     ; load x1            }
LD    mr2    mem10    ; load desired0      }
LD    sr0    mem17    ; load mu0           }
LD    sr1    mem18    ; load mu1           }  Load need
LD    sr2    mem19    ; load sigma0        }  parameters
LD    sr3    mem20    ; load sigma1        }
LD    sr4    mem26    ; load eta_weight    }
LD    sr5    mem27    ; load eta_bias      }
LD    sr6    mem28    ; load eta_mu        }
LD    sr7    mem29    ; load eta_sigma     }
LD    sr8    mem14    ; load wight0        }
LD    sr9    mem15    ; load wight1        }
LD    sr10   mem16    ; load bias          }

SUB   mr3    mr0    sr0     ; x0 - mu0
MOV   sr11   mr3
MUL   mr3    mr3    sr11    ; (x0 - mu0)^2
SUB   mr4    mr1    sr0     ; x1 - mu0
MOV   sr11   mr4
MUL   sr11   mr4    sr11    ; (x1 - mu0)^2
ADD   mr3    mr3    sr11    ; (x0 - mu0)^2 + (x1 - mu0)^2
MOV   mr4    sr2
MUL   sr11   mr4    sr2
MUL   sr11   sr11   cr3     ; 2 * sigma0^2
DIV   mr3    mr3    sr11    ; || x - mu ||^2 / (2 * sigma0^2)
MUL   sr11   mr3    cr18    ; (1/8)*x
SUB   sr11   sr11   cr2     ; (1/8)*x - 1
MUL   sr11   sr11   cr17    ; ((1/8)*x - 1)*(1/7)
MUL   sr11   sr11   mr3     ; ((1/8)*x - 1)*(1/7)*x
ADD   sr11   sr11   cr2     ; ((1/8)*x - 1)*(1/7)*x + 1
MUL   sr11   sr11   cr16    ; (((1/8)*x - 1)*(1/7)*x + 1)*(1/6)
MUL   sr11   sr11   mr3     ; (((1/8)*x - 1)*(1/7)*x + 1)*(1/6)*x
SUB   sr11   sr11   cr2     ; ...
MUL   sr11   sr11   cr15    ; ...
MUL   sr11   sr11   mr3
ADD   sr11   sr11   cr2
MUL   sr11   sr11   cr14
