J Electr Eng Technol.2016; 11(5): 1466-1474
http://dx.doi.org/10.5370/JEET.2016.11.5.1466

ISSN(Print) 1975-0102
ISSN(Online) 2093-7423

A Data Path Design Tool for Automatically Mapping Artificial Neural Networks on to FPGA-Based Systems

Ibrahim Sahin* and Namik Kemal Saritekin†

Abstract – Artificial Neural Networks (ANNs) are usually implemented as software running on general purpose computers. On the other hand, when software implementations do not provide sufficient performance, ANNs are implemented as hardware on FPGA based systems for performance enhancement. Mapping ANNs to FPGAs is a time consuming and error prune process. In this study, a novel data path design tool, ANNGEN, has been proposed to help automate mapping ANNs to FPGA based systems. ANNGEN accepts ANN definitions in a NetList form. First, it parses and analyzes given NetList. Second, it checks the availability of the neurons. If all the neurons required by the NetList are available in its neuron Library, ANGENN performs the design procedure and produces VHDL code for the given NetList. ANNGEN has been tested with several different test cases, and it is observed that it is able to successfully generate VHDL codes for given ANN NetLists. Our practice with ANNGEN has showed that it effectively shortens the time required for implementing ANNs on FPGAs. It also eliminates the need for expert people. Additionally, ANNGEN produces error free code; thus, the debugging stage is also eliminated.

Keywords: Artificial Neural Networks, Design Automation, Field Programmable Gate Arrays, Software Tool

1. Introduction

Although, Artificial Neural Networks (ANNs) can be implemented as software, in some cases where software implementations are not sufficient in terms of performance, hardware implementations are desired [1]. There are three hardware implementation options for ANNs which are Application Specific Integrated Circuit (ASIC), Digital Signal Processing (DSP) and Field Programmable Gate Arrays (FPGA) implementations [2]. The ASIC implementation suffers being inflexible. Once an ANN is implemented as an ASIC, no further changes can be done. Moreover, recovering a bug in an ASIC implementation is very costly and time consuming process. The DSP chips are a kind of specialized microprocessors. For implementing a design in DSP, one still has to write code and this code must be executed instruction by instruction, similar to the general purpose microprocessors. Here, again DSP implementation may not exploit full parallelism in the ANNs, and may not provide desired performance [3].

The best hardware choice for implementing ANNs is the use of FPGA devices. They are flexible, can easily be reconfigured in the case of error, can fully exploit the potential parallelism in the applications, and implementation time is less than compared to ASICs [4-6]. Although FPGAs offer great advantages over the other hardware choices, they have some disadvantages too. For instance, application development time is still considerably long. Expert people who have knowledge in the areas of both hardware and software are needed to develop both hardware configurations and the software interfaces of the FPGA devices or boards [7]. Moreover, hand designed and coded FPGA implementations have always potential to contain bugs.

All of these disadvantages of the FPGAs can be overcome using a design automation tool. Fig. 1 shows the regular design flow for mapping ANNs on to FPGAs. In this design flow, the most important step is done by experts who convert given ANN specifications to Hardware Description Language (HDL) codes. In this research work, a novel Design Automation Tool, Artificial Neural Network data path GENerator (ANNGEN), has been developed to automate this step and to minimize the need for expert people while developing ANN applications on FPGAs.

ANNGEN takes an ANN definition in the form of a NetList and some other input files, and produces a Very high speed integrated circuit Hardware Description Language (VHDL) code which implements the desired ANN.

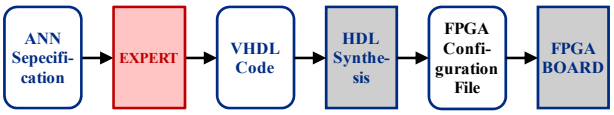


Fig. 1. Classical design flow of FPGA-based neural networks

† Corresponding Author: Dept. of Electrical Engineering, Duzce University, Turkey. (tekinfizikster@gmail.com)
* Dept. of Computer Engineering, Duzce University, Turkey. (ibrahimsahin@duzce.edu.tr)
Received: September 4, 2015; Accepted: April 1, 2016

ANNGEN has been tested with several test cases and the correctness of the outcome of it has been verified using the Xilinx's ISE design suit. Our practice with ANNGEN showed that it effectively shortens the time required for implementing ANNs on FPGAs, and it eliminates the need for expert people. Moreover, debugging stage of ANN implementations is eliminated, since, ANNGEN produces error free code. For functional verification and speedup measurement a special test case has been formed for predicting concrete compressive and tensile strength using ANNs. In this test case, we formed same ANN structure in both MatLab environment and in hardware environment using ANNGEN and compared them.

2. Background

2.1 Artificial neurons and neural networks

Fig. 2 shows the structure of a simple artificial neuron. In this model, each input (x_j) is weighted, and multiplied with its associated weight value (w_j), and weighted inputs are added up to a single value. At this stage, if the neuron is biased, a bias value is also added to this sum. This single value is passed through an activation function, also called transfer function, and is asserted as the output of the neuron.

