



**ÇUKUROVA UNIVERSITY**  
**ENGINEERING FACULTY**  
**DEPARTMENT OF COMPUTER ENGINEERING**

**GRADUATION THESIS**

**Generic Self Organizing Map implementation  
on FPGA using VHDL**

**By**

2019556460 – Ömer AKGÜL

**Advisor**

Assoc. Prof. Mustafa ORAL

January 2021

**ADANA**



**ÇUKUROVA UNIVERSITY**  
**ENGINEERING FACULTY**  
**DEPARTMENT OF COMPUTER ENGINEERING**

**GRADUATION THESIS**

**Generic Self Organizing Map implementation  
on FPGA using VHDL**

**By**

2019556460 – Ömer AKGÜL

**Advisor**

Assoc. Prof. Mustafa ORAL

January 2021

**ADANA**

## **Abstract**

In data mining applications, Self-Organizing Maps are used variously. With help of Self-Organizing Maps, classification and pattern recognition are done easily. When working on multidimensional data sets, Self-Organizing Maps can decrease dimensions. The main disadvantage of Self-Organizing Maps is the training time. As it requires comparing and modifying each neuron in the map for each iteration, it is a time-consuming method to apply. In this study, using a field-programmable gate array (FPGA) device implemented a Self-Organizing Map. VHDL language is used to implement the design on FPGA. To test the performance, the same algorithm written on C++ was tested on a modern CPU and compared with the FPGA implementation. FPGA implementation is tested on an ARTY A7 board with a 100mhz clock. When the results were compared, around a %79 of performance increase was measured. The project is shared on GitHub (Akgül, 2021).

## Index

Abstract .....	i
Symbols and Abbreviations Used .....	iv
List of Figures.....	v
List of Tables.....	vi
1. Introduction .....	7
2. Literature Review .....	8
3. Self-Organizing Map Algorithms .....	9
4. FPGA Implementation of SOM.....	11
4.1. Top Design.....	11
4.2. Random Weight Generation .....	12
4.2.1. External Noise Generator .....	14
4.3. Controller.....	14
4.3.1. Get Input Vector Process.....	15
4.3.2. Find BMU and Train Process.....	15
4.4. KMap (Kohonen Map) .....	17
4.4.1. Init Process.....	18
4.4.2. Find BMU Process .....	18
4.4.3. Train Process.....	18
4.4.4. Get Out Process .....	19
4.5. Serial Entity.....	19
4.6. Serial Interface .....	20
4.7. Testing .....	21
5. Performance Comparison with Generic CPU Based Implementation .....	22
Results and discussion.....	23

Future Work .....	24
References .....	25
Acknowledgements .....	27
Resume .....	28

**Symbols and Abbreviations Used**

FPGA: Field-programmable gate array

VHDL: Very High-Speed Integrated Circuit Hardware Description Language

SOM: Self-Organizing Map

CPU: Central Process Unit

LED: Light Emitting Diode

EEPROM: Electrically Erasable Programmable Read-Only Memory

LFSR: Linear-Feedback Shift Register

RAM: Random Access Memory

DRAM: Distributed Random-Access Memory

BRAM: Block Ram

BMU: Best Matching Unit

FIFO: First in First Out

FF: Flip Flop

## List of Figures

Figure 1 Top Design Block Diagram.....	11
Figure 2 LFSR Design .....	13
Figure 3 Random Weight Generator .....	14
Figure 4 Controller Block Diagram.....	15
Figure 5 Find BMU and Train Process Flow Diagram .....	16
Figure 6 KMap Entity.....	18
Figure 7 Python Script and Its Output .....	20
Figure 8 R Output Visualized.....	21
Figure 9 Random Map Visualized.....	21
Figure 10 SOM on CPU Result .....	22
Figure 11 SOM on FPGA Result.....	22

**List of Tables**

Table 1 Status Signals.....	12
-----------------------------	----



## 1. Introduction

In many data-related applications, pattern recognition and clustering are important steps. To achieve pattern recognition and clustering, neural networks are successfully used. There are supervised and unsupervised approaches for those neural network-based pattern recognition and clustering algorithms. Supervised neural network methods require the data to be tagged properly. This tagging work is done by humans and relatively expensive to do. On the other hand, unsupervised neural networks work without any intervention and group the data without supervision. Also, dimension reduction can be done on multidimensional data.

Kohonen Self-Organizing Map (SOM) (Kohonen, 1982) is one of the most suitable neural network models for unsupervised applications. SOM can show relationships between the input values by organizing a map according to similarities between input values.

There are many different approaches to lower computing time. Some focus on reducing computational complexity by using more efficient algorithms (Chaudhary et. al., 2014). Also, different implementations are used to create faster SOM. Many implementations are working on general-purpose CPUs as well as on specially designed hardware.

Even if general-purpose CPUs are easy to program and more accessible, implementing a complex algorithm on general-purpose CPUs slows down the algorithm because each computational step is done with combinations of predefined instruction sets and takes longer to execute compared to special hardware. Designing sequential application-specific hardware increases efficiency.

## 2. Literature Review

The discovery of SOMs (Kohonen, 1982), made clustering and dimension reduction way easier. Also found a wide application area in engineering such as preprocessing and feature extraction, systems analysis, statistical pattern recognition, visuomotor control of robot arm, telecommunication telecommunications, etc. (Kohonen et. al., 1996). Most of these applications are using a general-purpose CPU implementation. Due to the high complexity of the SOM algorithm and not optimized for the SOM workflow of CPUs, implementing SOM on CPUs is inefficient.

Research on FPGA technology and neural networks shows that FPGA implementations of neural networks are lowering the cost significantly (Zhang, et al., 2016) (Zhang and Prasanna, 2017). This approach might also be applied in SOM algorithms.

There are many hardware implementations of SOM done before. In the early days, researchers implemented parallel SOM hardware accelerators using VLSI Technology (Porrman, et. al., 2003). With improvements in FPGA technology, more advanced SOM hardware implementations were developed (Hikawa and Maeda, 2015) (Appiah, et al., 2009). Many different approaches to implement the SOM algorithm have been researched for years. This thesis is focused on implementing a standard and generic SOM algorithm.

