POLITECNICO DI TORINO

Master Degree Course in Electronic Engineering

Master Thesis

High-Level Synthesis for Ultralow Power FPGAs

Supervisor:
Prof. Mihai Lazarescu

Candidate:
Davide Salusso (257688)

Academic year 2019-2020

Acknowledgements

The path I followed was not easy, but it taught me what it means to have constancy and perseverance in doing things. I thank my friends who helped me even just by asking about what I was working on or how the exams were going. I thank my family who allowed me to undertake this path and helped in times of difficulty. I thank my girlfriend, Sara, the real engine of my strength, a reference point that gave me the motivation to improve myself more and more.

Thank you all.

Contents

List of Tables V

List of Figures VII

1 Introduction 1
- 1.1 Detection techniques for people indoors . . . 1
- 1.2 Capacitive sensing compared to other techniques . . . 2
- 1.3 Data processing with neural networks on the sensor . . . 3
- 1.4 High level neural networks synthesis . . . 3

2 Capacitive Sensors 5
- 2.1 Operating principles . . . 5
- 2.2 Localization and identification applications . . . 7
- 2.3 Types of reading front end . . . 8
- 2.4 Data processing on the sensor . . . 14

3 Neural networks for sensor data processing 15
- 3.1 What is a neural network . . . 15
  - 3.1.1 Learning method . . . 16
  - 3.1.2 Advantages and disadvantages of a neural network . . . 18
- 3.2 Types of neural networks . . . 18
- 3.3 Multi-Layer Perceptron neural networks . . . 19
- 3.4 Description of the neural network used . . . 20

4 Very low power FPGA 25
- 4.1 FPGAs types and performances compared . . . 25
  - 4.1.1 Lattice FPGA features . . . 26
  - 4.1.2 Microsemi FPGAs features . . . 26
  - 4.1.3 Other low power FPGAs . . . 26
  - 4.1.4 Microcontrollers . . . 27
- 4.2 FPGA programming mode . . . 29
  - 4.2.1 Manual programming . . . 29

4.2.2 High level synthesis 30
4.2.3 Optimizations 30

5 High level synthesis for FPGA 33
5.1 High level synthesis flow 33
5.2 Limits of Vivado HLS 35
5.3 Code used for neural network and optimizations 36
5.3.1 Synthesis diretives 38
5.3.2 Design space exploration 40

6 Experimental results 43
6.1 Target devices 43
6.2 Tools 43
6.2.1 Radiant Software 2.0 44
6.2.2 Libero SoC v11.9 46
6.3 Workflow 47
6.4 FPGA manual programming results 52
6.4.1 Manual implementation on Lattice ICE40UP5K 52
6.4.2 Manual implementation on Microsemi AGLN250 52
6.4.3 Manual implementation on Microsemi M1AGL600 53
6.4.4 Manual implementation on Microsemi M1AGL1000 53
6.4.5 Achieved manual results 54
6.5 Results and discussion 55
6.5.1 DSP mapping on Lattice FPGA 58
6.5.2 Lattice ICE40UP5K FPGA 61
6.5.3 Microsemi AGLN250 FPGA 67
6.5.4 Microsemi M1AGL600 FPGA 72
6.5.5 Microsemi M1AGL1000 FPGA 77
6.5.6 Comparison with objectives and manual flow 81

7 Conclusion 89

A Appendix 91
A.1 Code of the MLP neural network (.c++) 91
A.2 Code of the MLP neural network (.h) 94
A.3 Code of DSP (.c++) 94
A.4 Code of DSP (.h) 95

Bibliography 97

List of Tables

2.1 Variation of dielectric properties in different body tissues at different frequencies. 6
4.1 Features of ultralow-power FPGAs used in the reference programming. 27
4.2 Features of low power FPGAs used in the reference programming. . 27
4.3 Results of manual programming on MCUs. 28
5.1 Summary of the advantages and disadvantages of the directives used. 40
6.1 Performance and resource on FPGAs in the manual programming experiment. 55
6.2 Resource occupation of the test of DSPs on Lattice FPGAs. 61
6.3 Directives used in the solutions for the Lattice FPGA. 63
6.4 Performance of the solution 12 of the Lattice FPGA ICE40UP5K. . 63
6.5 Performance of the solution 13 of the Lattice FPGA ICE40UP5K. . 64
6.6 Performance of the solution 14 of the Lattice FPGA ICE40UP5K. . 65
6.7 Report data for the solution of Lattice FPGA. 66
6.8 Comparison of resources occupation between Vivado and Radiant. . 67
6.9 Directives used in the solutions for the Microsemi AGLN250 FPGA. 69
6.10 Performance of the solution 6 of the Microsemi FPGA AGLN250. . 69
6.11 Performance of the solution 7 of the Microsemi FPGA AGLN250. . 70
6.12 Power and energy comparison for solution 6 and 7 of Microsemi FPGA AGLN250 between VCD and vectorless analysis. 71
6.13 Report data for the solutions of Microsemi AGLN250 FPGA. 71
6.14 Directives used in the solutions for the Microsemi M1AGL600 FPGA. 72
6.15 Performance of the solution 11 of the Microsemi FPGA M1AGL600. 74
6.16 Performance of the solution 12 of the Microsemi FPGA M1AGL600. 74
6.17 Performance of the solution 13 of the Microsemi FPGA M1AGL600. 75
6.18 Power and energy comparison for solution 11, 12 and 13 of Microsemi M1AGL600 between VCD and vectorless analysis. 76
6.19 Report data for the solution of Microsemi M1AGL600 FPGA. . . . 76
6.20 Directives used in the solutions for the M1AGL1000 of Microsemi FPGA. 77
6.21 Performance of the solution 16 of the Microsemi FPGA M1AGL1000. 79
6.22 Performance of the solution 17 of the Microsemi FPGA M1AGL1000. 80

6.23 Power and energy comparison for solution 16 and 17 of Microsemi FPGA M1AGL1000. 80
6.24 Report data for the solution of Microsemi M1AGL1000 FPGA. . . . 81
6.25 Performance of manual and HLS programming. 82
6.26 Resource occupation of manual and HLS programming. 82

List of Figures

2.1 Principal four capacitance of a load mode capacitive sensor. 5
2.2 Capacitive sensor based on 555 integrated circuit. 7
2.3 A circuit used to determine the capacitance. 9
2.4 Resistor network for low-pass filter and experimental setup. 10
2.5 Front end interface used to measure the difference in amplitude between the output of two RC low-pass filters and experimental setup. 11
2.6 Front end interface used to reduce the environmental noise and experimental setup. 13
2.7 Main blocks of sensor node and base station. 14
3.1 Typical Architecture of neural network. 16
3.2 Structure of a neuron. 17
3.3 Behavior of ReLU function. 20
3.4 Flow of data augmentation. 21
3.5 Structure of multi-layer perceptron neural network. 22
5.1 Vivado HLS flow design to test the operation. 34
5.2 Flow of directives used to find the best performance. 42
6.1 Workflow of Radiant Software 45
6.2 Workflow of Microsemi Software 46
6.3 Workflow followed to obtain .bin file for programming. 48
6.4 Part of the Power Analysis to see the power of the inputs. 50
6.5 Part of the Power Analysis of Libero. 51
6.6 Ideal scheduling of the operation of the neural network for the manual programming. 53
6.7 Trend of the clock period based on the modification at the HLS level with no directives. 57
6.8 Trend of the clock period based on the modification at the HLS level with directives. 58
6.9 Netlist Analyzer and Technology View of DSP used as MAC write manually 60
6.10 Report of Radiant on area used of DSP write manually. 60
6.11 Datapath of circuit with one DSP written in HLS. 61

6.12 Inference time and Energy for different 14 Solution for Lattice FPGA ICE40UP5K. 62
6.13 Inference time and Energy for different 7 Solution for Microsemi FPGA AGLN250. 68
6.14 Inference time and Energy for different 13 Solution for Microsemi FPGA M1AGL600. 73
6.15 Inference time and Energy for different 17 Solution for Microsemi FPGA M1AGL1000. 78
6.16 Scheduling find on Vivado HLS in the Analysis view for the solution of Lattice FPGA 83
6.17 Scheduling of the multiplications to obtain one output of the neuron of the neural network for the solution for Microsemi FPGA AGLN250. 85
6.18 Scheduling find on Vivado HLS in the Analysis view for the solution of Microsemi FPGA M1AGL600. 86
6.19 Scheduling find on Vivado HLS in the Analysis view for the solution of Microsemi FPGA M1AGL1000. 88

Abstract

The subject of this thesis is the capacitive sensors used in locating people indoors, a field that is becoming very important especially for the health care of the elderly. This sensors have numerous advantages: cost, power, privacy and ease of installation while their main disadvantage is the steep loss of sensitivity by increasing the distance between the sensor and the person. To reduce the cost and installation complexity, the sensors are used in load mode, i.e. only one plate of the capacity is needed because the other plate is the person's body along with the surrounding environment.

In this thesis instead of processing the data on board the sensor with a microcontroller we used an ultralow-power FPGA like a hardware accelerator, and to process the data we use a neural network of the Multi-layer perceptron type synthesized with the high-level-synthesis (HLS) method. We will then compare the results obtained in terms of inference time, energy, power, clock frequency, clock cycles and area compared to the data obtained in a manual implementation [1]. The neural network (NN) used is of the multi-layer perceptron type, in which there are two hidden layers formed by eight neurons each and an output layer formed by a neuron. The NN takes six 16-bit inputs and generates one output of four-bit. The C code of this NN will be synthesized using Vivado HLS, which also allows to insert directives to optimize the generated code. Subsequently the created VHDL code will be compiled using the FPGA tools, which are: Radiant tool for ICE40UP5K of the Lattice family and Libero tool for AGLN250, M1AGL600 and M1AGL1000 of the Microsemi family. These FPGAs, that have all different resources, have been chosen as they are ultra low-power. So, the code created by HLS, is implemented on these FPGAs. The FPGA-specific synthesis tool may not be the completely compatible with the Xilinx Vivado HLS generated code.

The method followed to collect the data to be compared with those found by manual programming has been: start from HLS where the C code is synthesized by also adding directives to optimize it, and they have been added in such a way as to being able to explore multiple solutions, starting from a low level of parallelism and then increasing it from solution to solution and seeing how the various performance and resource metrics changed. After that, through Co-simulation, again on Vivado-HLS, it was verified that the synthesized code still worked as expected. From here, the two tools of the two families of FPGAs were used to implement the code on them and collect the various data. Many solutions have been created for the different FPGAs, and there are different implementations of the same solution as we tried to change the speed (clock period) imposed on HLS and see what effect it would have on the final implementation.

The trade-off is between performance and resource occupation, controlled using synthesis directives, and the ensuing Vivado HLS overhead. In addition to the area occupied by the optimizations, there is also the area occupied by the state machine that Vivado HLS creates for the control of the project. The maximum parallelism may not be achieved always, due to resource limitations. This is also due to the imperfect compatibility of the HDL code created by HLS with the two synthesis tools of the families of target FPGAs.

The results of the manual programming of the NN achieved by the reference manual implementation were not achieved for all the FPGAs, in particular for the smallest ones, AGLN250, and the M1AGL600, even if in any case the values of the data found are not far from the target ones. Excellent results were obtained for the Lattice FPGA, with, the lowest power and energy of both HLS and manual programming 5.23 mW and 5.78 nJ are obtained, against 6.24 mW and 5.79 nJ find in the manual implementation. This is because on this FGPA there are blocks that perform dedicated operations, DSP, and in particular, multiplications, which are the operations that require more area, have been assigned to these blocks thus not consuming any other area and thus allowing to add directives to obtain maximum parallelism. Also the inference time find is good, 1.10 μs, which is very close to that find in manual programming, 0.93 μs. With the Microsemi M1AGL1000 FPGA, the largest, it was possible to achieve excellent results, in particular the lowest inference time, 0.86 μs among all FPGAs with HLS, while in reference programming 0.74 μs. Also a good energy was obtained, 17.27 nJ against 19 nJ of the manual programming.

Chapter 1

Introduction

Identifying people indoors is extremely useful in a lot of fields, especially in the health care of the elderly. Over the years many techniques have been adopted but they always had a counterpart that made their use counterproductive, for example techniques based on image, video and gait recognition have very good performance, but are very expensive and violate privacy of the user, or techniques in which the person have to brought a device to be located, but this can be uncomfortable and can be forgotten. It turned out that capacitive sensors could be used, which had great benefits in terms of cost and consumption. Their big problem was that they lost reliability with increasing distance of the person to be located. To improve the data processing on board the sensor, it was decided to use a neural network installed on a microcontroller or on a FPGA (Field-Programmable Gate Array), which can be implemented through manual programming, therefore the programmer describes the network by creating the HDL (Hardware Description Language) file himself, or through high level synthesis (HLS), then starting from the network code written at high level (C or C++), the tool created the HDL file by itself.

1.1 Detection techniques for people indoors

The techniques for detecting people in buildings or rooms have become extremely important for a wide range of applications such as home or building automation, security, video games with virtual reality and in trying to make life of elderly people more confident and assisted.

Over the years, the scientific community has proposed many solutions for this research, where the main objectives were to create a device easy-to-install, inexpensive, low-power, small and, above all, respecting the privacy of those who use it. Another fundamental requirement is that they are not harmful to the human body, ultrasound transceivers [2] have been created for the localization of people or objects, but exposure for a long time to ultrasound can be harmful in fact to

prevent these emissions many countries have imposed a limit on the body's exposure to ultrasound, so as to prevent health problems [3]. The results provided by these detections have to be reliable, D.Yang et al. proposed a technique based on pyroelectric infrared (PIR) sensors and an accessibility map of the indoor environment [4], but the results provided by these sensors were not optimal as they are prone to errors given by the environment and sources of heat, and to obtain good results the system need more sensor, so the cost increases. Other techniques based on Wi-Fi [5] or Bluetooth [6], on the other hand, require people to carry a device with them, which can be both unreliable and uncomfortable, as well as the fact that they can often forget to wear it. Techniques that use devices that instead violate people's privacy terms are those proposed by Y.Zheng et al. [7] and G. Lu et al. [8], who respectively proposed a technology based on an indoor vision-based navigation system and a system based on thermal images for localization, and these techniques, in addition to not respecting privacy, are expensive.

The alternative to these techniques are capacitive sensors, which can meet most of the indicated requirements.

1.2 Capacitive sensing compared to other techniques

Capacitive sensors are used in many short-range applications such as the smartphone touchscreen, in tunnel-effect scanning microscopes, in editing systems and also in musical instruments. Thanks to their properties, it has been thought of being able to use them also in the recognition of people, in their identification and localization. The type of capacitive sensors used for this purpose operates in load-mode using a single plate while other plate is made by the environment and human body. This sensor measures the capacity of the human body at different frequencies, and this measurement will be influenced by the composition of it (height, gender, weight, muscles, etc.) and environment which must be compensated. Given these differences, the electrical and dielectric properties of the users will change, and from them people can be identified. So being that they can be built using simply with a plate, they are very easy to install (can be install near the light switches [9] or in the bed [10] to monitor the sleep disorders), they are not expensive (like system based on thermal imaging for indoor localization [11]), they do not invade the user's privacy like [11] and do not require any additional devices that must be worn like technique based on Wi-Fi or Bluetooth. Thanks to their ease of installation there are other uses of capacitive sensors, at lower ranges than localization in a room, for example they can be installed in a computer so that they can interact during a game [12], they can be used in beds to determine sleep patterns [10] or on a chair to prevent the patient from falling [13]. They can also be used to detect different types of walks [14].

The downside of these sensors is the loss of sensitivity with increasing distance. They become very sensitive to environmental factors such as electrostatic and magnetic noise, humidity or temperature. From [15] it turns out that at a distance greater than 10 times the size of the sensor, the variation of the capacity measurement is below 0.01%, therefore hardly distinguishable from noise, this latter in fact affect the measurements of the voltage across the capacitance for example, but not affect the capacitance itself, that depends on the electric and dielectric properties, and geometry of the sensor and environment.

1.3 Data processing with neural networks on the sensor

The accuracy of the capacitive sensor is an essential quality to obtain good results, to improve it can can be used Machine Learning algorithms, especially Neural Networks (NNs), such as 1-D Convonutional NNs (CNNs) or Long-Short Term Memory NNs (LSTM). In [16] a study is carried out on the recognition of people using 1D CNNs, a Recursive Neural Network (RNN) and a hybrid architecture with various sensors, in [17] LSTM networks are used together with 1D CNN to find and to classify rare acoustic sounds, in [18] the authors use LSTM networks for localization in closed environments using light and magnetic sensors.

Neural networks can be implemented in software (SW) using microcontrollers (MCUs) or in hardware (HW) using FPGA, and the goal of both implementations is to accelerate the neural network by trying to consume as little energy as possible. In the article by Braga et al. [19] the performance, accuracy, use of resources and power consumption of an NN implemented both in SW (using Xilinx Microblaze microprocessor) and in HW (using FPGA of the Xilinx family) are analyzed. The authors of [20] propose a method to optimize accuracy by minimizing the area for a Multi-Layer Perceptron (MLP) neural network implemented on FPGA. To try to improve the critical path (CP), and consequently improve operating frequency, of an MLP neural network, Bahoura et al. [21] propose an HW implementation with the use of the pipeline. In [22], Zhai et al. propose a low latency real-time system for gas classification using an accurate MLP neural network implemented on Xilinx FPGA with fixed point arithmetic. In the study of Marwen et al. [1] the performance of an MLP neural network is tested on two low power MCUs and subsequently on four different ultralow-power FPGAs.

1.4 High level neural networks synthesis

The HLS is the link between hardware and software domain. The tools used for HLS, starting from the high level code (written in C, C++ or SystemC) and generates the code written at low level, register transfer level (RTL), in Verilog or VHDL,

which can then be implemented on FPGA. HLS provides benefits both from the HW point of view, in fact it improves productivity because it allows the HW programmer to work at a higher level of abstraction and at the same time to create HW with optimization, but also from the point of view of the software programmer, allowing to speed up the calculation of the algorithms on the future FPGA used. For these reasons, various tools [23] have been created over the years for this purpose, like Vivado HLS which synthesizes high-level code and then implements it on the FPGA of the Xilinx family.

The experiment performed is based on the use of this tool to synthesize an neural network of the MLP type, used in the field of capacitive sensors to increase its accuracy, and try to compare the performances obtained in terms of power, energy, inference time, operating frequency and latency, with the performance obtained in [1] in which HLS is not used for the creation of the NN, but only low level language (RTL). Both experiments were performed on 4 different ultralow-power FPGAs that do not belong to the Xilinx family (one of them is from the Lattice iCE40 family while the other three are from the Microsemi IGLOO family), then the RTL code generated by Vivado HLS, which should be implemented on FPGAs of the family mentioned above, will not perfectly adapt to the design of the ultralow-power FPGAs used for the experiment, which have been selected from those of Xilinx for their low static power [1].

Chapter 2

Capacitive Sensors

For locating people indoors, many techniques have been used, including the capacitive sensors. They have several advantages, first of all the low power consumption. There are various front-end configurations used each with a different purpose.

2.1 Operating principles

The capacitive sensors can be used to localize and monitor persons or objects. They are used in load-mode, where the human body is used as second plate of capacitance, and this mode needs only a single transducer plate that form with the human body and the environment several capacitances, in particular four: C_{sg}, capacitance between sensor and ground, C_{bg}, capacitance between the human body and ground, C_{se}, capacitance between sensor and environment and C_{sb}, capacitance between sensor and human body, like Figure 2.1 (taken from [24]) shows.

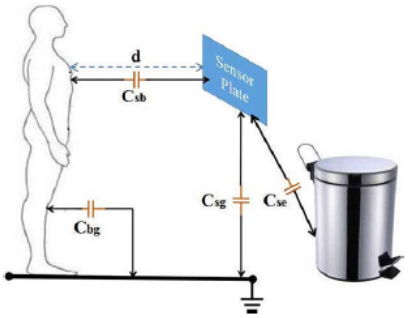


Figure 2.1. Principal four capacitance of a load mode capacitive sensor, taken from [24]: C_{sg}, between sensor and ground, C_{bg}, between the human body and ground, C_{se}, between sensor and environment and C_{sb}, between sensor and human body.

To identify the person is taken the value of C_{sb} at different frequencies, in particular, the person dielectric and electric properties change with different composition of the human body, in fact the values of permittivity (ϵ) and conductivity (σ) of the human body change with the frequencies of the electric fields, so, thanks to this difference, is possible to recognize the members of a small group of people like a family. In this regard, Gabriel [25] made a study on how the dielectric properties change according to the parts of the body, they are summarized in Table 2.1, at the frequencies used by our sensors.

Table 2.1. Variation of dielectric properties (ϵ and σ) in different body tissues (muscle, fat, bones and breast fat) at different frequencies, taken from [25].

Frequency	Muscle		Fat		Bones		Breast fat	
(kHz)	ε_r	σ (S/m)	ε_r	σ (S/m)	ε_r	σ (S/m)	ε_r	σ (S/m)
5	6.0×10^4	4.0×10^{-1}	1.2×10^3	1.8×10^{-2}	1.4×10^3	3.0×10^{-3}	1.0×10^3	11.0×10^{-3}
10	3.0×10^4	5.0×10^{-1}	1.0×10^3	1.8×10^{-2}	1.2×10^3	3.0×10^{-3}	1.3×10^2	11.0×10^{-3}
20	1.0×10^4	6.0×10^{-1}	1.4×10^2	1.8×10^{-2}	1.1×10^3	3.0×10^{-3}	1.1×10^2	11.0×10^{-3}
40	1.4×10^3	6.0×10^{-1}	1.1×10^2	1.9×10^{-2}	1.05×10^3	3.0×10^{-3}	1.0×10^2	11.0×10^{-3}
80	1.1×10^3	6.0×10^{-1}	1.0×10^2	1.9×10^{-2}	1.0×10^3	4.0×10^{-3}	1.4×10^1	11.0×10^{-3}
160	2.9×10^2	7.0×10^{-1}	1.5×10^1	1.9×10^{-2}	1.9×10^2	4.0×10^{-3}	1.3×10^{-1}	11.0×10^{-3}

The capacitance measured (in Farad) between two parallel infinitely large plates with a small distance between them is given by the formula (2.1):

$$C = \frac{\varepsilon_0 k A}{d} \tag{2.1}$$

where:

- k: is the relative dielectric permittivity of the material between plates, that in the case of free space is 1.
- ϵ_0: is the absolute dielectric permittivity of free space (equal to 8.854 $\times$ 10^{-12} F/m).
- d: is the distance between the capacitors plates.
- A: is the area of capacitors plates.

Capacitance plate strongly depends both on the area of the capacitive sensor and on the distance at which the measurement is carried out since the sensor works in load-mode.

So C_{sb} depends from the distance of the human body and the sensor, and if this distance grows up, the measurement it is no longer accurate because will be affected by environmental factors like noise, temperature and humidity. In particular, the noise can limit the sensor range to 10-15 times the diagonal of their plate [26].

A possible type of circuit, used to measure the capacitance between the transducer and the human body while changing the distance, is the one shown in Figure 2.2, taken from [27], which is formed by an oscillator based on a 555 integrated circuit (IC) configured as an astable multivibrator, for which the oscillation frequency is given by the formula (2.2):

$$\text{Frequency} = \frac{1}{0.7\,(R_1 + 2R_2)\,C} \tag{2.2}$$

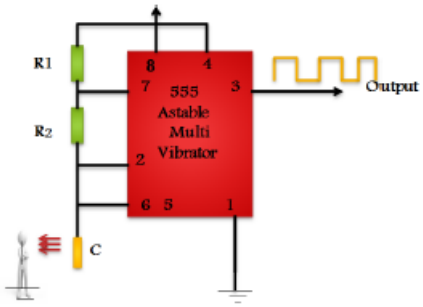


Figure 2.2. Structure of a capacitive sensor, taken from [27], based on 555 integrated circuit configured as an astable multivibrator.

2.2 Localization and identification applications

In addition to capacitive sensors, other techniques are used for locating people indoors. For example, J. Rivera-Rubio et al. did a study on a video-based localization method in which the person brought the camera with him and the localization carried out through solutions based on simultaneous localization and mapping (SLAM) algorithms [28].

Other techniques that require that the person to be located wear a device are those based on radio frequency such as GPS, Bluetooth, RFID and Smart phone [29], [30], [31], [32]. These could be problematic as not always people, especially the elderly, remember to wear it, in addition to fact that maybe the device is not comfortable to wear. Instead, techniques based on image, video and gait recognition are often used. Ding et al. presents a pose-invariant facial recognition technique [33], or in [34] a technique has been proposed for facial recognition through the use of deep learning. While for gait recognition a system based on arbitrary view transformation model [35] can be used. There are many techniques based on these principles and the results that they give are very good, their problem is that they are expensive and hardly susceptible for long-term battery-powered operations. Furthermore, being based on images or videos, they violate the user privacy. Other

techniques for the identification of the person are: wireless techniques, such as [36], ultra-wideband sensors [37] and techniques that use footstep induced structural vibrations [38].

Capacitive sensors, on the other hand, thanks to their low cost, their easy installation method and their low power consumption, have been designed for identification and tracking of the person in indoor environment. A.R. Akhmareh et al. studied a system based on capacitive sensors for indoor tagless identification in a room [27]. Using the capacitance of the sensors he built an RC oscillator whose frequency is dependent on the distance of the human body, the author also had to apply many noise reduction techniques as at a long distance the sensor loses sensitivity and therefore the movements are no longer captured correctly, in particular at distances greater than the dimensions of the transducter [39] it becomes sensitive to environmental factors. The performance of this localization method depends strictly on the sensor performance. A [15] study was also done to improve the sensitivity of the sensor at distances greater than 10 times its diameter. As explained in the Section 2.1, a capacitance, C_{sb}, is formed between the human body and the sensor, which depends on the dielectric properties of the body, and based on this, Iqbal et al. [40] presents an identification method based on capacitive sensors, in which the capacitance of the human body at different frequencies is measured and it will be linked to the different structure of the body. Several improvements have been made to this method in [41] that have improved the resolution. In [27] another method has been proposed for the identification of a single person inside a room, also improving the sensitivity on measurement at long distances. The method is based on the indirect measurement of the transducer capacitance by measuring the frequency through an oscillator, after which a baseband digital filter is used to attenuate the measurement noise, finally through localization algorithms the position of the person is found using all the data of the various sensors inserted in the room. Based on this work, in [42], the performance of the localization system is studied using different Machine Learning (ML) algorithms, which are very important if the improvement of the performance of the system is the goal.

2.3 Types of reading front end

In the studies done over the years, several front-end interfaces for capacitive sensors have been developed in this research group, all based on the principle of measuring the capacitance that forms between the human body and the sensor at different frequencies. In [41], an attempt was made to measure the capacity through its reactive effects at different stimulating frequencies. The circuit used is the one in Figure 2.3, in which the capacitive sensor C_s is connected in a first order RC low pass filter configuration, and a sine wave is applied as input, V_{in}, and knowing the amplitude and the frequency is read out of the filter using a high impedance unitary gain buffer thus obtaining the output signal V_{out}.

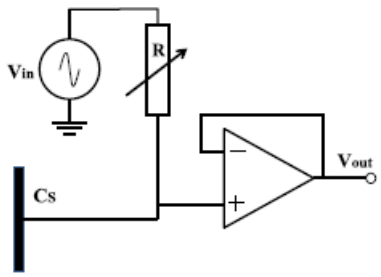


Figure 2.3. Circuit used to determine the capacitance made of a first order RC low pass filter, a sine wave applied as input and a high impedance unitary gain buffer, taken form [41].

In this specific experiment the author decided to use a frequency $f_m = 5$ kHz , so if a signal is put at the input:

$$V_{\text{in}}(t) = A\sin(2\pi ft + \theta) \tag{2.3}$$

where $f = f_m$. Then tries to calibrate the value of the LPF resistor R, in the event that there are no people in the sensor range, to obtain:

$$V_{\text{out}}(t) = \frac{A}{\sqrt{2}}\sin\left(2\pi ft + \theta - \frac{\pi}{4}\right) \tag{2.4}$$

This output is typical low-pass RC filer in which the cutoff frequency is equal to f_m. After finding the value of resistor R, it is called R_m, and on the basis of this the resistor array is built as shown in Figure 2.4 (a). Note that both the value of f_m and of the resistor do not have to be precise, because the method used to find the capacitance is a method based on relative measurements and therefore systematic errors usually cancel themselves out. While it is important that the value of the resistors remain stable and also the excitation frequency of the signal $V_{in}(t)$ which is tuned as:

$$f = 2^n f_m \tag{2.5}$$

Furthermore, for each frequency generated, the correct resistance is selected by means of an analog switch, S, which is therefore also tuned with the excitation frequency:

$$R = \frac{R_m}{2^n} \tag{2.6}$$

It is important to note that to eliminate excessive attenuation at high frequencies, thus improving the SNR, the cut-off frequency of the RC low pass filter must be kept tuned to the excitation frequency.