Eqs. (1, 2, and 3) show the mathematical model of a single neuron.

$$v = \sum_{j=1}^{n} w_j x_j + \beta \tag{1}$$

where x_j and w_j are the inputs of the neuron and the associated weight values for j=1, 2, 3…n, and β is the bias value, if present [8]. The activation function, $f(v)$, can be one of many different types defined in the literature. The neurons in our library have Pure Linear (*PureLin*) and Hard Limiting (*HardLim*) activation functions. Mathematical definitions of *PureLin* and *HardLim* activation functions are given in Eqs. (2) and (3), respectively.

$$PureLin(v) = v \tag{2}$$

$$HardLim(v) = \begin{cases} 1, & if \quad v \geq 0 \\ 0, & otherwise \end{cases} \tag{3}$$

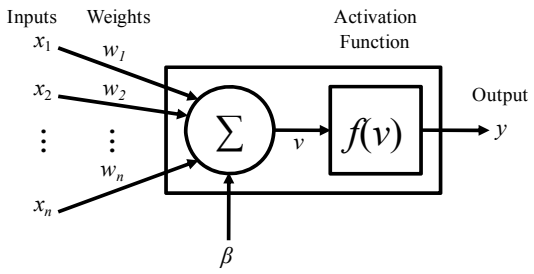


Fig. 2. Structure of an artificial neuron

Single neuron is not much of use in practice. Thus, neurons are connected together to form artificial neural networks and these networks are utilized in several tasks. Before being used, ANNs have to go through a learning process [9]. After the learning process, ANNs are used in several key areas such as pattern recognition, interpretation of data, optimization, function approximation, controlling, classification, clustering and filtering of data, power consumption estimation, and prediction [10], etc. Some common features of ANNs are learning, generalization, fault tolerance, parallel processing, nonlinear retention of knowledge [11, 12].

2.2 FPGA chips

Field Programmable Gate Arrays (FPGAs) are a type of chips that are completely prefabricated and contain special features for customization. The user of these chips can implement digital circuit designs by configuring them. A typical FPGA device contains three configurable parts which are Configurable Logic Blocks (CLBs), a programmable interconnection network and programmable input/output (I/O) blocks. CLBs contain look-up tables, flip-flops and multiplexer, and can be configured to implement combinational or sequential circuit pieces. The interconnection networks are completely programmable and are used to establish connection either between CLBs or between CLBs and I/O blocks. The I/O blocks are also configurable and are used to configure pins of the chip as input, output or bidirectional pins [13, 14]. FPGA chips are not used solely for developing applications. They usually are utilized on electronic cards with several support components such as I/O ports and memory units. These cards are simply called as FPGA boards. Depending on the design, some FPGA boards work independently while some others work as an attachment to a host computer through a bus structure [15, 16].

3. Related Works

Several research studies have been conducted on mapping different types of ANNs on to FPGA based systems. Sahin *et. al.* developed artificial neurons with exponential activation functions, mapped to FPGAs and measured their performance [17]. Cavuslu *et. al*. implemented an ANN training algorithm on FPGAs using IEEE-754 floating-point data format [18]. They also developed another ANN structure for locating license plate of a vehicle in a digital image [19]. PipeRench is a new reconfigurable fabric and it combines the flexibility of general-purpose processors with the efficiency of customized hardware to achieve extreme performance speedups [20]**.** D. Crookes *et. al.* presented a high level software environment for FPGA-based image processing [21]. A new method for hardware-software partitioning utilizes the runtime reconfiguration capabilities

of FPGAs [22]. Some of the other FPGA based ANN studies are implementation of ANN Classifier for Gas Sensor Array Applications [23], Architectures of Nervous System Models Embedded on High-Performance FPGAs [24], Neural Network Implementation of FPGA Chips used in Turkish Phonemes Classification [25], Neural Network-Based Embedded Controller on FPGA for Power Electronics Applications [26], and a Gradual Neural Network Approach for FPGA Segmented Channel Routing Problems [27]. In all of these studies, researchers first developed their neurons and later built their neural networks and mapped them to FPGA based systems manually. These neuron developments and FPGA mapping processes are time consuming and error prune processes.

4. Sample Neuron Library

In the proposed work, a sample neuron library has been developed to be used with ANNGEN. The library contains a total of six neurons, 2- and 4-input neurons with two different activation functions which are Pure Linear (*PureLin*) and Hard Limit (*HardLim*). The neurons are designed to process single precision (32-bit) IEEE-754 floating-point data.

Fig. 3 shows the standard top level block diagram of the neurons. Here, each neuron has a 32-bit WeightIn and a WeightOut signals. These signals are used for serially initializing the weights and, if available, the bias registers. Since a neural network contains several neurons, all neurons in the network can be connected as a chain, and all weights can be loaded from one WeightIn (from the first neurons WeightIn signal). For serial load operation, the Load signal has to be asserted each time a new weight value is presented to WeightIn. This kind of design enables us to reduce several weight inputs to a single input. Depending on the type, neurons have either 2 or 4 32-bit stimulus inputs. EnN and EnN_Out signals are used to synchronize data flow through the neurons. Whenever a set of stimuli are available at a neuron's inputs, EnN signal must be asserted to let the neuron read stimuli and begin processing them. When the neuron finishes processing a set of stimuli and produces a result, it asserts EnN_Out signal.

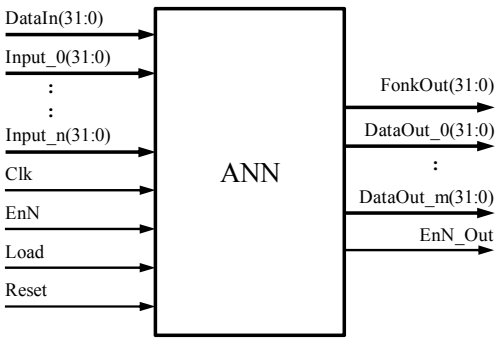


Fig. 3. Common top-level block diagram of the neurons

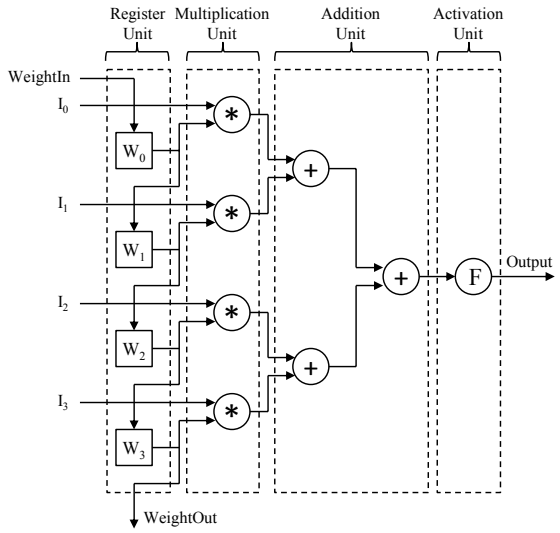


Fig. 4. Detailed view of the 4-input neuron design

This signal can be used to activate neurons in the next layer of the network. The result of the neuron is outputted via FonkOut signal.