MUL   sr11   sr11   mr3       Use Taylor-
SUB   sr11   sr11   cr2       Maclaurin series
MUL   sr11   sr11   cr13      (n=8)
MUL   sr11   sr11   mr3
ADD   sr11   sr11   cr2
MUL   sr11   sr11   cr12
MUL   sr11   sr11   mr3     ; ...
SUB   sr11   sr11   cr2     ; ...
MUL   sr11   sr11   mr3     ; ...
ADD   mr3    sr11   cr2     ; exp( -|| x - mu ||^2 / (2 * sigma0^2) )
                                                   Hidden
                                                   Layer
                                                   Calculation
```

(a) Forward Process of RBF Net

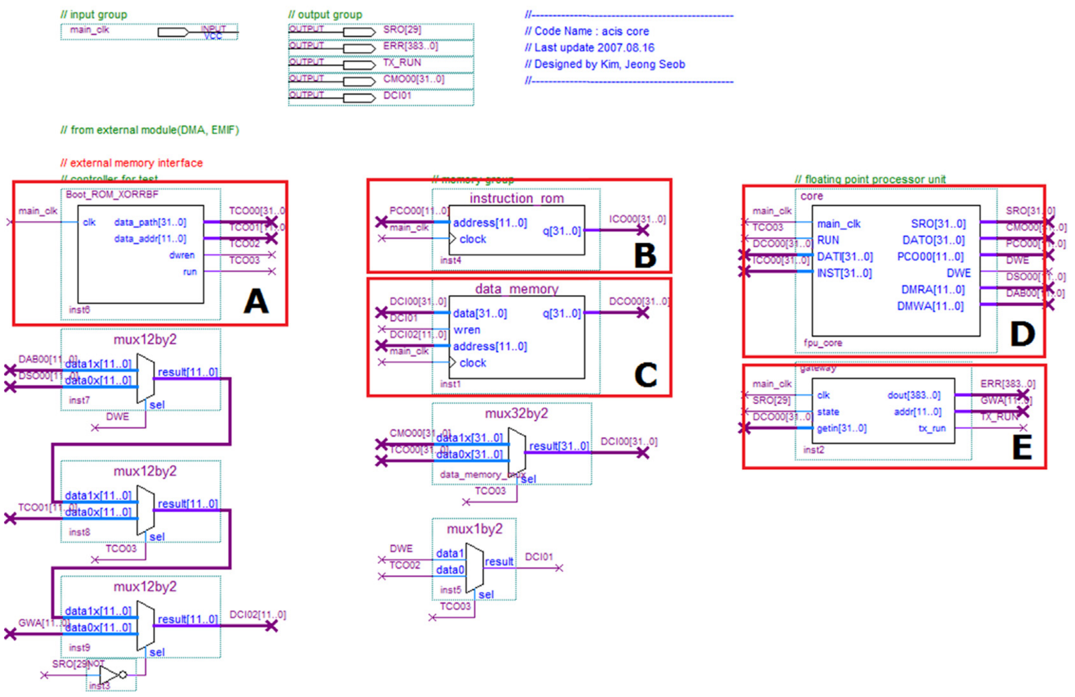
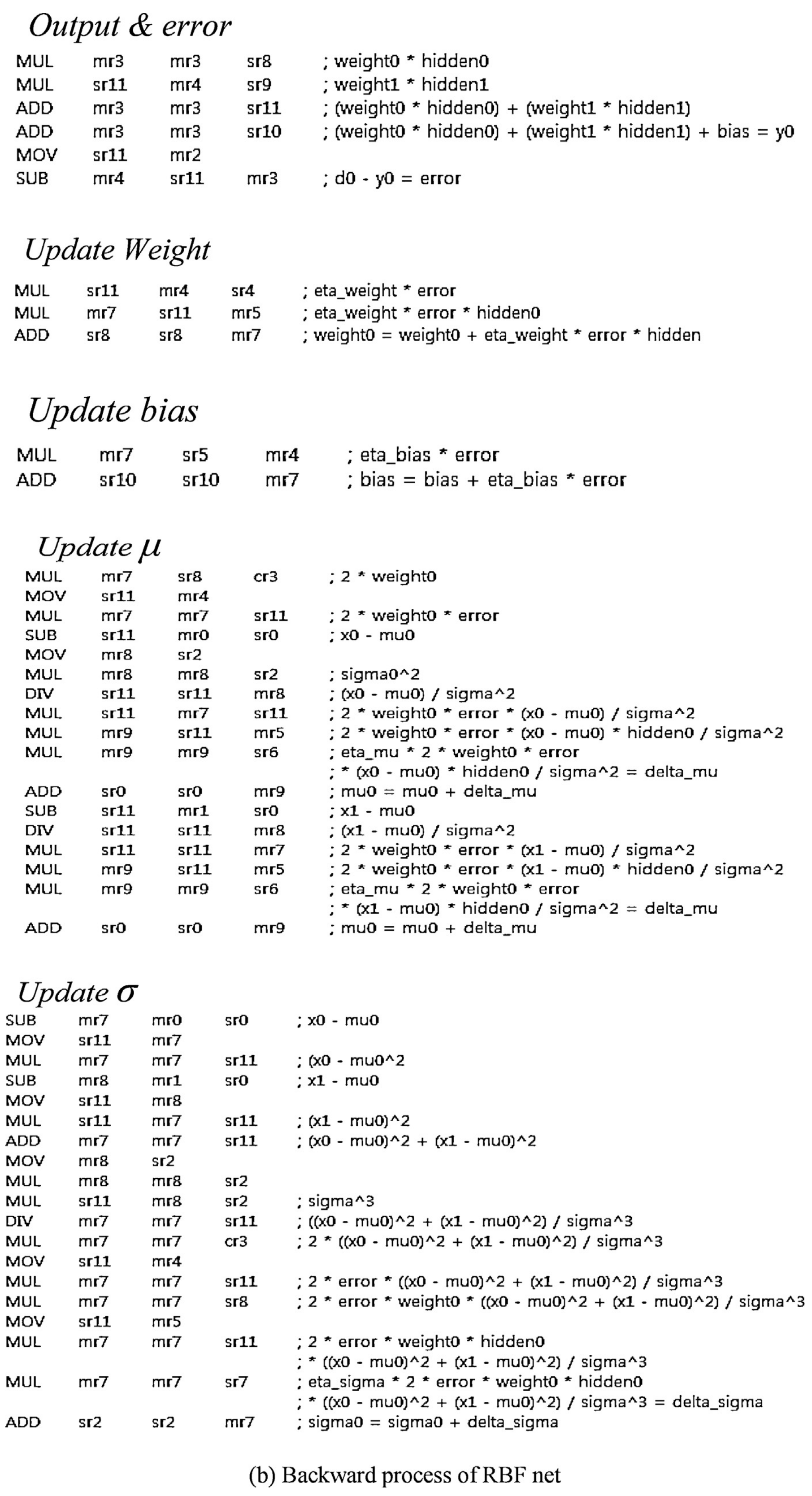


**Fig. 11.** RBF network design with HDL.



*Output & error*

```
MUL   mr3    mr3    sr8     ; weight0 * hidden0
MUL   sr11   mr4    sr9     ; weight1 * hidden1
ADD   mr3    mr3    sr11    ; (weight0 * hidden0) + (weight1 * hidden1)
ADD   mr3    mr3    sr10    ; (weight0 * hidden0) + (weight1 * hidden1) + bias = y0
MOV   sr11   mr2
SUB   mr4    sr11   mr3     ; d0 - y0 = error
```

*Update Weight*

```
MUL   sr11   mr4    sr4     ; eta_weight * error
MUL   mr7    sr11   mr5     ; eta_weight * error * hidden0
ADD   sr8    sr8    mr7     ; weight0 = weight0 + eta_weight * error * hidden
```

*Update bias*

```
MUL   mr7    sr5    mr4     ; eta_bias * error
ADD   sr10   sr10   mr7     ; bias = bias + eta_bias * error
```

*Update $\mu$*

```
MUL   mr7    sr8    cr3     ; 2 * weight0
MOV   sr11   mr4
