POLITECNICO DI TORINO

MASTER'S DEGREE IN COMPUTER ENGINEERING

Master Degree Thesis

Logic-In Memory implementation on FPGA

Supervisors
Prof. Mariagrazia Graziano
Prof. Massimo Ruo Roch
Prof. Giovanna Turvani

Candidate
Coralie Allioux
matricola 287268

July 25, 2022

Acknowledgements

I would like to thank all people that I have ever encountered in my path at University, who open to me wonderful opportunities that make the person I stand to be today.

These past years, I have learned that working as a team, side by side with other people, could be and should be an occasion to grow and go ahead, on personal and technical aspects. That is why, I dedicated this little space to all people that ever support me.

I warmly thank Andrea Coluccio, I could not express how glad I am to have been followed during this thesis by him. You told be one day that you wanted me to feel comfortable and supported during this thesis: I can tell, I truly do and I feel so grateful of it.
I sincerely thank the professors that make this thesis possible, in particular Mariagrazia Graziano. Her courses have always stimulated me and gave me the confirmation of my professional path, as the way I would like to improve myself as a person.

I would like to thank Gabriel, my dear friend, with which I shared this awesome experience in all the aspects.
Thank you to all my friends that I have met here in Italy, that ever believed in me and show me another part of myself.

I thank my home University, Grenoble-INP ENSIMAG, and Politecnico di Torino for the education that I have received. I cannot imagine myself without this 2-years experience in Italy, where I feel now home.

Glossary

BCNN Binary Convolution Neural Network. 20, 21, 87

CNN Convolution Neural Network. 20

DWT Data Watchpoint Trigger. 12, 57, 58

IFMAP Input Feature MAP. 12, 21, 31, 33–35, 38–40, 43, 49, 50, 54

Interface Decoder Part of the XNOR-Net implementation: connects the two LiM arrays. 11, 21, 32, 36, 39

K Weight input. 12, 21, 31, 33–35, 37–40, 43, 49, 50, 54, 55

LiM Logic In Memory. 11, 18–20, 87

LiM arrays TO BE WRITTEN. 21

LiM Ones Counter Part of the XNOR-Net implementation: LiM array that holds an half adder in each memory cell. 11, 21, 22, 32, 33, 36–38

LiM XNOR Part of the XNOR-Net implementation: LiM array that holds in every memory cell a xnor gate. 11, 21, 32, 33, 35–38, 54, 55

Logic-In Memory Architecture that merges computation and memory. 11, 18, 19

NN Neural Network. 20

OFMAP Output Feature MAP. 12, 31, 39–41, 43, 49, 50, 54, 56

Pop Counting Logic Part of the XNOR-Net implementation: performs the difference of 1s and 0s within a word. 21, 32

VirtLAB Board on which the LiM is implemented. 11, 12, 23–26, 43–45, 49, 52, 53, 59, 61–63, 87

XNOR-Net A binary convolution neural network. 11, 12, 20, 21, 31–39, 41–43, 49–52, 54–56, 87

Contents

Glossary 4

List of Figures 11

List of Tables 15

1 State of the art 17

1.1 Von Neumann architecture . . . 17
1.2 Memory bottleneck problem . . . 18
1.3 Logic in Memory architecture . . . 18
1.3.1 Advantages and drawbacks over Von Neumann . . . 19
1.3.2 Possible applications . . . 20
1.4 Neural network background . . . 20
1.4.1 Neural network . . . 20
1.4.2 Convolutional neural network . . . 20
1.4.3 Binary convolutional neural network . . . 20
1.5 XNOR-Net application . . . 21
1.5.1 Behavior . . . 21
1.5.2 XNOR-Net LiM Architecture . . . 21

2 VirtLAB 23

2.1 Overview . . . 23
2.2 Setup tutorial . . . 24
2.3 Master side . . . 24
2.3.1 QSPI Flash . . . 25
2.3.2 Digital Storage Oscilloscope . . . 27
2.4 User side . . . 28

3 Hardware implementations: XNOR-net 31
3.1 Gitlab versioning . . . 31
3.2 VHDL description . . . 31
3.2.1 Overview . . . 31
3.2.2 Control Unit . . . 33
3.2.3 LiM XNOR . . . 33
3.2.4 Interface Decoder . . . 36
3.2.5 LiM Ones Counter . . . 37
3.2.6 Pop Logic . . . 38
3.3 MCU based: C implementation . . . 39
3.3.1 XNOR operation . . . 40
3.3.2 Pop Count operation . . . 40
3.3.3 Pop Logic operation . . . 41
3.3.4 Algorithm cost . . . 42
3.4 FPGA based . . . 42
3.4.1 Overview . . . 42
3.4.2 Extended I/O using a Bread Board . . . 44
3.4.3 Debounce circuit . . . 45
3.4.4 Debounce Software . . . 47
3.5 MCU-FPGA based . . . 49
3.5.1 Overview . . . 49
3.5.2 Upgraded XNOR-net: changes overview . . . 49
3.5.3 Upgraded XNOR-net: Control Unit . . . 51
3.5.4 Communication Protocol characteristics . . . 52
3.5.5 Writing operation . . . 54
3.5.6 Launch computation operation . . . 55
3.5.7 Read operation . . . 56

4 Measurements Methods 57
4.1 Timing measures . . . 57
4.1.1 MCU . . . 57
4.1.2 VirtLAB Oscilloscope . . . 61
4.1.3 ModelSim . . . 61
4.2 Power measurements . . . 62
4.2.1 VirtLAB DSO: Observable channels . . . 62
4.2.2 VirtLAB DSO: Digital trigger IO9 . . . 62
4.2.3 VirtLAB: Oscilloscope GUI . . . 63
4.2.4 VirtLAB: Oscilloscope CLI . . . 64
4.3 Area occupation . . . 67

5 Results 69
5.1 Performance result approach . . . 69
5.1.1 Current measurements on VirtLAB . . . 69
5.1.2 Timing measurements . . . 70
5.1.3 Area occupation measurements . . . 70
5.2 MCU-FPGA based . . . 70
5.2.1 Area occupation . . . 71
5.2.2 Timing . . . 71
5.2.3 Power . . . 73
5.2.4 Energy . . . 78
5.3 MCU-based . . . 79
5.3.1 Timing . . . 79
5.3.2 Power consumption . . . 80
5.3.3 Energy . . . 81
5.4 Comparison . . . 82
5.4.1 Execution time . . . 82
5.4.2 Energy . . . 83

6 Conclusions and Future works 87

Bibliography 89

A Additional measures 91
A.1 MCU-FPGA based: FPGA a 32KHz . . . 91
A.1.1 LiM is computing . . . 91
A.1.2 MCU is writing and LiM is computing . . . 91
A.1.3 Energy with FPGA at 32KHz . . . 91

B Tutorial VirtLab 95
B.1 Introduction . . . 2
B.2 Connecting the VirtLab to a PC . . . 2
B.3 Needed software . . . 4
B.3.1 General overview: Interaction between software and VirtLab . . . 4
B.3.2 Stlinkbridge . . . 5
B.3.3 JDK 17 . . . 5
B.3.4 VirtLab GUI . . . 5
B.3.5 STM32CubeProg . . . 6
B.3.6 STM32CubeIDE . . . 6
B.3.7 Quartus . . . 7

B.4 Setting up the environment 9
B.4.1 For Windows 10/11 9
B.4.2 For GNU/Linux 9
B.5 First use of VirtLab 16
B.6 Program on VirtLab 20
B.6.1 Program MCU User 20
B.6.2 Program FPGA User 20
B.6.3 Program FPGA Master 23

List of Figures

1.1 Von Neumann's classical architecture composed of CPU and memory . . . 17
1.2 Memory bottleneck problem . . . 18
1.3 Logic-In Memory (LiM) novel architecture that merges computation and memory . . . 19

2.1 Overview of the VirtLAB: main components and external connections . . . 23
2.2 Overview of the VirtLAB: detailed schematics with all components . . . 24
2.3 QSPI Master: How to use with the dedicated Java App . . . 26
2.4 QSPI User: How to use with the dedicated Java App . . . 27

3.1 XNOR-Net V1: overview . . . 32
3.2 XNOR-Net V1: Control Unit . . . 34
3.3 XNOR-Net V1: LiM XNOR . . . 35
3.4 XNOR-Net V1: Interface Decoder . . . 36
3.5 XNOR-Net V1: LiM Ones Counter . . . 37
3.6 XNOR-Net V1: Pop Logic . . . 38
3.7 XNOR-Net C program: Overview . . . 39
3.8 XNOR-Net C program: Xnor operation . . . 41
3.9 XNOR-Net C program: Pop Count operation . . . 42
3.10 XNOR-Net C program: Pop Logic operation . . . 43
3.11 FPGA-based: Interconnection between the VirtLAB and the external breadboard . . . 44
3.12 Breadboard connected to VirtLAB . . . 44
3.13 Debounce circuits on the breadboard to compensate the misbehavior of the push buttons . . . 46
3.14 Debounce Circuit Anomaly . . . 46
3.15 Debounce Software waveform: how does it react . . . 47