Fig. 4 shows the second level block diagram of a non-biased 4-input neuron. The neurons were designed in four sections which are Register Unit, Multiplication Unit, Addition Unit, and Function Unit. The Register Unit keeps neuron's weight values. As mentioned above, the registers were connected to form a chain. The last register's output also goes to the output of the neuron so that the next neuron's weight registers can be added to the same register chain. Multiplication and addition units contain parallel running multipliers and adders.

The stimulus inputs and weight values are first multiplied in the multiplication units and then added up to a single value. This single value is then feed to the Function Unit. The Function Unit evaluates this value and produces the output value of the neuron. Neurons with 2 inputs were designed in a similar fashion. The multipliers and adders in the neurons have 8 clock cycles latency. The transfer function has one cycle latency. When a set of stimuli is given to a 4-input neuron the result is available at FonkOut after 25 clock cycles. On the other hand, since, all the adders and multipliers are pipelined, at each clock cycle, a set of stimuli can be given to the neurons. The first result appears at the output after 25 clock cycles. But the subsequent results are produced in the subsequent cycles. Floating-point multiplication and addition units, used in the design of neurons, were built using the IP Core Generator of Xilinx's ISE Web Pack 14.7 EDA tool.

Since the neurons in the library were designed like jigsaw puzzle pieces, neural networks with *i*-input, *z*-output, and *m*-layer can be formed using these neurons easily. Fig. 5 shows how these neurons should be connected to form an ANN. The first layer of the ANN is the input layer.

In this layer, loadable registers should be used to hold stimuli. The input register can be connected serially as

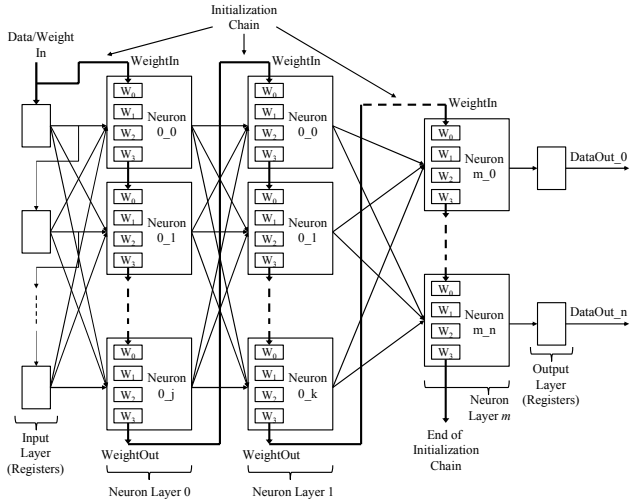


Fig. 5. Proposed neural network with *i*-inputs, *z*-outputs, and *m*-layers

shown in the figure to reduce the number of data inputs. They can also be connected in parallel to maximize the stimulus input rate. Same type of neurons should be used through a neuron layer. If different types of neurons needed to be used in a single neuron layer, latency of the neurons must be considered and delay units must be added to the outputs of the neurons whose latencies are less than the other neurons in the layer. Currently, ANNGEN does not support using neurons with different latencies in the same layer. The output layer is also constructed using registers. These registers are optional. If desired, the output of the last layer can be used directly without being stored in registers.

Before using such an ANN constructed with the neurons, the weight registers of all neurons have to be initialized with proper weight values by pushing them from the first neurons weight input one by one in a proper order until all weight values are loaded.

5. The ANNGEN

Artificial Neural Network GENerator (ANNGEN) is a software tool that automatically generates VHDL code for a given neural network description. Fig. 6 shows the location of ANNGEN in proposed FPGA-based ANN design flow. When all inputs are ready, it performs the design procedure and generates VHDL code in less than a second. The generated code has to be synthesized by an FPGA chip vendor's synthesis tools for a desired FPGA chip.

5.1 The Inputs of ANNGEN

ANNGEN requires three input files. These are a text-based artificial neural network definition file (NetList), the module library (Library) and the template file (Template).