### 3. Self-Organizing Map Algorithms

SOM is a special type of feature map discovered by the author (Kohonen, 1982) which is to form a topologically correct nonlinear map from a high dimensional data set to a lower-dimensional map with unsupervised learning algorithms.

SOM contains a neuron array with an  $n$ -dimensional weight vector. Here,  $n$  corresponds to the feature size of the inputs. Weights are adjusted throughout the learning process according to input vectors. (Kohonen et. al., 1996)

A general algorithm for SOM usually follows these steps (Vesanto and Alhoniemi, 2000):

- 1) Initialize: Assign each neuron a random weight vector. Define the neighborhood distance and learning rate.
- 2) Calculate distances between a randomly chosen input and each neuron.
- 3) Determine the closest neuron to the input.
- 4) Adjust weights of the neurons in the neighborhood of the best matching unit according to distance from the BMU and the current iteration count.
- 5) Decrease the neighborhood distance and repeat step 2 for  $N$  iterations.

In step 2, to calculate the distance between each neuron, the high dimensional Euclidean distance formula (Tabak, 2014) is used. After calculating the distances for each neuron, the distances are compared and in step 3, the neuron with the lowest distance to the input is determined as the BMU.

Step 4 is the training step. In this step, to calculate the new weight of each neuron, Equation 1 is used. In this equation,  $W()$  is the weight function.  $\Theta(t)$  is the

neighborhood rate, which is calculated with Equation 2,  $L(t)$  is the learning rate which is calculated by Equation 3, and  $V(t)$  is the input (ai-junkie).

$$W(t+1) = W(t) + \Theta(t)L(t)(V(t) - W(t))$$

*Equation 1*

$$\Theta(t) = \exp\left(-\frac{dist^2}{2\sigma^2(t)}\right) \quad t = 1, 2, 3, \dots$$

*Equation 2*

$$L(t) = L_0 \exp\left(-\frac{t}{\lambda}\right) \quad t = 1, 2, 3, \dots$$

*Equation 3*

## 4. FPGA Implementation of SOM

In this thesis, VHDL is used to implement a basic SOM on the ARTY A7-35T FPGA board. The design is divided into four main parts. Each part is designed as an individual entity and all the entities are put together in the top design. For non-repetitive pseudo-random number generation, an ATTiny85 processor is used as an external noise generator. To get inputs from a serial device and transmit the output vector to the serial device there is a serial interface.

Each entity is designed as generic entities. It is possible to modify the SOM for different map sizes and feature sizes. The input vector size is determined dynamically from the serial interface.

### 4.1. Top Design

The general flow of the implementation can be seen in Figure 1. Four signals are interfering with external devices. RX and TX signals are used to interfere with the serial device over the serial protocol. The input vector is received from the serial device and when training ends the output map is sent to the serial device.

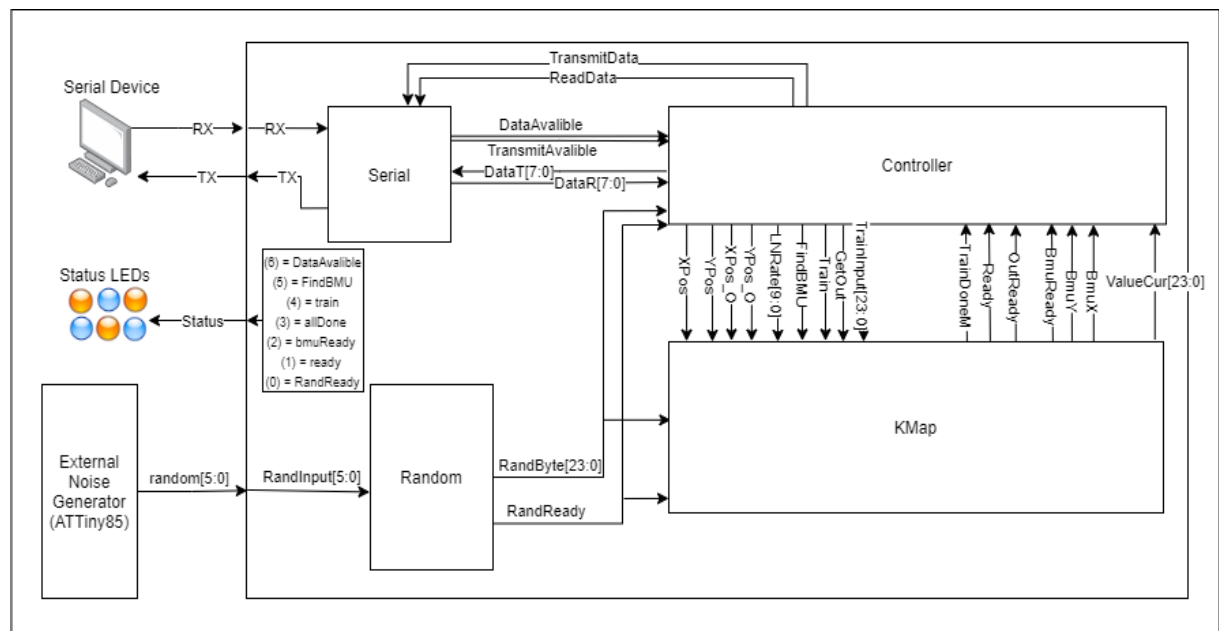


Figure 1 Top Design Block Diagram

There is a total of 6 status LEDs used to display the current situation of the system running to debug any possible error that occurs. The Status LEDs and corresponding signals with their source are listed in Table 1.