3.16 Debounce Software Top entity 47
3.17 Debounce Software Control Unit 48
3.18 Updated XNOR-Net: Overview 50
3.19 Updated XNOR-Net: Control Unit 51
3.20 Protocol: Signals between MCU and FPGA 53
3.21 Protocol diagram: Write IFMAP 54
3.22 Protocol diagram: Write K 55
3.23 Protocol diagram: Compute 55
3.24 Protocol diagram: Read OFMAP 56

4.1 Process of use of the DWT register 58
4.2 GPIO interrupts example: how to measure computation time in MCU-FPGA based implementation 59
4.3 Example of use of timers to measure computation time on MCU-based implementation 60
4.4 Measurement of execution time on ModelSim waves 61
4.5 Measurement of execution time on ModelSim waves 62
4.6 DSO menu in the Java Application of the VirtLAB 63
4.7 Java App: DSO home 64
4.8 DSO CLI: connection with Putty on physical COM port . . . 65
4.9 DSO CLI: connection successful, command *og* to verify the current status 65

5.1 Used area by LiM on FPGA 71
5.2 Power consumption of the FPGA Core while MCU is writing and LiM is computing 75
5.3 Power consumption FPGA Core and the MCU: while MCU is writing and LiM is computing 76
5.4 Power consumption of the FPGA while computing the LiM XNOR-Net 78
5.5 Energy of the LiM while computing 79
5.6 Power consumption of the MCU while computing the XNOR-Net 83
5.7 Energy of the MCU while computing 84
5.8 Timing performance comparison LiM and MCU: with FPGA at 10MHz 84
5.9 Energy performance: how much the LiM saves on MCU 85

B.1 VirtLab board conected to a PC 3
B.2 Interaction between software and VirtLab 4

B.3 Selection of RBF output file. 8
B.4 STEP 1: Search for the environment variable control panel . . 10
B.5 STEP 2: Click on *Environment variables...* 11
B.6 STEP 3: Select *Path* in *System Variables* and click *Modify* . . 12
B.7 STEP 4: Create a new entry by clicking on *New* 13
B.8 STEP 5: Click on *Browse...* and select the directory where *STLinkbridge* program (executable and library files) is 14
B.9 STEP 6: Done! Dismiss all windows by clicking on *OK* (three times) 15
B.10 STEP 4: Go to MCU tab 17
B.11 STEP 5: Select *Master* and click *Apply* 17
B.12 VirtLab: MCU Master selected, ORANGE LED 18
B.13 Connect to the ST's programmer present on the board by clicking *Connect* 18
B.14 STEP 7: Click on *Open file* and choose the firmware called *virtlab-master.elf*. Current code on the MCU is depicted in the middle. 19
B.15 STEP 8: Click on *Download* to upload on the MCU the file previously selected. 19
B.16 VirtLab GUI MCU: choose MCU user and apply 20
B.17 VirtLab: MCU User selected, GREEN LED 21
B.18 STEP 4: Connect to the communication port by clicking *Connect*. Then, Got to FPGA tab. 22
B.19 STEP 5: Select *File RBF* and ***User FPGA***. Then choose an RBF file by using *Browse* button. Finally click on the centered *arrow*, and wait for the status bar on the bottom right to be full. 22
B.20 VirtLabUI: Upload RBF file on FPGA Master 23

List of Tables

3.1 Constraints on operations given by external signals: states in which they are processed 52

5.1 Timing performance of the LiM while computing 72
5.2 MCU-FPGA: **power consumption** for a **32-bit word** during the *writing LiM + computation* of the LiM XNOR-Net 73
5.3 MCU-FPGA: **power consumption** for a **64-bit word** during the *writing LiM + computation* of the LiM XNOR-Net 74
5.4 MCU-FPGA: **power consumption** for a **128-bit word** during the *writing LiM + computation* of the LiM XNOR-Net . . 74
5.5 MCU-FPGA: **power consumption** for a **32-bit word** during the *computation only* of the LiM XNOR-Net 75
5.6 MCU-FPGA: **power consumption** for a **64-bit word** during the *computation only* of the LiM XNOR-Net 76
5.7 MCU-FPGA: **power consumption** for a **128-bit word** during the *computation only* of the LiM XNOR-Net 77
5.8 MCU-FPGA: **energy** during the *computation only* of the LiM XNOR-Net 79
5.9 **32-bit word**: Timing performance of the MCU while computing 80
5.10 **64-bit word**: Timing performance of the MCU while computing 80
5.11 **128-bit word**: Timing performance of the MCU while computing 81
5.12 MCU: current and power measures for a **32-bit word** during the *write + computation* of the XNOR-Net 81
5.13 MCU: current measures for a **64-bit word** during the *computation* of the XNOR-Net 82
5.14 MCU: current measures for a **128-bit word** during the *computation* of the XNOR-Net 82
5.15 MCU: **energy** during the *computation* 83

A.1 MCU-FPGA: **power consumption** for a **32-bit word** during the *computation only* of the LiM XNOR-Net 91
A.2 MCU-FPGA: **power consumption** for a **64-bit word** during the *computation only* of the LiM XNOR-Net 92
A.3 MCU-FPGA: **power consumption** for a **128-bit word** during the *computation only* of the LiM XNOR-Net 92
A.4 MCU-FPGA: current measures for a **32-bit word** during the *writing LiM + computation* of the LiM XNOR-Net 92
A.5 MCU-FPGA: current measures for a **64-bit word** during the *writing LiM + computation* of the LiM XNOR-Net 93
A.6 MCU-FPGA: current measures for a **128-bit word** during the *writing LiM + computation* of the LiM XNOR-Net 93
A.7 MCU-FPGA: **energy** during the *computation only* of the LiM XNOR-Net 93

Chapter 1

State of the art

1.1 Von Neumann architecture

The Von Neumann architecture is a computer architecture based on two main components as shown on Figure 1.1: a central processing unit and a memory unit. This paradigm is usually used in modern computing architecture.

Those two elements are constantly communicating between them to exchange data. Indeed, the processing unit, here called CPU, is computing on the data provided by the memory. In this configuration, the CPU and the memory are strongly separated.

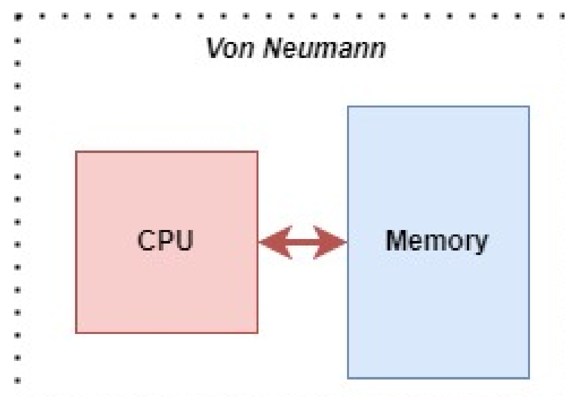


Figure 1.1: Von Neumann's classical architecture composed of CPU and memory

1.2 Memory bottleneck problem

The CPU is a very fast computational unit. On the contrary, the memory is slower than CPU, due to the time required to read and write data.

As shown on Figure 1.2, the memory does not follow the trend of CPU regarding performance. As a consequence, the CPU is constantly waiting for the data from the memory.

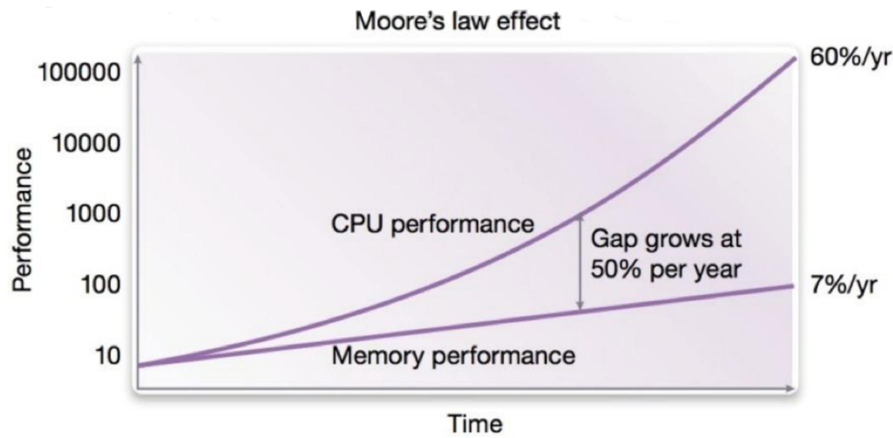


Figure 1.2: Memory bottleneck problem

In other words, a lot of power, energy and time is spent moving the data back and forth between those two elements, making the communication between the two really heavy.