MUL   mr7    mr7    sr11    ; 2 * weight0 * error
SUB   sr11   mr0    sr0     ; x0 - mu0
MOV   mr8    sr2
MUL   mr8    mr8    sr2     ; sigma0^2
DIV   sr11   sr11   mr8     ; (x0 - mu0) / sigma^2
MUL   sr11   mr7    sr11    ; 2 * weight0 * error * (x0 - mu0) / sigma^2
MUL   mr9    sr11   mr5     ; 2 * weight0 * error * (x0 - mu0) * hidden0 / sigma^2
MUL   mr9    mr9    sr6     ; eta_mu * 2 * weight0 * error
                            ; * (x0 - mu0) * hidden0 / sigma^2 = delta_mu
ADD   sr0    sr0    mr9     ; mu0 = mu0 + delta_mu
SUB   sr11   mr1    sr0     ; x1 - mu0
DIV   sr11   sr11   mr8     ; (x1 - mu0) / sigma^2
MUL   sr11   sr11   mr7     ; 2 * weight0 * error * (x1 - mu0) / sigma^2
MUL   mr9    sr11   mr5     ; 2 * weight0 * error * (x1 - mu0) * hidden0 / sigma^2
MUL   mr9    mr9    sr6     ; eta_mu * 2 * weight0 * error
                            ; * (x1 - mu0) * hidden0 / sigma^2 = delta_mu
ADD   sr0    sr0    mr9     ; mu0 = mu0 + delta_mu
```

*Update $\sigma$*

```
SUB   mr7    mr0    sr0     ; x0 - mu0
MOV   sr11   mr7
MUL   mr7    mr7    sr11    ; (x0 - mu0^2
SUB   mr8    mr1    sr0     ; x1 - mu0
MOV   sr11   mr8
MUL   sr11   mr7    sr11    ; (x1 - mu0)^2
ADD   mr7    mr7    sr11    ; (x0 - mu0)^2 + (x1 - mu0)^2
MOV   mr8    sr2
MUL   mr8    mr8    sr2
MUL   sr11   mr8    sr2     ; sigma^3
DIV   mr7    mr7    sr11    ; ((x0 - mu0)^2 + (x1 - mu0)^2) / sigma^3
MUL   mr7    mr7    cr3     ; 2 * ((x0 - mu0)^2 + (x1 - mu0)^2) / sigma^3
MOV   sr11   mr4