*Table 1 Status Signals*

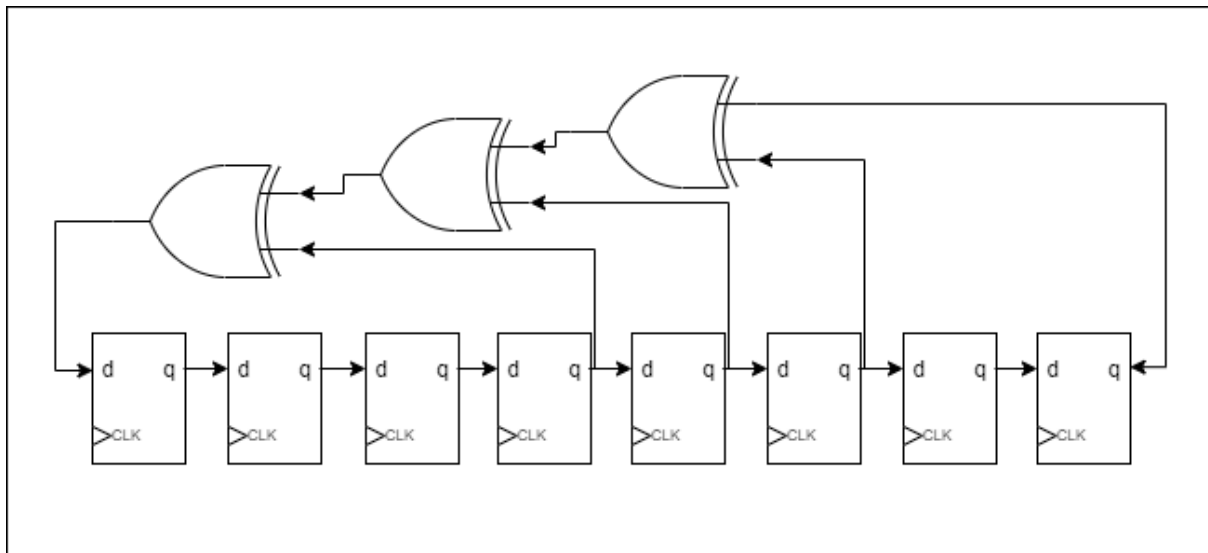
Led Number	Signal	Source of the signal
0	RandReady	Random Entity
1	ready	KMap Entity
2	bmuReady	KMap Entity
3	allDone	Controller Entity
4	train	Controller Entity
5	FindBMU	Controller Entity
6	DataAvalible	Serial Entity

For the random entity to work, it requires an external noise generator. In this project, an ATTINY85 microprocessor is used as a noise generator for its internal EEPROM. It generates a 6-bit random output.

#### **4.2. Random Weight Generation**

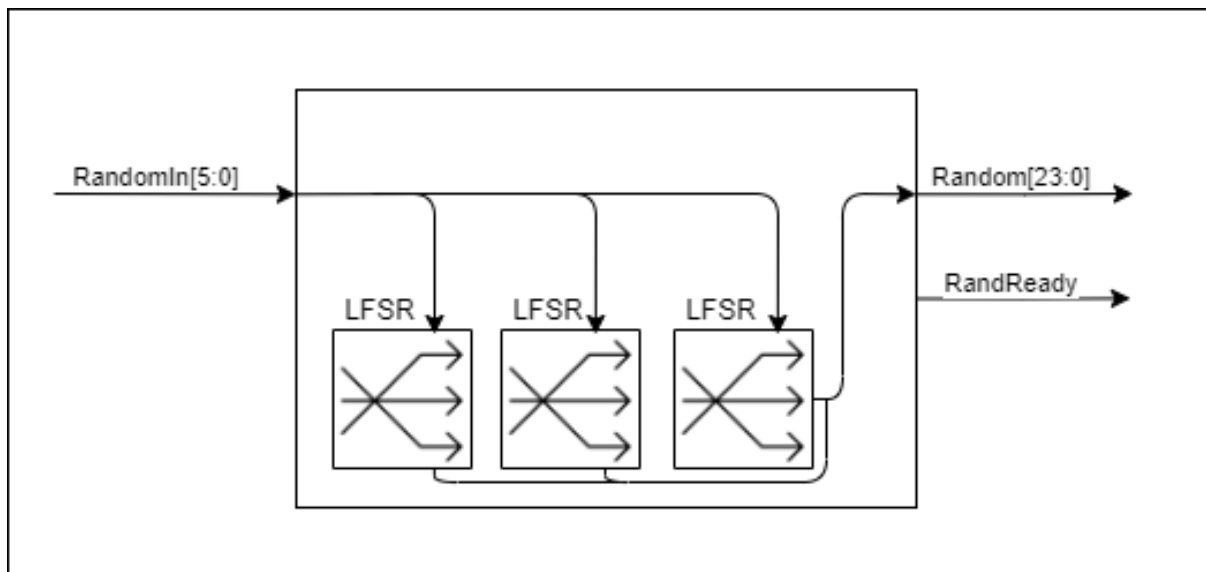
To fill the weights of each neuron in the map with random values, there is a random byte generation entity. It generates  $(8*N)$  bits random byte. N is for feature size. As a default, N is set to 3.

To generate random output, there are N number of LFSRs based pseudo number generators (Alfke, 1996). LFSR is a special type of shift register. Every clock cycle, it shifts byte one to the right. Instead of a regular shift register, it feeds the leftmost bit with some bits XORed. As seen in Figure 2, in this application, bits XORed are (0), (2), (3), and (4).



*Figure 2 LFSR Design*

The problem with using LFSR with FPGA is that, generated random output is predictable and repetitive for each run because the initial state is determined by a constant. Using regular LFSR results in generating the same pseudo-random map each run. To prevent that problem, it is necessary to feed the initial state with a noise source. In Figure 3 the general working principle is visualized. There are 3 LFSRs as a default. They are generating a 24-bit random bit vector. To initialize the registers, a 6-bit bit vector is fed through the registers. As the random vector is only 6 bits and there are 24 bits to initialize, the registers are initialized sequentially in 6 steps. Each time the random vector changes, the initializing process continues. When initializing done, the RandReady signal is set.



*Figure 3 Random Weight Generator*

To achieve that, an ATTINY85 microprocessor is used as a random noise generator. This microprocessor has six digital pins and a 512-byte EEPROM.