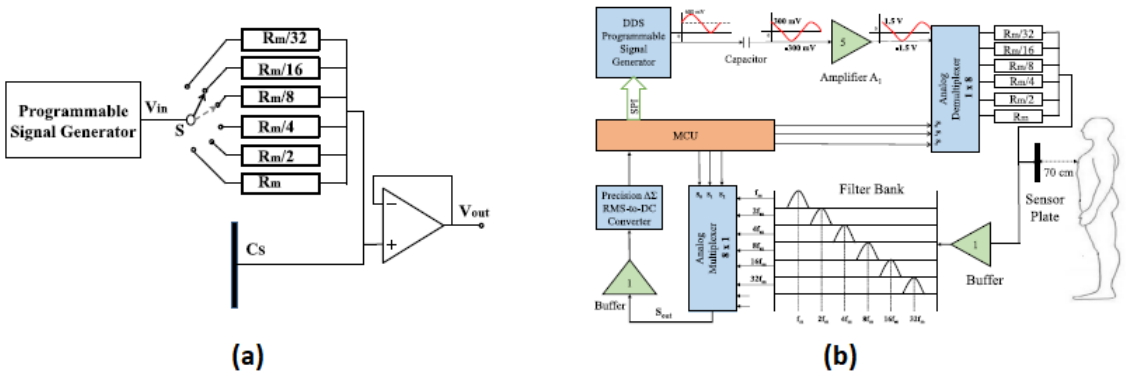


Figure 2.4. Resistor network used to create the first-order LPFs tuned for every excitation frequency (from 5 kHz to 160 kHz) and experimental setup of the circuit, taken from [41].

The circuit just described is then part of a final circuit used in the experiments, and is the one shown in Figure 2.4 (b), in which using a direct digital synthesys (DDS) programmable signal generator the excitation frequency is created, it is then programmed from a microcontroller using a serial peripheral interface (SPI). The sine wave generated by the DDS has 600 mVpp and an offset of 300 mV DC then eliminated by a capacitance in series. After that there is an amplifier A_1 which brings the signal entering the switch, S, to an amplitude of 3 Vpp. The switch is controlled by both the MCU and the DDS in order to tune the cutoff frequency of the RC filter. The output of the filter is then buffered and brought into input to a filter bank formed by narrow-band bandpass filters, each of which is centered at the frequency given by (2.5). The filter used will be selected via a multiplexer from the MCU, once the output from the filter bank is obtained, it is converted to DC using a $\Delta\Sigma$ RMS-to-DC converter and then measured by the MCU.

Another front-end capacitive sensor interface is shown in Figure 2.5 (a), taken from [24]. It is used to measure the difference in amplitude between two RC low-pass filters on the same sinusoidal input signal V_{in}. The first RC filter is made up of R_1C_s, which varies with C_s, while the second is R_2C_1 which is therefore fixed with C_s. Two equal high impedances are used to read the filter output voltage repeaters. Then, before entering the difference amplifier, the capacities C_2 and C_3 eliminate the DC offset. As said before, both R_2 and C_1 are fixed so V_{ref} has constant amplitude and phase, while V_{sense}, being C_s variable, will have non-constant amplitude and phase. The task of the difference amplifier is to subtract V_{sense} from V_{ref}, and then amplify the difference by a factor $G = R_4/R_3$, at the output there will be V_{diff} which will be modulated as C_s varies . Finally, two narrow band high Q-factor

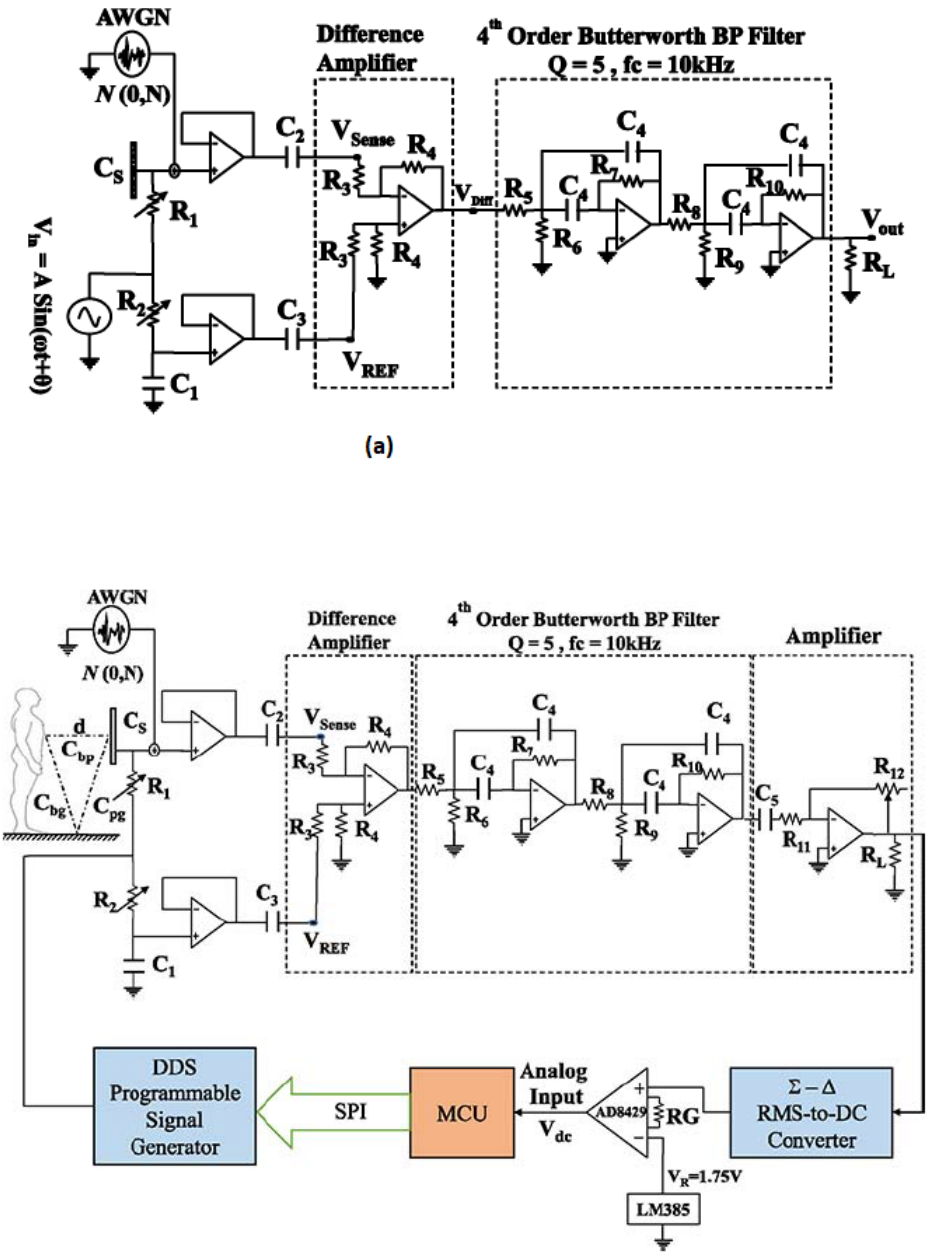


Figure 2.5. Capacitive sensor front end interface used to measure the difference in amplitude between the output of two RC low-pass filters receiving the same sinusoidal input signal and experimental setup used, taken from [24].

band-pass filter are used to eliminate the noise narrow band high Q-factor band-pass filters. This circuit, as shown in Figure 2.5 (b), is then inserted in the final diagram, as was also done with the previous circuit described.

Another front-end interface is the one described in the [15] article, shown in Figure 2.6 (a). The interface measures the phase shift variation of a low pass filter, in which there is also the transducer capacitance. Thanks to this way of measuring it is possible to greatly reduce the environment noise while still having high sensitivity to variations in capacitance. It consists of two analog parts excited by the same signal $V_{in}(t)$. In the upper part the signal enters a low-pass filter formed by R_1C_s, and the characteristics of the filter depend precisely on C_s which in turn depends on the distance from the person's body. After that the filter output enters a high impedence voltage repeater. In the lower part it is sent to a reading circuit similar to the previous one.

The circuit then, as in the two previous cases, is inserted in the same way in the final circuit used for the experiment, 2.6 (b).

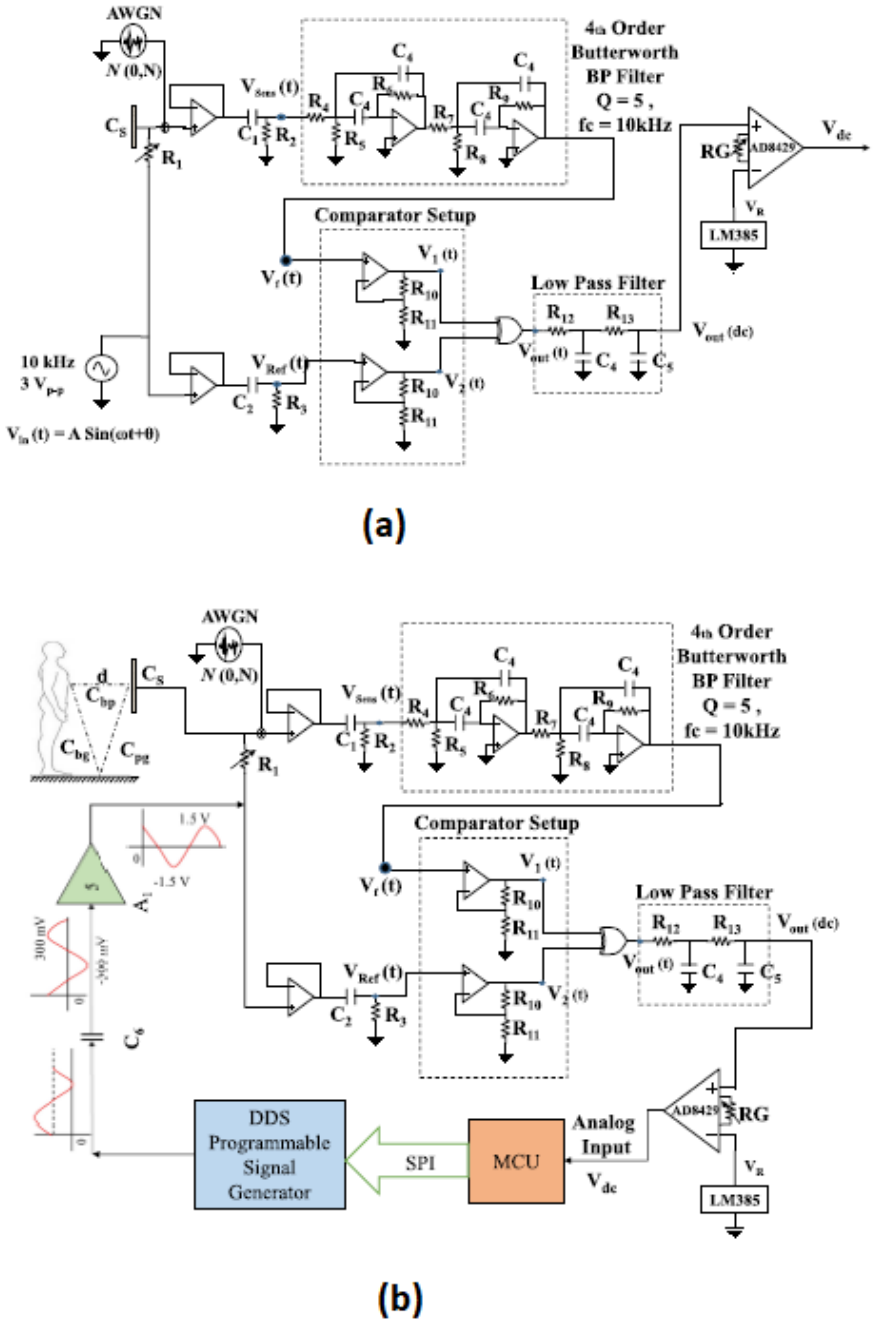


Figure 2.6. Capacitive sensor front end interface used to reduce the environmental noise: circuit and experimental setup, taken from [15].

2.4 Data processing on the sensor

To detect and localize a person in a room, in [27] use a lot of sensors installed in the room, for example in walls or chairs, sofas, etc. The detection system is therefore formed by these sensors and a base station that processes the data collected by them, to estimate the position using various methods such as: Naïve Bayes, k-Nearest-Neighbors, Support Vector Machine.

Figure 2.7 shows the architecture of the detection system. The movement of the person changes the transducer capacity with the distance to the human body. Using the oscillator, like the one in Figure 2.2, this change is converted into a variation in frequency, in fact when the person is close to the sensor plate, the capacitance C_{sb} becomes larger and the oscillation frequency, according to (2.2), becomes smaller. Then the sensor MCU measures the oscillator frequency with a certain rate, and then via a Zigbee radio module sends it to the base station. In the base station there is a Zigbee radio receiver which receives a continuous stream of raw frequency measurements from all sensors installed in the room. Finally, the data streams received are contained from the base station to a personal computer via a USB connection, so as to perform further processing.

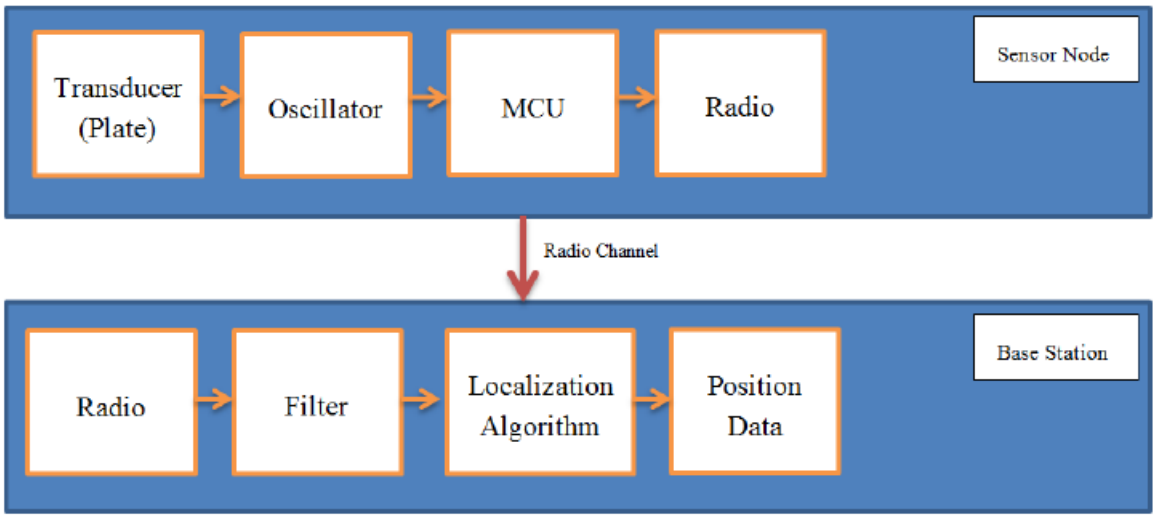


Figure 2.7. Architecture of localization system based on a Zigbee radio transmitter and receiver, taken from [27].

Chapter 3

Neural networks for sensor data processing

Machine Learning (ML), builds through data analysis analytical models, and is used in many sectors, especially in the field of artificial intelligence. Artificial neural networks (ANN), a part of ML, are mathematical models composed of neurons, whose structure resembles that of a human brain. There are many neural network types since they are used in many fields: Feedforward Neural Network (FNN), like Multilayer Perceptron (MLP), Recurrent Neural Network (RNN), etc. In this thesis, an MLP neural network was used for locating people indoors through capacitive sensors, and it is well suited for classification tasks.

3.1 What is a neural network

A neural network is a mathematical model whose layered structure remember the structure of the human brain. It is used to solve Artificial Intelligence engineering problems related to different technological fields, for example it can learn from data, so it can be trained to recognize speech patterns or images, like the human brain does, an image of typical NNs is in Figure 3.1 taken from [43].

It is composed of artificial neurons arranged and connected to each other in different layers, usually there is at least one layer for the inputs (input layer) and one for the output (output layer), but the more complex the problems to solve, the more layers there will be (layer between input and output are call hidden layers). Each neuron has a set of inputs and its own weight, to indicate the different importance of each of them, which is regulated, according to a specific rule, automatically neuron by neuron during training until the neural network correctly performs the desired activity (in the beginning they are set randomly). The various neurons of each layer operate in parallel, the first layer receives external inputs, and every neurons with their weights calculate their outputs and then through an activation function

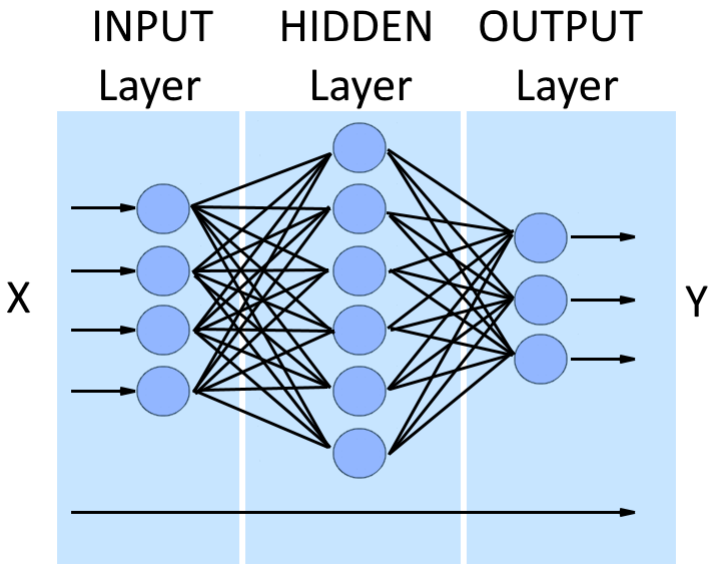


Figure 3.1. Typical Architecture of neural network in which there is an input layer, a hidden layer, and an output layer, taken from [43].

and a threshold value the output value of the neuron is weighted, and then it will enter as input into all neurons of the next layer, and this procedure is follow until the end. An image of a neuron is Figure 3.2, taken from the article of Marwen [1], where all inputs are multiplied with their weights, then all the results are added together with also a bias which is used to adjust the output along with the weighted sum of the inputs to the neuron, thus, bias is a constant which helps the model in a way that it can fit best for the given data. Then the output is calculated through the transfer (activation) function, that is used to determine the output of NN like "yes" or "no", so it map the output in range that can go from 0 to 1, -1 to 1 etc.

3.1.1 Learning method

To understand how to behave and solve the specific problem for which it will be used, the neural network will need to be trained, then modify their structure by changing the value of the weights according to the learning method used. Three main algorithms are used for this:

- **Supervised Learning**: with this method the NN is supplied with both an input set (call training set) and the corresponding desired outputs that should be obtained with that specific input set. Through them, the network is able to understand the link between input and output, so that, having understood this connection, the network is able to reuse this rule with other similar problems. The operator will have the task of adjusting the various weights in order to

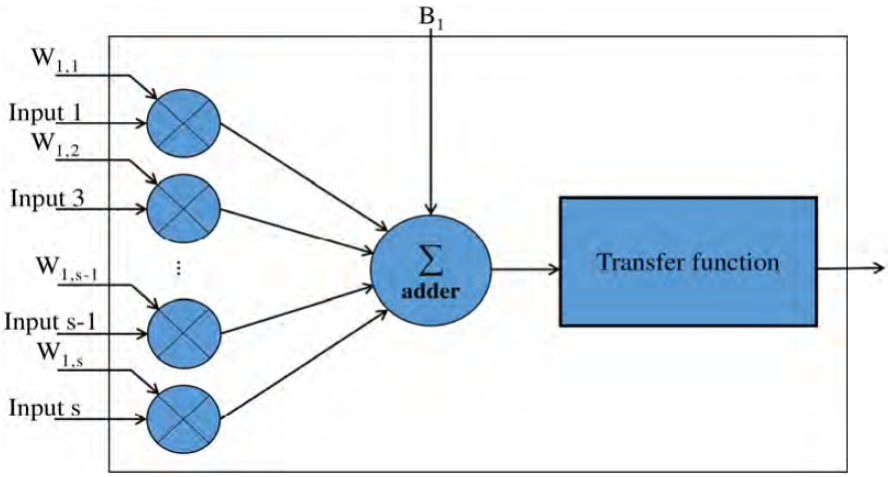


Figure 3.2. Structure of a neuron, taken from [1]: all inputs are multiplied with their weights, then the results are added together with also the bias and finally pass through the activation function.

have better and better outputs, therefore there is an Error-back Propagation mechanism in this method. If the training is successful, the network learns the relationship between input and output and therefore from now on will be able to predict the exit even when it is unknown. Neural networks of the MLP type use this method.

- **Unsupervised Learning**: with this method, only a set of inputs is provided to the NN, and according to them the neural network learns the scheme and the logical structure of the inputs. The weight value will be modified during the learning of the same nodes of the network. A neural network that uses this method is the Hopfield network.

- **Reinforcement learning**: in this method the NN does not try to find an input-output association as in the previous methods, but it learns by interacting only with the environment trying to get to the desired result through a mechanism of "rewards" (actions that allow to get closer to the result) and "punishment" (errors to be eliminated but from which one tries to learn). Then through a feedback mechanism, the algorithm is guided to the right learning process.

3.1.2 Advantages and disadvantages of a neural network

The fundamental advantages of neural networks, which have led to their use in vast fields, are:

- They can work in parallel thus treating a large amount of data.
- Fault tolerance, this is a consequence of the first advantage, in fact the possibility of an error occurs but it is unlikely that it will block the whole system, perhaps it will only reduce performance.
- Noise tolerance, often even if the inputs provided are not precise, it still manages to operate correctly.
- It is able to self-regulate in case of environmental changes.

However they also have limits:

- They are "Black box", that is the internal calculations cannot be fully analyzed, even if the final output is correct or otherwise acceptable.
- The training phase to fix the weights of individual neurons can take a long time if the number of available variables is very large.
- The output often are not the perfect output, but this in many cases is not necessary.
- The creator of the network has a fundamental role, because there are no algorithms that define an optimal network, everything lies in the skill of the operator.

3.2 Types of neural networks

There are several neural network architectures, which differ from each other in the learning method used and the specific problem for which they are used. The most important are:

- **Feedforward Neural Network**: they are the simplest neural networks. The data only propagate in input-output direction. They consist of an input layer and an output layer, and between them they can have, or not have, one or more hidden layers. Every layer have a set of neurons, and their output are the input of the neuron of the next layer. This network can be used for the recognition of shapes or calculation of simple functions.

- **Multilayer Perceptron (MLP)**: it is a type of network that has the input layer, the output layer and one or more hidden layer. Each layer will have a series of neurons, the outputs of which will be connected, as inputs, to all the neurons of the next layer. It can calculate any function and is also widely used in the field of facial recognition.
- **Convolutional Neural Network (CNN)**: this network is a subset of the MLP networks. It is made up of various layers, at least 5, and the outputs of each layer are used as inputs in the next layer, to which the convolutional operations are applied which make the network deeper but with fewer parameters. The fields of application of this network are vast: image or video recognition, natural language processing but they are also applicable in the field of Signal Processing and image classification.
- **Recurrent Neural Network(RNN) – like Long Short Term Memory**: in these networks the output of the upper layers are used as inputs of the lower layers, this to allow the system to create a memory supply so as to better predict the output. In this way, if the output was wrong, the system warns itself of the error so that in back-propagation it corrects it. Used for example in text-to-speech conversion.
- **Modular Neural Network**: works with different networks each independent of the other therefore they do not interact with each other during the calculation, but each one works alone to arrive at the final output. So being that they work independently the calculation process will be complex but faster being that they don't have to interact with each other.

These are only the main neural networks used in the various fields of applications but they are not all, we can still mention:

- Radial basis function Neural Network, based on the radial basis function.
- Modular Neural Network, in which there are a lot of small NN that cooperate or compete from each other to compute the output.
- Physical Neural Network, in which there are physical electric resistance to simulate artificial synapses.

3.3 Multi-Layer Perceptron neural networks

The NN used in [1] by Marwen is a MLP neural network, that is of the type Feed Forward, made up of at least 3 layers, an input layer that receives external inputs, one or more hidden layers, and an output layer that calculates the final result. Its training is supervised learning so as to find weights and biases in order

to minimize the error. In every layers there is a set of neurons, each of them do the multiplications between every inputs and their weights, and then sum all these results with the bias of the corresponding neuron, finally this output go through Activation Function. The two typical activation functions used for this type of network can be sigmoids:

$$y(x_i) = \tanh(x_i) \tag{3.1}$$

and

$$y(x_i) = \left(1 + e^{-x_i}\right)^{-1} \tag{3.2}$$

The range of the first is between -1 and 1, while the range of the second is between 0 and 1, but these functions often create numerical problems (Vanishing Gradient Problem [44]) so are replaced by the Rectifier Linear Unit (ReLU) activation function, also call ramp function, and his range, like Figure 3.3, taken from [45], shows, go from 0 to infinity, in particular the output is zero when it is less then zero while is equal to own value when it is equal or greater than zero.

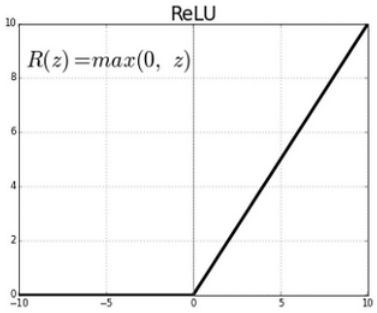


Figure 3.3. ReLU function, taken from [45]. The output will be zero, if the input is less than zero, otherwise it will be equal to the input value if equal or greater than zero.

3.4 Description of the neural network used

In the experiment of Marwen [1], an MLP network is used, which is suitable for this purpose. He uses the experimental data taken from [42], in which the data are taken from four different people at six different frequencies: 5 kHz, 10 kHz, 20 kHz, 40 kHz, 80 kHz, and 160kHz. To optimize and train this NN he used an augmented experimental data because the data available are too few for a good NN training. The flow used for the augmented data, using Matlab R2017a, is that shown in Figure 3.4, taken from [1].

First of all, Gaussian noise has been added, through noise generators, to significantly increase the number of data and avoid overfitting of the specific data sets

used for training, but trying to keep the main characteristics of the original set, they are 10000 labeled six-tuples were then generated for each of the four people (40,000 tuples in total). The data are then normalized between 0 and 1 to improve the convergence of NN training. Finally, the data is split into 3 groups: 70% of the tuples (so 28000) are used for the training set (so they are used to fit the model), 15% (so 6000) for the validation (to give a partial evaluation of the model adapted to the training set during the optimization for the model parameters) and optimization and 15% (so 6000) for the test of NN performance.

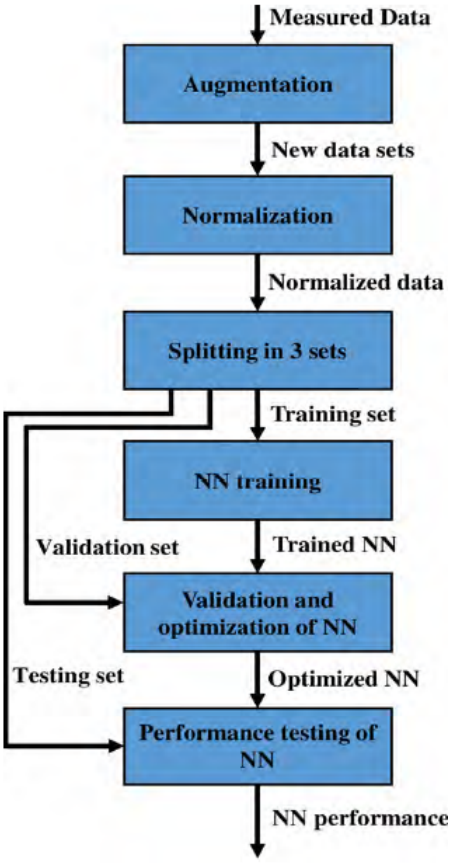


Figure 3.4. Flow of data augmentation for the training, validation and optimization of a neural network, taken from [1].