MUL   mr7    mr7    sr11    ; 2 * error * ((x0 - mu0)^2 + (x1 - mu0)^2) / sigma^3
MUL   mr7    mr7    sr8     ; 2 * error * weight0 * ((x0 - mu0)^2 + (x1 - mu0)^2) / sigma^3
MOV   sr11   mr5
MUL   mr7    mr7    sr11    ; 2 * error * weight0 * hidden0
                            ; * ((x0 - mu0)^2 + (x1 - mu0)^2) / sigma^3
MUL   mr7    mr7    sr7     ; eta_sigma * 2 * error * weight0 * hidden0
                            ; * ((x0 - mu0)^2 + (x1 - mu0)^2) / sigma^3 = delta_sigma
ADD   sr2    sr2    mr7     ; sigma0 = sigma0 + delta_sigma
```

(b) Backward process of RBF net

**Fig. 12.** Calculation of RBF network.

**Table VIII**
Pattern output after learning.

| | Desired value | Actual value | Pattern error |
|---|---|---|---|
| (0.1, 0.1) | 0.1 | 0.100996 | −0.000996 |
| (0.1, 0.9) | 0.9 | 0.898341 | 0.001659 |
| (0.9, 0.1) | 0.9 | 0.898341 | 0.001659 |
| (0.9, 0.9) | 0.1 | 0.101117 | −0.001117 |
| Sum of error | | | 0.000008 |

**Table IX**
RMS error between MATLAB and FPGA.

| Parameters | RMS error |
|---|---|
| Sum of error | 0.000034 |
| $w_0$ | 0.002204 |
| $w_1$ | 0.000024 |
| Bias, $\theta$ | 0.000061 |
| $\mu_0$ | 0.000010 |
| $\mu_1$ | 0.000049 |
| $\sigma_0$ | 0.000010 |
| $\sigma_1$ | 0.000713 |

**Table X**
Final parameter values of the RBF network.

| Parameters | Value |
|---|---|
| $w_0$ | −0.921395 |
| $w_1$ | −1.018259 |
| $\theta$ | 1.122028 |
| $\mu_0$ | 0.113313 |
| $\mu_1$ | 0.889585 |
| $\sigma_0$ | 0.453306 |
| $\sigma_1$ | 0.402052 |

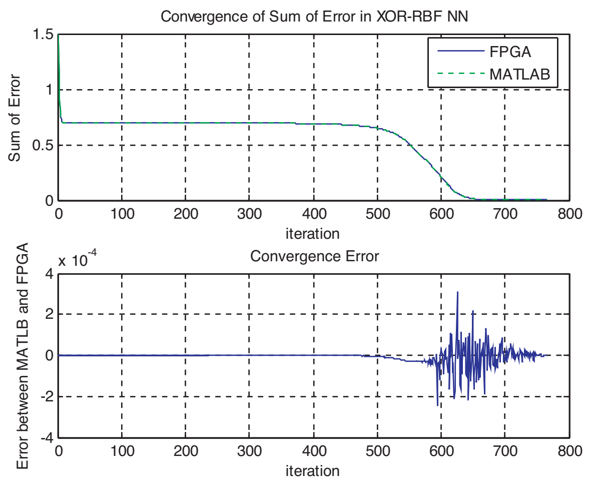