A possible way to solve that problem could be to merge those two components in some way: the Logic-In Memory solution is enlightened in this thesis.

1.3 Logic in Memory architecture

Logic-In Memory (LiM) is an architecture aiming at solving the memory bottleneck problem introduced by the Von Neumann architecture, by merging the processing unit and the memory, as depicted in Figure 1.3. This merge should reduce as much as possible the exchanges between CPU and memory and, by extension, lower the impact of the memory bottleneck on performance.

As an example, this approach could equip each memory cell of the usual

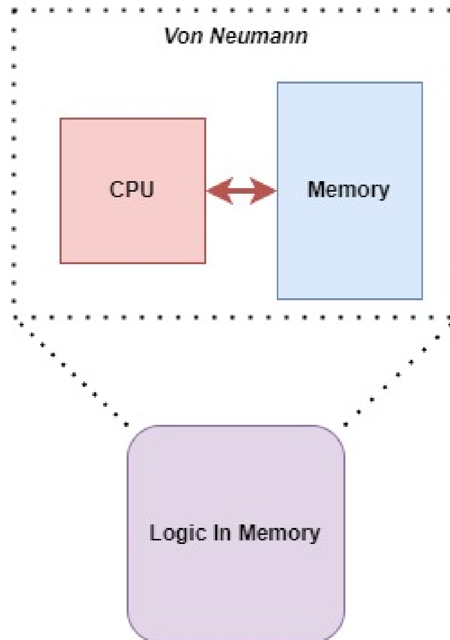


Figure 1.3: Logic-In Memory (LiM) novel architecture that merges computation and memory

store unit and an additional logical unit: like simple gates. Therefore, the computation is done directly where the data is and does not require any communication process between the store unit and the logical unit. The consequence is clear: the exchanges between CPU and memory are drastically reduced.

1.3.1 Advantages and drawbacks over Von Neumann

LiM is a promising architectural solution to Von Neumann bottleneck because:

- less energy and computational time are required;
- it possesses a high degree of parallelization: the computation is done on each word at the same time, on the contrary on CPU the computation is sequential.

On the other hand, it still has some drawbacks in term of power and area. It is explained by the dedicated logic unit added in each memory cell, making the overall memory more complex.

1.3.2 Possible applications

As said in section 1.3.1, the LiM architecture possesses a high degree of parallelization. Therefore, the main applications are focused on those based on a parallel algorithm. For instance: machine learning, convolution neural networks, cryptography, and much more.

In this thesis, we emphasize on a specific application: the binary convolutional neural network called XNOR-Net.

1.4 Neural network background

A minimal background would be useful to better understand the behavior of the XNOR-Net, which is implemented in this thesis.

1.4.1 Neural network

A neural network (NN) is a network of nodes that is exploited to approach and perform very complex tasks (for example image recognition). Those tasks are memory intensive, which is why it could be an ideal application to test the LiM.

1.4.2 Convolutional neural network

A convolutional neural network (CNN) behaves as a filter, performing the convolution between weights and the input, that slides among the input features.

This can achieve really high accuracy but at the cost of an improved computational complexity.

1.4.3 Binary convolutional neural network

A binary convolutional neural network (BCNN) aims at reducing the computational complexity of a standard CNN by approximating the input and the weights: only two values are used and then represented by 0 and 1.

This approximation reduces the complexity, but unfortunately, it also reduces the accuracy of the NN.

1.5 XNOR-Net application

The XNOR-Net is a BCNN. The chosen architecture to be implemented on FPGA is the one described in [1].

1.5.1 Behavior

The XNOR-Net computes the binary convolution between IFMAP and K, by performing several steps:

1. Having already IFMAP and K as binary values, applies the bitwise xnor operation between those two inputs;
2. Counts the number of $1s$, also known as pop-counting;
3. Computes the difference between $1s$ and $0s$.

1.5.2 XNOR-Net LiM Architecture

As described, the entire architecture is composed of four main elements [1], from which two of them are LiM arrays:

- LiM XNOR: performs the XNOR bitwise operation;
- Interface Decoder: sends the results from LiM XNOR to LiM Ones Counter;
- LiM Ones Counter: performs the pop-counting operation;
- Pop Counting Logic: computes the difference between $1s$ and $0s$.

The LiM XNOR contains in each memory cell a XNOR gate, in order to perform the binary product between the content of the cell (store unit) and an external binary input. The storing unit, which is here a flip-flop, keeps the binary input value called IFMAP. The external binary input is actually the associated weight.
In this way, each memory row is actually a convolution window.

On the other hand, the LiM Ones Counter has a flip-flop and a half-adder inside each memory cell. The half-adder that are part of the same word are connected together, where the first is fed by the Interface Decoder.
This way, when all bits have been received from the Interface Decoder, the pop-count is read directly from the value held by the storing units.

Finally, the pop-counting computation logic, to perform the difference between 0s and 1s within a single word, is made of:

- a multiplexer: selects the current word
- a shifter: multiplies by two the value the LiM Ones Counter
- a subtractor: subtracts by the length of the word (number of bits)

Chapter 2

VirtLAB

2.1 Overview

The VirtLAB [3] [4] [12] (Figure 2.1) is a board created for teaching purposes. It is composed of 2 distinct parts: the Master side and the User one, which are both equipped with a MCU (STM32L4) and a FPGA (Cyclone10).

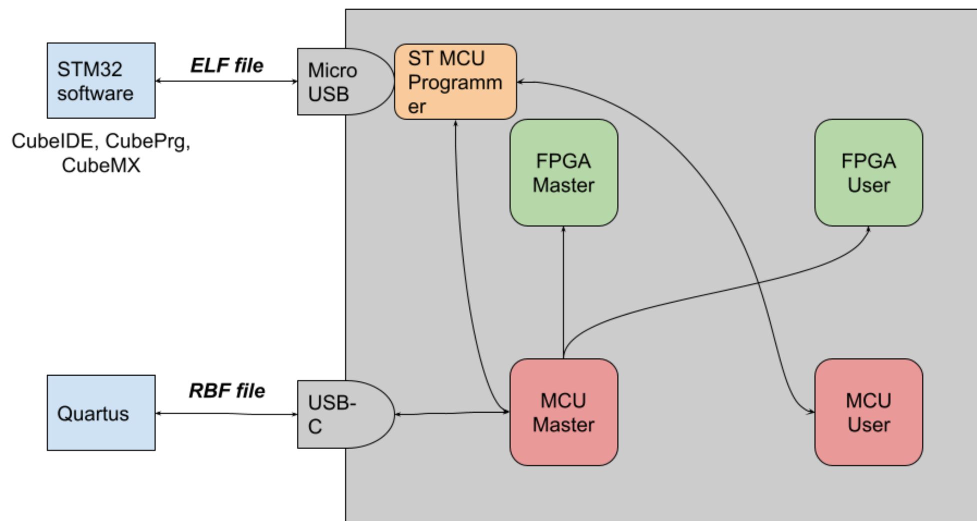


Figure 2.1: Overview of the VirtLAB: main components and external connections

The VirtLAB is directly connected to the host with a USB-C and a micro-USB. The communication is then possible through the following software: STMCube software, Quartus and a dedicated Java App.

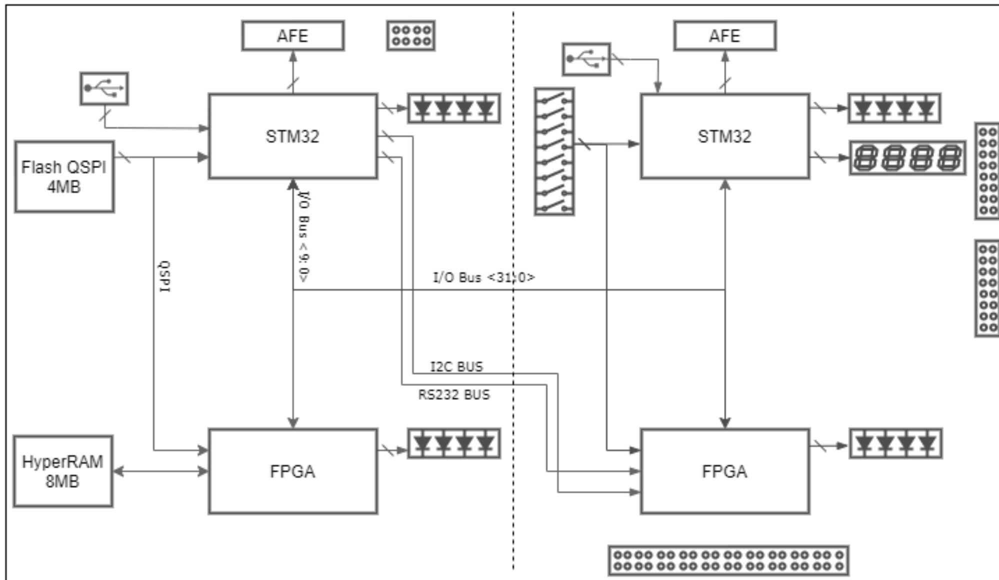


Figure 2.2: Overview of the VirtLAB: detailed schematics with all components

This thesis concentrates on the User side: all the implementations are done exploiting mainly the FPGA-User and the MCU-User.