After that the author decided how many hidden layers, how many neurons per layer and which activation function to use. For the latter, the sigmoid functions (LogSig) were used at the beginning, but given the excessive use of resources of these activation functions [42], they have been replaced by ReLU, for all the layers apart from the output layer in which it is used a linear identity activation. For the optimal number of hidden layers a search was made by narrowing the field down to

neural networks with the same number of neurons per layer. The architecture with 2 hidden layers each of eight neurons was chosen, because the accuracy would have changed little if multiple hidden layers had been used (Table 3 of [42]). An image of the NN used is Figure 3.5, taken from [1]. Note that the first hidden layer will have only 6 multiplication for each neuron because the inputs are six, while in the second there will be eight multiplications per neuron.

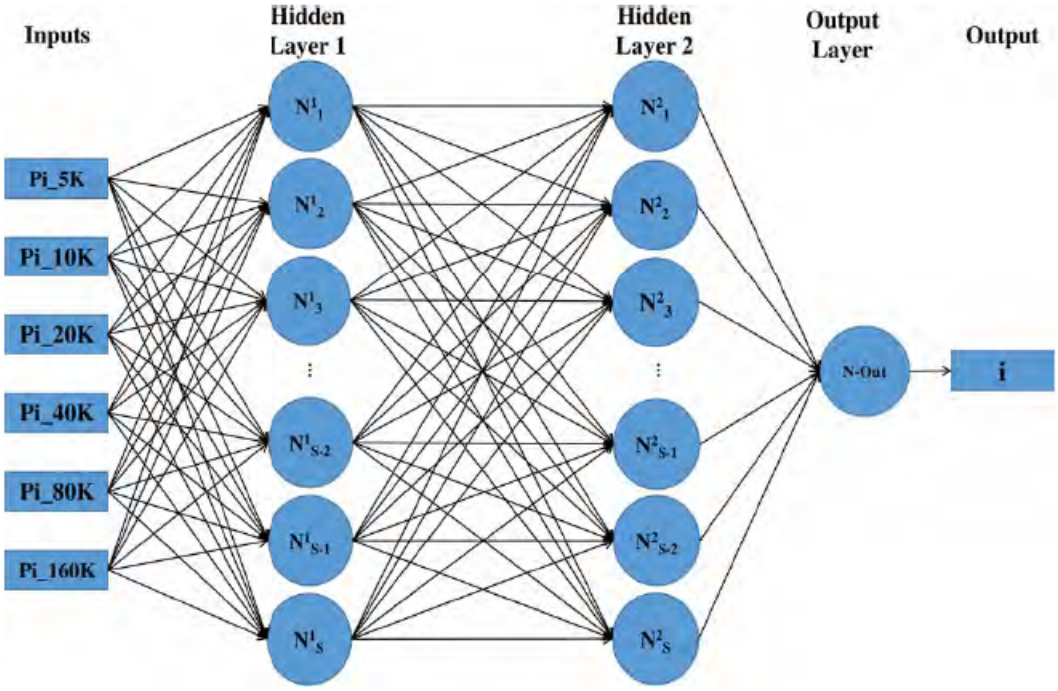


Figure 3.5. Multi-layer perceptron neural network with two hidden layer made of eight neurons and one output layer with one neuron, taken from [1].

Below, Algorithm 1 is a pseudocode of the MLP NN, taken from [1]. The first two for the loops constitute the first hidden layer: the first loop goes from 0 to the number of neurons of the first hidden layer, in this case eight, the second loop instead goes from 0 to the number of inputs, in this case six. Inside this last loop the various results of the neuron multiply and accumulate, once the loop is released the bias of that neuron is added to the final result and then the activation function is applied to it. After that each time the final output of the neuron is calculated, the accumulation variable is reset. The second hidden layer is implemented with the same structure but the second for now goes from 0 to all the outputs of the first hidden layer, so in this case eight. The last layer, Output layer, is implemented only with a for loop, which goes from 0 to all the outputs of the second hidden layer, then all the multiplications are performed and the results are accumulated,

and finally the bias is added to it and we obtain the person's ID.

Algorithm 1 Neural network MLP pseudocode, taken from [1]

First hidden layer
for i = *All neurons on first hidden layer* **do**
    for j = *All inputs* **do**
        $Product \leftarrow input\_data[j] \times weights\_0[i][j]$
        $Sum \leftarrow Sum + Product$
    end for
    $Hidden\_L0[i]Sum \leftarrow b0[i]$
    if *Hidden_L0 [i]* < 0 **then**
        $Hidden\_L0[i] \leftarrow 0$
    end if
    $Sum \leftarrow 0$
end for
Second hidden layer
for i = *All neurons on second hidden layer* **do**
    for j = *All outputs of first hidden layer* **do**
        $Product \leftarrow Hidden\_L0[j] \times weights\_1[i][j]$
        $Sum \leftarrow Sum + Product$
    end for
    $Hidden\_L1[i]Sum \leftarrow b1[i]$
    if *Hidden_L1 [i]* < 0 **then**
        $Hidden\_L1[i] \leftarrow 0$
    end if
    $Sum \leftarrow 0$
end for
Output layer
for i = *All neurons on second hidden layer* **do**
    $Product \leftarrow Hidden\_L1[j] \times weights\_2[i]$
    $Sum \leftarrow Sum + Product$
end for
$Person\_ID \leftarrow sum + b2$

Chapter 4

Very low power FPGA

In this experiment neural networks are implemented on MCUs or on ultralow-power FPGAs. The performances achieved by the latter are better than those of the MCUs. The ways in which they can be programmed are either manually (therefore writing the VHDL or Verilog code by hand) or through high level synthesis using a special tool which, starting from the code written in C or C++, creates the HDL code.

4.1 FPGAs types and performances compared

The use of neural networks, in recent years, has intensified a lot, this because they are applicable to a large number of fields, also obtaining excellent results. Their problem, however, is that more accuracy is desired, more complicated will be the NN and therefore they are really difficult to integrate into an embedded system such as a smart-phone or a robots, this is because their consumption of resources is very heavy. So, to overcome this problem, the neural network used for locating people indoors through the use of capacitive sensors, can be implemented via software, using powerful CPU or GPUs, or via hardware, using FPGAs as HW accelerators, which bring benefits both from the point of view of validity, of power [46] but also from the point of view of accuracy and resources used [47].

The families of FPGAs used by Marwen in the experiment [1], are of the ultralow-power type, therefore that from the point of view of power (both static and dynamic) they consume very little. In particular, two families of FPGAs were chosen, Lattice iCE40 family and Microsemi IGLOO family, of which respectively were chosen:

- ICE40UP5K for Lattice.
- IGLOO AGLN250, IGLOO M1AGL600 and IGLOO M1AGL1000 for Microsemi.

4.1.1 Lattice FPGA features

The Lattice ICE40 UltraPlus FPGA is used in smart-homes, factories and cities. It allows to have a low power of computing resources by also implementing complicated systems such as neural networks, in fact it is manufactured using a CMOS low power process. It has:

- 5280 Look-Up Table (LUTs), or structures data/tables used to replace a complex calculation with a consultation of this table, thus speeding up the operation.
- DSP blocks, in particular it has 8 blocks, and they are blocks capable of doing specific operations such as accumulation, multiplication, multiplication or accumulation (MAC) without going to use LUTs. It could be used as a 16x16 multiplier and as a 32 bit accumulator, therefore as a MAC block.
- 1024 Kbits Single Port Random Access Memory (SPRAM).
- Standby current is typically less than 100 uA.

4.1.2 Microsemi FPGAs features

From the Microsemi family, 3 ultralow-power FPGA IGLOOs were chosen, created by a 130-nm flash process. IGLOO devices use a Flash*Freeze technology that allows to enter and exit the ultralow-power mode which consumes less than 5 uW keeping system memory data and data registers. The three FPGAs used are:

- AGLN250: it is the smallest FPGA on which the experiment was made, in fact it has only 6144 VersaTiles (which are the equivalent of the LUTs for Lattice). It has no dedicated DSP blocks, it has 36 blocks of RAM 1024 Kbits. It has the lowest static power of all FPGA used in this experiment, 0.08 mW.
- M1AGL600: it has 13824 VersaTiles, has no dedicated DSPs and static power is 0.13 mW, also has 108 blocks RAM 1024 Kbits.
- M1AGL1000: it is the largest Microsemi FPGA has 24576 VersaTiles, has no dedicated DSPs and static power is 0.21 mW, also has 144 blocks RAM 1024 Kbits.

The Table 4.1 summarizes the characteristics just mentioned. Note that the static power information is taken from [1].

4.1.3 Other low power FPGAs

The choice to use the ultralow-power FPGAs of the Lattice and Microsemi family falls on the fact that they have a much lower static power than other FPGAs.

Table 4.1. Description of features (LUTs, DSPs, Pstatic and Ram) of the FPGAs used in the reference programming, taken from [1].

FPGA's Family	Device	LUTs	DSPs	Pstatic	Ram
Lattice ICE Ultraplus	ICE40UP5K	5280	8	0.28 mW	4 Kbits
Microsemi IGLOO	AGLN250	6144	N/A	0.08 mW	36 Kbits
Microsemi IGLOO	M1AGL600	13824	N/A	0.13 mW	108 Kbits
Microsemi IGLOO	M1AGL1000	24579	N/A	0.21 mW	144 Kbits

In fact, FPGAs from the Xilinx or Altera family could also have been used, in particular the first would have been excellent since the code generated by Vivado HLS is suitable for the FPGAs of this family. The problem is that comparing the power consumption of the latter devices, it is not comparable to that of the Lattice and Microsemi FPGAs, being they only low-power and not ultra-low-power. In particular, the FPGAs of the Xilinx and Altera family that could be used are those shown in the Table 4.2, in fact, it can be noticed that even if there are many LUTs and also DSPs are many and therefore they could be used for specific operations, the static power at least an order of magnitude greater than the ultralow-power FPGA.

Table 4.2. Description of the features (LUTs, DSPs and Pstatic) of other low power FPGAs not ultralow-power, taken from [1].

FPGA's Family	Device	LUTs	DSPs	Pstatic
Xilinx Spartan 3	XC3S200	4320	12	41 mW
Xilinx Spartan 3	XC3S200A	4032	16	43 mW
Xilinx Artix 7	XC7A100T	162240	240	41 mW
Altera Cyclone II	EP2C8	8256	18	40 mW

4.1.4 Microcontrollers

In addition to FPGAs, to implement the neural network, is possible use the software approach, i.e. implementation on a CPU or GPU. Unlike FPGAs, CPUs are suitable for running code sequentially and not with parallel implementation. The hardware architecture of a CPU is defined by the programmer, therefore it is not modifiable. In addition, the neural network or the system to be implemented will be in C/C++ while in the FPGA in Verilog or VHDL.

The neural network, show in Figure 3.5 was done on two ARM Cortex M3 MCUs [1] :

- STM32L152RE: that is optimize for ultralow-power, the field of the applications is various: medical, gaming, PC pheripheral, GPS and sport equipment etc, alarm systems, wired and wireless sensor.
- STM32F103RB: that is optimize to reach good performance, typical applications are motor drives and application control, medical, industrial application, printers and scanners, alarm systems, video intercom.

These MCUs include HW multipliers, and thanks to the way in which they were created there is the possibility to measure the current consumption during the execution of the program. The results find in [1], are show in Table 4.3

Table 4.3. Comparison between the results obtained by manual programming implementing the neural network on the two MCUs under different operating conditions, taken from [1].

MCU Parameter	**MCU device**		
	STM32L152RE/STM32F103RB		
Clock frequency (MHz)	8/8	16/16	32/64
FLASH latency (wait states)	0/0	0/0	1/2
Clock cycles for NN inference	1474/1474	1474/1474	1740/1831
NN inference time (μs)	184/184	92.1/92.1	54.4/28.6
Current(mA)	1.85/4.00	3.60/6.80	8.60/23.9
Voltage (V)	3.0/3.3	3.0/3.3	3.3/3.3
Power (mW)	5.55/13.2	10.8/22.4	28.4/78.9
Energy (μJ)	1.02/2.43	0.99/2.06	1.54/2.26

For STM32L152RE the best results from the point of view of energy are at 16 MHz, for the power instead it is at 8 MHz (5.55 mW), with a low energy however. Instead from the point of view of time, the best result is found with a clock of 32 MHZ, and the result found is about half the time that it is at 16 MHz, this is because the frequency in the second case is higher and therefore the time taken is less but the energy and power increase. The clock cycles for 8 MHz and 16 MHz are the same, while for 32 MHz they increase (from 1474 to 1740) due to the latency of the FLASH which is higher.

For STM32F103RB the best results from the point of view of energy are found at 64 MHz, however there is an increase in power due to the high operating frequency, while for 8 MHz there is still a low energy (only 0.17 uJ more than the previous case), but the power, being the lowest frequency, is much lower (13.2 mW).

4.2 FPGA programming mode

In order to implement a circuit on an FPGA, it must be programmed. There are two main programming modes:

- By schematic: that is, the user manually inserts every single block that forms the project.
- Through hardware description language (HDL): that is, the programmer describes the behavior of the project in Verilog or VHDL.

The first method prefers the speed and simplicity of development as well as facilitating the visualization of the design of the project, on the other hand, however, it can only be used for simple schemes. Describing the hardware to be implemented with the second method leads to flexibility and maintainability of the project, also large and complex designs can also be managed. The most used HDL languages are VHDL and Verilog.

Regarding the latter programming method, the VHDL or Verilog files, which describe the project, can be created starting from the high level description in C or C++ language, after which, using a tool, Vivado HLS, the high level code is synthesized to create the HDL file to be implemented on FPGA, in doing so the designer describes the design at a higher level of abstraction, leaving the synthesis tool to create the HDL files.

4.2.1 Manual programming

With manual programming, the programmer must describe the project hardware by using the block diagram, or by writing the VHDL or Verilog code. In both cases, the programmer has absolute freedom on the HW choices that can be made, and has complete control over the components that will then be used in the project. As for programming through the use of blocks, it can be done on relatively simple projects, because if the project becomes too complex, the various connections between the blocks would not be simple, so the creation of the project would become slow. If, on the other hand, HDL languages are used to describe the project, the programmer can better describe the HW, without wasting too much time in creating and connecting the various blocks because the synthesizer will take care of these operations. The disadvantage of the latter method is that it is not a programming language but a HW description language, so if there is a program to synthesize in which there are IF, FOR, WHILE, etc. constructs it can be done but the programming is more complicated, so it would take time to create the project.

In both cases, however, any design can be described to be implemented on FPGA, the time it takes to write the program depends on the complexity, and the optimization of the code, must be done always by hand. However, the programmer have full control over the creation of the project, so if there is a need to look

for an error during the simulation phase, it would be easily traceable because the programmer in theory knows all the signals, which is not the case using the HLS technique, in which the components of the synthesized HDL code will have different names and there is no control over the hardware it will insert.

4.2.2 High level synthesis

High level synthesis interprets the high level description of a given algorithm, and transforms it into the equivalent algorithm but described in a hardware description language. This is an automated process, and the goal of the HLS is to allow the hardware designer to create a project efficiently, making available optimizations to improve it, give the possibility to verify the actual operation and accelerate the generation of the HDL.

The advantages of adopting this type of programming are:

- Increase in performance.
- Reduction of power consumption.
- Improve the level of abstraction of the algorithm by being written in C/C++.
- Reduces the time, compared to HDL, to write complex algorithms (such as digital signal processing (DSP)).

But there are also disadvantages:

- The generated HDL code is poorly readable.
- Loose control of the described hardware architecture.

Then, starting from the C/C++ code written by the programmer, HLS extracts a Finite State Machine (FSM), after which it identifies the operations and they will be mapped in each state, then the operations are scheduled by mapping them in the various clock cycles and finally assign operations to available functional units.

4.2.3 Optimizations

The tool used for high level synthesis is Vivado HLS, and it is a tool used to create HLD code suitable for the FPGAs of the Xilinx family. Directives can be applied to the code written in C, that is optimizations to make the code more efficient so that it can add the set objectives. There are two ways in which these directives can be applied:

- Directive file: Vivado HLS inserts all the directives adopted in a .tcl called diretive.tcl, this way is useful when the goal is to to see different optimizations on the same code so as to explore the design.

- Source File: in this case the directives are applied directly to the source code as a pragma.

The main optimizations that can be made are those to improve:

- Throughput: directives such as PIPELINE or UNROLL can be applied both to loops and to functions of the C code, so as to increase parallelism, while paying attention to problems related to data dependency. Together with the aforementioned directives, the ARRAY_PARTITION directive is also usually used to partition a given array, since it is implemented as a RAM block with a maximum of two data ports, so as to access it with a greater parallelism. Another useful directive is DATAFLOW, used to parallelize operations when there are a set of sequential tasks in the code.

- Latency: the LATENCY directive can be used to limit a certain loop to a certain number of clock cycles.

- Area: directives such as INLINE improve the occupied area because when a function is inlining the components inside it are better shared or optimized with the logic of the calling function, also the ARRAY_MAP directive is used to improve the area, with it in fact joining small arrays mapping them into a single larger array to reduce the number of RAM blocks. Directives such as ALLOCATION or RESOURCE instead allow to better control the hardware used.

An example of how the use of directives can improve the performance of an algorithm is shown in [48], where Vivado HLS is used to synthesize and optimize different algorithms such as: Mergesort, Depth-First Search and Breadth-First Search . In fact, it has been discovered that the use of directives such as unroll and pipeline which increase parallelism, bring benefits in terms of time taken to execute the algorithm and also in terms of speedup. In particular, it has been noted that these two directives work very well together, especially when using unroll with a low factor. Also in [49] it is shown how using HLS and the directives on a Non-binary low-density parity-check (LDPC) Decoders, it is possible to reach over 50% of the throughput reached with the RTL design.

Chapter 5

High level synthesis for FPGA

In this thesis, the technique of using high level synthesis to create the HDL file to be implemented on FPGAs is adopted. With HLS there are numerous advantages (better performance, speed in creating the project, the possibility of inserting optimizations) but there are also disadvantages such as that one does not have control of the hardware included in the project or some construct of C that are not synthesizable. The software used is Vivado HLS by Xilinx, and various directives (optimizations) have been inserted in the original code to improve performance, after which the generated HDL code is inserted in the Lattice and Microsemi tools, so it will not adapt perfectly to the design of these FPGAs since the code is suitable for FPGAs of the Xilinx family.

5.1 High level synthesis flow

The creation of HDL files starting from the code written at high level follows a very precise flow, in particular, the flow of the Vivado HLS tool follows the following steps:

- Start by writing the project in C or C++ code Figure 5.1 (a), and this code will then be compiled, executed and debugged.
- Synthesize the C code in the RTL implementation, Figure 5.1 (c), possibly adding directives to optimize, Figure 5.1 (b).
- Generate synthesis reports to get information on the resources used and timing and analyze the design to see how operations are scheduled in the various clock cycles.

- Through Co-Simulation, Figure 5.1 (d), it is verified that the RTL implementation created still works correctly, and this is done through a test bench.
- Package the RTL implementation into a selection of IP blocks that will be integrate into the HW system, Figure 5.1 (e).

The Figure 5.1, taken from [50], summarizes the design flow of Vivado HLS and shows inputs that can be receive and which outputs creates a design synthesized by Vivado HLS. In particular, as input it can receive: functions written in C, C++ or SystemC, constraint, directive and the test bench in C. While the outputs are: the RTL files created in HDL (or in VHDL (IEEE 1076-2000) or in Verilog (IEEE 1364-2001)) and report files.

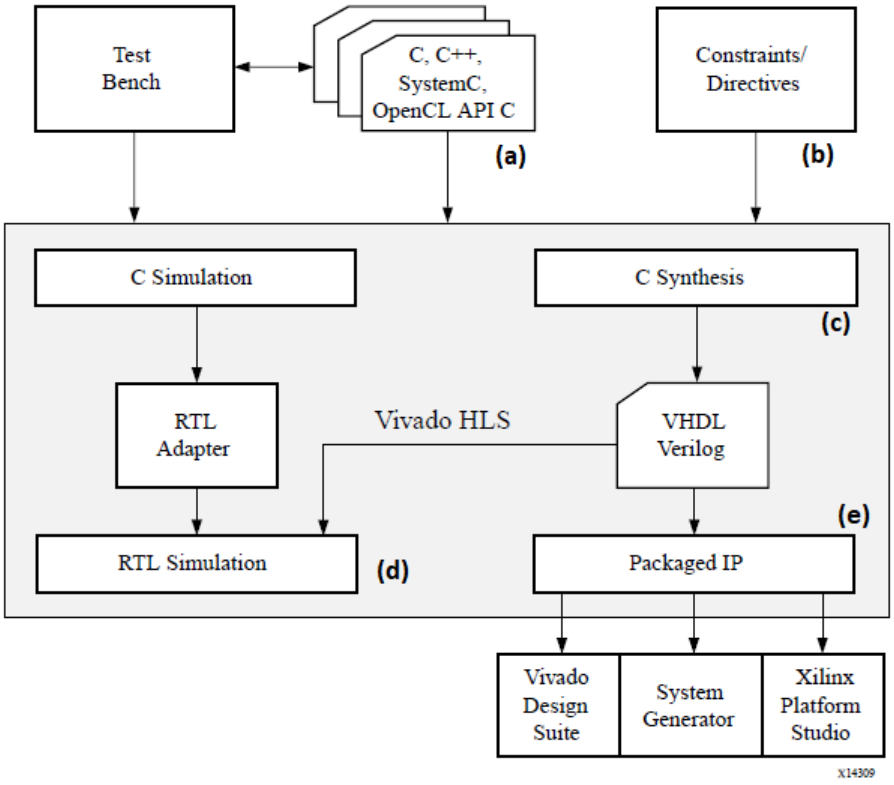


Figure 5.1. Vivado HLS flow design to test the operation. Test bench, directives and C-function are taken like input. The output are HDL files and report files, taken from [50].

After verifying with the Co-simulation that the VHDL or Verilog created continues to behave correctly, these files are taken and implemented on the Lattice and Microsemi FPGA, therefore Vivado is not continued to be used as it should be given that the HDL created is suitable for the design of the FPGAs of the Xilinx

family. The tools used for ultralow power FPGAs are: Radiant for Lattice FPGA, Libero for Microsemi FPGAs.

After inserting the .vhd source files created by Vivado HLS, the steps are equivalent for both tools:

- Set the correct FPGA to implement the program.
- Synthesize the VHDL code.
- Design is mapped.
- Place and Route.
- Finally the power analysis is done.

In addition there is also the possibility of simulating the program before and after the synthesis and also after the place and route. Once these steps are done the .bin programming file can be created, which will be placed in the FPGA. The way HLS synthesizes the C code is as follows:

- Top-level function arguments are synthesize as RTL I/O ports.
- The functions in C code are synthesized in blocks in the RTL hierarchy.
- Loops are kept rolled by default.
- The arrays are synthesized in RAM blocks.

5.2 Limits of Vivado HLS

There are many types of C language that Vivado HLS supports, but there are some constructs of these languages that cannot be synthesized. System calls, for example, are not supported because the functions that they perform are related to the C program and therefore this tool ignores the functions that have no impact on the algorithm (for example printf() and scanf()). Then all system calls that have to do dynamic memory usage (for example malloc(), alloc(), free()), and these functions will have to be removed from the code. Even the use of pointers is limited, in fact general pointer casting are supported only between native C types, pointers arrays but not that are refer to other pointers, instead function pointers are not supported. Finally, even the recursive functions cannot be synthesizable. In addition to these problems, it must be noted that the HDL code produced by HLS is hardly readable by the programmer, because all the signals or variables that were initialized in the C code will have different names and also the various HW blocks (adder, multiplier) will be difficult to be traced. Overall, all the code will be difficult to modify if there is an error during the Co-simulation, because determining the cause also through the use of a wave simulation is a difficult task.

5.3 Code used for neural network and optimizations

The neural network code used in this experiment is A.1. It receive an array composed by six data, that are the six input, and it provides an array of 1 dimension that is the output. The code is composed of three large loops, formed in turn by an internal loop.

The first loop *FirstHiddenLayer*, and also the internal loop *NeuralInputLayer*, is the loop that implements the first layer of the neural network 2. The six multiplications between weights and inputs are performed for the generic neuron (in the inner loop), and this is done then for all the eight neurons of the first layer (upper loop), making a total of 48 multiplications. It should be noted that of these 48 multiplications (6 multiplications for each of the eight neurons), there are eight in which the input is multiplied by a value of weights which is equal to zero, so this operation, since the result is necessarily zero, is discarded by Vivado HLS, so in each neuron there will be only five multiplications, so 40 in all. When the multiplications of the neuron are performed they are added up from time to time and finally the bias is also added and then the activation function is applied. The outputs of this first loop are array of two dimensions call outputs_middle, in particular the first dimension of these array identify the outputs of the first loop.

Algorithm 2 First Loop of Neural Network

```
1: FirstHiddenLayer:
2: for short j = 0; j < NEURAL_HIDDEN_LAYER; j++ do
3:     sum = 0;
4:     NeuralInputLayer:
5:     for short i = 0; i < NEURAL_INPUT_LAYER; i++ do
6:         t = weights[0][i][j] * inputs[i];
7:         sum = sum + t;
8:     end for
9:     sum = sum + biases[0][j];
10:    outputs_middle[0][j] = Relu(sum);
11: end for
```

The second loop *NeuralNextLayer*, and also the internal loop *NeuralThisLayer*, implement the second layer 3. Each neuron will do the 8 multiplications between weights and inputs for all 8 neurons, therefore in all 64 multiplications. The outputs of the second loop are always outputs_middle, but this time on the second dimension of the array. Finally the last loop *OutputLayer* and also the internal loop

Algorithm 3 Second Loop of Neural Network

```
1: NeuralNextLayer:
2: for short j = 0; j < NEURAL_HIDDEN_LAYER; j++ do
3:     sum = 0;
4:     NeuralThisLayer:
5:     for short i = 0; i < NEURAL_INPUT_LAYER; i++ do
6:         t = weights[1][i][j] * outputs_middle[0][i];
7:         sum = sum + t;
8:     end for
9:     sum = sum + biases[1][j];
10:    outputs_middle[1][j] = Relu(sum);
11: end for
```

LastHiddenLayer implement the output layer and the final eight multiplications 4. Note that in this loop the activation function ReLU is no longer used but Linear.

Algorithm 4 Last Loop of Neural Network

```
1: OutputLayer:
2: for short j = 0; j < NEURAL_OUTPUT_LAYER; j++ do
3:     sum = 0;
4:     LastHiddenLayer:
5:     for short i = 0; i < NEURAL_HIDDEN_LAYER; i++ do
6:         t = weights[2][j][i] * outputs_middle[1][i];
7:         sum = sum + t;
8:     end for
9:     sum = sum + biases[LAYER - 1][j];
10:    outputs_middle[2][j] = Linear(sum);
11: end for
```

There is also an initial loop 5 that is used to serialize the inputs of the NN, because if, for example, the parallelism of NN rise up, like in the case of directive PIPELINE, more inputs are processed, and this is a problem for the FPGA pin number being exceeded, to avoid this loop has been added so only one input at the time is taken, and then are processed all in parallel to respect the number of the pin. In addition to the file on which there is the code of NN, there is another

Algorithm 5 Initialization of Neural Network

```
1: Initialization_input:
2: for int i = 0; i < NEURAL_INPUT_LAYER; i ++ do
3:     inputs[i] = input[i]
4: end for
```

file (the header A.2) in which there are the general settings of the neural network such as: number of layers, number of neurons in the input layer and hidden layer etc. There is the possibility in Vivado HLS to define the size of the data that is managed, so the programmer has the possibility to choose the size of the data, so, for example, instead of having the generic int of 32 bit, it can be set to a smaller number of bits, thus saving area because all internal logic will work with a lower number of bits. In the case of this neural network, the inputs should have 17 bits and the output 4 bits, but to save on the area it was decided to opt for a number of bits for the inputs equal to 16 because even with a bit less the final result varies but only slightly. The data with which it works (inputs, weights, biases, outputs and all the various internal results) are not integers but are fixed point, in particular the inputs and all internal results are of the type data_t, that have 16 bits of which 4 represent the integer part, in order to be able to represent the values of the weights, inputs and biases correctly, and 12 bits for the fractional part, while the output is of type output_t that has 4 bits, of which 2 for the integer part and two for the fractional part.

There is also a need to handle the interface, because in a C code all the operations are done in zero time, but when it is transformed into HDL code the various signals will pass through ports and must be synchronized through an appropriate interface. In my case the interface protocol ap_ctrl_hs was used which is a type of Block-Level Interface protocol.