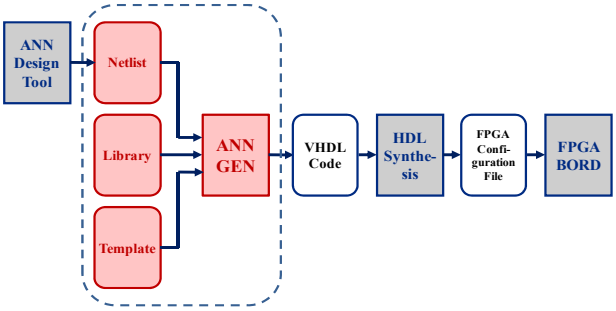


Fig. 6. Proposed FPGA-based neural network design flow

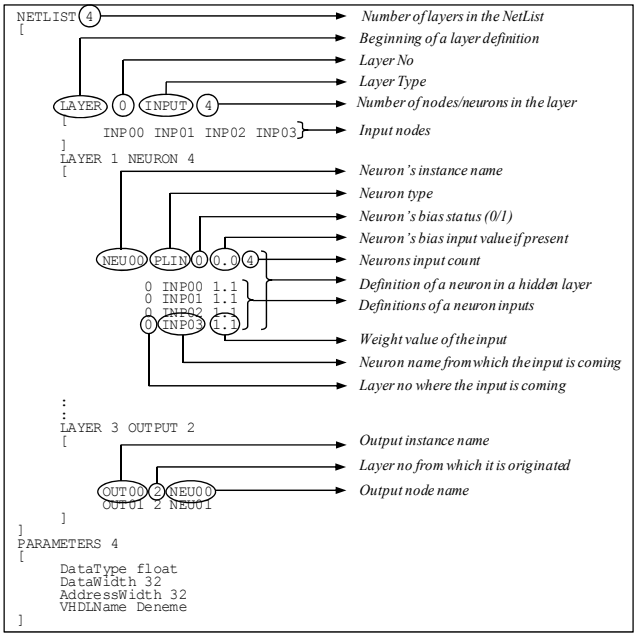


Fig. 7. Structure of the NetList file

NetList is a text file which contains the text based definition of an ANN that will be mapped to an FPGA device. The ANN structure is defined hierarchically in this file. Fig. 7 shows the structure of the NetList file. It clearly defines layers, neurons in the layers, connections between the neurons and the layers, weight values of the neurons, bias statuses, types of the neurons, and some other parameters. The NetList file can be either written by the users or it can be generated using Artificial Neural Network Design and Education Software (ANNDES) [28] which is another design tool developed before ANNGEN in the same project framework.

The Library file contains the list of available neurons that can be used by ANNGEN. This file is also a text based file and each record in the file defines the name of a neuron, the number of inputs of the neuron, and the bias status of it. Fig. 8 shows the internal format of the library file.

The template file contains some VHDL templates needed for code writing, VHDL codes of the Neurons, and some other component definitions such as registers, multiplier, and adders. ANNGEN uses these templates

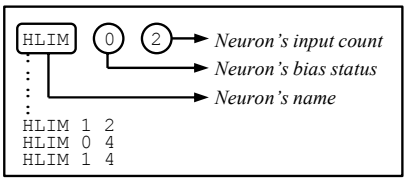


Fig. 8. Internal format of the Library file

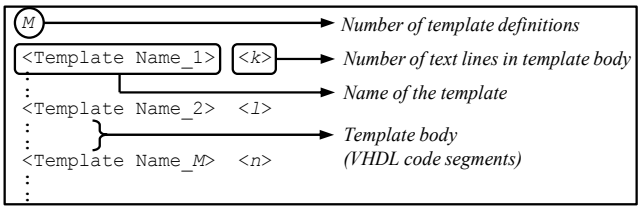


Fig. 9. Internal format of the template file

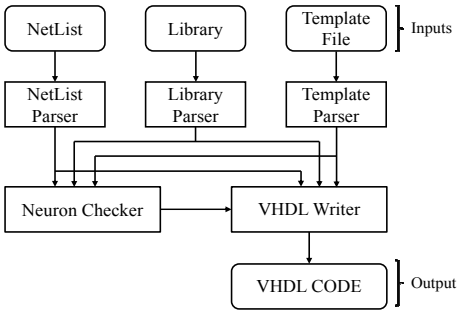


Fig. 10. General structure of ANNGEN

when writing VHDL code for a requested ANN design. Fig. 9 shows the internal format of the template file. The first number in this file defines the number of temples found in the file. Each template is declared with a template name and a number which specifies the number of text lines in the template body. The template body contains VHDL code fragments.

As new neuron designs are available, they can easily be adapted to the system by defining these new neurons both in the template file and in the Library file. ANNGEN automatically recognizes and uses them in ANN designs.

5.2 Components of the ANNGEN

ANNGEN consists of five main components which are NetList, Library, and Template Parsers, Neuron Checker, and VHDL Writer. Fig. 10 depicts the data flow between the components.

Fig. 11 shows the basic algorithm of ANNGEN. It first reads and parses three input files using three separate parsers, and parsed information is stored in separate flexible data structures. Then, it checks the NetList to determine whether all the neurons specified in the NetList are available in the library or not. If the test result is a success, ANNGEN performs the design procedure and starts producing VHDL code using parsed information,

ANNGEN()
1 Read and Parse given NetList file.
2 Read and Parse given Library file.
3 Read and Parse given Template file.
4 If (All neurons in NetList are available in Library)
Write VHDL code.
Else
Send warning messages.

Fig. 11. ANNGEN main algorithm

otherwise; it gives some warning messages and stops.

The NetList Parser's task is to read and parse the NetList file and store extracted information in a flexible data structure for future use. First, it reads the number of layers in the network. Then, it reserves enough memory space to hold that many layers. Later, it reads the layers individually. There can be three different types of layers which are *Input*, *Output,* and *Neuron Layers*. Regardless of the type, all three kinds of layers are stored in the same data structure with a layer type indicator. While reading a layer, again, the reader first reads the number of neurons or number of elements in the layer, reserves that much memory space, and then, reads the layer. After reading all layers in the NetList, it also reads the parameters at the end of the NetList file and stores them in a suitable data structure. Currently, there are 4 parameters that are DataType = {float | integer}, DataWidth = {16 | 32 | 64}, AddressWidth = {16 | 32 | 64}, and VHDLName = {"sample.vhd"}. These parameters tell ANNGEN that what kind of data is going to be processed in the ANN, how wide the data and address busses of the ANN are going to be, and finally, what is going to be the file name of the produced VHDL code.

The structure of the library file is very simple. Each line in this file contains a Neuron name, bias status of the neuron, and the number of inputs to the neuron. The Library Parser reads these data items and stores them in a structured array.

Template file includes the VHDL code fragment for writing VHDL code and VHDL code for neurons and some other components which are needed for forming ANN hardware. The template parser reads and parses given template file and stores this information in a data structure.