#### **4.2.1. External Noise Generator**

ATTINY85 as a noise generator is an LFSR pseudo-random number generator. The reason why there is an external noise generator is to generate a random initialization seed. This seed needs to be non-repetitive. To achieve that, the initialization seed is changed each run. When the ATTINY85 powers on, the seed is read from the internal EEPROM, and then the EEPROM is rewritten with a changed version of the initial seed. This way, each time the device is powered, the seed changes. This noise generator could also be used as a random number generator but because it runs at 16 MHz and the FPGA runs at 100 MHz using this external noise generator alone would slow down the initialization process.

#### **4.3. Controller**

The controller is the entity that controls the flow of the system. There are three main sequential processes in the controller. To use in processes, there are two lookup tables which has precalculated function solutions. To store inputs from the serial interface, there is an input vector DRAM.



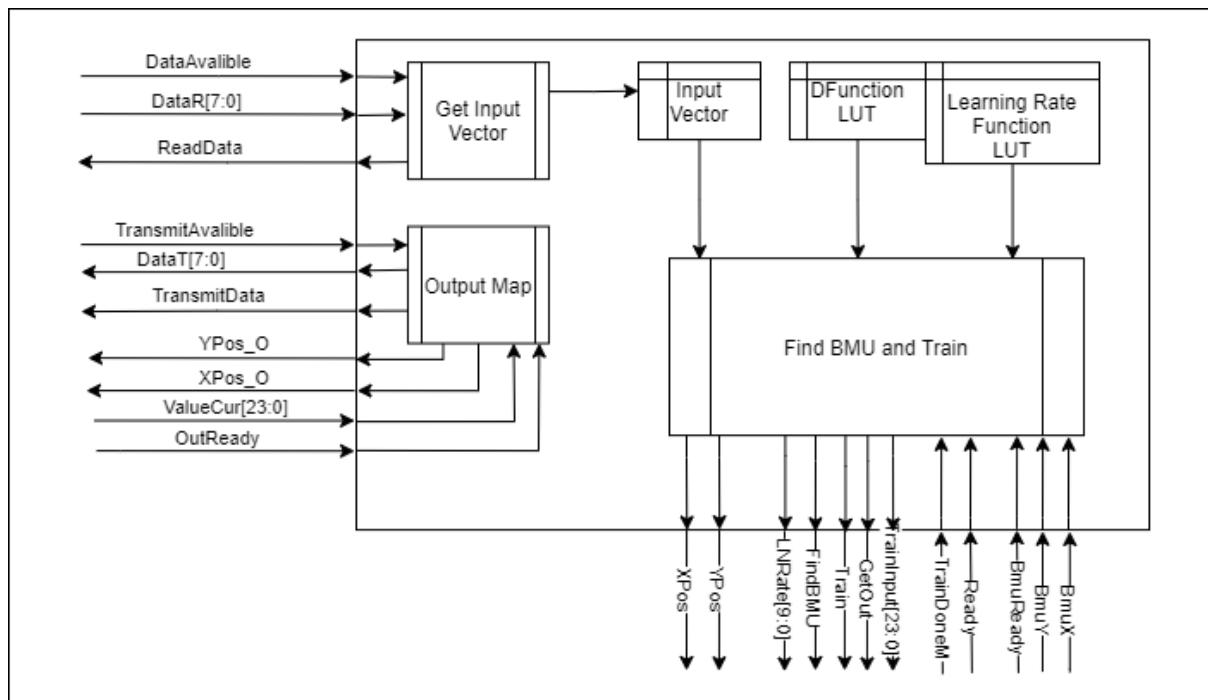


Figure 4 Controller Block Diagram

#### 4.3.1. Get Input Vector Process

The Input vector sent from the serial device with serial byte arrays is deserialized and written into the input vector DRAM. In the initial state, after a reset, the process waits until the first byte is received. The first byte after a reset is always considered as input count. Following bytes are feature bytes of the inputs. For each input, the process is expecting N bytes. N is feature size and as a default set to three. For every N byte is received, an input vector is read. When the input count is equal to the count of the input vectors, the input read process is done and a flag is set.

#### 4.3.2. Find BMU and Train Process

The controller is controlling the KMap entity to accomplish training. The flowchart for control logic is shown in Figure 5. Each clock cycle, the control process checks if the input vector is read, and if the train is not finished. If the input vector is ready and the train is not done, the train state machine starts. This state machine has five main steps. The initial state is WaitS. In this state, the process checks if the current iteration is less than the iteration count. If it is not, then that means the training process is done and the TrainDone flag is set. If the current iteration is less than the iteration

count, then the state is set to bmu\_wS state and a random input from the input vector is chosen.