5.3.1 Synthesis diretives

Before Synthesize the code, Vivado HLS gives the possibility to insert (in the code via #PRAGMA or in a .tcl file) directives in order to improve the performance according to our goal (area, latency, etc.). As explained in the Chapter 4 in the Section 4.2.3, the directives can be used to improve throughput, latency and area.

In my case, to explore as much as possible the space of solutions, the directives

have been inserted through directive file (directive.tcl), that is, the different solutions that have been created each have their own directives. The directives used are:

- Pipeline: directive used to allow operations to be performed concurrently, just like on an assembly line. It improves both throughput and initiation interval and can be applied to both loops and functions with the difference that if applied to loops the pipeline will be executed until the end of the loop, while in the functions it will be performed forever. In the case of the neural network it will be applied on the three main loops, and via parameter II can be choose after how many clock cycles a new value can be applied, so as to ensure that the pipe chain is always full and thus avoid the insertion of bubbles that would degrade performance.

- Partition Array: usually is used together with the PIPELINE directive, because if the pipe is applied the parallelism is increase, therefore multiple accesses are given to the same array at the same time, and it, being synthesized as a RAM block has a maximum number of two data ports, is a limit for the throughput of an algorithm that must read multiple values of that array at the same time. So the solution is to split the array into smaller arrays thus increasing the number of ports. The partition of the array can be done in the following way:

  - Block: the original array is split into blocks, of the same size, of consecutive elements
  - Cyclic: the original array is split into blocks, of the same size, of interleaving elements
  - Complete: the original array is split into individual elements.

  Also the dimension of the array it's a parameter that can be partitioned, for example, in one of the solutions, for the weights array which has three dimensions, I partitioned it on the third dimension which is equal to 8, thus having 8 arrays instead of only one.

- Unroll: allows to unroll the loop thus increasing throughput and decreasing the clock cycles to do and operate a loop, but as a counterpart there is a large increase in area because there are more operations to do together. Unroll can be applied to both loops and functions, also deciding the degree so how long to roll the loop, if for example the loop takes 4 clock cycles to execute, with a degree two unroll it will only take two, with degree four it will only take one.

- Allocation: directive used to keep under control the number of resources used, in my case it is used to modify the number of multipliers to be used in the

various solutions of the FPGAs, since the one I choose from Vivado HLS is not of the same family and therefore will not have the same DSP available.

- Function Inlining: this directive allows to save some area inlining the function and thus allowing to share the components within it and be optimized in the logic of the calling function. This directive in my case is used indirectly, in the sense that Vivado HLS uses it automatically for the ReLU and Linear function.

The Table 5.1 summarizes the advantages and disadvantages of the directives used.

Table 5.1. Summary of the advantages and disadvantages of the directives used.

Directive	**Advantages**	**Disadvantages**
PIPELINE	Grow up Parallelism, Throughput and Latency	Increase Area
PARTITION ARRAY	It allows to make multiple accesses to the same array	More access, therefore more signals and therefore more energy is consumed
UNROLL	Grow up Parallelism, Throughput and Latency	Increase Area
ALLOCATION	It limits resources therefore less area used	With less resources the speed is slower

5.3.2 Design space exploration

To find among all the possible solutions the one that comes closest to the performance obtained by the manual programming, I have adopted the following scheme, Figure 5.2, that I have used for all FPGAs:

- I start allocating the number of multipliers with the ALLOCATION directive, but I don't allocate them all immediately, but first start with a lower number and then increase them. When the limit of performance are at the limit (perhaps checking latency or seeing that as I added directives the area increased but the performances remained the same). So for example in the case of the Lattice FPGA in which up to eight multipliers could be inserted, I started by putting only two, then four and finally eight.
- Then I start piplining the inner loop of all three loops, thus seeing how latency improves.
- I use low grade unroll together with the pipe always in the inner loops, and usually this is the limit of performance using pipes and unroll in the inner loops.

- Since using the directives in the internal loops does not achieve the predetermined performance, the use of the pipe applies at the external loops and together with this also partition, in complete mode, the arrays inputs on dimension 1, and ouputs_middle on dimension 2, since now the parallelism is increased and so more access are made on these array.
- After which the unroll directive is also used together with the pipe, first using a low degree and then increasing it from solution to solution. Together with them I will also have to partition the weights array, on dimension 3, and the biases array, on dimension 2, in order to make more access correctly. With these (when all the multipliers available through the allocation directive are available) the performance is achieved best.

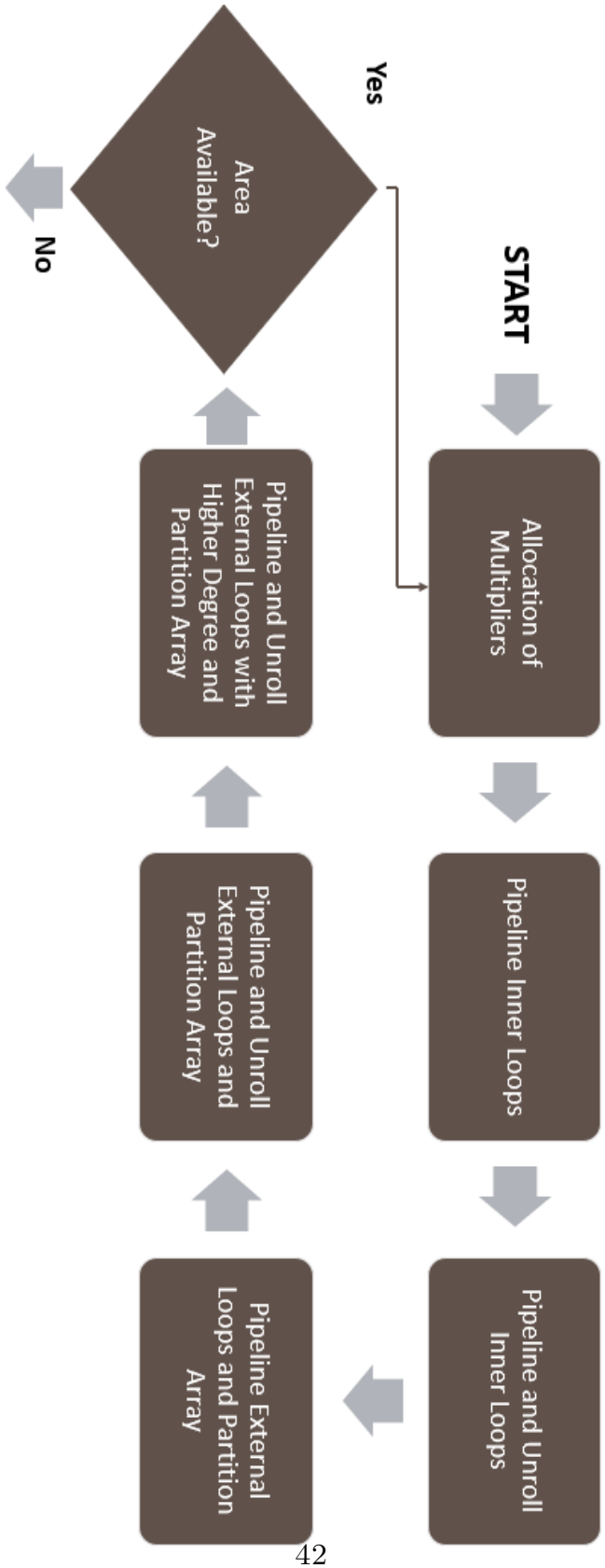


Figure 5.2. Flow of directives used to find the best performance. Start with allocation of multipliers, then pipeline and then unroll of inner loops, then can pass to pipeline, unroll and partition arrays of the upper loops. If the performance are not reached, this flow is repeat.

Chapter 6

Experimental results

The analysis of the tools, used to carry out the experiments, are made and the values of the data obtained for the various FPGAs, finally comparing them with those obtained with Marwen's manual programming [1].

6.1 Target devices

The main objective of this thesis is to reach or surpass the experimental results obtained by Marwen in [1] using HLS. As already written before, the software used to generate the HDL code at the HLS level is Vivado HLS, while the two software used for the two FPGA families under test are: Radiant and Libero, for Lattice iCE40 family and Microsemi IGLOO family respectively. The target devices are the FPGAs of these two families described in Chapter 4 in the Section 4.1, so ICE40UP5K, AGLN250, M1AGL600 and M1AGL1000. The first obstacle that can be encountered is the non-adaptability of the code generated by Vivado HLS for the two FPGAs examined, since the generated code should be implemented on FPGAs of the Xilinx family, therefore some code constructs may not be understood by the two tools and therefore implemented incorrectly using resources that may not actually be used. The two target parameters that have to reach are inference time and energy, after which area, clock frequency, power and clock cycles were also compared. After having achieved the performances reached by reference programming, there is a need to extract the .bin programming file to be inserted into the FPGAs.

6.2 Tools

The HDL code generated by Vivado HLS will then be implemented on the tools of the two families, namely: Radiant Software 2.0, for the Lattice iCE40 FPGA, and Libero SoC v11.9 for Microsemi IGLOO FPGAs. In order to use these tools I had

to download the license for both of them from their corresponding website and in both I checked that in the version of the tool downloaded there were the FPGAs that I should have tested.

6.2.1 Radiant Software 2.0

The tool used for Lattice's FGPA is Radiant in the version 2.0. The workflow that should be followed to obtain the .bin programming file to be inserted into the FPGA is the one shown in Figure 6.1, taken from [51].

It should be noted that the tool is equipped with numerous features:

- ALDEC Active-HDL, Figure 6.1 (a): which is a simulator provided by the tool, therefore through the use of a test bench and adding the waves to understand the behavior, a verification is made, before synthesis, that the inserted HDL code works correctly.

- Inspect Strategy Settings, Figure 6.1 (b): the tool gives the possibility to set the best strategy to be adopted for the project. In fact it is a set of settings that will then be used in the various stages of the implementation of the project (synthesis, map, place & route, etc.). With the strategy is possible also choose whether to prioritize the speed or the area of the project. Radiant tool also provides a default strategy, with the possibility, however, to modify it according to the objectives to be achieved.

- Set Timing and Location Assignments, Figure 6.1 (c): by setting the constraints for timing and location the achievement of the requirements of the design is possible, because they allow the optimization algorithms to work in the most efficient way possible.

- Synthesizers, Figure 6.1 (d): the synthesis translates the HDL file, therefore RTL, into a process-specific gate-level netlist which in this case will be optimized for Lattice Semiconductor FPGAs. The synthesizers used by this tool are two: Synopsys Synplify Pro for Lattice and Lattice Synthesis Engine (LSE), and they are fully integrated, meaning that all options are directly set from the Radiant tool.

- Power Calculator, Figure 6.1 (e): it estimates the power dissipated by the project and is integrated into the Radiant tool. The power estimate can be made either in Estimation Mode, i.e. based on the resources of the device or template that the designer will have to provide, or in Calculation Mode, i.e. the power estimate will be made on the device resources taken from a .udb file of the design or from an external .vcd file, and these are files that contain the real delays of the signals of the project, and therefore with them a more precise estimate can be done. In this thesis the second modality was adopted.

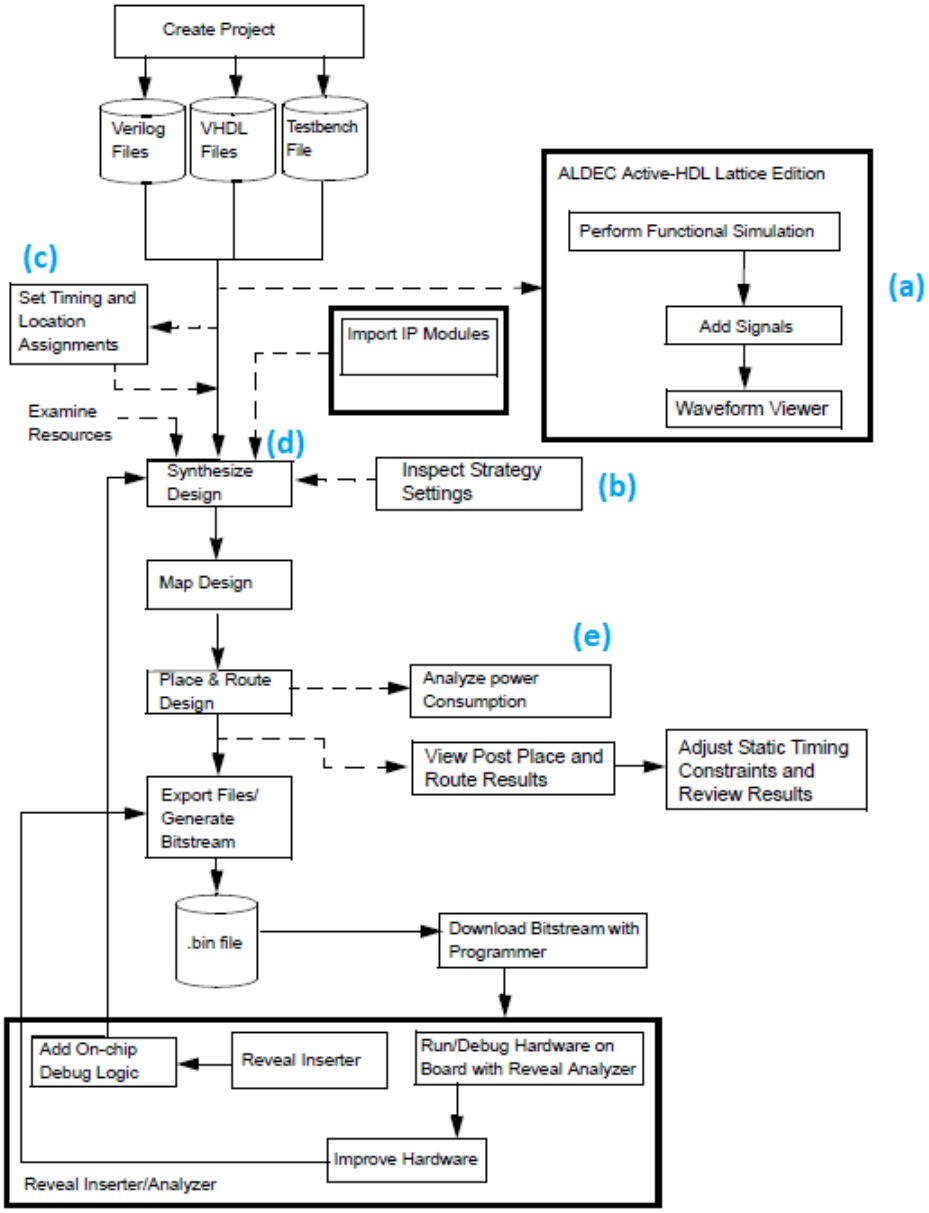


Figure 6.1. Workflow of Radiant Software, taken from [51]. Several tools are integrated into this software: ALDEC Active-HDL (a), to simulate, Synplify Pro for Lattice and Lattice Synthesis Engine (LSE) (d), to synthesis, Power Calculator (e), to calculate power.

6.2.2 Libero SoC v11.9

For the FPGAs of the Microsemi family, the Libero tool is used. The workflow of this tool is shown in Figure 6.2, taken from [52].

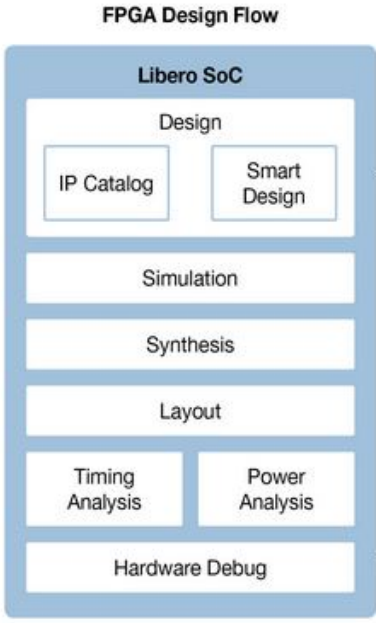


Figure 6.2. Workflow of Microsemi Software, taken from [52]. Start with creation of design, simulate it with Modelsim, synthesize the design with Synplify Pro, do the place and route then do the timing and power analysis and finally create the .bin file for programming.

The features of this tool are:

- Modelsim Simulation: using this tool already within Libero, the simulation of the design can be made, using a testbench. The simulation can be done pre-synthesis, post-synthesis and post-place & route.
- Synthesizing the Design using Synplify Pro: this synthesizer compiles and synthesizes the design.
- Timing analysis: using Smart time a static timing analysis can be done thus seeing if there is any path that has negative slack, and if there is change the clock frequency so that from this analysis the slack is small and positive.
- Power analysis: there is also the possibility to analyze how much power our design dissipates. This analysis can be done vectorless, so the tool makes an estimate of the possible delays of the signals, or through the file .vcd created after the simulation, where there are the real delays of the signals.

6.3 Workflow

In order to use the neural network used in the capacitive sensor project, the .bin programming file have to be created to be loaded on the FPGA in question. The workflow followed by me in this thesis to create the final file is the one shown in Figure 6.3. At the top, when Vivado HLS is used, the steps to follow are equal for both projects, whether it will then be implemented on Lattice's FPGA or on Microsemi's. After the Co-simulation different paths are followed even if the steps are similar but placed in different order in the two software, after checking the timing they return to be the same.

The steps therefore are:

- Choose FPGA and Import C files: import the project at a high level and after that choose which FPGA to work on, being that on Vivado HLS there are no FPGAs of the Lattice and Microsemi family but only those Xilinx, I choose the FPGA of the latter family that was closest in number of LUTs and DSPs to the FPGA that I had to test.

- Insert Directives: directives are then inserted in order to achieve the performance established according to the diagram in Figure 5.2.

- Sythetize project: the project is now synthesized with directives inserted on Vivado HLS in order to create the .vhd (in VHDL) or .v (in Verilog) files. The clock period can be set before the synthesis, this is used to go faster or slower thus affecting the created files. After the synthesis in the Analysis panel can be seen how the various operations are scheduled.

- Co-simulation: to verify that the created HDL code still behaves correctly, Vivado HLS gives the possibility to do a Co-Simulation. So I created a test bench, giving as inputs the real values that are used in manual programming, after which there is the possibility to debug and check that each result provided is correct by comparing them with those found by the reference programming provided to me via an Excel sheet. It should be noted that using only four output bits, due to the few pins available on Lattice FPGA, the results will not be as precise as those of the reference model but approximated being on a lower number of bits, I have tried anyway, just for verification, to increase the number of bits and in fact the same results, hence the code continues to behave correctly.

Now the flow takes different (but similar) paths depending on which FPGA you are testing, whether that of Lattice or those of Microsemi.

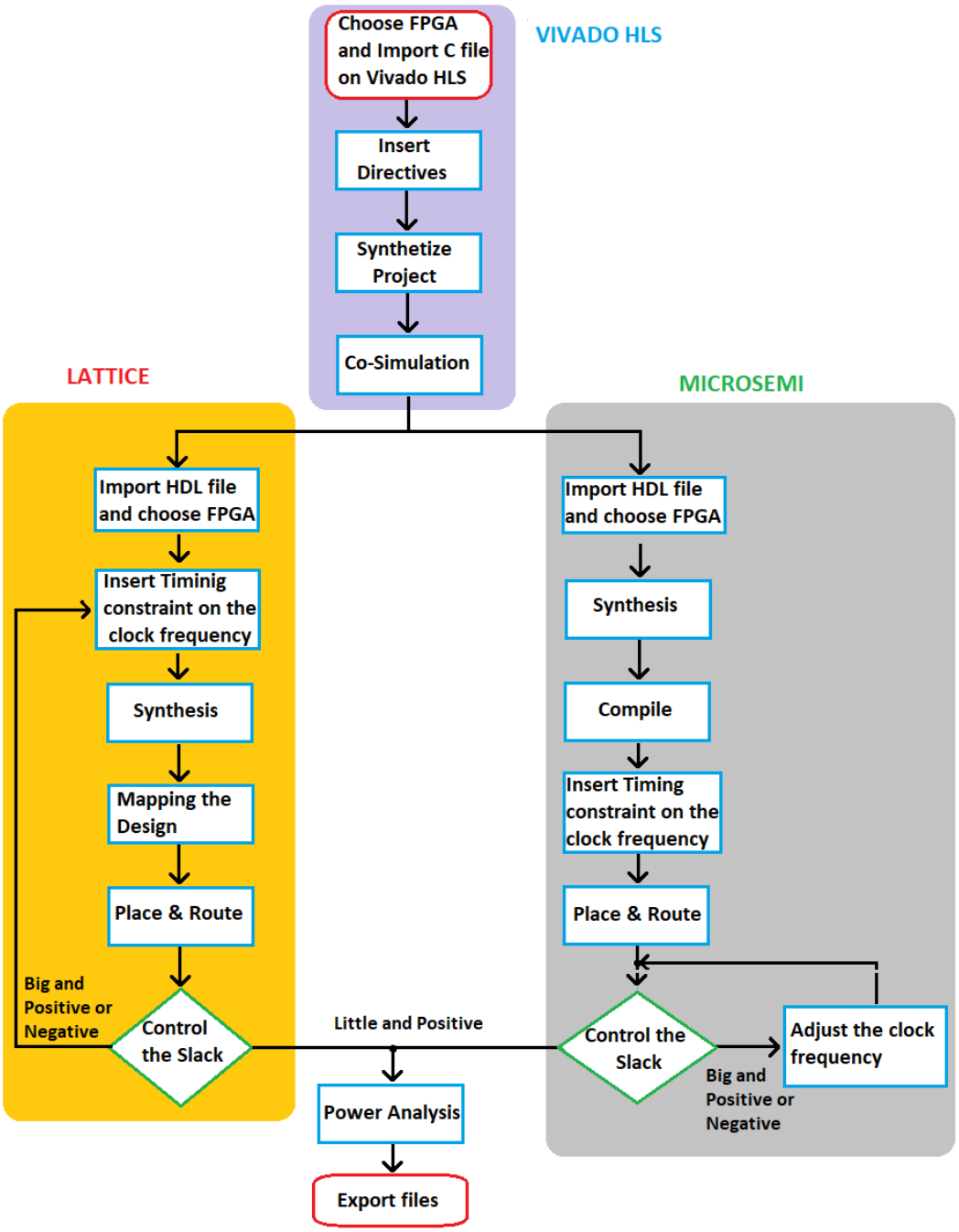


Figure 6.3. Workflow follow to obtain .bin file for programming. At the top, violet square, there are the steps followed on Vivado HLS. At the left, yellow square, there are the steps followed in the Lattice tool. At the right, grey square, there are the steps followed in the Microsemi tool. At the end, power analysis and then export files.

For Lattice:

- Import HDL file and choose FPGA: import the created HDL files and choose the FPGA to test, which in this case is only one.
- Insert clock frequency: now insert the reference frequency, that is found in the manual programming, for this FPGA to see if my project can also run at the same frequency.
- Synthetize: synthesize the HDL code into gate-level netlist.
- Mapping and Place and Route.
- Check the slack: I now check that the slack, based on the previously set clock frequency. If is small and positive, is a sign that the design is efficient that goes perfectly against the assigned frequency. In what was not so there are two other possibilities: the first is that the slack is positive but big, this means that my design can go faster than the set frequency, so I will have to increase the frequency to decrease this slack, the second instead it is that the slack is negative, this means that the project does not meet all the critical paths of the clock and therefore has to go slower, so the design will have lower the operating frequency.
- Power Analysis: After verifying that the slack is not violated, the power analysis can be done. The final operating frequency of the project is set, and now the dynamic and static power can be seen. It should be noted that in this power analysis the power relative to the inputs is also calculated, Figure 6.4 in red, being that my purpose of the thesis is to test the operations inside the neural network, the dynamic power of the inputs will be subtracted from the total dynamic power since it is beyond the scope of my thesis.
- Export: finally the .bin programming file can be exported to be inserted in the FPGA.

For Microsemi:

- Import HDL file and choose FPGA: like in Lattice but this time different Microsemi FPGAs, depending on which one I want to test, can be chosen.
- Synthesis and Compile.
- Insert clock frequency.
- Place and route.

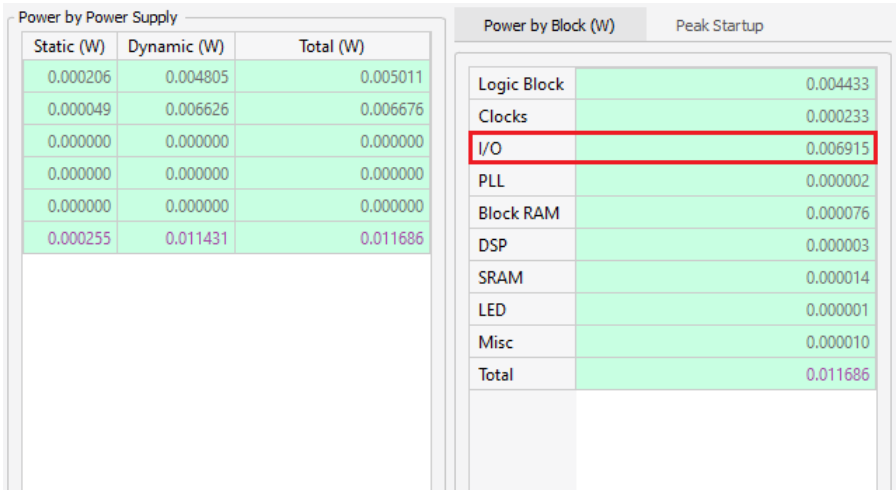
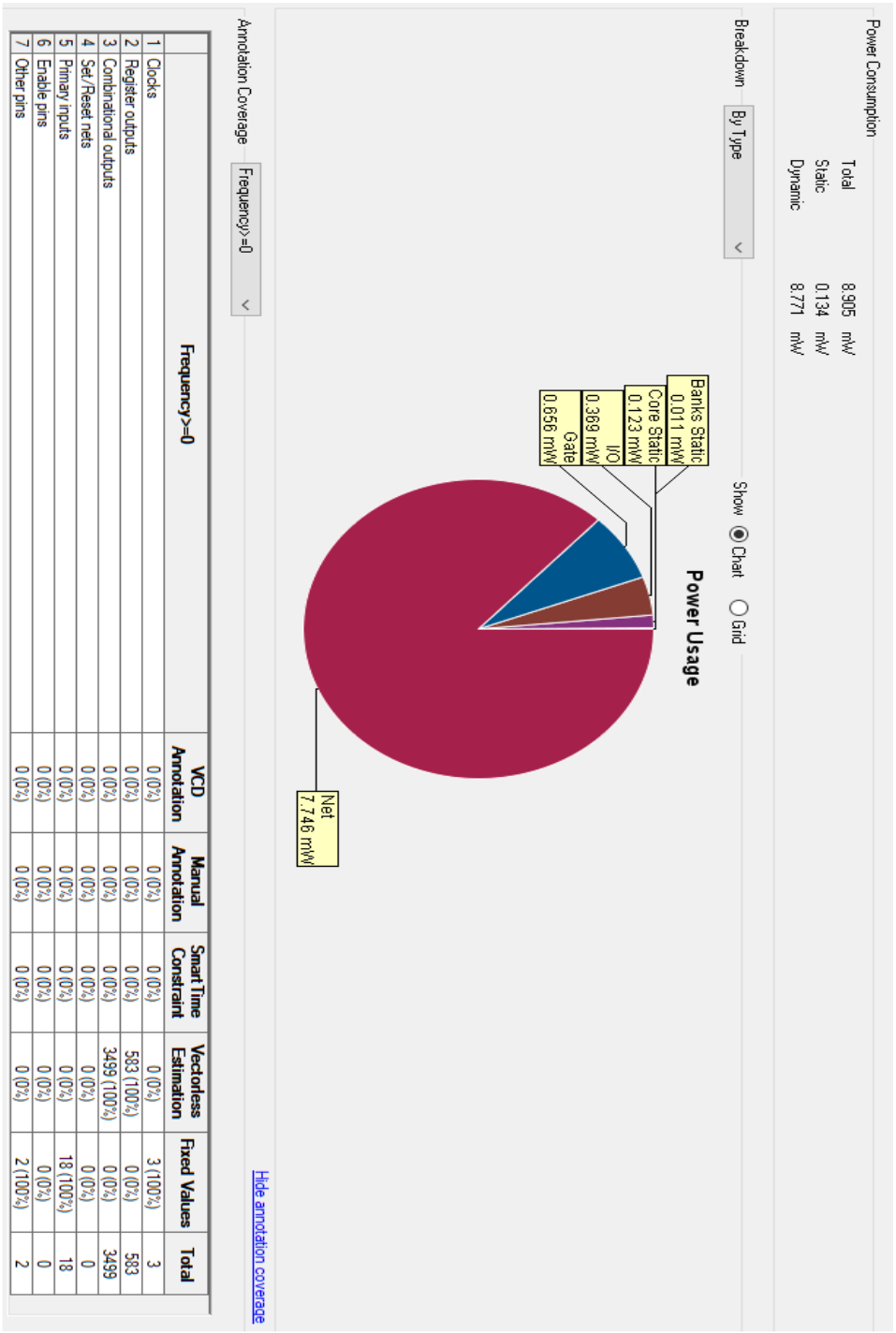


Figure 6.4. Part of the Power Analysis to see the power of the inputs (in red). In my analysis this contribute is not important, so I have to subctract it from the total dynamic power.