**Fig. 13.** Error convergent plot when the 50th order is used.

## 6.2. Convergence of weights

The learning stops when the error satisfies the tolerance specified in Table III. After the learning process, all patterns are tested for the classification. The resultant classification is given in Table VIII. We clearly see that learning of the neural chip has been done correctly. Classification is well done. Table IX shows the difference of the weight values between the neural hardware and the MATLAB program. Finally converged values of weights are listed in Table X.

## 6.3. Convergence of weights when different learning rates

Next experiment is done when the learning rates, $\eta_w = 0.1$, $\eta_\theta = 0.1$, $\eta_\mu = 0.01$, $\eta_\sigma = 0.01$ are used. The goal is to see the difference under a different learning condition. The deviation errors between the neural hardware and the MATLAB program are clearly compared.

Table XI shows the classification result after learning. The difference of the final weight values between the neural hardware and the MATLAB program is listed in Table XII. Table XIII compares weights values when two different learning tasks have been executed. We see the closeness in weight values between two different learning tasks although different learning rates are used.

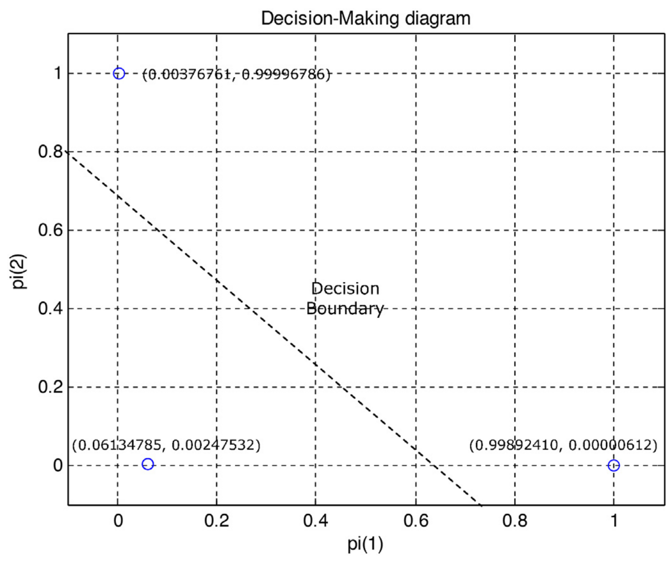


**Fig. 14.** The decision boundary for the XOR logic.

**Table XI**
Pattern recognition after learning.

| | Desired value | Actual value | Pattern error |
|---|---|---|---|
| (0.1, 0.1) | 0.1 | 0.101586 | −0.001586 |
| (0.1, 0.9) | 0.9 | 0.898415 | 0.001585 |
| (0.9, 0.1) | 0.9 | 0.898415 | 0.001585 |
| (0.9, 0.9) | 0.1 | 0.101244 | −0.001244 |
| Sum of error | | | 0.000009 |

**Table XII**
RMS error between MATLAB and FPGA.

| Parameters | RMS error |
|---|---|
| Sum of error | 0.000069 |
| Weight0, $w_0$ | 0.000157 |
| Weight1, $w_1$ | 0.000052 |
| Bias, $\theta$ | 0.000067 |
| $\mu_0$ | 0.000029 |
| $\mu_1$ | 0.000125 |
| $\sigma_0$ | 0.000035 |
| $\sigma_1$ | 0.000713 |

**Table XIII**
Weight changes by different learning rates RMS.