Before writing VHDL code for a given NetList, ANNGEN ensures that VHDL definitions of all neurons in the NetList are available in the Library. For this purpose, the Neuron checker spans through the parsed NetList data and identifies all neuron types. Identified neurons types are searched in the Library. If any of the neurons is not available in the library, it returns "false", causing main program to give warning messages to the user, and terminate. Otherwise, neurons checker returns "true" and let the VHDL Writer perform the design procedure and produce VHDL code.

VHDL writer is the most sophisticated section of the ANNGEN. It is activated, if the neuron checker returns "true". Fig. 12 shows the simplified algorithm of the

VHDL_Writer()
1 Write VHDL templates of necessary components such as registers, adders, and multipliers to the target file.
2 Span NetList data structure and write all needed neurons' VHDL templates to the target file.
3 Form the top level Entity part for the ANN and write it to the target file.
4 Form the top level Architecture part.
4.1 Form the header of the Architecture part.
4.2 Define all necessary internal signals.
4.3 Instantiate Input Registers and make connections.
4.4 Instantiate Neurons and make connections.
4.5 End the Architecture part.

Fig. 12. Simplified algorithm for the VHDL Writer

VHDL writer. After creating a text file using the filename provided in the parameters section of the NetList, it first writes VHDL templates of necessary components to the target file. Second, it spans the NetList Data and writes VHDL templates for each neuron type to the file. In the third step, VHDL writer forms the Entity section of the top level of the ANN begin constructed and writes to the file. In this section, inputs and outputs of the ANN are defined. Most of the ports are standard ports such as reset, clock, and data-in, and they do not vary from one ANN to another. The only variation in this section occurs in the ANN outputs. The Writer defines one data output for each output of the given ANN and writes to the target file.

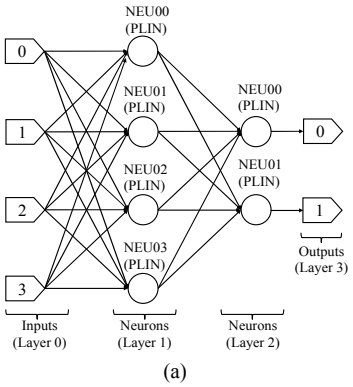
In the next step, it forms the Architecture part of the given ANN. The Architecture is formed in five sub-steps. First, the header of it is written to the target file. Second, the writer defines internal connection signals. Three kinds of signals are defined here. These are data flow signals which are used to connect one neuron's output to another neuron's input, initialization chain signals which are used to form weight and bias initialization chain, and the enable signals which are used to trigger neurons as soon as valid data is available at the neurons' inputs. After the signal definitions, component instantiations are completed in the third and fourth sub-steps. The input registers are instantiated before neurons and their connections are finished. Later, the neurons are instantiated and their connections are done according to the NetList. During these instantiations, signals defined in the second sub-step are used to connect components. In the last sub-step, the Architecture definition is ended. All information is dumped in to the target text file as they are formed. Finally, the text file is closed and ANNGEN ends.

6. Testing ANNGEN

Before we tested the ANNGEN, we designed six example neurons. The neuron designs were coded in VHDL and synthesized with ISE tool. Table 1 lists the hardware statistics of the neurons on Virtex-6 chip (xc6vlx75t). 7th column of the Table shows the theoretical

Table 1. Summary of the neurons in the Library

Neuron Type	Input Cnt.	Slice Regs. (%)	LUTs (%)	Occup. Slices (%)	Latency (Clk C.)	Num.of Neurons
HLIM	*2*	1	3	4	17	25
HLIM	*4*	4	8	10	25	10
HLMS	*2*	1	3	4	17	25
HLMS	*4*	4	8	10	25	10
PLIN	*2*	1	3	4	17	25
PLIN	*4*	4	8	10	25	10




```
NETLIST 4
[
LAYER 0 INPUT 4
     [
          INP00 INP01 INP02
INP03
     ]
     LAYER 1 NEURON 4
     [
          NEU00 PLIN 0 0.0 4
                0 INP00 1.1
                0 INP01 1.1
                0 INP02 1.1
                0 INP03 1.1
          NEU01 PLIN 0 0.0 4
                0 INP00 1.1
                0 INP01 1.1
                0 INP02 1.1
                0 INP03 1.1
          NEU02 PLIN 0 0.0 4
                0 INP00 1.1
                0 INP01 1.1
                0 INP02 1.1
                0 INP03 1.1
          NEU03 PLIN 0 0.0 4
                0 INP00 1.1
                0 INP01 1.1
                0 INP02 1.1
                0 INP03 1.1
     ]
     LAYER 2 NEURON 2
     [
          :
          :
     ]
     LAYER 3 OUTPUT 2
     [
          OUT00 2 NEU00
          OUT01 2 NEU01
     ]
]
PARAMETERS 4
[
     DataType float
     DataWidth 32
     AddressWidth 32
     VHDLName Deneme
]
```

(b)

Fig. 13. Example ANN and its NetList definition

upper limits of the number of neurons that can fit in a single Virtex-6 chip. These values were calculated based on the number of Occupied Slices. The xc6vlx75t chip is the smallest device of Virtex-6 family. Using a larger chip such as xc6vlx760 up to 10 times more neurons can be fitted into a single device.

ANNGEN has been successfully tested with several test cases. In one of the test cases, an ANN was needed for predicting concrete compressive and tensile strength values using amount of cement, amount of water, consistency, and temperature values of the concrete composition. Our purpose in this test case was to compare the ANN formed using ANNGEN with an ANN formed in MatLab environment in terms of calculation accuracy and speed. The test data was collected from 68 different concrete compositions. Fig. 13 shows the ANN structure and its associated NetList definition that was used in the test. This ANN is composed of two hidden layers with four and two neurons, respectively. The neurons which are available in our library have *PureLin* activation functions.