- Timing Analysis: through this analysis it is possible to see if the slack is violated as in the case of Lattice FPGA. The difference is that now there is no need to do the synthesis, mapping and place and route again like in the case of Lattice iCE40 family, but just change the clock frequency until the slack is small and positive.
- Power Analysis: the power analysis is similar to that of Lattice FPGA except for the fact that the dynamic power of the inputs must not be subtracted from the total dynamic power. In this case, a vectorless analysis can be done or by loading the .vcd file created after the simulation. A typical screen of the power analysis in Libero is the one shown in Figure 6.5, where is possible to see the various power contributions of the project and understand which is the most dominant.
- Export: export the back annotation files.



	Frequency>=0	VCD Annotation	Manual Annotation	SmartTime Constraint	Vectorless Estimation	Fixed Values	Total
1	Clocks	0 (0%)	0 (0%)	0 (0%)	0 (0%)	3 (100%)	3
2	Register outputs	0 (0%)	0 (0%)	0 (0%)	583 (100%)	0 (0%)	583
3	Combinational outputs	0 (0%)	0 (0%)	0 (0%)	3499 (100%)	0 (0%)	3499
4	Set/Reset nets	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0
5	Primary inputs	0 (0%)	0 (0%)	0 (0%)	0 (0%)	18 (100%)	18
6	Enable pins	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0
7	Other pins	0 (0%)	0 (0%)	0 (0%)	0 (0%)	2 (100%)	2

Figure 6.5. Part of the Power Analysis of Libero. On the top the main contribution of dynamic and static power. The diagram shows all the contributions of the power. The table shows if the analysis is made vectorless or with .vcd file.

As for the possibility of simulating the project on the software of the two families of FPGAs, it has not been included in the project flow because in reality after having done the Co-Simulation on Vivado HLS there is the certainty that the project works. The post-synthesis simulation will be done for the Microsemi FPGAs to have the .vcd file so real delays to have a precise power estimate. For Lattice, on the other hand, a .udb file is used for the power analysis in which there is an estimate of the possible signal delays, and this is already a very good estimate as even inserting the .vcd file created by the simulation, I have verified that the power estimate does not change.

6.4 FPGA manual programming results

Manual programming by [1] is based on using ultralow-power FPGAs on which to implement the hand-programmed MLP neural network, and compare the performance between the various devices. The main objective is to analyze the performance, in terms of inference time to process the data and energy spent to do it, of an MLP neural network on board capacitive sensors. It is therefore implemented via HW on ultralow-power FPGAs, so as to consume as little energy as possible.

6.4.1 Manual implementation on Lattice ICE40UP5K

Since this FPGA provides the use of DSPs, the multiplications of the neuron will be implemented with them, and since there are eight DSPs, they can be used in parallel to implement the multiplications between inputs and weights while for the additions and the activation function the LUTs can be used. So, since the NN has two hidden layers each of 8 neurons, there are 16 neurons to be processed, and using the DSP-LUTs scheme for multiplications and additions, after 17 clock cycles the output is obtained, the last clock cycle is used to apply the linear activation function (Figure 6.6 (a), taken form [1]). All eight DSPs and 2047 out of 5280 (38.77%) LUTs of the FPGA were used with this implementation.

6.4.2 Manual implementation on Microsemi AGLN250

Since the FPGA is smaller, with few VersaTiles available, only 2 multipliers can be implemented on it, those occupy 1071 VersaTiles each, and all other elements occupy another 2747. Since there can only be 2 multipliers, they can be scheduled in parallel only 2 multiplications each clock cycle, so it takes a total of 4 clock cycles to process a neuron, plus an additional clock cycle to add up all the results and apply the activation function (Figure 6.6 (b), taken form [1]).

6.4.3 Manual implementation on Microsemi M1AGL600

Eight multipliers can be implemented on this FPGA, being larger than the previous one, each of them will occupy 1071 VersaTiles, while all the other elements will occupy another 1463 VersaTiles, which are less than in the previous case thanks to the more level of parallelism that simplifies the design. Since 8 multipliers can be implemented in this FPGA, the scheduling is identical to that of the Lattice FPGA, so the eight multipliers perform the multiplications of the neuron in parallel, and a final adder will sum the results (Figure 6.6 (a), taken form [1]).

6.4.4 Manual implementation on Microsemi M1AGL1000

This is the largest FPGA, in fact 16 multipliers are implemented on it, from 1071 VersaTiles each, and the other elements occupy another 706, less and less than before for the highest level of parallelism that can be obtained. With the 16 multipliers and two adder, two neurons can be processed together, and then the scheduling of the neural network can be finished after nine clock cycles, the last one being used to calculate the final output (Figure 6.6 (c), taken form [1]).

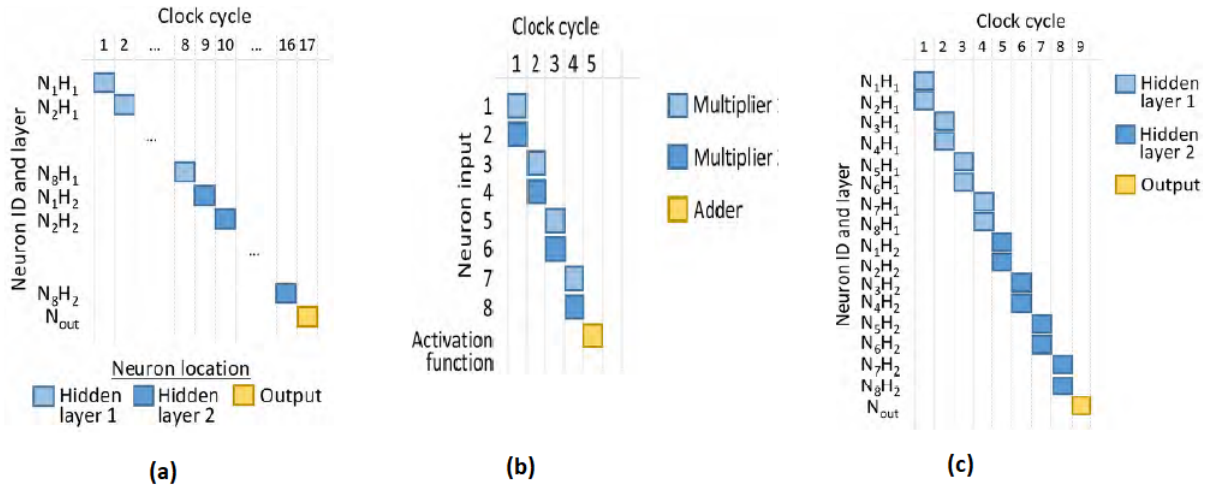


Figure 6.6. In order scheduling, taken form [1], of the operations of: ICE40UP5K and M1AGL600 which takes 17 clock cycles to get the output, AGLN250 which takes 5 clock cycles to get the first multiplication of the first neuron, M1AGL1000 which takes 9 clock cycles to get the output.

6.4.5 Achieved manual results

The results of the experiment on the four FPGA was made for two cases, this is because, with the smaller FGPA (AGLN250), only two multiplications can be done in parallel, therefore it is necessary to have registers that save the multiplications of the whole neuron and then added up at the end, if this were not done, the two multiplications would be overwritten from time to time. Then inserting registers to keep the multiplication result splits the longer critical path, and increases both the clock cycles and the operating frequency. To make comparable the results of the four FPGAs, the registers are added to all the FPGAs ("short critical path" (SCP)), even if the others in theory would not need it since they manage to do all eight parallel multiplication, but it was also done without registering to keep the multiplication results ("long critical path" (LCP)), so these two types of data were collected.

The resources employed for both solutions (SCP and LCP) are shown in the Table 6.1 taken from [1]. As described, two, eight and sixteen multipliers implemented with VersaTiles are used for the three FPGAs of Microsemi (from the smallest to the largest), and for all three implementations, the occupation of resources exceeds 70%. While for the Lattice FPGA, since the specific eight DSPs are used for multiplication, the LUTs used are around 38%.

The Table 6.1 taken from [1], shows also the performances achieved. Lattice FPGA for SCP reaches the highest clock frequency (45.82 MHz), while maintaining lower energy than all SCP results (10.4 nJ). In the LCP case, the lowest energy (5.79 nJ) and also the lowest active power (6.24 mW) are reached. Thanks to the high parallelism that can be achieved with Microsemi's larger FPGA (M1AGL1000), the lowest clock cycles in LCP mode (12) and also the lowest inference time (0.74 μs) for LCP but also for SCP (0.80 μs). For the latter, even if it has lower clock cycles than that of Lattice, a lower frequency is obtained, both for SCP and for LCP, compared to LAT, probably because it has the dedicated DSP and the creation process at 40-nm CMOS. Microsemi FPGAs are those that consume less from the point of view of static power. In particular the AGLN250 is the lowest one (0.079 mW). The latter also has the lowest active power (7.50 mW). If are made the comparison only between Microsemi FPGAs, which are all produced with the same manufacturing process at 130-nm, it can be seen that the most energy efficient for both implementations (SCP and LCP) is the MS-S (17.8 nJ for LCP and 24.0 for SCP), from the point of view instead of inference time the worst is the AGLN250 (3.59 μs), this because it has only two multiplier compared to the M1AGL1000 which has 16.

Table 6.1. Performance and resource, taken form [1], on FPGAs in the manual programming experiment in the "short critical path" and "long critical path" implementation.

		Perfomance						Resource
FPGA/arch		Clock (MHz)	Active power (mW)		Neural network inference			Total logic blocks
			Dynamic (mW)	Static (mW)	Clock cycles	Times (μs)	Energy (nJ)	
LAT	LCP	21.58	6.24		20	0.93	5.79	2047 of 5280 (38.77%)
			5.97	0.277				
	SCP	45.82	12.8		37	0.81	10.4	2093 of 5280 (39.64%)
			12.6	0.277				
MS-S	SCP	24.26	7.58		87	3.59	27.2	4889 of 6144 (79.57%)
			7.50	0.079				
MS-M	LCP	16.83	15.0		20	1.19	17.8	10031 of 13824 (72.57%)
			14.9	0.131				
	SCP	26.97	17.5		37	1.37	24.0	10212 of 13824 (73.87%)
			17.4	0.131				
MS-L	LCP	16.26	25.7		12	0.74	19.0	17842 of 24576 (72.60%)
			25.5	0.213				
	SCP	26.15	34.7		21	0.80	27.9	18674 of 24576 (75.89%)
			34.5	0.213				

6.5 Results and discussion

This section describes the results obtained from my experiments. For each of the four FPGAs under observation I created different solutions, always starting from a low level of parallelism, then increasing it from solution to solution as long as the FPGA area allowed it. The directives inserted in the various solutions are based on the description in Figure 5.2. It is also fair to say that Vivado HLS is very efficient from the point of view of optimizations, because, for example, if there is a multiplication for 0, the tool ignore that, because the result is always 0, so is useless do this operation. Another concrete example in all my solutions is the automatic insertion of the INLINE directive for the neuron activation function. For each single solution with the same directives, three or two different VHDL files were created by setting in each of them a different clock period on HLS so as to make the solution go more or less fast, and see what effect it had on the analysis of inference time and energy, a clock period around 20 ns is usually chosen for the slower, one around 5 ns for the faster and one that is in the middle between these two clock periods to have a clear analysis on the behavior of the neural network.

An in-depth analysis was also done to be sure that a change in the clock period in HLS corresponds to a decreasing minimum clock period also in the software of the two FPGAs. The analysis was done by changing the period in HLS from 15 ns to 3 ns for the Lattice FPGA and for the smaller Microsemi FPGA (AGLN250) and was done in case no directives were inserted and in case they are inserted. The expected trend is that from 15 ns there should be a descending line that reaches down to 3 ns. Figure 6.7 shows the graphs with no directives and Figure 6.8 shows the graphs with directives insert. The graphs show the clock period in ns on the Y axis and on the X axis the period set on Vivado HLS, while the blue line indicates the period estimated by HLS in ns, the orange one the period estimated by the FPGA software.

In order:

- the graph (a) of Figure 6.7 represents the values obtained without directives for the Microsemi FPGA, the target period to be reached is 41.22 ns and it is possible to reach it around 10 ns and 11 ns of period in HLS. As can be observed, the trend of the period in Microsemi is quite linear, in fact there is an unusual increase to 8 ns of the HLS period, but otherwise the data obtained follow an almost linear trend.

- the graph (b) of Figure 6.7 represents the values obtained without directives for the Lattice FPGA. The target period to be reached is 46.33 ns which I can't get even at 15 ns, in which there is a lower period, a sign that the design can go faster than the target period. Also in this case the trend is rather linear. graph (a) of Figure 6.8 represents the values obtained with directives for the Microsemi FPGA. the target period to reach is 41.22 ns. The trend is also linear here, apart from 7 ns there is a 0 as this design exceeded the LUTs available on the FPGA, and therefore was not implementable. The target period however I manage to reach it at 5 ns of HLS in which I get 38.88 ns in the tools.

- graph (b) of Figure 6.8 represents the values obtained with directives for the Lattice FPGA. The target period to reach is 46.33 ns, which I get between a period between 4 ns and 3 ns in HLS. The trend is mostly monotonical in this case too.

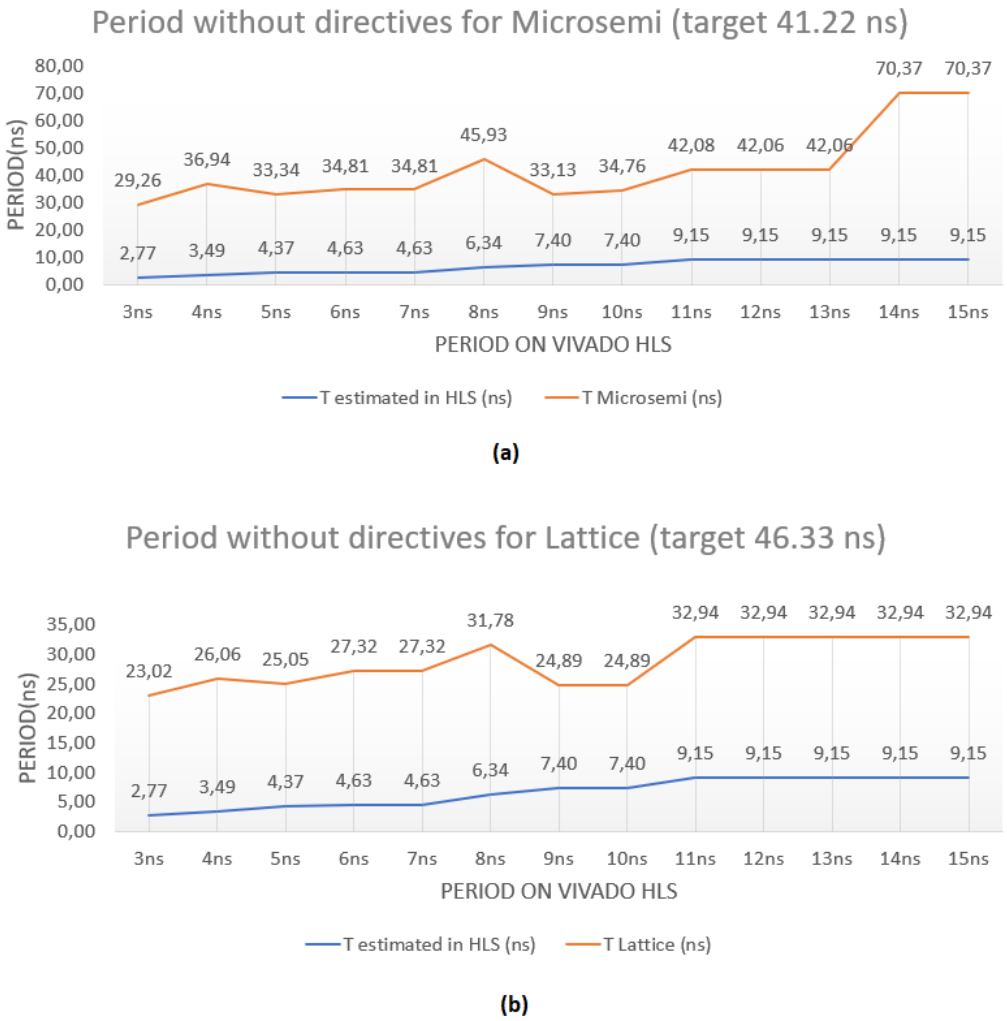


Figure 6.7. Trend of the clock period on the tools of the FPGAs based on the modification at the HLS level from 15 ns to 3 ns with no diectives.

Therefore from this analysis it is possible to establish a certain dependence between the period set on HLS and the one obtained by the tools of the two FPGAs, and apart from certain values that have an upward peak, the trend is still monotonical. Once this analysis has been made, the data can be collected to be compared with those [1].

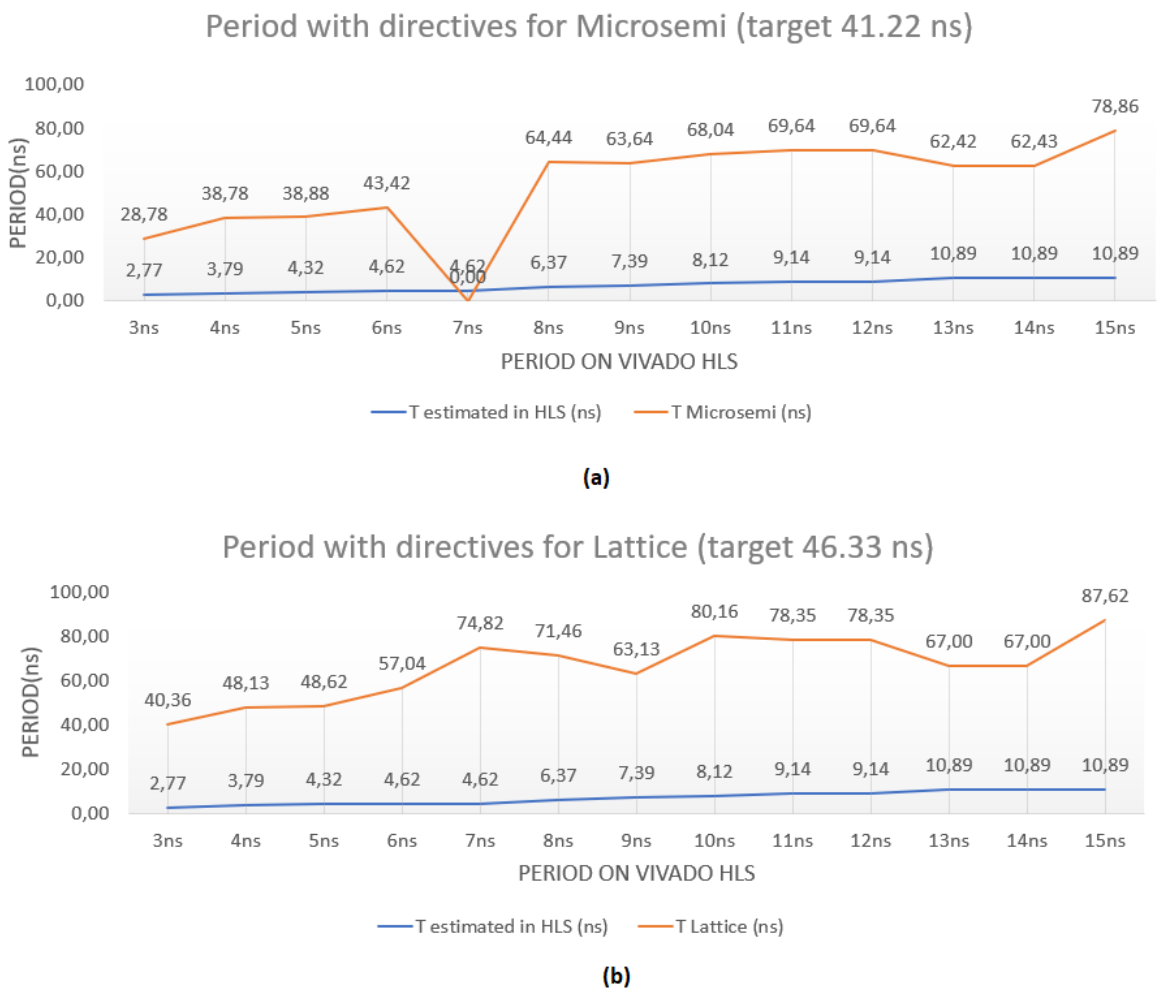


Figure 6.8. Trend of the clock period on the tools of the FPGAs based on the modification at the HLS level from 15 ns to 3 ns with diectives.

6.5.1 DSP mapping on Lattice FPGA

Before to start to take all the data I do an analysis on the DSP of the Lattice FPGA. Among the four FPGAs that are under examination, is the only one that has DSPs on board (it have 8 DSPs), that are dedicated blocks that can be used as a single 16x16 multiplier, or as two independent 8x8 multipliers, or as a 32 MAC bit etc. without going to use so other area. Being that in the neural network code each neuron multiplies the inputs with the weights and then adds these results together, it would be ideal to use DSPs as MAC, the problem is that it can be a maximum of 32 bits, and being that the inputs of the neural network are 16 bits, entering the multiplier and subsequently the adder, these 32 bits of the DSP block

used as MAC are exceeded. So also in the reference programming the DSP is used as a single multiplier of 16x16 while used the LUTs to implement the adders and the activation function. A verification was therefore made to demonstrate that the code that implements multiplication and accumulation written at HLS and then synthesized and put on Radiant tool, behaved in the same way as a code written manually, in particular in both the cases I have to obtain the same result of [1], ie the DSP used as a multiplier while the LUTs as adder.

The handwritten VHDL code tries to best mimic the operations done by the neuron, then the multiplication between two values of 16 bits, after which add them, accumulate the sum and once finished take the output. The latter was taken only on one bit because otherwise the maximum number of pins on this FPGA would have been exceeded. In the Figure 6.9 taken from the Netlist Analyzer of Radiant tool, it can be seen how the multiplier (the red circle) is implemented as DSP while the adder (the green circle) is implemented by a series of Full-Adders (FAs) and therefore with LUTs, where for the dimension of the image I have reported only one FA. Then in Figure 6.10 there is a report of the area used, where one DSP is used.

I then switch to HLS to see if even writing the code at a high level the behavior remained the same. The code was then written which exactly replicates the behavior of the MAC described in VHDL, and the HDL code generated by HLS is then implemented on Radiant tool. In Figure 6.11, the circuit is slightly different from the previous one (I have omitted from the image the part of the datapath related to the state machine that Vivado HLS creates by default), first of all it can be noted that there is another adder (blue circle) used to sum the temporary data, while the final adder (green circle) is used only for the sum of the final result. Apart from this difference which increase the area, and other increase is for the logic of the state machine, the circuit is similar to the previous one and also in this case the multiplier (red circle) is implemented with the DSP while the adder with the FA.

After that, the same experiment was done with the same code A.3, deselecting the interested parties, increasing the number of DSPs first to two and then to three. The behavior remains consistent with that of the single DSP, first two and then three DSPs are used as multipliers and FAs as adders. I have not included the images of the datapaths of the latter for size reasons but in Table 6.2 there are the data of the reports of the four tests made, where is possible to see that for the handwritten DSP the resources used are very few because it was the designer who controlled them, when instead in the DSP written in HLS, the resources increase, and this is due to both the adder that is added to add up the temporary results, but also to the logic that Vivado HLS inserts for the state machine. However, it can be noted that increasing the number of DSPs also increases the resources quite proportionally.

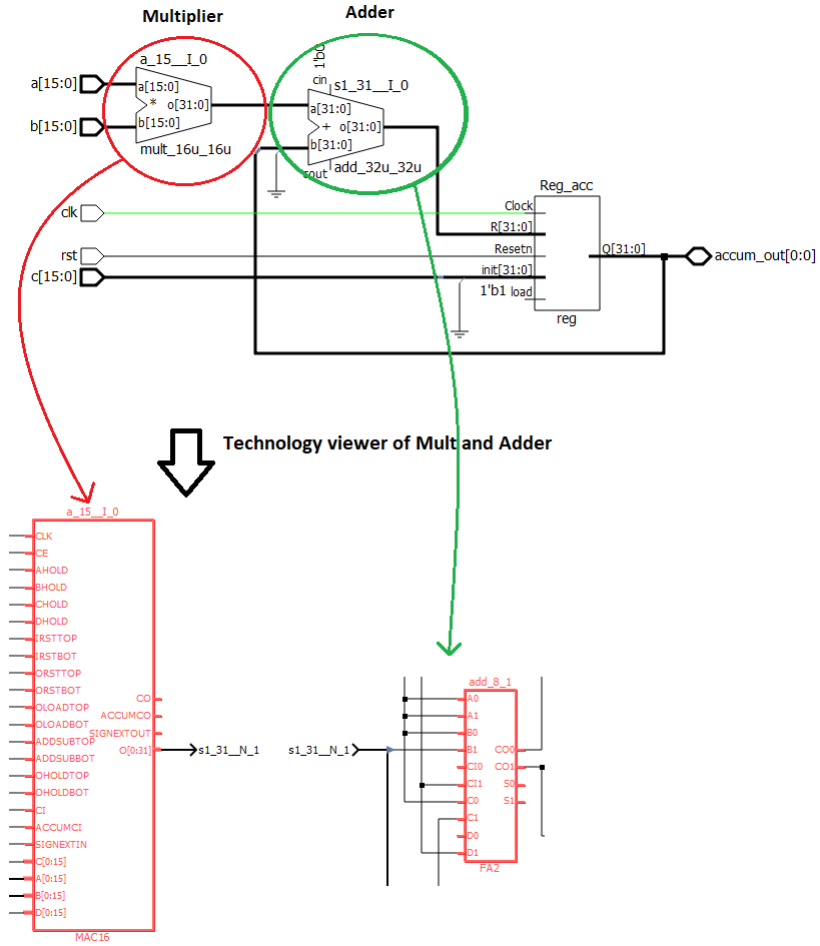


Figure 6.9. Netlist Analyzer: there is the multiplier (red circle) and the adder (green circle) and the register that is the accumulator. Technology View: the multiplier is the DSP and the adder is mapped in a series of full-adders.

	LUT4	PFU Registers	IO Buffers	DSP MULT	Carry Cells
▼ mac	4(0)	2(0)	36(36)	1(1)	10(10)
Reg_acc	4(4)	2(2)	0(0)	0(0)	0(0)

Figure 6.10. Report of Radiant on area used of DSP written manually. List one column DSP used like as multiplier.

After doing this analysis it is shown that the code generated by HLS, even if it has not been adapted for the Lattice family, continues to utilize an amount of resources consistent with the number of DSPs..

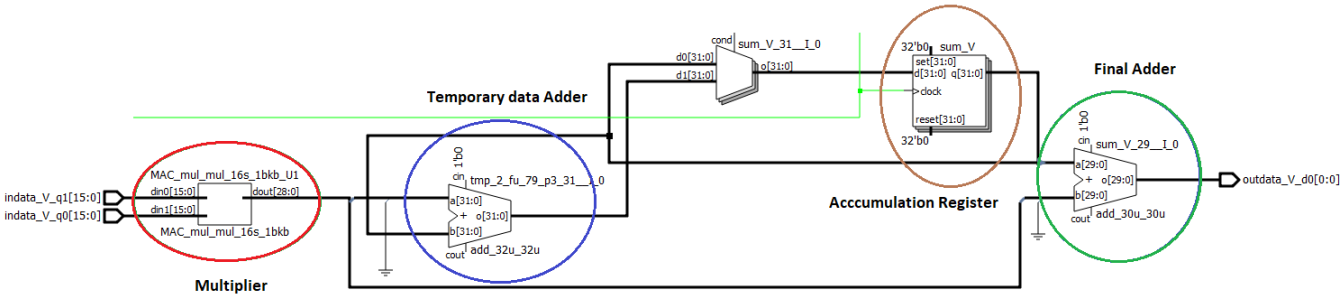


Figure 6.11. Datapath of circuit with one DSP written in HLS. There is the multiplier (red circle) of the two inputs, the adder (blue circle) to sum the temporary data, the accumulation register (brown circle) and the final adder (green circle) that give the output.