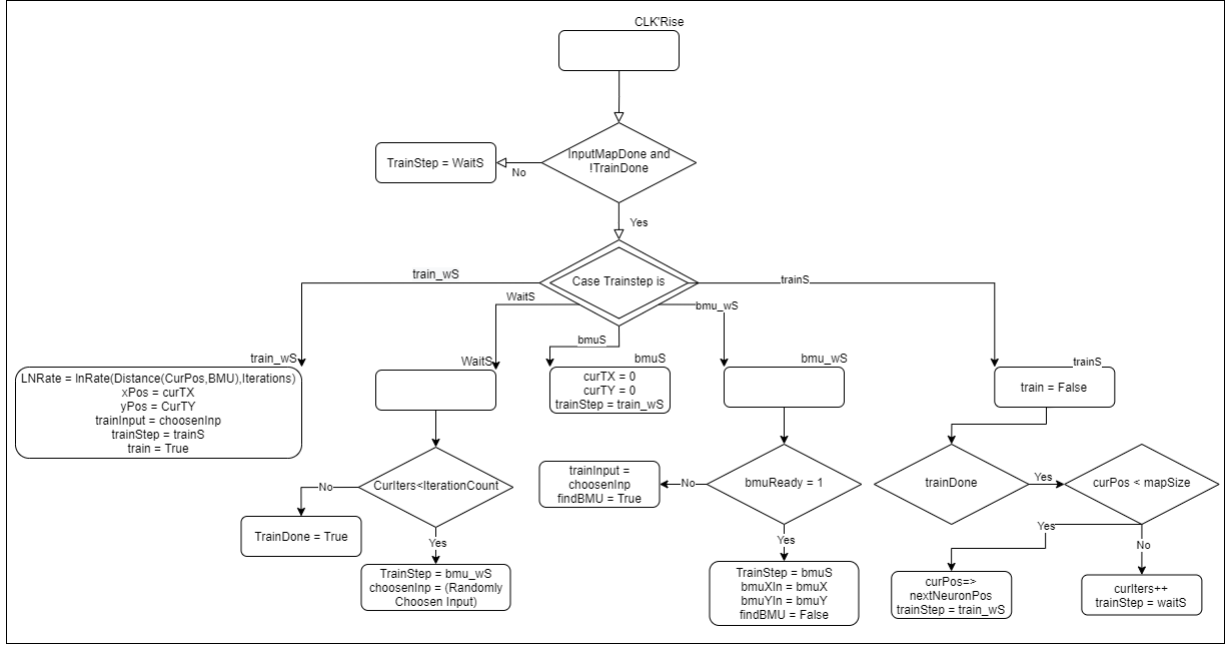


Figure 5 Find BMU and Train Process Flow Diagram

The bmu\_wS and bmuS states are for detecting the best matching unit. First, bmu\_wS runs to send findBMU signal to KMap. This signal stays “True” until BMU is found. The bmuReady signal is set when the BMU is found. So bmu\_wS state is checking if bmuReady is set. If it is not set, we set the findBMU signal and pass chosenInput to trainInput. If bmuReady is “True” then bmuXIn and bmuYIn is set to bmuX and bmuY, trainStep is set to bmuS, and findBMU signal is set to “False”. With bmuS state, the process is prepared for train steps by setting curTX and curTY to 0 and then trainStep is set to train\_wS.

Train\_wS state is to send the signal to the KMap to start the training process. It sets xPos and yPos to curTX and curTY to determine which neuron to train. The training algorithm is adjusting the weights of the neuron according to Equation 1 (ai-junkie).  $L(t)$  is the learning rate and calculated with Equation 3. It is starting from  $L_0$  and decreases each iteration.  $L_0$  is set to 0.1 by default.  $\Theta(t)$  is the neighborhood rate and calculated with Equation 2. It changes according to the iteration and the distance between BMU and the current neuron.

The training function is calculated in the KMap entity but the  $L(t)$  and  $\Theta(t)$  are calculated in the controller and passed to the KMap with the LNRate signal. Calculating exponential functions on FPGA is a complex problem. Instead of calculating it in realtime, the results for 100 iterations and 100x100 map size are precalculated with the MATLAB Software. Then the results are written into LUTs in FPGA to access them when needed. Train\_wS state sets LNRate signal with D Function and LearningRate Function LUTs according to current iteration and the distance between BMU and current neuron training. After setting the LNRate signal, it sets the training signal and sets trainStep to trainS.

TrainS state is to set current neuron training. It waits until the trainDone signal is set and if it is set then checks if the curPos signal is smaller than map size to know if all the neurons are trained. If not, it increments the iteration count and sets the trainStep to waitS. If the curPos signal is smaller than the map size, then increments curPos and sets the trainStep to train\_wS.

This process is looping until the training process is done for each iteration.

#### **4.4. KMap (Kohonen Map)**

KMap entity is part of the design where the Kohonen Map is initialized and processed. As seen in Figure 6, there are four processes and a BRAM in the KMap entity. The BRAM is used to form a neuron weight map. It has 10000 addresses which are 24 bits wide. X and Y positions of the neurons are converted to addresses by multiplying X-1 by 100 and adding the result to Y. 24-bit width is for the weights. Each weight is 8 bits wide and as default, by taking the specification count as 3, the weight vector for each neuron is 24 bits wide.

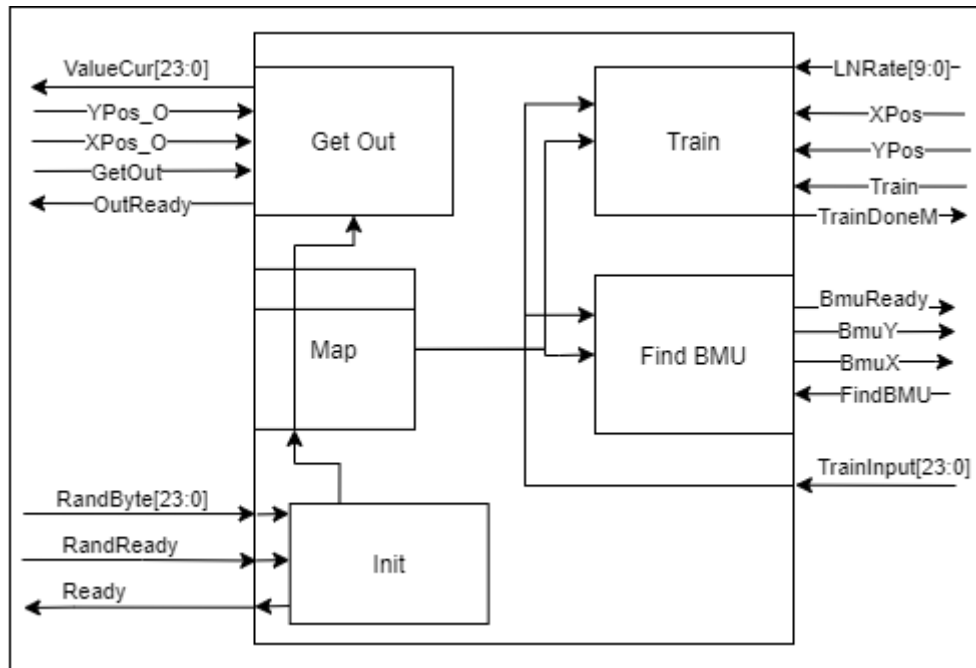


Figure 6 KMap Entity

#### 4.4.1. Init Process

After a reset, the initialization of the KMap starts with the Init process. Initialization is setting all weights of all neurons to random numbers. To do that, the init process needs a random input. RandReady signal is set when there is a random input ready in the RandByte signal. Init process sets the weights from address zero to 10000 to RandByte one by one each cycle. When the process is done, the Ready signal is set.