The following sections describe briefly in more detail the functionalities of the VirtLAB and what is used for the development of this thesis.

2.2 Setup tutorial

A setup tutorial of the VirtLAB has been written for teaching of research purposes: it is exploited by courses at Politecnico di Torino and the research team which works on the VirtLAB.

The full tutorial can be found in Appendix B.

2.3 Master side

The main components on the Master side (left side of Figure 2.2) are:

- **MCU-Master**
- **FPGA-Master**

- **LEDs**: 4 for the MCU and 4 for the FPGA, mainly used for information purposes
- **Oscilloscope**: allows to perform some measures on the board (power and IO pin value)
- **QSPI Flash**: hold firmware of FPGAs

The Master side holds a predefined firmware for the MCU and the FPGA, done by the professor (available on DropBox [11]). It is not to be modified because it handles some critical parts of the board, like:

- Communicate with the ST Programmer to control which MCU is to be programmed;
- Program the FPGAs;
- Implements the DSO;
- Manages the versioning of firmware with the QSPI Flash.

As a consequence, no firmware for the Master side was developed to preserve its crucial functionalities.

2.3.1 QSPI Flash

The QSPI Flash on the VirtLAB is used to hold the firmware of the FPGA User and Master. Indeed, when the VirtLAB is turned off and then on, the firmware downloaded on the FPGAs are lost.

Writing these firmware on the QSPI Flash allows the MCU-Master to program the FPGAs with those ones at the start of the VirtLAB.

Note that to write on the QSPI Flash, the VirtLAB must be connected and turned on. For more information, see the setup tutorial in Appendix B.

FPGA-Master firmware on QSPI flash

Figure 2.3 shows the configuration to write a firmware for the FPGA-Master on the QSPI.

The steps are:

- Connect the VirtLAB to the computer with the USB-c and micro-USB and turn it on.

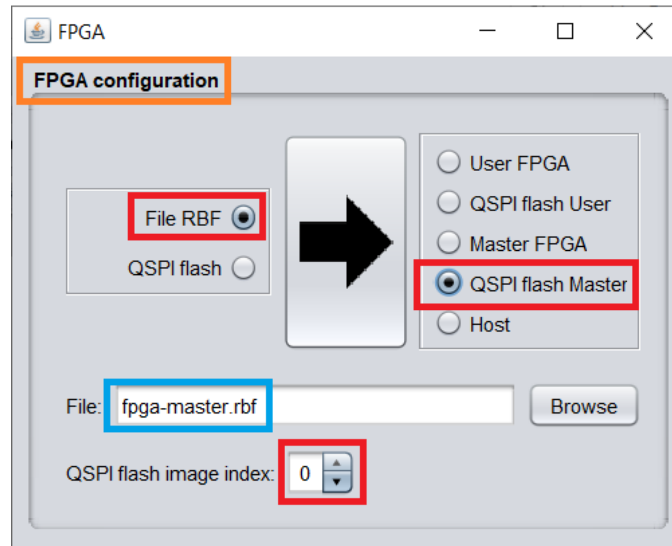


Figure 2.3: QSPI Master: How to use with the dedicated Java App

- Open the Java App of the VirtLAB.
- Select the FPGA configuration.
- Select **File RBF** and **QSPI flash Master**.
- Select the firmware to be about by clicking on **Browse**. Here the target file is *fpga-master.rbf*.
- Select the index **0** for the QSPI flash image.
- Finally, click in the big arrow on the center of the graphical interface.
- Wait for the process to be finished.

FPGA-User firmware on QSPI flash

Figure 2.4 describes the needed configuration to write a firmware for the FPGA-Useron the QSPI.

The steps are similar to the FPGA-Master procedure and are the followings:

- Connect the VirtLAB to the computer with the USM-c and micro-USB and turn it on.
- Open the Java App of the VirtLAB.

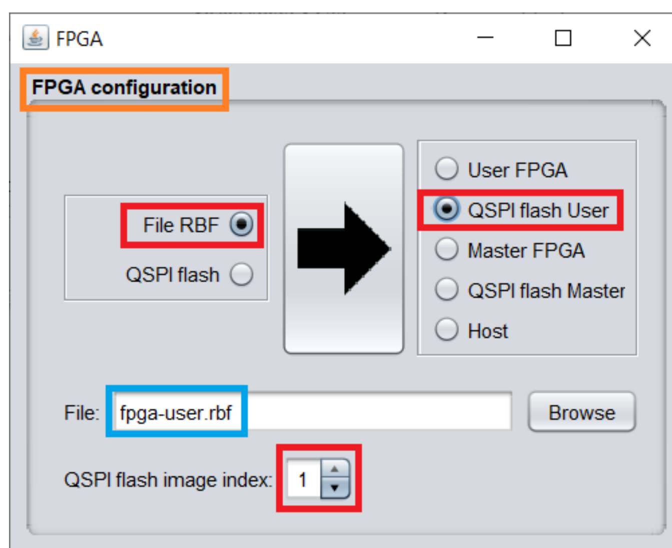


Figure 2.4: QSPI User: How to use with the dedicated Java App

- Select the FPGA configuration.
- Select **File RBF** and **QSPI flash User**.
- Select the firmware to be about by clicking on **Browse**. Here the target file is *fpga-user.rbf*.
- Select the index **1** for the QSPI flash image.
- Finally, click in the big arrow on the center of the graphical interface.
- Wait for the process to be finished.

2.3.2 Digital Storage Oscilloscope

The DSO (details on the characteristics in [12]) offers the possibility to measure the supply current of:

- User FPGA I/O (3.3 V);
- User FPGA PLL (2.5 V);
- User FPGA core (1.2 V);
- User MCU (3.3 V).

The DSO could be triggered by the followings channels:

- IO9: digital trigger;
- Channel 1;
- Channel 1;
- User FPGA I/O (3.3 V);
- User FPGA PLL (2.5 V);
- User FPGA core (1.2 V);
- User MCU (3.3 V).

With the modes bellow:

- automatic;
- normal;
- single;
- halted.

The DSO stores the read values into a buffer. The trigger is used as a temporal reference, which would be the middle of the buffer: then, the first part of the buffer are values that have been sampled before the trigger and the second part after the trigger.

The different methods to exploit the DSO are described in section 4.2.

2.4 User side

The User side (on the right of Figure 2.2) is the part of the board which is programmed for this thesis.

The main components on the User side are:

- **MCU-User**
- **FPGA-User**
- **LEDs**: 4 for the MCU and 4 for the FPGA, mainly used for debug purposes

- **8 Switches**: mainly used to interact manually with the design put on the FPGA
- **LCD**: not used here
- **GPIO pins**: 32 IO interconnect the MCU-User and the FPGA-User

Chapter 3

Hardware implementations: XNOR-net

3.1 Gitlab versioning

The sources of the thesis are shared and versioned on Gitlab [13].

Those sources collect all the versions of the XNOR-Net that are described in the following sections, including the STM32CubdeIDE and Quartus projects, to be ready to use. They are supported by ReadMe to explain how to use them.

3.2 VHDL description

The XNOR-Net described in [1] is then described in VHDL.

3.2.1 Overview

As describe in [1], the XNOR-Net for LiM architecture computes the binary convolution of IFMAP and weights, called K. The result of the convolution is returned by OFMAP. The XNOR-Net is composed of five components, as shown on Figure 3.1:

- **Control Unit**: generates the control signals to command the behavior of the other components, which only compute the data;

- **LiM XNOR**: memory which contains an XNOR gate for each cell, to perform XNOR bitwise operations on inputs;
- **Interface Decoder**: sends data from LiM XNOR to LiM Ones Counter;
- **LiM Ones Counter**: memory which contains an half-adder in each memory cell, to compute the pop counting, together with Pop Counting Logic;
- **Pop Counting Logic**: some logic to finish the pop counting computation started by LiM Ones Counter.