| Parameters | $\eta_w = \eta_\theta = 0.05$, $\eta_\mu = \eta_\sigma = 0.005$ | $\eta_w = \eta_\theta = 0.1$, $\eta_\mu = \eta_\sigma = 0.01$ |
|---|---|---|
| $w_0$ | −0.85266680 | −0.8761408 |
| $w_1$ | −0.84844798 | −0.8705631 |
| $\theta$ | 0.95275021 | 0.9776968 |
| $\mu_0$ | 0.11095808 | 0.0977247 |
| $\mu_1$ | 0.90131116 | 0.9075751 |
| $\sigma_0$ | 0.33398876 | 0.3640978 |
| $\sigma_1$ | 0.23129223 | 0.2313576 |

## 7. Conclusion

The floating-point based processor is designed in the HDL to allow us to design the RBF neural network. The hardware implementation of the RBF neural network is written in assembly codes and embedded on the FPGA chip. The performance of classifying the nonlinear classifier of XOR logic is successfully tested. Experimental results show that 50th order is enough to give an acceptable range of deviation error by the Taylor series for nonlinear function approximation. Therefore, the computational error due to the numerical approximation can be accepted. The neural chip successfully conducts the nonlinear classification by the on-line back propagation learning algorithm. Since we confirm that the RBF neural chip works properly, the neural chip can be used as a controller for controlling nonlinear dynamical systems in the future.

## References

[1] C. Lau, Neural Networks: Theoretical Foundations and Analysis, IEEE Press, 1992.
[2] S. Lee, M.K. Rhee, A Gaussian potential function network with hierarchically self-organizing learning, Neural Netw. 4 (1991).
[3] M. Miyamoto, M. Kawato, T. Setoyama, R. Suzuki, Feedback error learning, Neural Netw. 1 (1988) 251–265.
[4] F.L. Lewis, S. Jagannathan, A. Yesildirek, Neural Network Control of Robot Manipulators and Nonlinear Systems, Taylor & Francis, London, UK/Philadelphia, USA, 1999.
[5] H. Chaoui, P. Sicard, W. Gueaieb, ANN based adaptive control of robotic manipulators with friction and joint elasticity, IEEE Trans. Ind. Electron. 56 (8) (2009) 3174–3187.
[6] C.S. Chen, Dynamic structure neural–fuzzy networks for robust adaptive control of robot manipulators, IEEE Trans. Ind. Electron. 55 (9) (2008) 3402–3414.
[7] S. Jung, S.S. Kim, Hardware implementation of real-time neural network controller with a DSP and an FPGA for nonlinear system, IEEE Trans. Ind. Electron. 54 (1) (2007) 265–271.
[8] S. Jung, H.T. Cho, T.C. Hsia, Neural network control for position tracking of a two-axis inverted pendulum system: experimental studies, IEEE Trans. Neural Netw. 18 (4) (2008) 1042–1048.
[9] M.A. Cavuslu, C. Karakuzu, F. Karakaya, Neural identification of dynamic systems on FPGA with improved PSO learning, Appl. Soft Comput. 12 (2012) 2707–2718.
[10] F.M. Pouzols, A.B. Barros, D.R. Lopez, S. Sanchez-Solano, Enabling fuzzy technology in high performance networking via an open FPGA-based development platform, Appl. Soft Comput. 12 (2012) 1440–1450.
[11] P. Brox, A.I. Barturone, S. Sanchez-Solano, Fuzzy logic-based embedded system for video de-interlacing, Appl. Soft Comput. 14 (2014) 338–346.
[12] R. Sepulveda, O. Montiel, O. Castillo, P. Melin, Embedding a high speed internal type-2 fuzzy systems for a real plant into an FPGA, Appl. Soft Comput. 12 (2012) 988–998.
[13] Y. Maldonado, O. Castillo, P. Melin, Particle swarm optimization of internal type-2 fuzzy systems for FPGA applications, Appl. Soft Comput. 13 (2013) 496–508.