#### 4.4.2. Find BMU Process

SOM algorithm requires to find the best matching neuron for the input each iteration. The find BMU process gets the TrainInput signal and calculates the distance between each neuron. To calculate the distance, it is using the high dimensional Euclidean distance formula (Tabak, 2014). After calculating for each neuron, by comparing each distance, the process finds the best matching neuron. The position of the neuron is passed to BmuX and BmuY signals and after that, the BmuReady signal is set.

#### 4.4.3. Train Process

The training process is assigned to train a neuron at a given position with given input and a given learning rate. When the train signal is on the rising edge, the training

process starts. The position of the neuron is given by XPos and YPos signals. Using Equation 1 sets the neuron weights to (original weight + LNRate (input weight - weight)). When the training process is done, sets the TrainDoneM signal.

#### **4.4.4. Get Out Process**

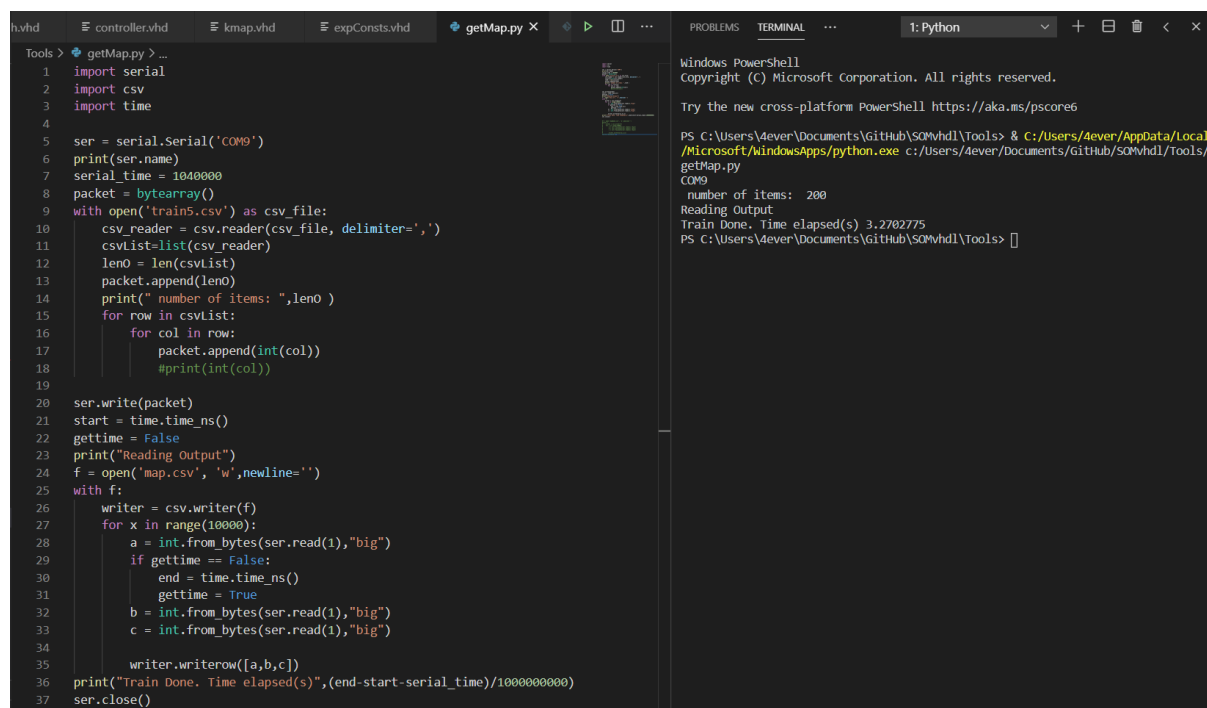
The get-out process is to output the weight vector of a given neuron. It reads an input at positions XPos\_O and YPos\_O when the GetOut signal is set then puts the weight vector on the ValueCur signal. After that, sets the OutReady signal.

### **4.5. Serial Entity**

The serial entity is the entity that handles serial communication with serial devices. It reads the input vector from the serial device and writes the final map to the serial bus when the training is done. It uses an 8 data bit, 0 parity, and 1 end bit serial protocol in 9600 baud rates (Osborne, 1980). There is a FIFO device in the receiver to prevent incoming data from being missed. The transmitter is transmitting the data on the DataT signal when the TransmitData signal is set and while transmitting the data, sets the TransmitAvalible signal to the ground. When data is received, it is put into the FIFO device. When there is data in the FIFO, the DataAvalible signal is set. When the ReadData signal is set, the FIFO device pops, and the output of the FIFO is put to the DataR signal.

## 4.6. Serial Interface

The serial interface is the way for the FPGA to communicate with the outside world. Using this interface, it is possible to send the input vector and receive the map data. To control communication, there is a python script. This script reads inputs from a .csv file and after receiving the map, writes the output to output.csv file. It also keeps track of time and calculates the precise training time. The Python script and its output are shown in Figure 7.



```

Tools > getMap.py > ...
1 import serial
2 import csv
3 import time
4
5 ser = serial.Serial('COM9')
6 print(ser.name)
7 serial_time = 1040000
8 packet = bytearray()
9 with open('train5.csv') as csv_file:
10     csv_reader = csv.reader(csv_file, delimiter=',')
11     csvList=list(csv_reader)
12     len0 = len(csvList)
13     packet.append(len0)
14     print(" number of items: ",len0 )
15     for row in csvList:
16         for col in row:
17             packet.append(int(col))
18             #print(int(col))
19
20 ser.write(packet)
21 start = time.time_ns()
22 gettime = False
23 print("Reading Output")
24 f = open('map.csv', 'w',newline='')
25 with f:
26     writer = csv.writer(f)
27     for x in range(10000):
28         a = int.from_bytes(ser.read(1),"big")
29         if gettime == False:
30             end = time.time_ns()
31             gettime = True
32         b = int.from_bytes(ser.read(1),"big")
33         c = int.from_bytes(ser.read(1),"big")
34
35         writer.writerow([a,b,c])
36 print("Train Done. Time elapsed(s)",(end-start-serial_time)/1000000000)
37 ser.close()