The ANN structure explained above was first formed in the MatLab environment, trained, and simulated with sample test data. During the training phase 50 of 68 records were used and the remaining records were used for testing

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY Deneme IS
   PORT (Clock : in STD_LOGIC;
      Reset : in STD_LOGIC;
      LConst : in STD_LOGIC;
      LVar : in STD_LOGIC;
      DataIn : in STD_LOGIC_VECTOR (31 downto 0);
      OUT00_out : out STD_LOGIC_VECTOR (31 downto 0);
      OUT01_out : out STD_LOGIC_VECTOR (31 downto 0));
END ENTITY Deneme;
ARCHITECTURE RTL OF Deneme IS
   --Internal data flow signal
   SIGNAL S_INP00_0 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_INP01_0 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_INP02_0 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_INP03_0 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU00_1 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU01_1 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU02_1 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU03_1 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU00_2 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU01_2 : STD_LOGIC_VECTOR (31 downto 0);
   --Init Chain "IC" signals
   SIGNAL S_NEU00_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU01_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU02_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU03_IC_1 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU00_IC_2 : STD_LOGIC_VECTOR (31 downto 0);
   SIGNAL S_NEU01_IC_2 : STD_LOGIC_VECTOR (31 downto 0);
   --EnN Signal Definitions
   SIGNAL EnN_Out_1_0 : STD_LOGIC;
   SIGNAL EnN_Out_1_1 : STD_LOGIC;
   SIGNAL EnN_Out_1_2 : STD_LOGIC;
   SIGNAL EnN_Out_1_3 : STD_LOGIC;
   SIGNAL EnN_Out_2_0 : STD_LOGIC;
   SIGNAL EnN_Out_2_1 : STD_LOGIC;
   SIGNAL EnN : STD_LOGIC;
BEGIN
   EnN <= '1';
   --Register Instantiations
   I_INPUT_0: entity work.RegGen Generic Map (32)
              Port Map (Clock, Reset, LVar, DataIn, S_INP00_0);
   I_INPUT_1: entity work.RegGen Generic Map (32)
              Port Map (Clock, Reset, LVar, S_INP00_0, S_INP01_0);
   I_INPUT_2: entity work.RegGen Generic Map (32)
              Port Map (Clock, Reset, LVar, S_INP01_0, S_INP02_0);
   I_INPUT_3: entity work.RegGen Generic Map (32)
              Port Map (Clock, Reset, LVar, S_INP02_0, S_INP03_0);
   --Neurons
   --Layer No : 1
   I_NEURON_1_0: entity work.PLIN04 Port Map (Clock, Reset, LConst, S_INP00_0,
                 S_INP01_0, S_INP02_0, S_INP03_0, DataIn,
                 S_NEU00_IC_1, S_NEU00_1, EnN, EnN_Out_1_0);
   I_NEURON_1_1: entity work.PLIN04 Port Map (Clock, Reset, LConst, S_INP00_0,
                 S_INP01_0, S_INP02_0, S_INP03_0, S_NEU00_IC_1,
                 S_NEU01_IC_1, S_NEU01_1, EnN, EnN_Out_1_1);
   I_NEURON_1_2: entity work.PLIN04 Port Map (Clock, Reset, LConst, S_INP00_0,
                 S_INP01_0, S_INP02_0, S_INP03_0, S_NEU01_IC_1,
                 S_NEU02_IC_1, S_NEU02_1, EnN, EnN_Out_1_2);
   I_NEURON_1_3: entity work.PLIN04 Port Map (Clock, Reset, LConst, S_INP00_0,
                 S_INP01_0, S_INP02_0, S_INP03_0, S_NEU02_IC_1,
                 S_NEU03_IC_1, S_NEU03_1, EnN, EnN_Out_1_3);
   --Layer No : 2
   I_NEURON_2_0: entity work.PLIN04 Port Map (Clock, Reset, LConst, S_NEU00_1,
                 S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU03_IC_1,
                 S_NEU00_IC_2, OUT00_out, EnN, EnN_Out_2_0);
   I_NEURON_2_1: entity work.PLIN04 Port Map (Clock, Reset, LConst, S_NEU00_1,
                 S_NEU01_1, S_NEU02_1, S_NEU03_1, S_NEU00_IC_2,
                 S_NEU01_IC_2, OUT01_out, EnN, EnN_Out_2_1);
END ARCHITECTURE RTL;
```

Fig. 14. VHDL code fragment produced by ANNGEN for the NetList shown in Fig. 13

the ANN. After the training phase, the weight values were saved to be used with the ANN formed with ANNGEN.

Later, ANNGEN was run with this example NetList, Template, and Library files and it produced VHDL code for ANN implementation. Fig. 14 partially shows the code. The actual code is more than 300 lines. Component definitions were removed in the Figure for simplicity.

The code was added to a newly created VHDL project in ISE to verify the correctness of it. First, a syntax check was done successfully. Later, it was synthesized and Register Transfer Level (RTL) view of the code is generated as shown in Fig. 15. In this RTL view, all instantiations and connections are checked against the original NetList and correctness of the code is verified.

Finally, the ANN formed by ANNGEN was simulated using the weight and input values. The output values calculated by ANNGEN formed ANN were compared to the values collected from MatLab formed ANN as shown in Table 2. The difference between the output values calculated by both ANNs is only in the last digits of the some records. These differences happen due to the