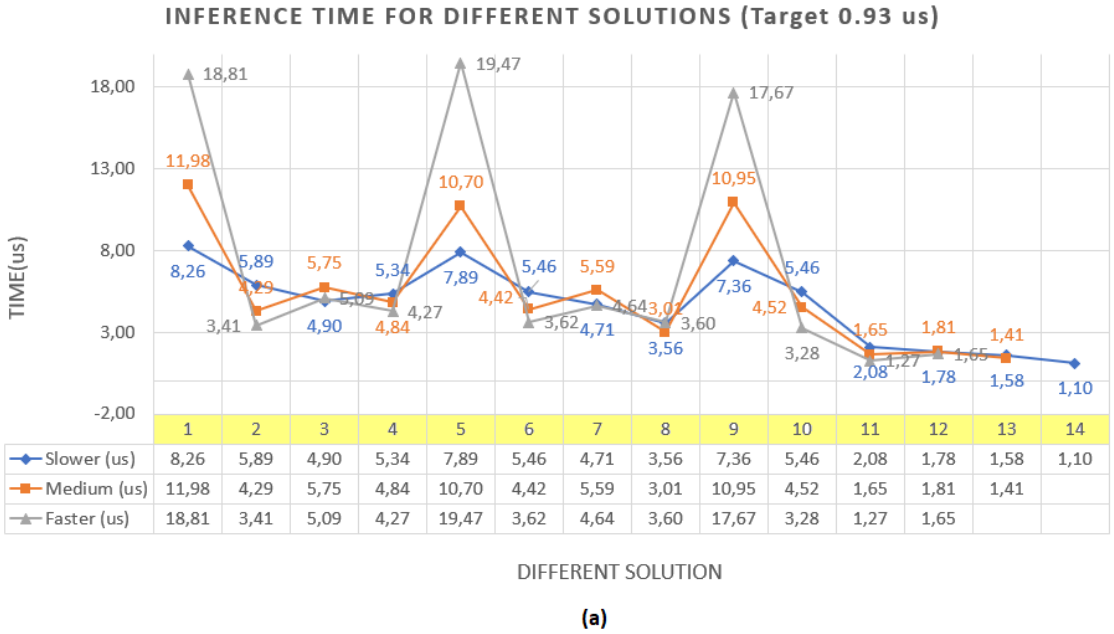
Table 6.2. Resource occupation in terms of LUTs, registers and DSPs of the various tests done on DSPs (DSP written by hand, one DSP written in HLS, two DSP written in HLS and three DSP written in HLS).

DSP under test	LUT4	Registers	DSP
DSP wrote by hand	4	2	1
One DSP wrote in HLS	31	29	1
Two DSP wrote in HLS	63	74	2
Three DSP wrote in HLS	88	101	3

6.5.2 Lattice ICE40UP5K FPGA

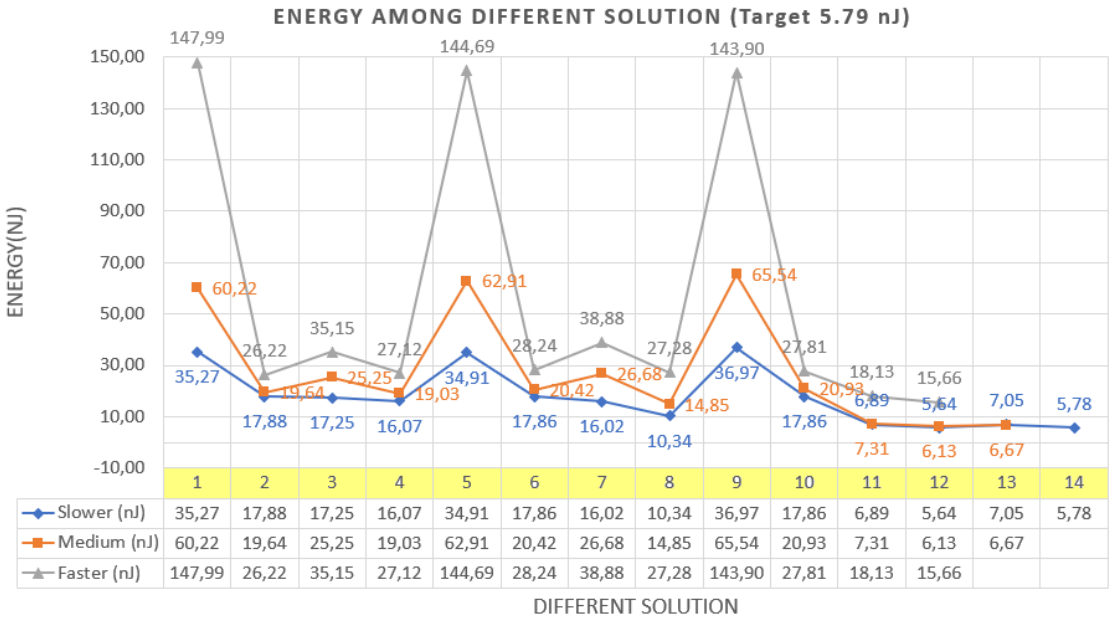
The first FPGA whose performance is analyzed is that of Lattice iCE40 family. The insertion of directives to improve performance is based on the Figure 5.2, in particular I started by limiting the number of multipliers with the ALLOCATION directive to two, then increasing it to four and finally to eight. Switching from one number of multipliers to the other is due to achieving maximum performance with that number of multipliers. The solutions explored are therefore 14 in all and the FPGA choose in HLS to have a behavior similar to that of Lattice is the: Spartan 7 package FTGB196. The graphs of Figure 6.12 summarize the inference time (a) and the energy (b) found in Radiant tool in the analysis for all 14 different solutions. The three lines indicate the different periods set by HLS. The peaks that are seen are due to the change in the number of multipliers, and changing it the performance returns low and the directives are reinserted. It can therefore be noted that the data closest to those of [1], in terms of inference time and energy, are those of solutions 12, 13 and 14, because they are the ones that are closest to

the target inference time (0.93 μs) and target energy (5.79 nJ).



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Slower (us)	8,26	5,89	4,90	5,34	7,89	5,46	4,71	3,56	7,36	5,46	2,08	1,78	1,58	1,10
Medium (us)	11,98	4,29	5,75	4,84	10,70	4,42	5,59	3,01	10,95	4,52	1,65	1,81	1,41	
Faster (us)	18,81	3,41	5,09	4,27	19,47	3,62	4,64	3,60	17,67	3,28	1,27	1,65		

(a)



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Slower (nJ)	35,27	17,88	17,25	16,07	34,91	17,86	16,02	10,34	36,97	17,86	6,89	5,64	7,05	5,78
Medium (nJ)	60,22	19,64	25,25	19,03	62,91	20,42	26,68	14,85	65,54	20,93	7,31	6,13	6,67	
Faster (nJ)	147,99	26,22	35,15	27,12	144,69	28,24	38,88	27,28	143,90	27,81	18,13	15,66		

(b)

Figure 6.12. Inference time (a) and Energy (b) find on Radiant for different 14 Solution for Lattice FPGA ICE40UP5K, for three different clock periods set on HLS (faster, the grey line, medium, the orange line and slower the blue line).

Starting from solution 12, the directives used are shown in Table 6.3. I set the limit on multiplications to eight, set a pipeline with depth two for the first two loops while a pipe with depth one for the last loop, for problems of data-dependency, after which I partitioned all the arrays in such a way as to parallelize the accesses as much as possible and finally I used an unroll of two on all loops.

Table 6.3. Directives used in solution 12, 13 and 14 for Lattice FPGA. The directives are divided in Allocation, Pipeline, Unroll and Array Partition.

Solution	Allocation	Pipeline	Unroll	Array partition
Sol 12	Limit 8 Mult.	II 2 FirstHiddenLayer II 2 NeuralNextLayer LastHiddenLayer	factor 2 FirstHiddenLayer factor 2 NeuralNextLayer factor 2 LastHiddenLayer	complete dim 1 inputs complete dim 2 output_middle complete dim 3 weights complete dim 2 biases
Sol 13	Limit 8 Mult.	II 3 FirstHiddenLayer II 4 NeuralNextLayer LastHiddenLayer	factor 4 FirstHiddenLayer factor 4 NeuralNextLayer factor 4 LastHiddenLayer	complete dim 1 inputs complete dim 2 output_middle complete dim 3 weights complete dim 2 biases
Sol 14	Limit 8 Mult.	FirstHiddenLayer NeuralNextLayer LastHiddenLayer	factor 8 FirstHiddenLayer factor 8 NeuralNextLayer factor 8 LastHiddenLayer	complete dim 1 inputs complete dim 2 output_middle complete dim 3 weights complete dim 2 biases

With these optimizations, for the three different periods set by HLS, the following results are obtained Table 6.4:

Table 6.4. Performance of the solution 12 of the Lattice FPGA ICE40UP5K for three different clock periods set on HLS (25 ns, 20 ns and 5 ns).

Parameter Sol12	**HLS Period**		
	25 ns	**20 ns**	**5 ns**
Pdynamic (mW)	2.82	3.04	9.06
PStatic (mW)	0.34	0.34	0.41
Pactive (mw)	3.16	3.4	9.47
Clock (MHz)	15.15	14.92	22.99
Clock Cycle	27	27	38
Inference Time (μs)	1.78	1.81	1.65
Energy (nJ)	5.64	6.13	15.66
LUTs (max 5280)	1928 (36%)	1992 (37%)	2916 (55%)
DSP	8/8	8/8	8/8

From the point of view of the parameters that most affect the analysis, energy and inference time, the best solution regarding the first parameter is when the HLS clock period is 25 ns, in fact an energy of 5.64 nJ is obtained, and in this case the solution have also an active power very low (3.16 mW). Also the inference time

(1.78 μs) is quite low, but for this latter, the best solution is when the clock period is 5 ns, in fact by setting a smaller period, the inference time decrease and the operating frequency increases (41.84 MHz), at the expense of energy and power which increase three times compared to the two other cases (15.66 nJ and 9.47 mW). As for the occupied LUTs, it should be noted that in all three cases there is still area available, this is because the multipliers, which are the most expensive from the point of view of the area, do not use the LUTs but the dedicated DSPs of this FPGA. Note also that when the speed of the solution increases, the area also increases accordingly.

For solution 13 directives are shown in Table 6.3. The degree of unroll of all loops from 2 to 4 has been increased, this increase also leads to a change in the degree of the pipe.

Regarding this solution, the data is collected only for two clock periods of HLS as in addition to 15 ns it no longer satisfied the minimum period in Vivado HLS and consequently also on Radiant tool. In Table 6.5 there are the results:

Table 6.5. Performance of the solution 13 of the Lattice FPGA ICE40UP5K for two different clock periods set on HLS (25 ns, 15 ns).

Parameter Sol13	**HLS Period**	
	25 ns	**15 ns**
Pdynamic (mW)	4.02	4.27
PStatic (mW)	0.43	0.45
Pactive (mW)	4.45	4.72
Clock (MHz)	14.53	16.3
Clock Cycle	23	23
Inference Time (μs)	1.58	1.41
Energy (nJ)	7.05	6.67
LUTs (max 5280)	2952 (58%)	3010 (59%)
DSP	8/8	8/8

Since in this solution the unroll level and consequently the depth of the pipe has been increased, in order to avoid data-dependecy problems, inevitably the area increases and also the power and energy consumed for both the two speeds of the clock periods by HLS. On the other hand, the parameter that improves compared to the previous solution is the inference time, in fact by parallelizing more it is possible to gain something from the temporal point of view, thus reaching an inference time of 1.58 μs for the 25 ns of HLS and 1.41 μs for 15 ns, and also decrease the clock cycles compared to the previous solution (23 for both HLS periods). The clock frequency for 25 ns of HLS period, being that the directives increase the area and intensify the complexity of the project, is less than in solution 12 (14.53 MHz respect 15.15 MHz).

Solution 14, on the other hand, has only a speed set by HLS as if it had been increased further, the maximum available area of the FPGA would have been exceeded. The directives used are shown in Table 6.3. I have entered the unroll of the loops equal to eight, which is also the limit of the loop, along with the pipeline, thus maximizing the parallelism. Thus it is possible to make eight multiplications of the neuron in parallel. Furthermore, since the loops are totally unrolled, there is no need to insert higher pipeline degrees as in the previous cases.

The collected data are in Table 6.6:

Table 6.6. Performance of the solution 14 of the Lattice FPGA ICE40UP5K for one clock period set on HLS (25 ns).

Parameter Sol14	**HLS Period**
	25 ns
Pdynamic (mW)	4.64
PStatic (mW)	0.58
Pactive (mW)	5.23
Clock (MHz)	12.67
Clock Cycle	14
Inference Time (μs)	1.10
Energy (nJ)	5.78
LUTs (max 5280)	4266 (80%)
DSP	8/8

For this solution I obtain the best inference time of all (1.10 μs), a sign that the parallelization has increased the speed again. The energy is also very low (5.78 nJ), this is because having used an unroll of the loop equal to 8, the complexity has increased again and consequently the operating frequency of the clock and so the energy. This solution is also excellent from an active power point of view (5.23 mW) and has the lowest clock cycles of all solutions. As for the area, on the other hand, resource utilization is 80%, so the limit has been almost reached, again for the directives used.

Summing up in Table 6.7 the values of the data found for the solutions for the Lattice FGPA are compared. Data values are reported only for the clock periods for which optimal results have been found. Solution 14 is the best found from the point of view of inference time and energy (only 0.14 mW more than solution 12 with 20 ns period), the loss is only a little in the clock frequency compared to the other solutions, but also for clock cycles turns out to be the best. So this will be used for comparison to the performance found by manual programming.

Table 6.7. Report data of Lattice FPGA for solution 12, 13 and 14 for different clock periods set on HLS.

Parameter	**Sol 12**		**Sol 13**		**Sol14**
	25 ns	**5 ns**	**25 ns**	**15 ns**	**25 ns**
Pdynamic (mW)	2.82	9.06	4.02	4.27	4.64
PStatic (mW)	0.34	0.41	0.43	0.45	0.58
Pactive (mW)	3.16	9.47	4.45	4.72	5.26
Clock (MHz)	15.15	22.99	14.53	16.3	12.67
Clock Cycle	27	38	23	23	14
Inference Time (μs)	1.78	1.65	1.58	1.41	1.10
Energy (nJ)	5.64	15.66	7.05	6.67	5.78
LUTs (max 5280)	1928 (26%)	2916 (55%)	2852 (58%)	3010 (59%)	4266 (80%)
DSP	8/8	8/8	8/8	8/8	8/8

For this FPGA, the analysis with the real delays of the signals of the *.vcd* file was not carried out as doing it the power values previously found with the *.udb* file did not change.

Finally, a comparison was made between the area occupied by the project if it is implemented on the Lattice FPGA, and between the area occupied by the project if it were instead implemented, through Vivado, on a Xilinx FPGA (Spartan 7 package FTGB196), to see how much compatibility difference there is between the implementation between these two different families. The Table 6.8 summarizes the collected data. As expected, a device suitable for the code created is used, as in the case of Xilinx, the LUTs occupied are much less than implementing it on an FPGA in a different family. This is because the created code has constructs that are better suited to the family for which they are created, if instead these constructs are changed, may not be perfectly compatible and therefore takes up more area than necessary. It should be noted that this analysis makes sense to do it only with the Lattice FPGA as it is the only one that has the DSPs inside, so the remaining occupied area is comparable with that found in Vivado since also it uses DSPs. While for Microsemi, not having these dedicated units, if one compares the area with that found on Vivado it would certainly be less than that found on the latter as all the multipliers would be mapped on the DSPs and the remaining area on the LUTs, while on Libero, the area is all mapped to LUTs, so making this comparison would not make sense.

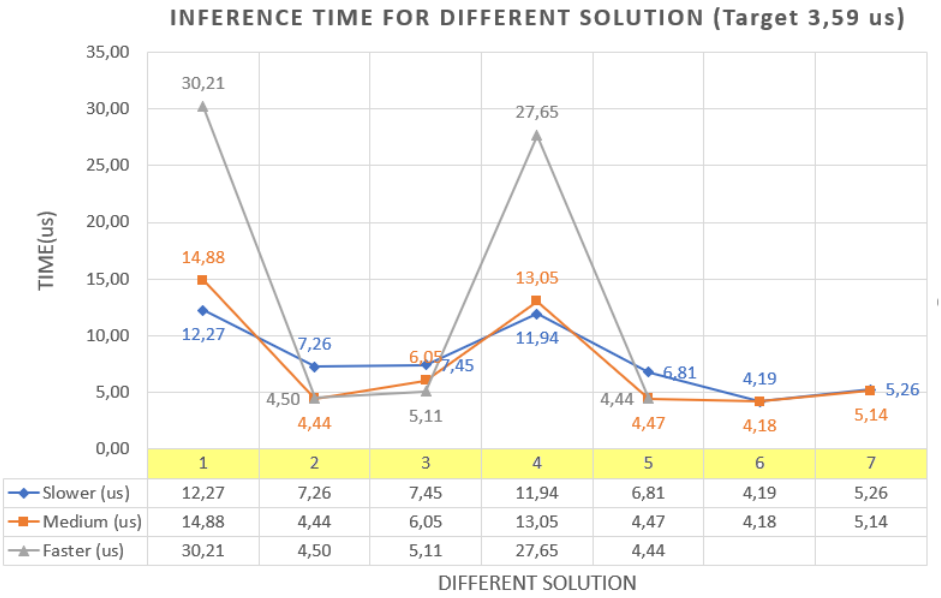
Table 6.8. Comparison between area occupation of the design implemented on the Radiant software using ICE40UP5K and implemented on Vivado using Xilinx FPGA Spartan 7 package FTGB196.

Sol	**HLS T**	**LUTs on Viv.**	**DSP on Viv.**	**LUTs on Lat.**	**DSP on Lat.**
		(max 14600)	(max 80)	(max 5280)	(max 8)
Sol12	25 ns	906 (6.21%)	8	1928 (36%)	8
	20 ns	912 (6.25%)	8	1992 (37%)	8
	5 ns	1093 (7,49%)	8	2916 (36%)	8
Sol13	25 ns	1562 (10.70%)	8	2952 (58%)	8
	15 ns	1682 (11.52%)	8	3010 (57%)	8
Sol14	25 ns	3114 (21.33%)	8	4266 (80%)	8

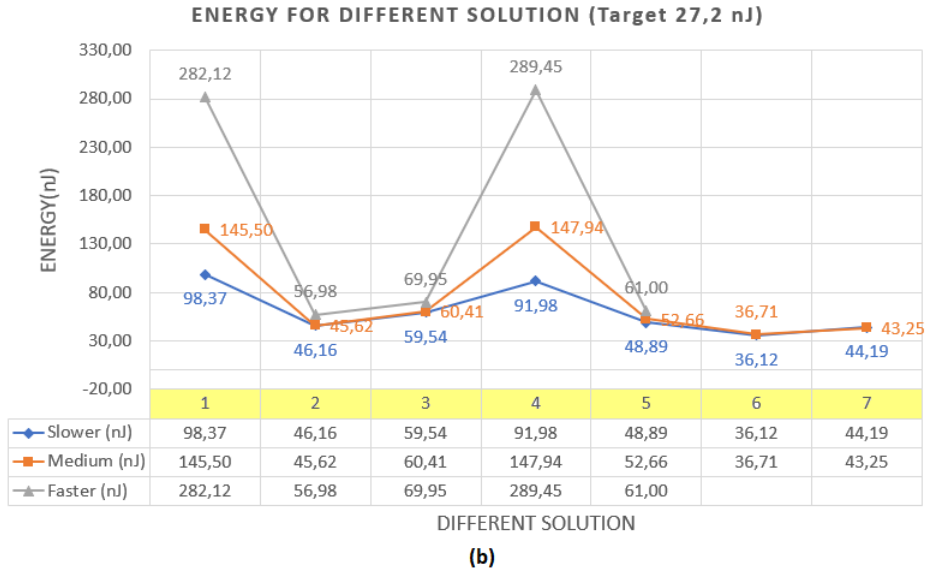
6.5.3 Microsemi AGLN250 FPGA

The FPGA of Microsemi AGLN250 is the smallest in terms of area, in fact only 2 multipliers could be implemented in parallel at a time, because putting more would go beyond the usable area. Seven different solutions have been created on this FPGA, less than the other FPGAs because, having little area available, the explorability is lower. Each solution, like for Lattice FPGA, has been implemented on different HLS clock periods and I put first only one multiplier and then two. The FPGA closest to the one under test chosen on HLS is the Spartan 7 package CPGA196. The graphs showing the evolution of the solutions according to the inference time and energy parameters are those in Figure 6.13, (a) and (b) respectively. The best solutions are number six and seven because they are the ones that are closest to the target inference time and energy (3.59 μs and 27.2 nJ).

Starting from solution 6 the directives used are shown in Table 6.9. As in the reference model I limited the number of multipliers to two, after which I used the pipe directive with different degree depending on the loop due to the few multipliers available and finally partitioned the output_middle and inputs arrays. Unlike the solution for Lattice FPGA, I partitioned only these two arrays for the reason that here I have a lower level of parallelism so partitioning the others too would have been useless.



(a)



(b)

Figure 6.13. Inference time (a) and Energy (b) find on Libero for different 7 solution for Microsemi FPGA AGLN250, for three different periods set on HLS (faster, the grey line, medium,the orange line and slower the blue line).

The collected data is shown in the Table 6.10. The periods set in HLS are only two (20 ns and 17 ns) and also very similar to each other, this is because if the period had been further reduced there would have been an excess of area and therefore the solution would not have been implementable (in fact for 17 ns there

Table 6.9. Directives used in solution 6 and 7 for Microsemi AGLN250 FPGA. The directives are divided in Allocation, Pipeline, Unroll and Array Partition.

Solution	Allocation	Pipeline	Unroll	Array partition
Sol 6	Limit 2 Mult.	II 4 FirstHiddenLayer II 3 NeuralNextLayer LastHiddenLayer	/	complete dim1 inputs complete dim 2 output_middle
Sol 7	Limit 2 Mult.	II 5 FirstHiddenLayer II 8 NeuralNextLayer LastHiddenLayer	/	complete dim1 inputs complete dim 2 output_middle

is an occupancy of 99%). As for the inference time it is almost the same for the two HLS clock periods (4.19 μs and 4.18 μs), same thing for the clock cycles (71 for both) and clock frequency (16.96 MHz and 16.98 MHz). While the energy and power are less for the solution with 20 ns period (8.63 mW and 36.12 nJ). This is because the design go slightly slower. As can be seen from the clock cycles, the output is calculated after 71 cycles, this delay is due to the few multipliers available and therefore to the low degree of parallelization.

Table 6.10. Performance of the solution 6 of the Microsemi FPGA AGLN250 for two different clock periods set on HLS (20 ns and 17 ns).

Parameter Sol6	**HLS Period**	
	20 ns	**17 ns**
Pdynamic (mW)	8.55	8.70
PStatic (mW)	0.077	0.077
Pactive (mW)	8.63	8.80
Clock (MHz)	16.96	16.98
Clock Cycle	71	71
Inference Time (μs)	4.19	4.18
Energy (nJ)	36.12	36.71
LUTs (max 6144)	6051 (98.5%)	6110 (99%)

Instead, for solution 7 the directives used are shown in Table 6.9 in which I tried to relax a little the pipe directive by increasing the degree.

Having relaxed this directive, the clock frequency decreased slightly for both periods (16.53 MHz and 16.91 MHz). But this was not enough to improve the other performances, Table 6.11, in fact, apart from the power, all the other parameters have increased compared to solution 6. Even the area has remained almost the same.

Table 6.11. Performance of the solution 7 of the Microsemi FPGA AGLN250 for two different clock periods set on HLS (20 ns and 17 ns).

Parameter Sol7	**HLS Period**	
	20 ns	**17 ns**
Pdynamic (mW)	8.32	8.33
PStatic (mW)	0.077	0.077
Pactive (mW)	8.40	8.40
Clock (MHz)	16.53	16.91
Clock Cycle	87	87
Inference Time (μs)	5.26	5.14
Energy (nJ)	44.19	43.25
LUTs (max 6144)	5937 (96%)	6061 (98%)

For the Microsemi FPGAs, a further analysis of the power was also made as it was made vectoreless in the first instance and therefore with estimates on the signal delays set by Libero. However, these estimates may not always be very precise, so a power analysis was made based on a .vcd file, created after the post-synthesis simulation, which contains the real delays of the signals, thus having value on the more realistic power and energy. The data of the vectorless and VCD analysis are shown in the Table 6.12, where it can be seen that for both parameters there is an increase with the VCD analysis having the real delays of the signals.

Table 6.12. Power and energy comparison for solution 6 and 7 of Microsemi FPGA AGLN250 between VCD and vectorless analysis for two different clock periods set on HLS (20 ns and 17 ns).

Sol	**Parameters**	**HLS Period**			
		Vectorless		**.vcd**	
		20 ns	**17 ns**	**20 ns**	**17 ns**
Sol6	Pactive (mW)	8.63	8.80	10.11	11.75
	Energy (nJ)	36.12	36.71	43.98	49.14
Sol7	Pactive (mW)	8.40	8.40	11.92	10.88
	Energy (nJ)	44.19	43.25	62.73	55.97

For the comparison with the performances found by reference programming, in Table 6.13 there are the the values of the data find for solution 6 and 7. The solution that has reached the best values is the solution 6 for the 20 ns of HLS period, as it consumes less energy and power than that at 17 ns of period and the inference time and frequency of clock they are practically similar. Solution 7, on the other hand, has worse performance than solution 6, so it has not been taken into consideration.

Table 6.13. Report data of Microsemi AGLN250 FPGA for solution 6 and 7 for different clock periods set on HLS.

Parameter	**Sol 6**		**Sol 7**	
	20 ns	**17 ns**	**20 ns**	**17 ns**
Pdynamic (mW)	8.55	8.70	8.32	8.33
PStatic (mW)	0.077	0.077	0.077	0.077
Pactive (mW)	8.63	8.80	8.40	8.40
Clock (MHz)	16.96	16.98	16.53	16.91
Clock Cycle	71	71	87	87
Inference Time (μs)	4.19	4.18	5.26	5.14
Energy (nJ)	36.12	36.71	44.19	43.25
LUTs (max 5280)	60.51(98.5%)	6110(99%)	59337(96%)	6061(98%)

6.5.4 Microsemi M1AGL600 FPGA

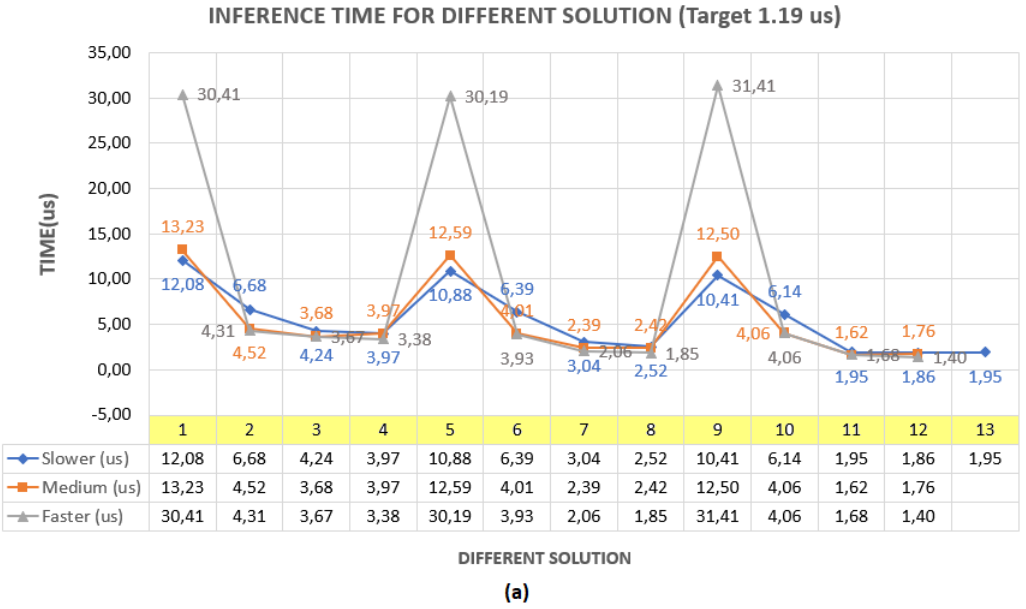
The second Microsemi FPGA is the M1AGL600, for it I experimented 13 different solutions, starting from a low level of exploration by inserting only two multipliers, then increasing to four and finally eight (for the available area 8 was the maximum). The FPGA chosen on the Vivado HLS closest to the one under review is the Artix 7 package CPG238. In Figure 6.14, the trend along the different solutions of inference time (a) and energy (b) is shown. The ones that come closest to target performance (1.19 μs and 17.8 nJ) are solutions 11, 12 and 13.