[14] N.M. Botros, M. Abdul Aziz, Hardware implementation of artificial neural network using field programmable gate array, IEEE Trans. Ind. Electron. 41 (6) (1994) 665–667.
[15] Y. Taright, M. Hubin, FPGA implementation of a multilayer perceptron neural network using VHDL, Proc. ICSP (1998) 1311–1314.
[16] J. Zhu, G.J. Milne, B.K. Gunther, Towards an FPGA based reconfigurable computing environment for neural network implementations, IEEE Proc. Artif. Neural Netw. (1999) 661–666.
[17] M. Krips, T. Lammert, A. Kummert, FPGA implementation of a neural network for a real time hand tracking system, in: IEEE International Workshop on Electronic Design, Test, and Applications, 2002.
[18] F. Sargeni, V. Bonaiuto, Digitally programmable nonlinear function generator for neural networks, Electron. Lett. 41 (2005) 3.
[19] K. Basterretxea, J.M. Tarela, I.D. Campo, G. Bosque, An experimental study on nonlinear function computation for neural/fuzzy hardware design, IEEE Trans. Neural Netw. 18 (1) (2007) 266–283.
[20] S. Himavathi, D. Anitha, A. Muthuramalingam, Feed-forward neural network implementation in FPGA using layer multiplexing for effective resource utilization, IEEE Trans. Neural Netw. 18 (3) (2007) 880–888.
[21] A. Dinu, M.N. Cirstea, S.E. Cirstea, Direct neural network hardware implementation algorithm, IEEE Trans. Ind. Electron. 57 (5) (2010) 1845–1848.
[22] D. Zhang, H. Li, A stochastic-based FPGA controller for an induction motor drive integrated neural network algorithms, IEEE Trans. Ind. Electron. 55 (2) (2008) 551–561.
[23] H.C. Huang, C.C. Tsai, FPGA implementation of an embedded robust adaptive controller for autonomous omnidirectional mobile platform, IEEE Trans. Ind. Electron. 56 (5) (2009) 1604–1616.
[24] J.S. Kim, S. Jung, Implementation of the RBF neural chip with the on-line learning back-propagation, IEEE WCCI (2008) 378–384.
[25] J.S. Kim, H.W. Jeon, S. Jung, Evaluation of embedded RBF neural chip with back-propagation FPGA-based general purpose algorithm for pattern recognition tasks, IEEE INDIN (2008) 1110–1115.
[26] J.S. Kim, S. Jung, Hardware implementation of a neural network controller on FPGA for a humanoid robot arm, IEEE AIM (2008) 1164–1169.
[27] F. Moreno, J. Alarcon, R. Salvador, T. Riesgo, Reconfigurable hardware architecture of a shape recognition system based on specialized tiny neural networks with online training, IEEE Trans. Ind. Electron. 56 (8) (2009) 3253–3263.
[28] A. Gomperts, A. Ukil, F. Zurfluh, Development and implementation of parameterized FPGA based general purpose neural networks for on-line applications, IEEE Trans. Ind. Inf. 7 (1) (2011) 78–89.
[29] T.O. Kowalska, M. Kaminski, FPGA implementation of the multilayer neural network for the speed estimation of the two mass drive system, IEEE Trans. Ind. Inf. 7 (3) (2011) 436–445.
[30] F. Yang, M. Paindavoine, Implementation of an RBF neural network on embedded systems: real-time face tracking and identity verification, IEEE Trans. Neural Netw. 14 (5) (2003) 1162–1175.
[31] R.S. Oreifej, R.F. DeMara, Intrinsic evolvable hardware platform for digital circuit design and repair using genetic algorithms, Appl. Soft Comput. 12 (2012) 2470–2480.
[32] H. Yu, T.T. Xie, S. Paszczynski, B.M. Wilamowski, Advantages of radial basis function networks for dynamic system design, IEEE Trans. Ind. Electron. 58 (December (12)) (2011) 5438–5450.