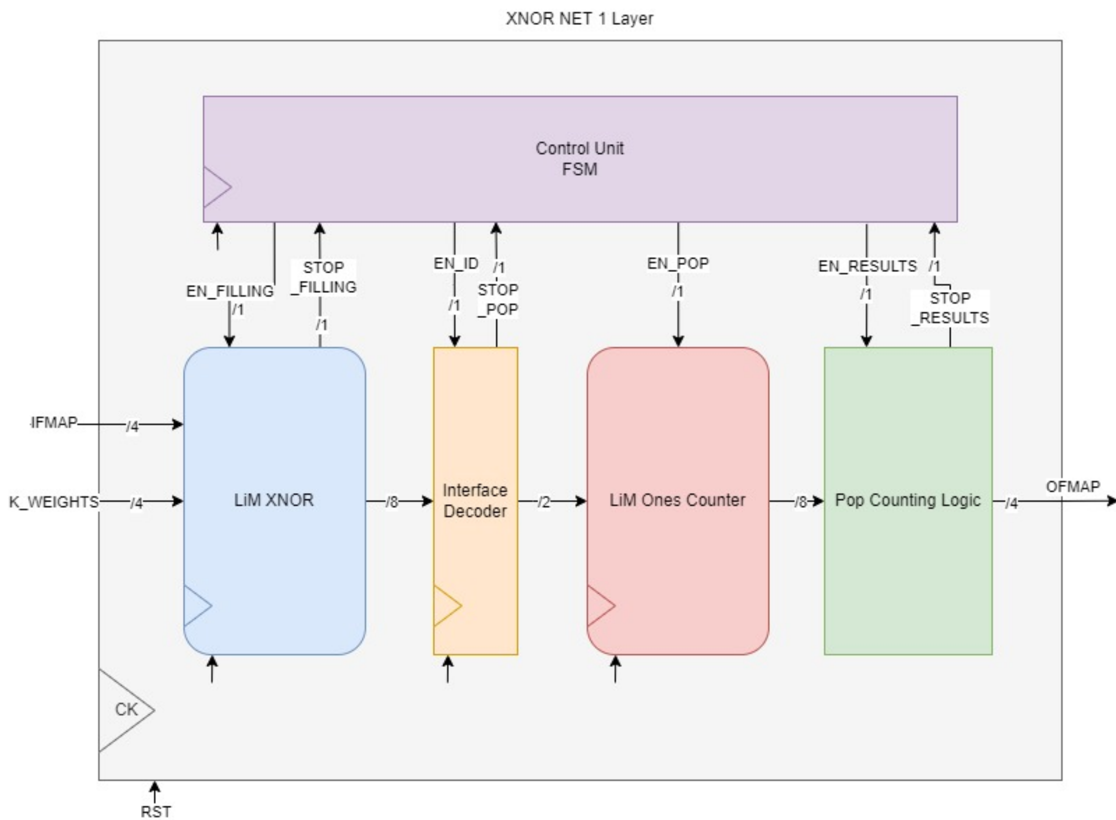


Figure 3.1: XNOR-Net V1: overview

This VHDL follows the XNOR-Net architecture described in [1], and additionally integrates a control unit.

The following sections go deeper into details for each component of the XNOR-Net VHDL description.

3.2.2 Control Unit

As already said, the XNOR-Net performs the binary convolution between IFMAP and K. The control unit manages the entire flow in each component to compute that overall operation.

In the following sections, let us consider that the memory is of size $N \times M$ with: N the number of bits per word; M the number of words.

As shown on Figure 3.2, the control unit is composed of six states:

- **Reset**: resets all components if RST is up
- **Idle**: resets counters, 1 *clock cycles*
- **Filling_Xnor**: writes inside LiM XNOR, M *clock cycles*
- **Pre_Pop_Computing**: starts sending LiM XNOR results to LiM Ones Counter, 1 clock cycle
- **Pop_Computing**: computes number of $1s$ inside each word coming from LiM XNOR, N *clock cycles*
- **Results**: sends difference between $0s$ and $1s$ for each word of the memory, one by one, M *clock cycles*

The Control unit does not need any external stimuli, except the reset signal: the current stage changes by itself, and the flow cannot be stopped without resetting. For this reason, this implementation requires a fully synchronous protocol.

Indeed, some counters directly integrated into some components are used for this synchronous protocol. In particular for the stage *FILLING_XNOR* and *RESULTS*.

For *FILLING_XNOR*, at each clock cycle are expected a IFMAP and K value to be written. When M clock cycles have passed, the control unit considers that the memory is full and that the computation can be launched.

There is the same idea behind *RESULTS* state: it automatically sends the results one by one. It means that the entity that wants to read those results must be fully synchronized to be able to sample each result.

3.2.3 LiM XNOR

The LiM XNOR (Figure 3.3) does the xnor bitwise operation between IFMAP and K inputs.

This is a memory of size $N \times M$ with:

- N: the size of one word, which corresponds to the size of K in bits

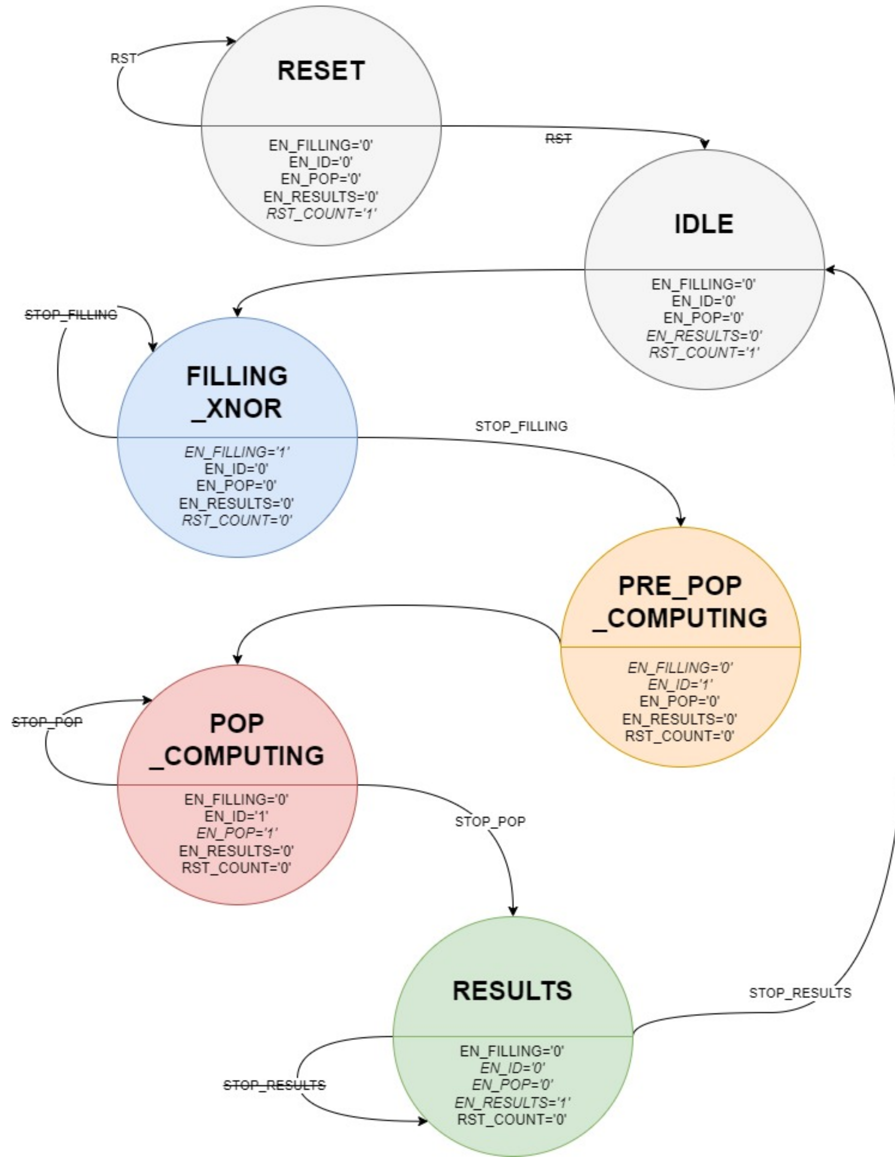


Figure 3.2: XNOR-Net V1: Control Unit

- M: number of words (memory depth), which correspond to the size of $size(IFMAP) \div size(K)$ in bit. Note that IFMAP must be a multiple of K.