The directives used for solution 11 are shown in Table 6.14, using the pipeline on each loop, and partitioning the arrays to increase parallelism and avoid over-reading errors.

Table 6.14. Directives used in solution 11, 12 and 13 for Microsemi M1AGL600 FPGA. The directives are divided in Allocation, Pipeline, Unroll and Array Partition.

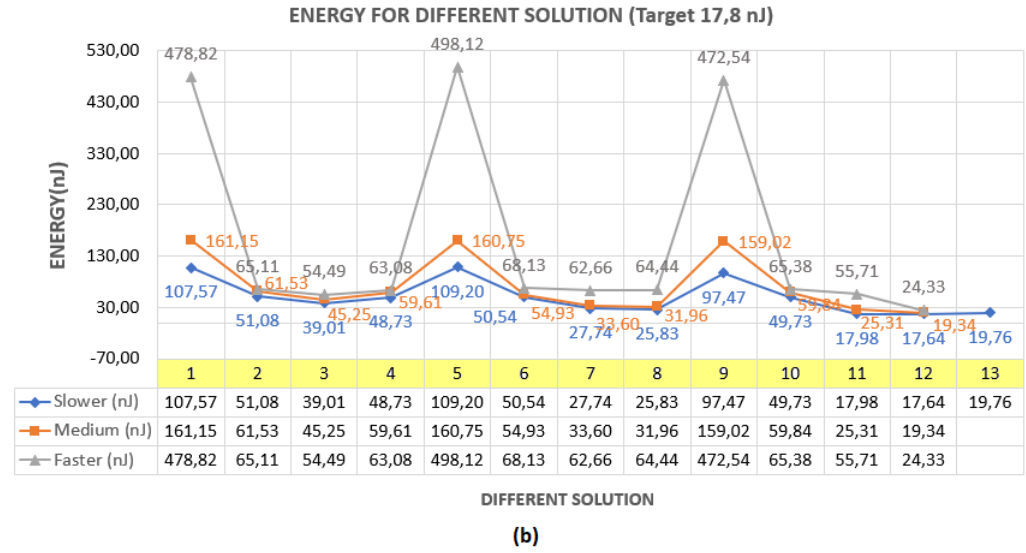
Solution	**Allocation**	**Pipeline**	**Unroll**	**Array partition**
Sol 11	Limit 8 Mult.	FirstHiddenLayer NeuralNextLayer LastHiddenLayer	/	complete dim 1 inputs complete dim 2 output_middle
Sol 12	Limit 8 Mult.	II 2 FirstHiddenLayer II 2 NeuralNextLayer LastHiddenLayer	factor 2 FirstHiddenLayer factor 2 NeuralNextLayer factor 2 LastHiddenLayer	complete dim 1 inputs complete dim 2 output_middle complete dim 3 weights complete dim 2 biases
Sol 13	Limit 8 Mult.	II 3 FirstHiddenLayer II 4 NeuralNextLayer LastHiddenLayer	factor 4 FirstHiddenLayer factor 4 NeuralNextLayer factor 2 LastHiddenLayer	complete dim 1 inputs complete dim 2 output_middle complete dim 3 weights complete dim 2 biases

For this solution the following data was collected, Table 6.15. It is noted that as regards the energy consumed, the best found is equal to 17.98 nJ for 25 ns of HLS period, and always for this period there is still a low inference time if compared to the others (1.95 μs), while the dissipated power (9.23 mW) is the lowest compared to the others. For the solution with the higher HLS period, the clock frequency is the lowest (16.43 MHz). The lowest inference time is found instead for the 20 ns period of HLS (1.62 μs), but the corresponding energy and active power are quite higher than in the previous case (25.31 nJ and 15.62 mW). The faster solution instead (3 ns of period of HLS) is not to be taken into consideration as there is an energy consumed far higher than the previous ones (55.71 nJ), this indicates that if the speed is increased, in fact the clock frequency is the higher (33.27 MHz), consumption will increase accordingly. As far as the occupation of the area is concerned, it can be seen that there is still some space available to insert other optimizations.



	1	2	3	4	5	6	7	8	9	10	11	12	13
Slower (us)	12,08	6,68	4,24	3,97	10,88	6,39	3,04	2,52	10,41	6,14	1,95	1,86	1,95
Medium (us)	13,23	4,52	3,68	3,97	12,59	4,01	2,39	2,42	12,50	4,06	1,62	1,76	
Faster (us)	30,41	4,31	3,67	3,38	30,19	3,93	2,06	1,85	31,41	4,06	1,68	1,40	

(a)



	1	2	3	4	5	6	7	8	9	10	11	12	13
Slower (nJ)	107,57	51,08	39,01	48,73	109,20	50,54	27,74	25,83	97,47	49,73	17,98	17,64	19,76
Medium (nJ)	161,15	61,53	45,25	59,61	160,75	54,93	33,60	31,96	159,02	59,84	25,31	19,34	
Faster (nJ)	478,82	65,11	54,49	63,08	498,12	68,13	62,66	64,44	472,54	65,38	55,71	24,33	

(b)

Figure 6.14. Inference time (a) and Energy (b) find on Libero for different 13 Solution for Microsemi FPGA M1AGL600, for three different periods set on HLS (faster, the grey line, medium, the orange line and slower the blue line.)

In solution 12, the directives inserted are shown in Table 6.14. Here the degree 2 unroll has been inserted into all loops and consequently the pipeline degree has been modified in order to avoid data-dependecy problems and partitioned all arrays.

The values of the parameters for this solution are in Table 6.16. Having increased

Table 6.15. Performance of the solution 11 of the Microsemi FPGA M1AGL600 for three different clock periods set on HLS (25 ns, 20 ns and 3 ns).

Parameter Sol11	**HLS Period**		
	25 ns	**20 ns**	**3 ns**
Pdynamic (mW)	9.09	15.49	32.96
PStatic (mW)	0.13	0.13	0.13
Pactive (mw)	9.23	15.62	33.1
Clock (MHz)	16.43	22.23	33.27
Clock Cycle	32	36	56
Inference Time (μs)	1.95	1.62	1.68
Energy (nJ)	17.98	25.31	55.71
LUTs (max 13824)	9810 (71%)	10599 (76%)	11773 (85%)

the parallelism with the introduction of the new directives, the area has increased compared to the previous one in all three cases of the HLS clock period. The best energy (17.64 nJ), is the one found with the slowest period, and it is also the best with respect to those found for solution 11. The best inference time, on the other hand, is obtained with 10 ns of period of HLS (1.40 μs), however, having increased the speed both the power and the energy increase compared to the others. The clock frequency increases together with the HLS clock period increase, finding the maximum with the fastest period (20.27 MHz). While the number of clock cycles is very similar for all three cases (27, 27 and 29).

Table 6.16. Performance of the solution 12 of the Microsemi FPGA M1AGL600 for three different clock periods set on HLS (25 ns, 20 ns and 10 ns).

Parameter Sol12	**HLS Period**		
	25 ns	**20 ns**	**10 ns**
Pdynamic (mW)	9.34	10.84	17.25
PStatic (mW)	0.13	0.13	0.13
Pactive (mw)	9.47	10.96	17.39
Clock (MHz)	14.50	15.31	20.72
Clock Cycle	27	27	29
Inference Time (μs)	1.86	1.76	1.40
Energy (nJ)	17.64	19.34	24.33
LUTs (max 13824)	11476 (83%)	12274 (88.79%)	12972 (93.84%)

Finally, with solution 13, directives in Table 6.14, I tried to increase the parallelism again, as far as the occupation of the area allowed me, so the degree of unroll was increased to 4 for the first two loops, with consequent adaptation of the degree of pipe.

With the increase in parallelism it was thought of a possible performance improvement, however, as shown in Table 6.17, the only data that has suffered an improvement compared to the previous solutions are the clock cycles (25). All the remaining data have suffered a worsening and this is a sign that even having increased the parallelism, the congestion and the complexity (in fact, here it occupies 99.93% of the area) that is created between the signals, instead of improving performance, deteriorate them. Furthermore, only one solution for a period of HLS (25 ns) has been explored as decreasing it would increase the area too much, exceeding the limit.

Table 6.17. Performance of the solution 13 of the Microsemi FPGA M1AGL600 for one different clock period set on HLS (25 ns).

Parameter Sol13	**HLS Period**
	25 ns
Pdynamic (mW)	9.98
PStatic (mW)	0.13
Pactive (mW)	10.12
Clock (MHz)	12.8
Clock Cycle	25
Inference Time (μs)	1.95
Energy (nJ)	19.76
LUTs (max 13824)	13815 (99.93%)

As for the AGLN250 FPGA, the power analysis was also made starting from the .vcd file, and also in this case, as in the FPGA above, an increase in both power and energy appears. If the energy and power are taken, Table 6.18, for the case VCD, of solution 11 for a HLS period of 20 ns, it can be seen that they are the lowest compared to all the others, and are almost comparable to those found with the vectorless analysis (9.87 mW and 19.24 nJ). The same parameters for the other solutions instead, compared to the vectorless case, are almost always double if not triple.

Table 6.18. Power and energy comparison for solution 11, 12 and 13 of Microsemi M1AGL600 between VCD and vectorless analysis for three different clock periods set on HLS.

Sol	Parameters	HLS Period					
		Vectorless			.vcd		
		Slower	Medium	Faster	Slower	Medium	Faster
Sol11	Pactive (mW)	9.23	15.62	33.1	9.87	19.42	35.94
	Energy (nJ)	17.98	25.31	55.71	19.24	31.46	60.5
Sol12	Pactive (mW)	9.47	10.96	17.39	17.29	19.79	47.62
	Energy (nJ)	17.64	19.34	24.33	32.18	34.9	66.62
Sol13	Pactive (mW)	10.12	/	/	28.32	/	/
	Energy (nJ)	19.76	/	/	55.31	/	/

Summing up in Table 6.19 the values of the data found for the solutions for the Microsemi M1AGL600 FGPA are compared. Data values are reported only for the clock periods for which optimal results have been found. The best solution found is solution 12 for the 20 ns period of HLS. In fact, there is a very low inference time (1.76 μs) with a correspondingly low energy and power consumption if compared to that of the other solutions. The clock frequency is also quite high and only two clock cycles are lost compared to the solution 13 which has 27, but there is a gain in all the other parameters.

Table 6.19. Report data of Microsemi M1AGL600 FPGA for solution 11, 12 and 13 for different clock periods set on HLS.

Parameter	Sol 11		Sol 12		Sol13
	25 ns	20 ns	20 ns	10 ns	25 ns
Pdynamic (mW)	9.09	15.49	10.84	17.25	9.98
PStatic (mW)	0.13	0.13	0.13	0.13	0.13
Pactive (mW)	9.23	15.62	10.96	17.39	10.12
Clock (MHz)	16.43	22.23	15.31	20.72	12.8
Clock Cycle	32	33	27	29	25
Inference Time (μs)	1.95	1.62	1.76	1.40	1.95
Energy (nJ)	17.98	25.31	19.34	24.33	19.76
LUTs (max 5280)	9810 (71%)	10599 (76%)	12274 (88.79%)	12972 (93.84%)	13815 (99.93%)

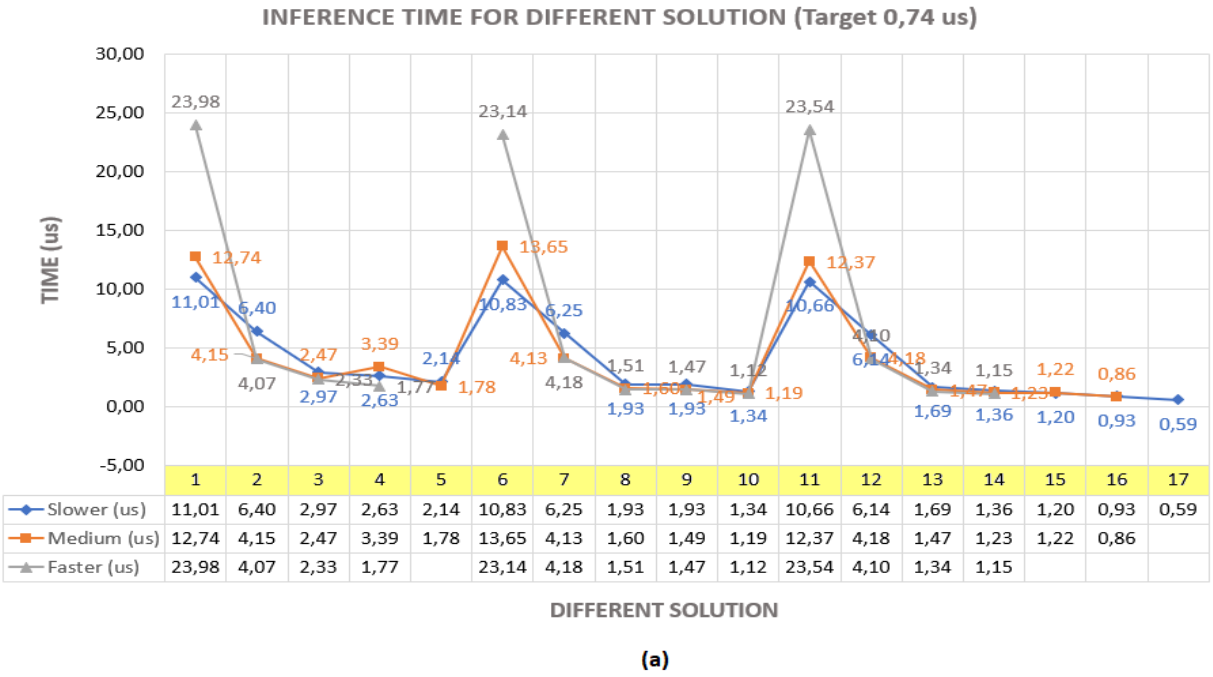
6.5.5 Microsemi M1AGL1000 FPGA

The last FPGA to look at is the M1AGL1000 of Microsemi FPGA, which is the largest in size. In fact, there are 17 solutions explored on it, starting from having only four multipliers, then eight and then even 16 given that the area allows it. The FPGA chosen on Vivado HLS that comes closest to the characteristics is the Spartan 7 package FTGB196. For these 17 solutions, the inference time (a) and energy (b) trends are shown in Figure 6.15, where it can be seen that the best performances are achieved by solutions 16 and 17.

The directives used in solution 16 are shown in Table 6.20. I limit the number of multipliers to 16, use grade 2 pipeline and grade 4 unroll on the first two loops, while pipeline and grade 8 unroll for the last loop. After that the arrays will be partitioned in order to parallelize access and avoid over reading.

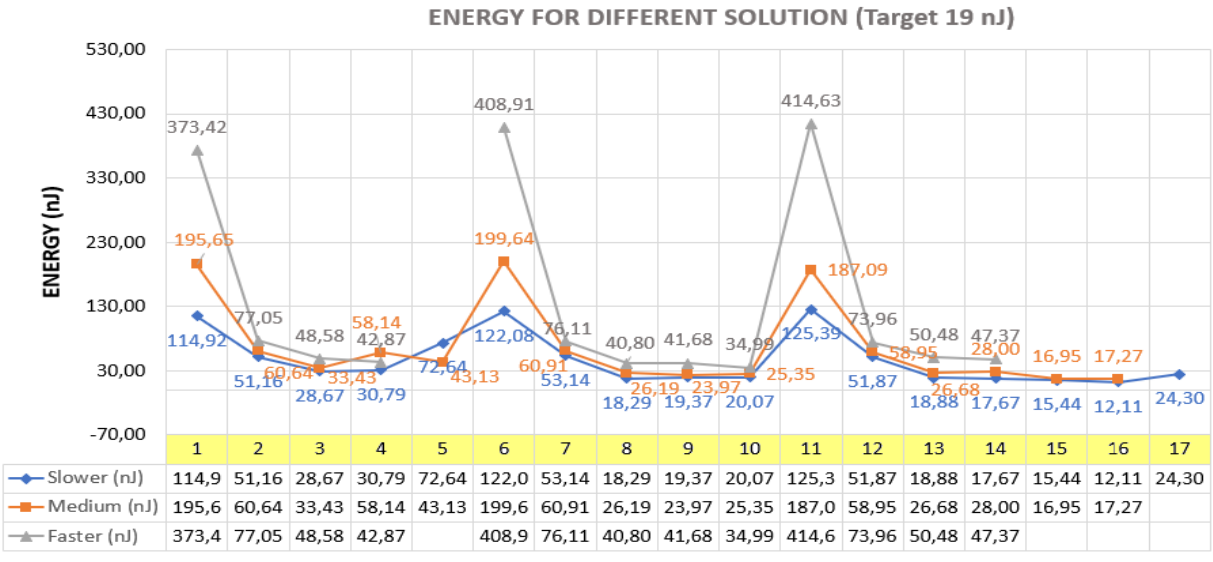
Table 6.20. Directives used in solution 16 and 17 for M1AGL1000 of Microsemi FPGA. The directives are divided in Allocation, Pipeline, Unroll and Array Partition.

Solution	**Allocation**	**Pipeline**	**Unroll**	**Array partition**
Sol 16	Limit 16 Mult.	II 2 FirstHiddenLayer II 2 NeuralNextLayer LastHiddenLayer	factor 4 FirstHiddenLayer factor 4 NeuralNextLayer factor 8 LastHiddenLayer	complete dim1 inputs complete dim 2 output_middle complete dim 3 weights complete dim 2 biases
Sol 17	Limit 16 Mult.	FirstHiddenLayer NeuralNextLayer LastHiddenLayer	factor 8 FirstHiddenLayer factor 8 NeuralNextLayer factor 8 LastHiddenLayer	complete dim1 inputs complete dim 2 output_middle complete dim 3 weights complete dim 2 biases



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Slower (us)	11,01	6,40	2,97	2,63	2,14	10,83	6,25	1,93	1,93	1,34	10,66	6,14	1,69	1,36	1,20	0,93	0,59
Medium (us)	12,74	4,15	2,47	3,39	1,78	13,65	4,13	1,60	1,49	1,19	12,37	4,18	1,47	1,23	1,22	0,86	
Faster (us)	23,98	4,07	2,33	1,77		23,14	4,18	1,51	1,47	1,12	23,54	4,10	1,34	1,15			

(a)



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Slower (nJ)	114,9	51,16	28,67	30,79	72,64	122,0	53,14	18,29	19,37	20,07	125,3	51,87	18,88	17,67	15,44	12,11	24,30
Medium (nJ)	195,6	60,64	33,43	58,14	43,13	199,6	60,91	26,19	23,97	25,35	187,0	58,95	26,68	28,00	16,95	17,27	
Faster (nJ)	373,4	77,05	48,58	42,87		408,9	76,11	40,80	41,68	34,99	414,6	73,96	50,48	47,37			

(b)

Figure 6.15. Inference time (a) and Energy (b) find on Libero for different 17 Solutions for Microsemi FPGA M1AGL1000, for three different periods set on HLS (faster, the grey line, medium, the orange line and slower the blue line).

The values of the collected data are in Table 6.21. Regarding the inference time, solution 16, for 15 ns of period of HLS, has the lowest one (0.86 μs), while both power and energy are a little larger (20.02 mW and 17.27 nJ) compared to values of the same solution but with a period of 25 ns (12.98 mW and 12.11 nJ). This because the latter one goes slower, therefore it consumes less. The number of clock cycles on the other hand are almost equal (13 and 14).

Table 6.21. Performance of the solution 16 of the Microsemi FPGA M1AGL1000 for two different clock periods set on HLS (25 ns and 15 ns).

Parameter Sol16	**HLS Period**	
	25 ns	**15 ns**
Pdynamic (mW)	12.76	19.80
PStatic (mW)	0.21	0.21
Pactive (mW)	12.98	20.02
Clock (MHz)	13.93	16.23
Clock Cycle	13	14
Inference Time (μs)	0.93	0.86
Energy (nJ)	12.11	17.27
LUTs (max 24576)	20876(84.74%)	23890(94%)

For solution 17, since for solution 16 with 25 ns of period there was still about 15% of usable area, I tried to parallelize it to the maximum, thus unrolling the loops with a degree of 8 together with the pipe. The directives are shown in Table 6.20

The data collected for this solution, Table 6.22 were collected only for a period of HLS (25 ns) being that, if it had been further increased, the available area would have exceeded. At first glance it seems that the data found for this solution are the best of all: in fact the inference time is lower than the previous solution (0.59 μs), even the clock cycles (8), even the clock frequency has not changed much (now 13.45 MHz vs 13.93 MHz). Having found these data, and since the area has also increased by 10% compared to before, one would have expected an increase in the energy consumed and power, perhaps with values similar to those found for solution 16 but for 20 ns of period, instead both the power and the energy have increased in an "anomalous" way. In fact, for the energy there is a value that is double compared to the previous case (24.30 nJ), but the biggest problem is the power which has a value that is almost quadruple (40.87 mW).

Given the anomaly of the power and energy values found, it was particularly useful in this case to perform the power analysis starting from the .vcd file, so that it is made to start from the real delays of the signals. As the Table 6.23 shows, for solution 16, in which consistent results were found, there is an increase in both energy and power, while for solution 17 there is a much lower power than in the

Table 6.22. Performance of the solution 17 of the Microsemi FPGA M1AGL1000 for one different clock period set on HLS (25 ns).

Parameter Sol17	**HLS Period**
	25 ns
Pdynamic (mW)	40.65
PStatic (mW)	0.216
Pactive (mW)	40.87
Clock (MHz)	13.45
Clock Cycle	8
Inference Time (μs)	0.59
Energy (nJ)	24.30
LUTs (max 24576)	23287(94.74%)

case vectorless (12.7 mW), and the same thing happens for energy (7.62 nJ). So in this particular case, doing the power analysis with real delays was more useful than the vectorless analysis.

Table 6.23. Power and energy comparison for solution 16 and 17 of Microsemi FPGA M1AGL1000 between VCD and vectorless analysis for two different clock periods set on HLS.

Sol	**Parameters**	**HLS Period**			
		Vectorless		**.vcd**	
		25 ns	**20 ns**	**25 ns**	**20 ns**
Sol16	Pactive (mW)	12.76	19.80	28.85	30.89
	Energy (nJ)	12.11	17.27	27.21	26.84
Sol17	Pactive (mW)	40.65	/	12.48	/
	Energy (nJ)	24.30	/	7.62	/

Finally, for the latest FPGA under review in Table 6.24 the values of the data found for the solutions for the Microsemi M1AGL1000 FGPA are compared. The best solution found is solution 16 for the 15 ns period of HLS. Solution 17 has excellent parameters, but due to the strange results on power and energy it cannot be taken into consideration, besides the fact that the comparison with manual programming results is done on the basis of a vectorless simulation.

Table 6.24. Report data of Microsemi M1AGL1000 FPGA for solution 16 and 17 for different clock period set on HLS.

Parameter	**Sol 16**		**Sol 17**
	25 ns	**15 ns**	**25 ns**
Pdynamic (mW)	12.76	19.80	40.65
PStatic (mW)	0.21	0.21	0.216
Pactive (mW)	12.98	20.02	40.87
Clock(MHz)	13.93	16.23	13.45
Clock Cycle	13	14	8
Inference Time (μs)	0.93	0.89	0.59
Energy (nJ)	12.11	17.27	24.30
LUTs (max 5280)	20876(84.74%)	23890(94%)	23287(94.74%)

6.5.6 Comparison with objectives and manual flow

After collecting the parameter values of all four FPGAs under review, a comparison is made with those found manual programming. As mentioned, only one of the proposed solutions is examined, the best, so as to have the best possible comparison. The solutions taken are:

- solution 14 for 25 ns of period of HLS for ICE40UP5K of Lattice (LAT).
- solution 6 for 20 ns of period of HLS for AGLN250 of Microsemi (MS-S).
- solution 12 for 20 ns of period of HLS for M1AGL600 of Microsemi (MS-M)
- solution 16 for 15 ns of period of HLS for M1AGL1000 of Microsemi(MS-L).

Below, Table 6.25, the data values for the various FPGAs of the chosen solutions compared with those of manual programming, and in Table 6.26, the comparison for the resources occupation.

The results are now shown by comparing manual programming solution found for a given FPGA with mine.

Table 6.25. Performance find in the experiment of manual programming and perfomance for the best solutions find in my experiment for ICE40UP5K FPGA of Lattice and for AGLN250, M1AGL600, M1AGL1000 FPGAs of Microsemi.

Parameters	**HLS Results**				**Manual programming Results**			
	LAT	**MS-S**	**MS-M**	**MS-L**	**LAT**	**MS-S**	**MS-M**	**MS-L**
Pdynamic (mW)	4.64	8.55	10.83	19.80	5.97	7.50	14.9	25.5
PStatic (mW)	0.582	0.077	0.134	0.214	0.277	0.079	0.131	0.195
Pactive (mW)	5.23	8.63	10.96	20.02	6.24	7.58	15	25.7
Clock (MHz)	12.67	16.53	15.30	16.23	21.58	24.26	16.83	16.26
Clock Cycle	14	71	27	14	20	87	20	12
Inference Time (μs)	1.10	4.19	1.76	0.86	0.93	3.59	1.19	0.74
Energy(nJ)	5.78	36.12	19.34	17.27	5.79	27.2	17.8	19

Table 6.26. Resource occupation find in the experiment of manual programming and resource occupation for the best solutions find in my experiment for ICE40UP5K FPGA of Lattice and for AGLN250, M1AGL600, M1AGL1000 FPGAs of Microsemi.

Param.	**HLS Results**				**Manual programming Results**			
	LAT	**MS-S**	**MS-M**	**MS-L**	**LAT**	**MS-S**	**MS-M**	**MS-L**
Neuron	1	1	1	2	1	1	1	2
Mult.	8	2	8	16	8	2	8	16
LUTs	4266 (80%)	6051 (98.5%)	12274 (88.8%)	23890 (94%)	2047 (38.77%)	4889 (79.57%)	10031 (73.6%)	17842 (75.98%)
DSP	8/8	/	/	/	8/8	/	/	/

- LAT: For the Lattice FPGA not all parameters found by the HLS solution were able to match those found with manual programming. For the HLS solution the parameters that have suffered an improvement are active power (5.23 mW against 6.24 mW) and clock cycles (14 against 20). The fact that the active power is lower is mainly due to the fact that in my solution the clock frequency is lower (12.67 MHz) than that manual programming (21.58 MHz), but despite this, the inference time found is 1.10 us which does not differ so much from 0.93 us of manual programming. As for the energy found, it is practically the same (5.78 nJ and 5.79 nJ). The occupied area, on the other hand, is much more than that found with HLS (80% compared to 38.77%), this is because, as explained above, there is no control over the blocks and therefore the area that HLS will use for the design, so it is very likely that it puts many more blocks, which for the own internal logic are correct, compared to those a designer who does elaborate the project by hand would put. Compare it to all the

others, this FPGA is the one that consumes less active power (5.23 mW) and energy (5.78 nJ) than the other results found by HLS but also those found with manual programming.

Another parameter to consider for comparison is the way operations are scheduled by HLS and with manual scheduling. Figure 6.16 reports the scheduling of operations from Vivado HLS. The latency to obtain the output is 14 clock cycle as shown also by the Table 6.25, against the 20 find on manual programming. Since both pipeline and a degree 8 unroll have been used, all loops are completely unrolled, and therefore at every clock cycle are performed 8 multiplications. In the first loop, as explained in the Chapter 5 Section 5.3, there are a total of 40 multiplications, so since eight multiplications are performed every clock cycle, they will be used five to perform all the multiplications of the neurons of the first loop. While for the second loop there are 64 multiplications in all, so they will be distributed over eight clock cycles. Finally the last loop only uses one clock cycle as it has only eight multiplications.

Scheduling LAT	Clock Cycles													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
First Loop	8 Mult													
	N1													
		8 Mult												
		N2,N3	8 Mult											
			N4	8 Mult										
				N5,N6	8 Mult									
					N7,N8									
Second Loop						8 Mult								
						N1	8 Mult							
							N2	8 Mult						
								N3	8 Mult					
									N4	8 Mult				
										N5	8 Mult			
											N6	8 Mult		
												N7	8 Mult	
													N8	
Last Loop														8 Mult
														Nout

Figure 6.16. Scheduling find on Vivado HLS in the Analysis view for the solution of Lattice FPGA. The green clock cycles belong to the first loop, the red belong to the second loop and the blue belong at the last loop.