```

```

Windows PowerShell
Copyright (c) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

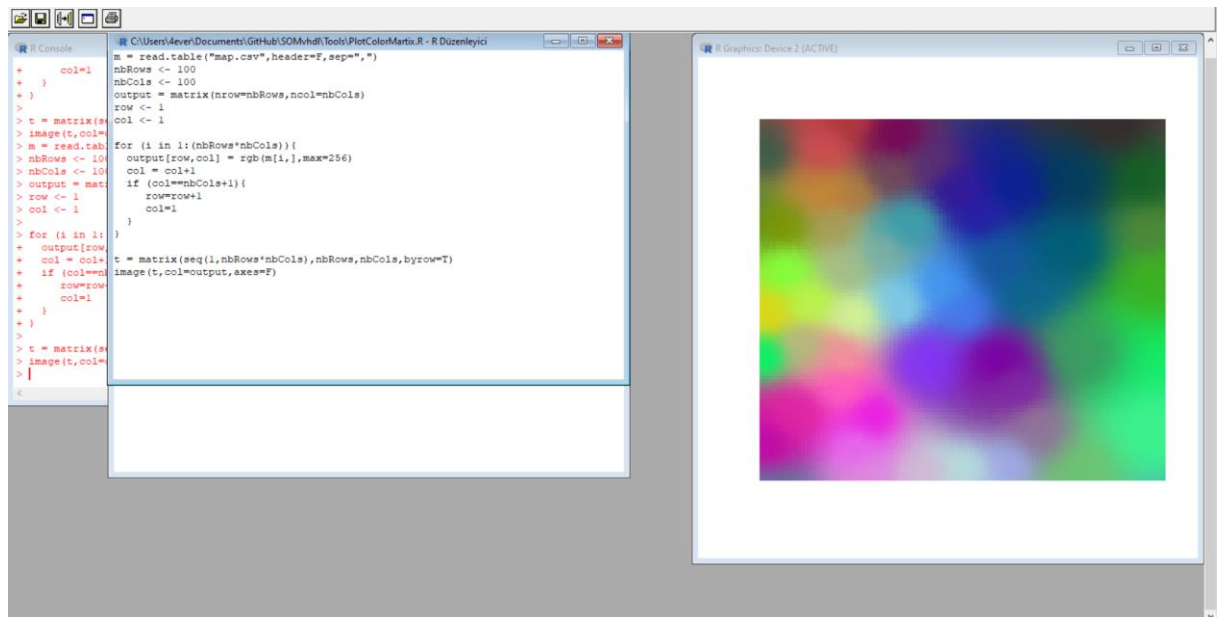
PS C:\Users\Aever\Documents\Github\SOMvhdl\Tools> & C:/Users/Aever/AppData/Local/
/Microsoft/WindowsApps/python.exe c:/Users/Aever/Documents/Github/SOMvhdl/Tools/
getMap.py
COM9
number of items: 200
Reading Output
Train Done. Time elapsed(s) 3.2702775
PS C:\Users\Aever\Documents\Github\SOMvhdl\Tools>

```

Figure 7 Python Script and Its Output

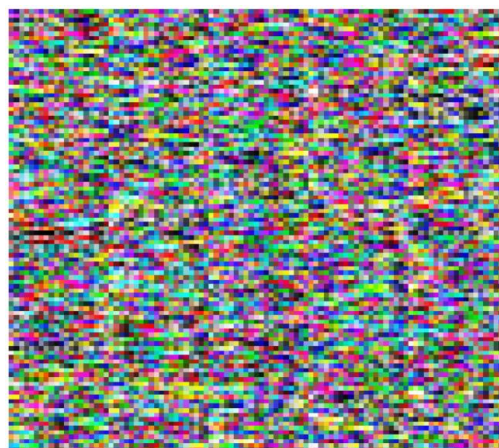
## 4.7. Testing

To test the results, RGB colors are used as input, and the expected result is that a color map with similar colors is closer together. Visualizing the output is done with help of the R Project. Using an R script to visualize the output vector (Ciriano, 2014). The script and example visualized map is seen in Figure 8.



*Figure 8 R Output Visualized*

There is also a mod to test the randomly generated map. When the BTN1 is pressed before training, the randomly generated map is sent through the serial interface. The results of the random test are shown in Figure 9. There is no noticeable pattern seen in the random map.



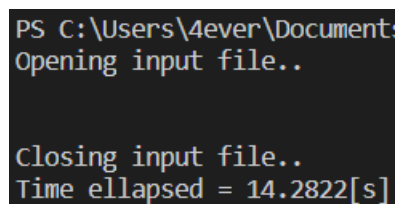
*Figure 9 Random Map Visualized*

## 5. Performance Comparison with Generic CPU Based Implementation

To test the performance of the FPGA implementation, a CPU implementation of the design created using C++. The results are compared for 1000 iterations and 100 inputs for a 100x100 map. Both the CPU and FPGA implementations are using the same algorithm.

For the CPU test, a PC with an i7-8565U processor runs at a 1.80 GHz base and a 1.99 GHz max clock frequency. The test is done in single-core as the FPGA implementation is also single-threaded. For the FPGA implementation test, an ARTY A7 FPGA board is used in 100 MHz clock frequency.

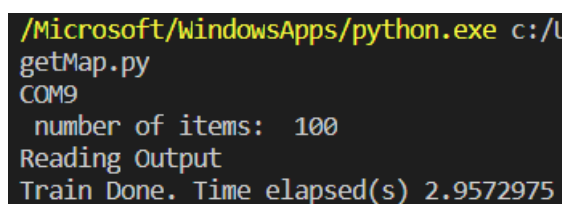
The result of the CPU implementation of SOM is shown in Figure 10. Overall training process took 14.28 seconds. File reading and writing times are not counted in this time measurement. On the other hand, as seen in Figure 11, the training process took only 2.95 seconds with the FPGA implementation of the same SOM algorithm. The serial transmission time is not calculated in the measurement to be fair.



```
PS C:\Users\4ever\Documents
Opening input file..

Closing input file..
Time elapsed = 14.2822[s]
```

*Figure 10 SOM on CPU Result*



```
/Microsoft/WindowsApps/python.exe c:/U
getMap.py
COM9
number of items: 100
Reading Output
Train Done. Time elapsed(s) 2.9572975
```

*Figure 11 SOM on FPGA Result*

This result shows that the FPGA implementation without any algorithm improvements, performed %79.34 better than the CPU implementation.