For instance, on Figure 3.3: K is 4-bit long, so four memory cells per word are used; IFMAP is 8-bit long, so two words, i.e. two lines in memory.

Each memory cell is composed of one flip flop and one xnor gate. The IFMAP value is held within the flipflops, and K is a combinatorial input

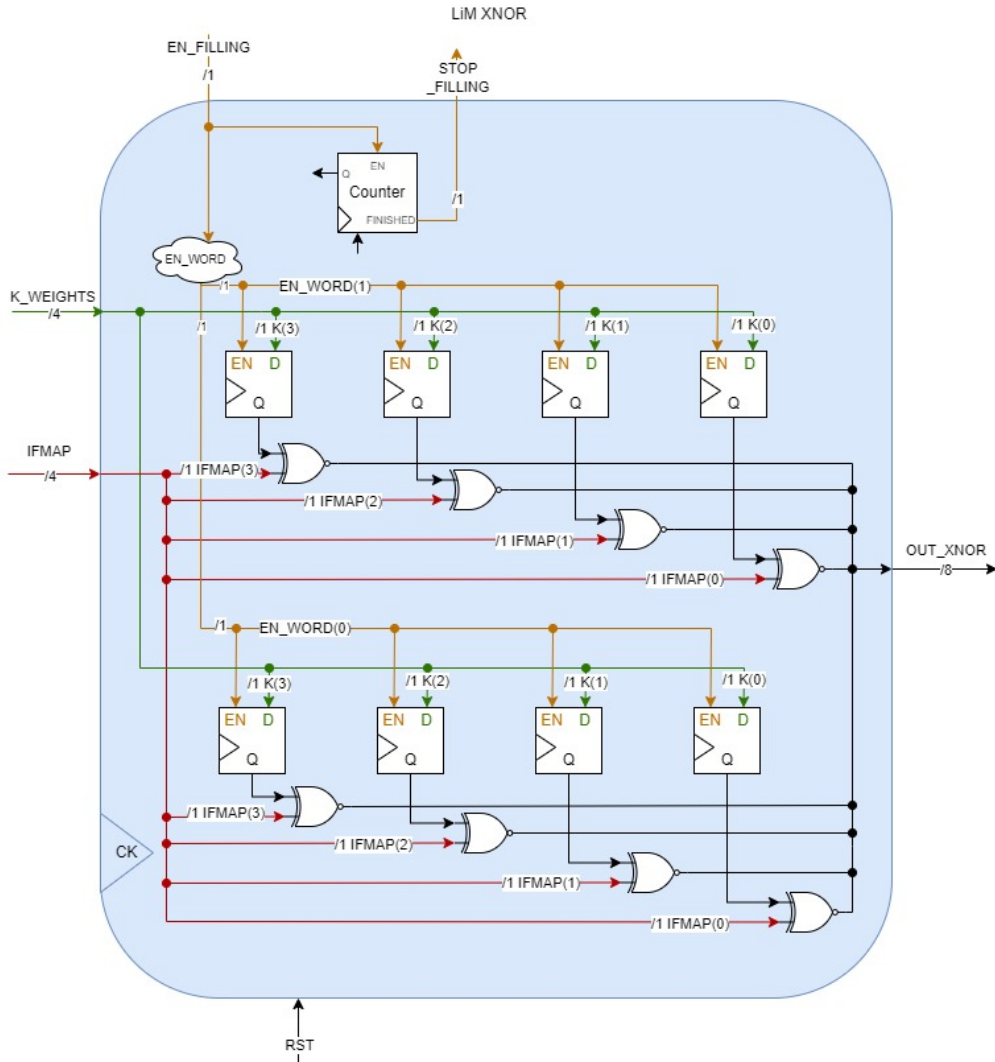


Figure 3.3: XNOR-Net V1: LiM XNOR

directly connected to the xnor gate.

The output, *OUT_XNOR*, is the result of xnor operation between IFMAP anf K.

The memory is filled word by word. Indeed, the signal **EN_WORD** controls the enable signal of each flipflop, depending on the **EN_FILLING** control signal value.

EN_WORD signal is built as follows:

- Each bit is associated to 1 word, which means that EN_WORD is M-bit long

– MSB: associated to word index M (first word generated, on top)
– LSB: associated to word index 1 (last word generated, on bottom)

The counter inside the LiM XNOR is used to inform the control unit that the LiM is finally full: each word of the memory has been written. This implies that the control unit will change state and go on with the computation.
As a consequence, the counter counts until the total number of words. In the example on Figure 3.3, it counts until 2 and then set **STOP_FILLING** at 1.

3.2.4 Interface Decoder

Once the LiM XNOR is full, the Interface Decoder (Figure 3.4 sends to the LiM Ones Counter the results bit per bit from LiM XNOR.

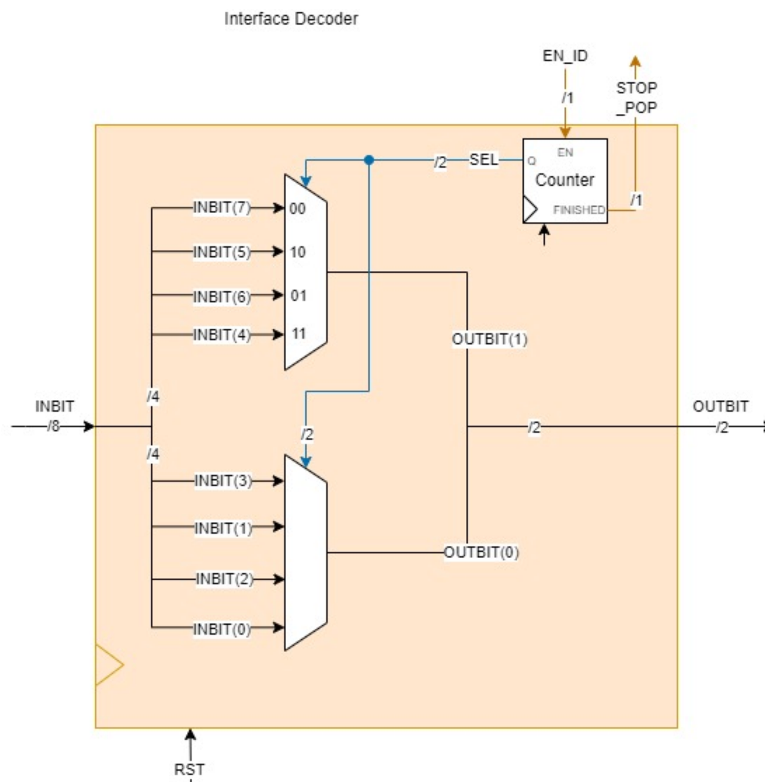


Figure 3.4: XNOR-Net V1: Interface Decoder

It is composed of M NTO1 muxes, with M the number of words and N the number of bits per word.

As depicted on Figure 3.4, a counter selects the bit to be sent: its output is used as a selector to the muxes. At each clock cycle, one bit of each word is sent to the LiM Ones Counter. When all bits have been sent, i.e. the counter reaches N and sets **STOP_POP** to 1, the computation is done: the control unit changes the state to *RESULTS*.

3.2.5 LiM Ones Counter

The LiM Ones Counter (Figure 3.5), as its name tells, counts the number of 1s inside each word coming from LiM XNOR.