- MS-S: For this FPGA, since being the smallest there was less possibility of exploration at the HLS level, manual programming results were not achieved but acceptable results were nevertheless obtained. The target values have not been reached because with the addition of the directives the area is also increased, and it is not much so the occupation immediately reached the limit,

moreover, as already explained, from HLS there is no control on the area that will be occupied, therefore in addition to the area for the added directives there is also the occupation due to all the logic that HLS adds. In fact, the occupied area is 99.5% against 79.57% of the target. As for the performance, the only improved parameter are the clock cycle (71 of HLS against 87 of manual programming), the remaining parameters are good because they are very close to those found in manual programming but are not better than them. For example, the inference time that I found is 4.19 μs, while that of the target is 3.59 μs, so it's not so different, about 16% more. Also for the energy of the same reasoning can be made, in fact in the case of HLS there is an increase of about 32% compared to reference programming (36.12 nJ compared to 27.2 nJ), which therefore remains quite low. Note that this FPGA is the one that consumes the least from the point of view of static power and have also the bigger clock frequency (16.53 MHz) respect to the other values find for the other FPGAs on HLS.

The scheduling for this FPGA have too many clock cycles, so I report only the scheduling of how the single neuron is obtained for each loop. Figure 6.17, shows the scheduling for ALGN250. The orange line where first loop is written indicates that that is a loop, specifically a loop with a trip count (i.e. how many times it is iterated) of eight since it has not been used to unroll on this FPGA. In the previous Lattice scheduling this is not present as all the loops have been unrolled. The first loop (a) lasts three clock cycles, and a total of five multiplications are done to get the output of the neuron of the first layer, and this loop is executed eight times. The second loop (b) lasts four clock cycles in which 2 multiplications are performed in each of them, and will also be performed eight times. The last loop (c), on the other hand, lasts only one clock cycle, multiplied by eight iterations to obtain the final output. If the total clock cycles are counted they are: $8\times3 = 24$ for the first loop, $8\times4 = 32$ for the second and $8\times1 = 8$ for the third for a total of 64 clock cycles, those that are missing to get to 71 are operations that Vivado HLS adds but does not schedule.

Scheduling MS-S	Clock Cycles		
	1	2	3
	First Loop		
Multiplication	2 Mult		
		2 Mult	
			1Mult
			N1

(a)

Scheduling MS-S	Clock Cycles			
	1	2	3	4
	Second Loop			
Multiplication	2 Mult			
		2 Mult		
			2 Mult	
				2 Mult
				N1

(b)

Scheduling MS-S	Clock Cycles
	1
	Last Loop
Multiplication	1 Mult
After 8 Iteration	Nout

(c)

Figure 6.17. Scheduling of the multiplications to obtain one output of the neuron in the various clock cycles for all three loops (green for the first loop, red for the second and blue for the last) of the neural network for the solution for Microsemi FPGA AGLN250.

- MS-M: Also for this FPGA good results are obtained but not all parameters are able to reach the target ones. The inference time value found by HLS is 1.76 μs which is 50% more than the value found from manual programming. While the comparison between the found energy is very similar (19.34 nJ vs 17.8 nJ) and also the found clock frequency and clock cycles (15.31 MHz and 27) are similar to those found with manual programming (16.83 MHz and 20). The discussion of the area is also valid in this case, in fact from HLS the design have an occupation of the area of 88.79% compared to 73.87%, the difference between the two decreases as being larger than the previous FPGA, the directives and the logic added by HSL fit better because they find a larger area.

  The scheduling of this FPGA is shown in Figure 6.18. In the columns I have reported the multiplications that are performed each clock cycle and the order of execution in the various iterations of the neurons. In the first loop, four iterations are performed for two clock cycles in which 10 multiplications are performed in all, so at the end of the fourth iteration all the neurons and 40 multiplications are performed. In the second loop the scheme is the same but at

the first clock cycle 8 multiplications are performed and in the second another 8, so there are 64 in all. In the last loop 2 multiplications are performed by four iterations so as to have the eight multiplications of the final neuron. In all, therefore, there are: $4\times2 = 8$ clock cycles for the second loop, $4\times2 = 8$ clock cycles for the first loop and $4\times1 = 8$ clock cycles for the last loop, so in all there are 20 clock cycles. The seven that are listed for 27 are added by Vivado HLS with the internal operations.

Scheduling MS-M	Clock Cycles				
	1	2	3	4	5
	First Loop				
Multiplication	8 Mult				
		2 Mult			
First Iteration	N1				
		N2			
Second Iteration	N3				
		N4			
Third Iteration	N5				
		N6			
Four Iteration	N7				
		N8			
			Second Loop		
Multiplication			8 Mult		
				8Mult	
First Iteration			N1		
				N2	
Second Iteration			N3		
				N4	
Third Iteration			N5		
				N6	
Four Iteration			N7		
				N8	
					Last loop
Multiplication					2 Mult
1,2,3 4 Iteration					Nout

Figure 6.18. Scheduling find on Vivado HLS in the Analysis view for the solution of Microsemi FPGA M1AGL600. The multiplication in every clock cycles are written in blue. In green there are every neurons calculated in the various clock cycles of the first loop, in red there are every neurons calculated in the various clock cycles of the second loop and in blue there is the neuron calculated in the various clock cycles of the last loop.

- MS-L: For the latest FPGA the results found are excellent, in fact the results of manual programming have been reached or exceeded. Starting from inference

time, a value of 0.86 μs was found which is only 17% compared to the 0.74 μs found in manual programming. The energy consumed by the HLS solution, on the other hand, is lower (17.27 nJ against 19 nJ) and also the active power (20.02 mW against 25 mW). The clock frequency and clock cycles, on the other hand, are very close to those of reference (16.23 MHz and 14 clock cycles against 16.26 MHz and 12 clock cycles). The occupation of the area in the HLS has almost reached the maximum (94.74% compared to 75.98% of the manual programming). Compared to the results found by HLS, this FPGA has the shortest inference time (0.86 μs), and only the correspondent in manual programming has the shortest inference time (0.74 μs).

The scheduling is shown in Figure 6.19. In the first loop two iterations are performed, each of two clock cycles, in which a total of 20 multiplications are performed, for two iterations, then 40. In the second loop the scheme is the same but in the two clock cycles a total of 32 multiplications are performed, for 2 times, therefore 64 in all. The last loop is completely unrolled, Vivado HLS chooses to execute six multiplications in the first clock cycle and in the last two to complete the eight multiplications of the final neuron. In all the clock cycles are: $2\times2 = 4$ of the first loop, $2\times2 = 4$ and two of the last, in all there are 10 and the 4 that are missing are due to the internal operations of Vivado HLS.

Scheduling MS-L	Clock Cycles					
	1	2	3	4	5	6
	First Loop					
	16 Mult					
Multiplication		4 Mult				
	N1,N2,N3					
First Iteration		N4				
	N5,N6,N7					
Second Iteration		N8				
			Second Loop			
			16 Mult			
Multiplication				16 Mult		
			N1,N2			
First Iteration				N3,N4		
			N5,N6			
Second Iteration				N7,N8		
					6 Mult	
						2 Mult
						Nout

Figure 6.19. Scheduling find on Vivado HLS in the Analysis view for the solution of Microsemi FPGA M1AGL1000. The multiplication in every clock cycles are written in blue. In green there are every neurons calculated in the various clock cycles of the first loop, in red there are every neurons calculated in the various clock cycles of the second loop and in blue there is the last neuron calculated in the various clock cycles.

Chapter 7

Conclusion

In the experiment I dealt with in the course of this thesis I tried to understand if was possible synthesize a neural network, used in the field of capacitive sensors, through the use of an HLS tool, Vivado HLS, and then implement the code created not on the FPGA of the same family of the tool, but on families of ultralow-power FPGAs, therefore problems of adaptability between code and FPGA could arise since it was not created specifically for that device. Different solutions have been created for the four FPGAs under test, and for each of these the period from HLS has also been changed, thus making it faster or slower, and we have seen the effect on the final result.

The best results were obtained for the Lattice FPGAs (ICE40UP5K) and the Microsemi M1AGL1000. For the first, since on this there are dedicated blocks that perform multiplications (DSPs), it was possible to explore the parallelism to the maximum, in fact it is the only one in which it was possible to insert the degree of unroll equal to 8 together with the pipe and at the same time obtain excellent results, the obtained active power (5.23 mW) and the energy (5.78 nJ) are the lowest among all the FPGAs both for programming via HLS and manual (6.24 mW and 5.79 nJ for the manual programming reference). For the second one, thanks to the large area available, it was possible to explore a lot, also obtaining better results than those of manual programming, in fact the energy found with HLS is 17.27 nJ against 19 nJ of the manual programming, and also the inference time is slightly greater than that found with manual programming (0.86 μs against 0.74 μs). For the remaining two Microsemi FPGAs, AGLN250 and M1AGL600, not having the dedicated blocks and not having a lot of area available, especially for the first one, it was not possible to explore at best so results of reference programming are not reach, but however, good results have been obtained. This is a sign that the code created by HLS, being that we do not have control over the blocks that it will insert and that will be inserted on a FGPA other than the pre-established family, it is better to implement it on FPGAs on which there is a lot of usable area or dedicated blocks for certain operations, so as to be able to achieve excellent performance.

Appendix A

Appendix

Appendix A reports all the codes used for the research done in this thesis.

A.1 Code of the MLP neural network (.c++)

```
#include <stdio.h>
#include <iostream>
#include "nn.h"
#include "hls_stream.h"
using namespace hls;

data_t Relu(data_t x)
{
    return (x > 0) ? x : (data_t)0;
}

data_t Linear(data_t x)
{
    return x;
}

data_t weights[LAYER][MAX_NEURAL][MAX_NEURAL]={{
    {-3.3811,-3.1615,0.8051,0.6828,-2.3433,-0.8871,-0.7088,-0.9692},
    {1.8252,1.5395,-1.2511,-1.0526,-1.03,-1.1274,0.0056,-0.2283},
    {0.538,0.474,1.0117,1.1828,-0.0423,-0.1053,1.8809,1.396},
    {-0.283,-0.2058,0.1461,1.0411,0.6863,0.5107,0.6447,1.114},
    {-0.9766,-0.8149,1.8329,-0.5494,0.7075,1.1299,0.217,-0.567},
                                {0,0,0,0,0,0}
                                                    },
```

```
25                                  {
26
      {0.6879,-1.1197,-1.2057,-0.1915,1.5561,-1.0803,1.6456,0.1638},
27
      {-0.0596,1.3794,1.3375,-1.1603,-0.938,-0.951,-1.4609,-1.6858},
28
      {0.1732,0.4295,-0.0323,0.8043,-0.1587,-0.1178,0.9939,-0.2684},
29
      {0.5053,0.6979,-0.5013,0.4808,-0.8809,0.8578,-0.8309,0.1895},
30
      {0.8711,-0.4226,0.0139,0.2811,1.0532,0.0501,-0.3013,1.8653},
31
      {-1.1001,0.7179,-0.7721,0.1211,-0.2538,0.3285,-0.0419,0.3673},
32
      {0.7311,-0.9601,-0.9548,1.0453,0.3698,-0.7131,-0.2295,-0.8121},
33
      {-0.5441,-0.4836,-0.3862,-0.7657,-0.3714,-1.0038,0.9135,-0.8717}
34
                                    },
35                                  {
36
      {1.4592,-0.3405,-1.1434,-0.6427,-1.2936,-2.0807,0.4467,0.8777}
38                                  }};
39
40 data_t biases[LAYER][MAX_NEURAL]={
41
42
      {-1.5116,1.2868,0.4261,-0.4145,1.2128,-1.1247,-1.5157,-2.122},
43
44
      {-2.2332,0.7328,0.0082,0.0122,0.1503,2.1233,1.3671,0.2673},
45
46                      {5.0282}
47                };
48
49
50
51 void ForwardTransfer(data_output outputs[NEURAL_OUTPUT_LAYER],
       data_t input[NEURAL_INPUT_LAYER])
52 {
53   data_t inputs[NEURAL_INPUT_LAYER];
54
55       Initialization_input:for (int i = 0; i < NEURAL_INPUT_LAYER; i
       ++)
56      {
57            inputs[i]=input[i];
58
59      }
60
61        data_t outputs_middle[LAYER - 1][MAX_NEURAL];
```

```
62      data_t t;
63      data_t sum;
64
65
66      t = 0;
67
68  //compute outputs of the first hidden layer
69  FirstHiddenLayer:
70      for (short j = 0; j < NEURAL_HIDDEN_LAYER; j++)
71      {
72          sum = 0;
73      NeuralInputLayer:
74          for (short i = 0; i < NEURAL_INPUT_LAYER; i++)
75          {
76              t = weights[0][i][j] * inputs[i];
77              sum = sum + t;
78          }
79          sum = sum + biases[0][j];
80          outputs_middle[0][j] = Relu(sum); //outputs of first
     hidden layer
81      }
82
83
84      NeuralNextLayer:
85          for (short j = 0; j < NEURAL_HIDDEN_LAYER; j++)
86          {
87              sum = 0;
88          NeuralThisLayer:
89              for (short i = 0; i < NEURAL_HIDDEN_LAYER; i++)
90              {
91                  t = weights[1][i][j] * outputs_middle[0][i];
92                  sum = sum + t;
93              }
94              sum += biases[1][j];
95              outputs_middle[1][j] = Relu(sum);
96          }
97
98
99  //compute outputs of output layer
100 OutputLayer:
101     for (short j = 0; j < NEURAL_OUTPUT_LAYER; j++)
102     {
103         sum = 0;
104     LastHiddenLayer:
105         for (short i = 0; i < NEURAL_HIDDEN_LAYER; i++)
106         {
107             t = weights[2][j][i] * outputs_middle[1][i];
108             sum = sum + t;
109         }
110         sum = sum + biases[LAYER - 1][j];
```

```
111             outputs[j] = Linear(sum);
112         }
113 }
```

A.2 Code of the MLP neural network (.h)

```
1 #pragma once
2 #include <ap_fixed.h>
3 #define LAYER 3 //number of layers
4 #define MAX_NEURAL 8 //MAX number of neurals in layers
5 #define NEURAL_HIDDEN_LAYER 8 //number of neurals in each hidden
      layer
6 #define NEURAL_INPUT_LAYER 6
7 #define NEURAL_OUTPUT_LAYER 1
8 #define DATA_WIDTH 16
9 #define INT_WIDTH 4
10
11 typedef ap_fixed<DATA_WIDTH,INT_WIDTH> data_t;
12
13 #define DATA_WIDTH_output 4
14 #define INT_WIDTH_out 2
15 typedef ap_fixed<DATA_WIDTH_output,INT_WIDTH_out> data_output;
16
17 void ForwardTransfer(data_output outputs[NEURAL_OUTPUT_LAYER],
      data_t input[NEURAL_INPUT_LAYER]);
18
19 data_t Relu(data_t);
20 data_t Linear(data_t);
```

A.3 Code of DSP (.c++)

```
1 #include "mac.h" // Provides default WINDOW_LEN if not user defined
2
3 // Function definitions:
4 void MAC(out_t outdata[1], in_t indata[3])
5 {
6   static prod_t sum,sum1,sum2;
7
8   sum += indata[0]*indata[1]; // First DSP
9     //sum1 += indata[2]*indata[1]; // Second DSP
10   //sum2 += indata[0]*indata[2]; // Third DSP
11
12   outdata[0]=sum;
13   //outdata[0]=sum+sum1; //for 2 DSP
14   //outdata[0]=sum+sum1+sum2; //for 3 DSP
15
16
17 }
```

A.4 Code of DSP (.h)

```
#include <stdint.h>
#include <math.h>
#include <ap_int.h>
// Define widths of fixed point fields
#define W_IN     16
#define IW_IN    3
#define W_OUT    1
#define IW_OUT   1

#define prod   32
#define prod_v   3
// Define fixed  point types for input, output and coefficients
typedef ap_fixed<W_IN,IW_IN> in_t;
typedef ap_fixed<W_OUT,IW_OUT> out_t;
typedef ap_fixed<prod,prod_v> prod_t;

void MAC(out_t outdata[1], in_t indata[3]);
```

Bibliography

[1] M. Roukhami, M. T. Lazarescu, F. Gregoretti, Y. Lahbib, and A. Mami. "Very Low Power Neural Network FPGA Accelerators for Tag-Less Remote Person Identification Using Capacitive Sensors". In: *IEEE Access* 7 (2019), pp. 102217–102231.

[2] L. Mainetti, L. Patrono, and I. Sergi. "A survey on indoor positioning systems". In: *2014 22nd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 2014, pp. 111–120.

[3] Michal Cieslak, Christoph Kling, and Andrea Wolff. "Ultrasound exposure in a workplace and a potential way to improve its measurement methodology". In: June 2020, pp. 172–176. DOI: 10.1109/MetroInd4.0IoT48571.2020.9138223.

[4] D. Yang, W. Sheng, and R. Zeng. "Indoor human localization using PIR sensors and accessibility map". In: *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*. 2015, pp. 577–581.

[5] C. Yang and H. Shao. "WiFi-based indoor positioning". In: *IEEE Communications Magazine* 53.3 (2015), pp. 150–157.

[6] Riya Lodha, Suruchi Gupta, Harshil Jain, and Harish Narula. "Bluetooth Smart Based Attendance Management System". In: *Procedia Computer Science* 45 (Dec. 2015), pp. 524–527. DOI: 10.1016/j.procs.2015.03.094.

[7] Y. Zheng, G. Shen, L. Li, C. Zhao, M. Li, and F. Zhao. "Travi-Navi: Self-Deployable Indoor Navigation System". In: *IEEE/ACM Transactions on Networking* 25.5 (2017), pp. 2655–2669.

[8] Guoyu Lu, Yan Yan, Li Ren, Philip Saponaro, Nicu Sebe, and Chandra Kambhamettu. "Where Am I in the Dark: Exploring Active Transfer Learning on the Use of Indoor Localization based on Thermal Imaging". In: *Neurocomputing* 173 (Aug. 2015). DOI: 10.1016/j.neucom.2015.07.106.

[9] Gabe Cohn, Dan Morris, Shwetak Patel, and Desney Tan. "Humantenna: Using the Body as an Antenna for Real-Time Whole-Body Interaction". In: (May 2012). DOI: 10.1145/2207676.2208330.

[10] Maxim Djakow, Andreas Braun, and Alexander Marinc. "MoviBed - Sleep Analysis Using Capacitive Sensors". In: June 2014, pp. 171–181. DOI: 10.1007/978-3-319-07509-9_17.

[11] Guoyu Lu, Yan Yan, Li Ren, Philip Saponaro, Nicu Sebe, and Chandra Kambhamettu. "Where Am I in the Dark: Exploring Active Transfer Learning on the Use of Indoor Localization based on Thermal Imaging". In: *Neurocomputing* 173 (Aug. 2015). DOI: 10.1016/j.neucom.2015.07.106.

[12] Raphael Wimmer, Paul Holleis, Matthias Kranz, and Albrecht Schmidt. "Thracker - Using Capacitive Sensing for Gesture Recognition". In: Jan. 2006, p. 64. DOI: 10.1109/ICDCSW.2006.109.

[13] Heather Knight, Jae-Kyu Lee, and Hongshen Ma. "Chair Alarm for Patient Pall Prevention based on Gesture Recognition and Interactivity". In: *Conference proceedings : ... Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Conference* 2008 (Feb. 2008), pp. 3698–701. DOI: 10.1109/IEMBS.2008.4650012.

[14] Marian Haescher, Denys Matthies, Gerald Bieber, and Bodo Urban. "CapWalk: A Capacitive Recognition of Walking-Based Activities as a Wearable Assistive Technology". In: July 2015. DOI: 10.1145/2769493.2769500.

[15] J. Iqbal, M. T. Lazarescu, A. Arif, and L. Lavagno. "High sensitivity, low noise front-end for long range capacitive sensors for tagless indoor human localization". In: *2017 IEEE 3rd International Forum on Research and Technologies for Society and Industry (RTSI)*. 2017, pp. 1–6.

[16] Jindong Wang, Yiqiang Chen, Shuji Hao, Xiaohui Peng, and Hu Lisha. "Deep Learning for Sensor-based Activity Recognition: A Survey". In: *Pattern Recognition Letters* (July 2017). DOI: 10.1016/j.patrec.2018.02.010.

[17] Hyungui Lim, Jeongsoo Park, Kyogu Lee, and Yoonchang Han. "RARE SOUND EVENT DETECTION USING 1D CONVOLUTIONAL RECURRENT NEURAL NETWORKS". In: May 2018.

[18] X. Wang, Z. Yu, and S. Mao. "DeepML: Deep LSTM for Indoor Localization with Smartphone Magnetic and Light Sensors". In: *2018 IEEE International Conference on Communications (ICC)*. 2018, pp. 1–6.

[19] A. L. S. Braga, C. H. Llanos, D. Göhringer, J. Obie, J. Becker, and M. Hübner. "Performance, accuracy, power consumption and resource utilization analysis for hardware / software realized Artificial Neural Networks". In: *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*. 2010, pp. 1629–1636.

[20] A. G. Blaiech, K. Ben Khalifa, M. Boubaker, and M. H. Bedoui. "Multi-width fixed-point coding based on reprogrammable hardware implementation of a multi-layer perceptron neural network for alertness classification". In: *2010 10th International Conference on Intelligent Systems Design and Applications*. 2010, pp. 610–614.

[21] M. Bahoura and C. Park. "FPGA-implementation of high-speed MLP neural network". In: *2011 18th IEEE International Conference on Electronics, Circuits, and Systems*. 2011, pp. 426–429.

[22] Xiaojun Zhai, Amine Ait Si Ali, Abbes Amira, and Faycal Bensaali. "MLP Neural Network Based Gas Classification System on Zynq SoC". In: *IEEE Access* PP (Oct. 2016), pp. 1–1. DOI: 10.1109/ACCESS.2016.2619181.

[23] O. Arcas-Abella et al. "An empirical evaluation of High-Level Synthesis languages and tools for database acceleration". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8.

[24] J. Iqbal, M. T. Lazarescu, O. B. Tariq, and L. Lavagno. "Long range, high sensitivity, low noise capacitive sensor for tagless indoor human localization". In: *2017 7th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI)*. 2017, pp. 189–194.

[25] C. Gabriel. "Compilation of the Dielectric Properties of Body Tissues at RF and Microwave Frequencies." In: 1996.

[26] O. B. Tariq, M. T. Lazarescu, and L. Lavagno. "Neural Networks for Indoor Human Activity Reconstructions". In: *IEEE Sensors Journal* (2020), pp. 1–1.

[27] Alireza Ramezani, Mihai Lazarescu, Osama Bin Tariq, and Luciano Lavagno. "A Tagless Indoor Localization System Based on Capacitive Sensing Technology". In: *Sensors* 16 (Sept. 2016), p. 1448. DOI: 10.3390/s16091448.

[28] Jose Rivera-Rubio, Ioannis Alexiou, and Anil Bharath. "Appearance-based indoor localization: A comparison of patch descriptor performance". In: *Pattern Recognition Letters* 10 (Mar. 2015). DOI: 10.1016/j.patrec.2015.03.003.

[29] Navid Fallah, Ilias Apostolopoulos, Kostas Bekris, and eelke folmer. "Indoor Human Navigation Systems: A Survey". In: *Interacting with Computers* 25 (Jan. 2013), pp. 21–33. DOI: 10.1093/iwc/iws010.

[30] Yuan Zhuang, Jun Yang, You Li, Longning Qi, and Naser El-Sheimy. "Smartphone-Based Indoor Localization with Bluetooth Low Energy Beacons". In: *Sensors* 16 (Apr. 2016), p. 596. DOI: 10.3390/s16050596.

[31] A. Athalye, V. Savic, M. Bolic, and P. M. Djuric. "Novel Semi-Passive RFID System for Indoor Localization". In: *IEEE Sensors Journal* 13.2 (2013), pp. 528–537.

[32] Zheng Yang, Chenshu Wu, Zimu Zhou, Xinglin Zhang, Xu Wang, and Yunhao Liu. "Mobility Increases Localizability: A Survey on Wireless Indoor Localization using Inertial Sensors". In: *ACM Computing Surveys* 47 (Apr. 2015), pp. 1–34. DOI: 10.1145/2676430.

[33] C. Ding, C. Xu, and D. Tao. "Multi-Task Pose-Invariant Face Recognition". In: *IEEE Transactions on Image Processing* 24.3 (2015), pp. 980–993.

[34] Erjin Zhou, Zhimin Cao, and qi Yin. "Naive-Deep Face Recognition: Touching the Limit of LFW Benchmark or Not?" In: (Jan. 2015).

[35] Daigo Muramatsu, Akira Shiraishi, Yasushi Makihara, Md Uddin, and Yasushi Yagi. "Gait-Based Person Recognition Using Arbitrary View Transformation Model". In: *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society* 24 (Nov. 2014). DOI: 10.1109/TIP.2014.2371335.

[36] Zahid Farid, Rosdiadee Nordin, and Mahamod Ismail. "Recent Advances in Wireless Indoor Localization Techniques and System". In: *Journal of Computer Networks and Communications* 2013 (Sept. 2013). DOI: 10.1155/2013/185138.

[37] G. Mokhtari, Q. Zhang, C. Hargrave, and J. C. Ralston. "Non-Wearable UWB Sensor for Human Identification in Smart Home". In: *IEEE Sensors Journal* 17.11 (2017), pp. 3332–3340.

[38] Irem Velibeyoglu, Shijia Pan, ningning wang, Yuqiu Qian, Hae Young Noh, and pei zhang. "Indoor Person Identification through Footstep Induced Structural Vibration". In: Feb. 2015. DOI: 10.1145/2699343.2699364.

[39] R. Wimmer, M. Kranz, S. Boring, and A. Schmidt. "A Capacitive Sensing Toolkit for Pervasive Activity Detection and Recognition". In: *Fifth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom'07)*. 2007, pp. 171–180.

[40] Javed Iqbal, Arslan Arif, Osama Bin Tariq, Mihai Teodor Lazarescu, and Luciano Lavagno. "A contactless sensor for human body identification using RF absorption signatures". In: *2017 IEEE Sensors Applications Symposium (SAS)* (2017), pp. 1–6.

[41] J. Iqbal, M. T. Lazarescu, O. B. Tariq, A. Arif, and L. Lavagno. "Capacitive Sensor for Tagless Remote Human Identification Using Body Frequency Absorption Signatures". In: *IEEE Transactions on Instrumentation and Measurement* 67.4 (2018), pp. 789–797.

[42] O. Bin Tariq, M. T. Lazarescu, J. Iqbal, and L. Lavagno. "Performance of Machine Learning Classifiers for Indoor Person Localization With Capacitive Sensors". In: *IEEE Access* 5 (2017), pp. 12913–12926.

[43] http://newsspazio.blogspot.com/2019/07/intelligenza-artificiale-06-imitare-il.html.

[44] Sepp Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (Apr. 1998), pp. 107–116. DOI: 10.1142/S0218488598000094.

[45] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[46] M. Bettoni, G. Urgese, Y. Kobayashi, E. Macii, and A. Acquaviva. "A Convolutional Neural Network Fully Implemented on FPGA for Embedded Platforms". In: *2017 New Generation of CAS (NGCAS)*. 2017, pp. 49–52.

[47] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang. "Accelerating low bit-width convolutional neural networks with embedded FPGA". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017, pp. 1–4.

[48] K. Georgopoulos, G. Chrysos, P. Malakonakis, A. Nikitakis, N. Tampouratzis, A. Dollas, D. Pnevmatikatos, and Y. Papaefstathiou. "An evaluation of vivado HLS for efficient system design". In: *2016 International Symposium ELMAR*. 2016, pp. 195–199.

[49] J. Andrade, N. George, K. Karras, D. Novo, V. Silva, P. Ienne, and G. Falcao. "Fast Design Space Exploration Using Vivado HLS: Non-binary LDPC Decoders". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2015, pp. 97–97.

[50] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf.

[51] https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiTj7mRrNfrAhUhpHEKHbqrA60QFjAAegQIAxAB&url=https%3A%2F%2Fwww.latticesemi.com%2Fview_document%3Fdocument_id%3D52758&usg=AOvVaw2mJCpnSH_NoTvWdZ3GXdon.

[52] www.microsemi.com/product-directory/libero-soc/5507-libero-soc-v11-9-archive#overview.