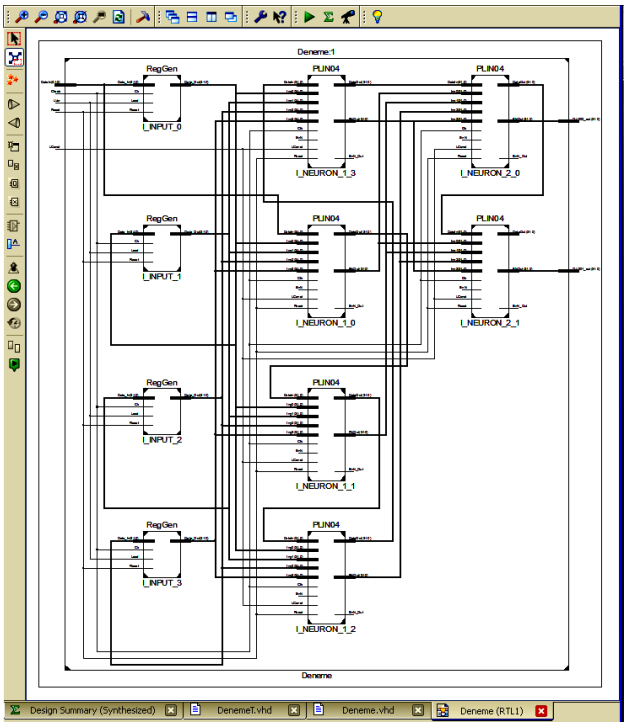


Fig. 15. RTL view of sample NetList generated by ISE®

Table 2. Comparison of Output values calculated by both ANNs

Output values calculated in MatLab env.		Output values calculated by ANNGEN-formed ANN		Differences	
Compress. Strength	Tensile Strength	Compressive Strength	Tensile Strength	Compressive Strength	Tensile Strength
0.3310065	0.3793904	0.3310065	0.3793904	0.0000000	0.0000000
0.1410547	0.1541590	0.1410547	0.1541590	0.0000001	0.0000000
-0.1161481	-0.0770886	-0.1161481	-0.0770886	-0.0000001	0.0000000
-0.1161481	-0.0770886	-0.1161481	-0.0770886	-0.0000001	0.0000000
0.2593421	0.2727866	0.2593421	0.2727866	0.0000000	0.0000000
0.3588953	0.2587150	0.3588953	0.2587151	0.0000000	-0.0000001
0.2403032	0.2655796	0.2403032	0.2655796	0.0000000	0.0000000
0.2403032	0.2655796	0.2403032	0.2655796	0.0000000	0.0000000
0.2446899	0.2570641	0.2446899	0.2570640	0.0000000	0.0000001
0.2193616	0.1874140	0.2193617	0.1874140	-0.0000001	0.0000000
0.4934929	0.4934361	0.4934930	0.4934362	-0.0000001	-0.0000001
0.2403032	0.2655796	0.2403032	0.2655796	0.0000000	0.0000000
0.1096080	0.0923398	0.1096080	0.0923398	0.0000000	0.0000000
0.2838376	0.2866638	0.2838376	0.2866639	0.0000000	-0.0000001
0.2943438	0.2703187	0.2943438	0.2703187	0.0000000	0.0000000
0.6122515	0.5004922	0.6122515	0.5004922	0.0000000	0.0000000
0.2077234	0.1313440	0.2077234	0.1313439	0.0000000	0.0000001
0.1832279	0.1174668	0.1832279	0.1174668	0.0000000	0.0000001

rounding errors and can be omitted. The mean square error between these two set of output values are 1.0833×10^{-15} and 2.1741×10^{-15} for compressive and tensile strength, respectively.

We also compared both ANNs in terms of calculation speed. The ANN in MatLab environment requires 4089700 ns to calculate 18 output values. The ANN formed by ANNGEN was synthesized for Xilinx's Virtex-6 chip

(xc6vlx75t). The synthesis tool reported that it can be clocked at 498.72 MHz. This means that if parallel inputs are used, the ANN can produce 498 million results per second. At this clock rate the ANN can calculate output values in 104.26 ns when serial input is used. The ANN formed by ANNGEN running on Virtex-6 is 39225.97 times faster than the ANN on MatLab environment.

7. Discussions and Conclusions

ANNs are utilized in several areas. They implemented on FPGA-based systems when high performance is needed. Implementing an ANN on FPGA is a time consuming process and requires experts. In this study, an ANN Generator (ANNGEN) has been developed to automated design process of ANN on FPGA-based system. ANNGEN was developed as a design tool that will help the implementation of ANN on FPGAs automatically.

Under normal conditions, designing and writing VHDL code for an ANN takes days or in some cases weeks depending on the structure of the ANN. ANNGEN reduces this design and implementation time significantly. It has been tested on several test cases using a sample neuron library to show its effectiveness. In each test case, it produced VHDL code successfully for the desired ANN design in less than a second without needing an expert person. Since the VHDL code produced by ANNGEN is error free, the debugging stage in ANN designs is also eliminated. In one specific test case, ANN formed using ANNGEN was compared to ANN formed in MatLab environment in terms of calculation accuracy and speed. Our result showed that ANN formed by ANNGEN calculated correct results with negligible rounding error and it very well outperformed the ANN in MatLab environment in terms of calculations speed.

Logic circuits defined using VHDL can also be implemented as ASIC. Since ANNGEN produces VHDL code, this VHDL code can also be used to implement ANNs as ASIC, too.

Currently, the ANNGEN library contains only a limited number of neurons. It has to be enriched by adding new neuron definitions. It was structured to recognize new neurons in the library, easily.

References

[1] Diessel O and Milne G, "Hardware compiler realizing concurrent processes in reconfigurable logic," *IEE Proc.-Comput. Digit. Tech.*, 148-4/5, 152-162, 2001.

[2] Gupta V, Khare K and Singh RP, "FPGA Design and Implementation Issues of Artificial Neural Network Based PID Controllers," *Proc. ARTCOM '09*, 860-862, 2009.

[3] Zulfikar MY, Abbasi SA and Alamoud ARM, "FPGA Based Walsh and Inverse Walsh Transforms for Signal Processing," *Elektron Elektrotech*, 18(8), 3-8, 2012.

[4] Lin Z, Dong Y, Li Y and Watanabe T, "A Hybrid Architecture for Efficient FPGA-based Implementation of Multilayer Neural Networks," *IEEE Circuits and Systems* (*APCCAS*), 616-619, 2011