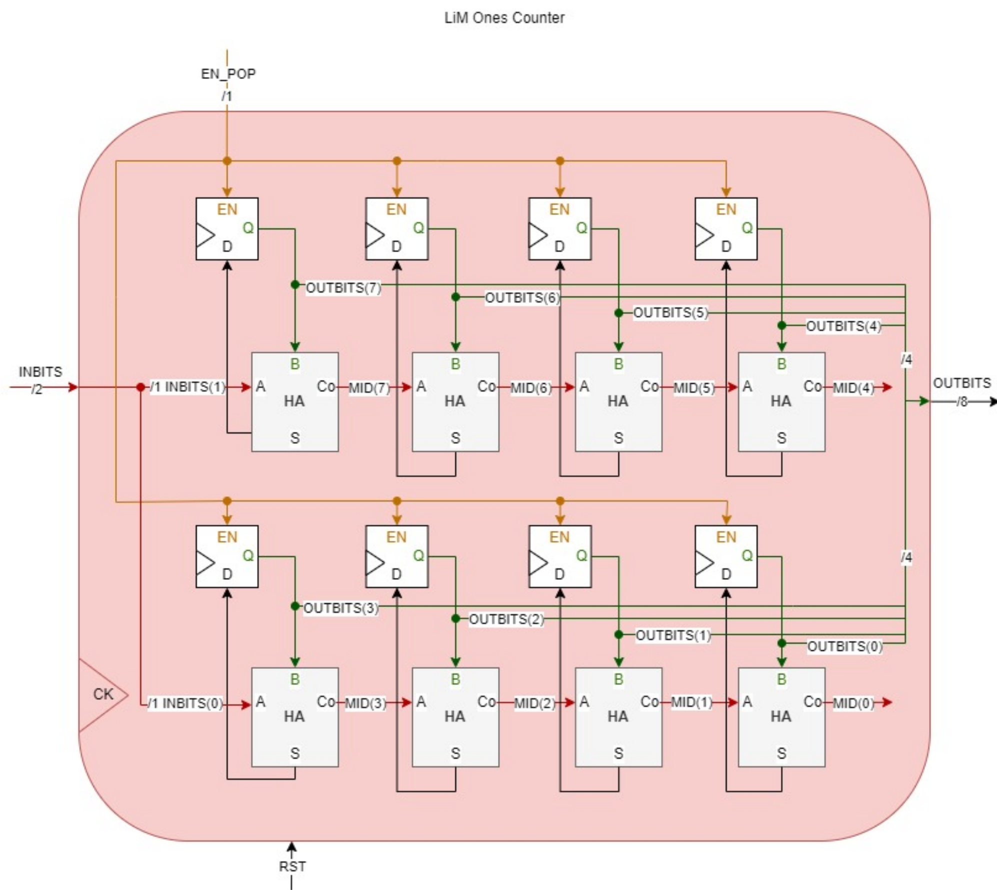


Figure 3.5: XNOR-Net V1: LiM Ones Counter

The LiM Ones Counter has the same dimensions than LiM XNOR, i.e. of size $N \times M$ with:

- N: size of one word, which correspond to the size of K in bit

- M: number of words (memory depth), which correspond to the size of $size(IFMAP) \div size(K)$ in bit. Note that IFMAP must be a multiple of K.

Here, each memory cell holds one flipflop and one half adder. When all bits of each word have been received, the fliflops representing one word (flipflops of the same line) hold the number of 1s present within the associated word in LiM XNOR.

3.2.6 Pop Logic

Finally, the Pop Logic component (Figure 3.6) computes the difference between 0s and 1s within each word held in LiM XNOR. Its computation is based on the one done earlier by the LiM Ones Counter.

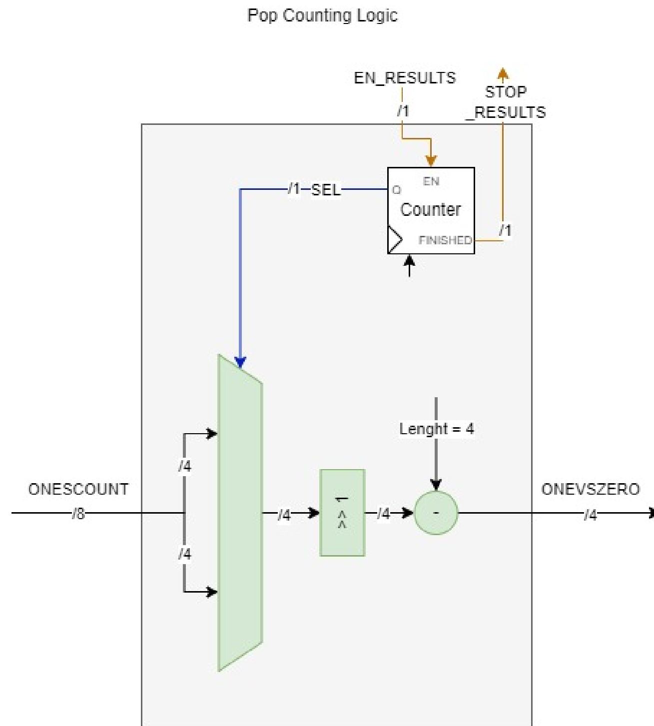


Figure 3.6: XNOR-Net V1: Pop Logic

It does not compute that result for each word in parallel but one by one. Indeed, a multiplexer, controlled by a counter, selects a word coming from the LiM Ones Counter one by one.

As a consequence, one result is sent to the output each clock cycle. When all results have been sent, i.e. the counter reaches two on Figure 3.6 and set

STOP_RESULTS to 1, the overall computation finally ends. Then, the control unit goes back to *IDLE* and is ready to launch another flow.

3.3 MCU based: C implementation

The MCU-based version is simply a C program that implements the XNOR-Net behavior. The overview of the flowchart is shown on Figure 3.7.

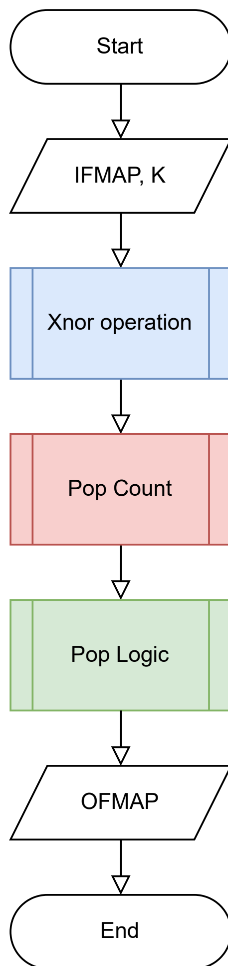


Figure 3.7: XNOR-Net C program: Overview

The program expects as inputs IFMAP and K, as for the hardware implementation. It performs the xnor operation on the inputs, then computes the popcount of each word, and finally computes the difference between $0s$ and $1s$ within each word. The results are directly written in OFMAP.

Note that the Interface Decoder phase is no longer present: the popcount in C does not need to be fed bit per bit. The results from the xnor operation are processed as a whole.

To define the size of the memory within the C program, three constants are used:

- NWORD: Number of words within the memory, i.e. depth
- NBIT: Number of bits per word
- N_UINT_PER_WORD: equals to $NBIT \div 32 + 1$, i.e. the number of `uint32_t` for representing the entire word

One bitwise value is represented by a single bit of a `uint32_t`. Thus, a single `uint32_t` can represents at most 32 values. In the case of a word longer than 32 bit, additional `uint32_t` are necessary. As a consequence, a single word is represented in the C program by: a single `uint32_t` if $NBIT \leq 32$; a list of `uint32_t` if $NBIT > 32$.

For this reason, the input IFMAP is a list of $NWORD \times N_UINT_PER_WORD$ `uint32_t` to contain all bits.

As for IFMAP, the input K is NBIT-long (i.e. the same size that a single word of IFMAP): it means that K is represented as a list of N_UINT_PER_WORD `uint32_t`.

On the contrary, OFMAP is a list of NWORD `int`. A single `int` value is quite enough for representing the difference between 0s and 1s within a single word. Indeed, the maximum value of a `int` is $+2147483647$ and minimum value is -2147483648. To reach those values, a single word should be 2147483647-bit long. Having a word as long as that does not have any relevance in this thesis. That is why the result is coded using a single `int` value.

3.3.1 XNOR operation

The XNOR operation (Figure 3.8) is applied on the inputs IFMAP and K during this phase. The computed result is written in the output xnorres.

The xnorres variable is described exactly as IFMAP: a list of $NWORD \times N_UINT_PER_WORD$ `uint32_t`.

Two loops are used to achieve this computation:

- One for processing all word
- The other for processing all `uint32_t` of a single word

The output xnorres is then updated for each `uint32_t`, as followed, performing the xnor bitwise operation between the corresponding `uint32_t` of IFMAP and K:

```
xnorres[i*N_UINT_PER_WORD + j] |=
                ~(ifmap[i*N_UINT_PER_WORD + j] ^ k[j]) & mask ;
```

The algorithm complexity of this phase is then: $\mathcal{O}(M \times N/32)$.

3.3.2 Pop Count operation

A built-in C method already exists to count the number of 1s within a bitwise variable. Then pop count operation (Figure 3.9) consists on applying this method, `__pop_count()`, on each `uint32_t` taken from xnorres.

The results are directly written in the OFMAP variable that will be modified in the next and final step.

The algorithm complexity of this phase is then: $\mathcal{O}(M \times N/32)$.