## **Results and discussion**

Hardware implementations of neural networks are widely used and have lots of advantages on cost (Chen, et al., 2012). In many data-related applications, Self-Organizing Maps are used for pattern recognition and clustering. Using FPGA technology, an application-specific implementation of SOM is made and achieved a %79.34 performance increase. The performance improvement shows that, even if the FPGA runs at a much lower clock frequency, an application-specific hardware implementation makes a big improvement in terms of time.

It is possible to use improved algorithms with FPGA and improving the performance much better. KMap can be divided into smaller parts and this way, parallel processing of the map can be done.

## **Future Work**

An ARTY A7 board is used in this thesis. It is an entry-level FPGA board and has a limited number of RAM, FF, and LUTs. To implement a bigger neuron map and more specifications, more RAM and LUTs are necessary. A better FPGA can be used in future research.

The algorithm implemented in this implementation is the most basic version of the SOM algorithm. A better algorithm can be developed and implemented such as a multithreading approach. There is a high potential in application-specific hardware implementations and this thesis proves it.

## References

- ai-junkie. (n.d.). *SOM tutorial*. Retrieved from ai-junkie.com: <http://www.ai-junkie.com/ann/som/som4.html>
- Akgül, Ö. (2021). SOMVhdl. Zenodo. doi:10.5281/zenodo.4445469
- Alfke, P. (1996, July 7). Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators. *Xilinx Application Note*.
- Appiah, K., Hunter, A., Meng, H., Yue, S., Hobden, M., Priestley, N., . . . Pettit, C. (2009). A Binary Self-Organizing Map and its FPGA Implementation. *Proceedings of International Joint Conference on Neural Networks*, (pp. 164-171). Atlanta, Georgia, USA,.
- Chaudhary, V., Bhati, R., and Ahlawat, A. (2014). An efficient Self-Organizing Map (E-SOM) learning. *International Journal of Computational Intelligence*, 7(5), 963–972.
- Chen, T., Chen, Y., Duranton, M., Guo, Q., Hashmi, A., Lipasti, M., . . . Temam, O. (2012). BenchNN: On the Broad Potential Application Scope of Hardware Neural Network. *IEEE International Symposium on Workload Characterization (IISWC)*.
- Ciriano, I. C. (2014). *SOMcpp*. Retrieved from <https://github.com/isidro/SOMcpp>
- Hikawa, H., and Maeda, Y. (2015). Improved Learning Performance of Hardware Self-Organizing Map Using a Novel Neighborhood Function. *IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS*, vol. 26, no. 11, pp. 2861-2873.
- Kohonen, T. (1982). Self-Organized Formation of Topologically Correct Feature Maps. *Biological Cybernetics*, 43, 59-69.
- KOHONEN, T., OJA, E., STMULA, O., VISA, A., and UNGAS, J. (1996). Engineering applications of the Self-Organizing Map. *Proceedings of the IEEE*, 84(10), 1358–1384.
- Osborne, A. (1980). *An Introduction to Microcomputers Volume 1: Basic Concepts*. Osborne-McGraw Hill Berkeley California USA.
- Porrman, M., Witkowski, U., and Rückert, U. (2003). A Massively Parallel Architecture for Self-Organizing Feature Maps. *IEEE TRANSACTIONS ON NEURAL NETWORKS*, 14(5), 1110–1121.
- Tabak, J. (2014). *Geometry: The Language of Space and Form*. Facts on File math library, Infobase Publishing.
- Vesanto, J., and Alhoniemi, E. (2000). Clustering of the Self-Organizing Map. *IEEE Transactions on Neural Networks*, 11(3), 586–600.

- Zhang, C., and Prasanna, V. (2017). Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*.
- Zhang, C., Wu, D., Sun, J., Sun, G., Luo, G., and Cong, J. (2016). Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. *Proceedings of the 2016 International Symposium on Low Power Electronics and Design - ISLPED '16*.

**Acknowledgements**

Special thanks to my friend Suna AYHAN who helped with the algorithm problems a lot. She also gave me lots of design ideas.

Thanks a lot, to my friend Mehmet Melih Özkurt for his support in the process of writing this thesis.

Thanks to my advisor Assoc. Prof. Mustafa ORAL for leading me to the artificial intelligent world and helping me with the project.

I would like to thank my grandfather Süleyman Günver for guiding me to programming field.

## Resume



Ömer Akgül was born in 1999 in Bursa, İnegöl. During his elementary school years, he learned web programming. At the age of 15, he managed to sell the rights of an application he developed for Android. Later he entered Bursa High School for Boys. When he came to the 10th grade, he was accepted to an international class at a school called "Ferdinand Porsche Gymnasium". He was educated by staying with a German family for 1 year. Later, when he returned to Turkey, he started doing orienteering sport. He also made projects for Tubitak competitions and participated in many competitions. After high school years, he got accepted at the Electrical and Electronics Department of Karadeniz Technical University. At the university, he first entered the K-Tech Team hybrid vehicle team and designed a telemetry system for 3 different vehicles. Then he produced an ROV for the Teknofest ROV competition with the Creatiny team he founded with his friends. With this team, he won 1st place for 2 consecutive years. He also developed an 8-bit processor using logical gates and exhibited it at Teknofest. After studying this department for 2 years, he changed his mind and wanted to transfer to computer engineering. He applied to the Computer Engineering department of Çukurova University and was accepted. He completed his internship in Aselsan and, within the scope of his internship, he developed the circuit that he previously printed on the PCB using logical gates and implemented it from the beginning with VHDL.