[5] Tamulevičius G, Arminas V, Ivanovas E and Navakauskas D, "Hardware Accelerated FPGA Implementation of Lithuanian Isolated Word Recognition System," *Elektron Elektrotech,* 3(99), 52-62, 2010.

[6] Polata Ö and Yıldırım T, "FPGA implementation of a General Regression Neural Network: An embedded pattern classification system," *Digital Signal Process,* 20, 881-886, 2010.

[7] Sahin İ, "A 32-Bit Floating-Point Module Design for 3D Graphic Transformations," *Sci Res Essays,* 5(20), 3070-3081, 2010.

[8] Gomperts A, Ukil A and Zurfluh F, "Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications," *IEEE T Ind Inform,* 7(1), 72-88, 2011.

[9] Chorowski J and Zurada JM, "Extracting Rules from Neural Networks as Decision Diagrams," *IEEE T Neural Network,* 99, 1-12, 2011.

[10] Yildiz O, "Döviz Kuru Tahmininde Yapay Sinir Ağlarının Kullanımı," *MS Thesis*, Eskisehir Osmangazi University, 2006.

[11] Sahin I, "A Compilation Tool for Automated Mapping of Algorithms onto FPGA Based Custom Computing Machines," *Dissertation*, North Carolina State University, Raleigh-USA, 2002.

[12] Levinskis A, "Convolutional Neural Network Feature Reduction using Wavelet Transform," *Elektron Elektrotech*, 19(3), 61-64, 2013.

[13] Togawa N, Yanagisawa M and Ohtsuki T, "Maple-opt: A Performance-Oriented Simultaneous Technology Mapping, Placement, and Global Routing Algorithm for FPGA's," *IEEE T Comput Aid D*, 17(9), 803-818, 1998.

[14] Goda BS, McDonald JF, Carlough SR, KrawczykJr TW and Kraft RP, "SiGe HBT BiCMOS FPGAs for fast reconfigurable computing," *IEE P-Comput Dig T,* 147(3), 189-194, 2000.

[15] Koyuncu İ, "A Matrix Multiplication Engine for Graphic Systems Designed to run on FPGA Devices," *MS Thesis*, Duzce University, 2008.

[16] Hauck S, "The Roles of FPGA's in Reprogrammable Systems," *P IEEE*, 86(4), 615-638, 1998.

[17] Sahin I and Koyuncu I, "Design and Implementation of Neural Networks Neurons with RadBas, LogSig, and TanSig Activation Functions on FPGA," *Elektron Elektrotech* 4(120), 51-54, 2012.

[18] Çavuşlu MA, Karakuzu C, Şahin S and Karakaya F, "Yapay Sinir Ağı Eğitiminin IEEE 754 Kayan Noktalı

Sayı Formatı İle FPGA Tabanlı Gerçeklenmesi," *GomSis*, 2008.
[19] Çavuşlu MA, Altun H and Karakaya F, "Plaka Yeri Tespiti İçin Kenar Bulma, Bit Tabanlı Öznitelik Çıkartma ve YSA Sınıflandırıcısının FPGA Üzerine Uyarlanması," GomSis 2008.
[20] Goldstein SC, Schmit H, Budiu M, Cadambi S, Moe M and Taylor RR, "PipeRench: a Reconfigurable Architecture and Compiler," *Computer*, 33(4), 70-77, 2000.
[21] Crookes D, Benkrid K, Bouridane A, Aiotaibi K and Benkrid A, "Design and implementation of a high level programming environment for FPGA-based image processing," *IEE P-Vis Image Sign*, 147, 377-384, 2000.
[22] Harkin J, McGinnityand TM and Maguire LP, "Partitioning methodology for dynamically reconfigurable embedded systems," *IEE P-Comput Dig T*, 147, 391-396, 2000.
[23] Benrekia F, Attari M, Bermak A and Belhout K, "FPGA implementation of a neural network classifier for gas sensor array applications," *6th International Multi-Conference on Systems, Signals and Devices,* 1-6 2009.
[24] Weinstein RK and Lee H, "Architectures for high-performance FPGA implementations of neural models," *J Neural Eng*, 3, 1-21, 2006.
[25] Uçar A, "FPGA implementation of a neural network for Turkish phoneme classification," *MS Thesis*, Hacettepe University, 2007.
[26] Bastos JL, Figueroa HP and Monti A, "FPGA implementation of neural network-based controllers for power electronics applications," *IEEE Twenty-First page Annual Applied Power Electronics Conference and Exposition*, 1443-1448, 2006.
[27] Funabiki N, Yoda M, Kitamichi J and Nishikawa S, "A gradual neural network approach for FPGA segmented channel routing problems," *IEEE T Syst Man Cy B*, 29, 481-489, 1999.
[28] Sahin I and Akkaya A, "ANNDES: An artificial neural network design and education software," *ICITS 2011 5th International Computer and Instructional Technologies Symposium*, 2011.

Ibrahim Sahin He has a M.S. Degree from Old Dominion University, Norfolk - USA, and a Ph.D. Degree from North Carolina State University, Raleigh-USA. Both degrees are from Department of Electrical and Computer Engineering. He is a faculty member of Faculty of Technology, Duzce University in Turkey. His main research interests are digital system design, reconfigurable computing, and hardware-software co-design. He is also interested in network protocols and computer graphics. Currently, he is teaching several courses at Duzce University.

Namik Kemal Saritekin He has a M.S. Degree from Duzce University, and a Ph.D. Degree from Abant Izzet Baysal University, Bolu-Turkey. He is a physics teacher in Duzce Science High School. His main research interest are artificial neural networks, reconfigureable computing and superconductors.