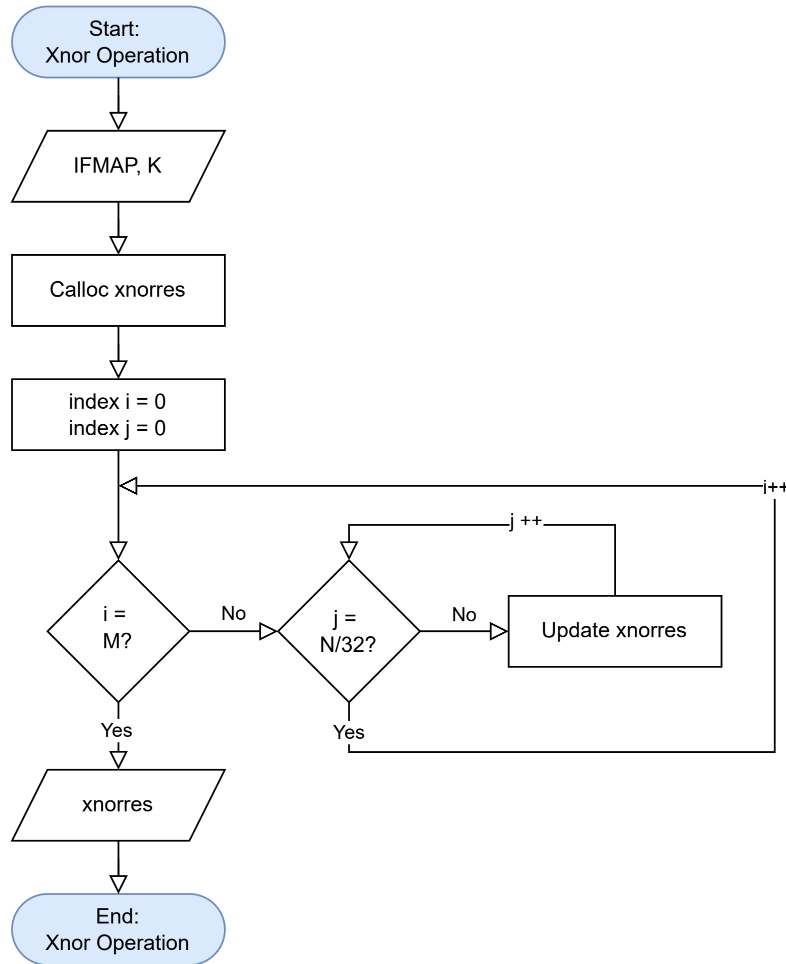


Figure 3.8: XNOR-Net C program: Xnor operation

3.3.3 Pop Logic operation

This phase computes as the logic circuit in the VHDL description the difference between $1s$ and $0s$ within each word.

To do so, this operation (Figure 3.10) uses again a for loop to compute the final result. It takes OFMAP as an input, and for each word: multiply the corresponding value by two and subtracts the result by the number of bits of one word.

The algorithm complexity of this phase is then: $\mathcal{O}(M)$.

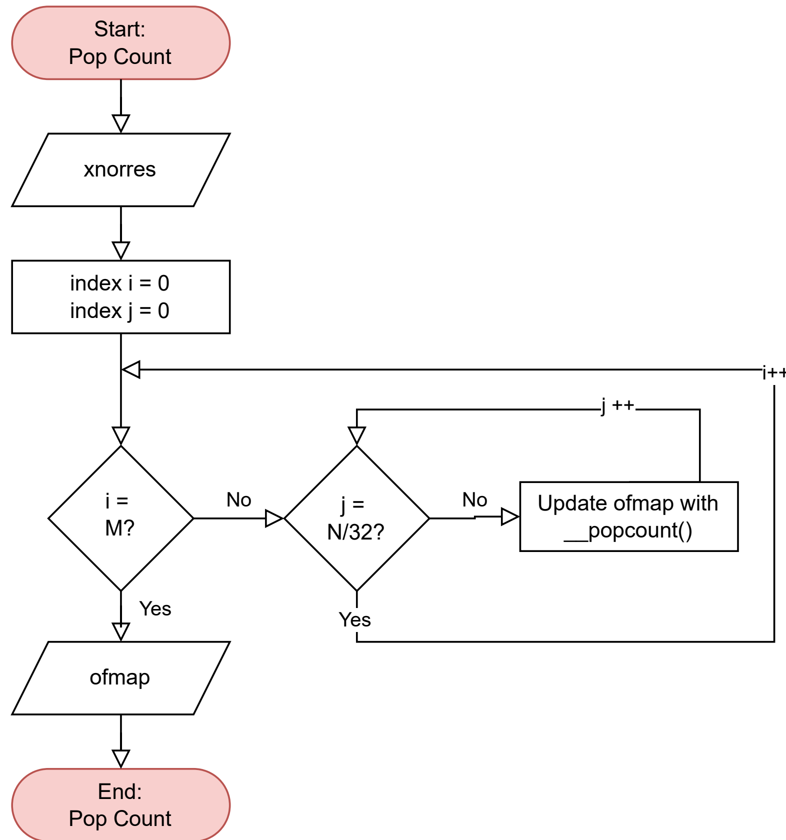


Figure 3.9: XNOR-Net C program: Pop Count operation

3.3.4 Algorithm cost

Grouping the cost of each phase, the overall cost is:
$\mathcal{O}(M) + 2 \times \mathcal{O}(M \times N/32) \approx \mathcal{O}(MN)$.

3.4 FPGA based

3.4.1 Overview

The FPGA based implementation uses the VHDL description of the XNOR-Net presented in section 3.2: this version is downloaded on the FPGA-User. This holds a version of the XNOR-Net 4×2. The objectives of this version are to:

- synthesize for the first time the XNOR-Net VHDL description with Quartus;

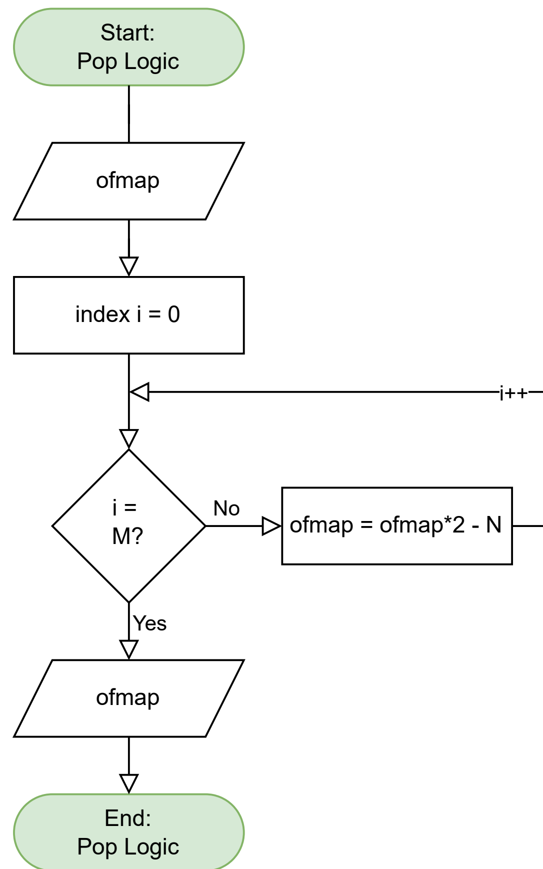


Figure 3.10: XNOR-Net C program: Pop Logic operation

- test a first instance of the XNOR-Net on the VirtLAB;
- debug and interact a simple version of the XNOR-Net.

The overall interconnection of this version is depicted on Figure 3.11.

The eight switches are putting the binary value of IFMAP, and K (4 bits each) of the FPGA are the input to be written in the LiM. The LEDs are set to the value of OFMAP.

Not having enough switches and LEDs, an external breadboard is connected to the IOs of the VirtLAB to provide to the LiM the reset signal and the clock and to observe the current state of the control unit.

The internal clock is not suitable for this implementation: human control is needed to understand what is happening within the design of the LiM and perfectly check whether the behavior is the one expected.

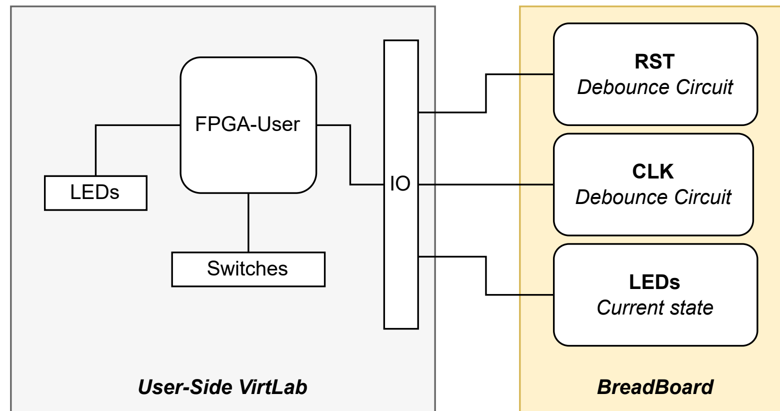


Figure 3.11: FPGA-based: Interconnection between the VirtLAB and the external breadboard

3.4.2 Extended I/O using a Bread Board

A picture of the breadboard is on Figure 3.12.

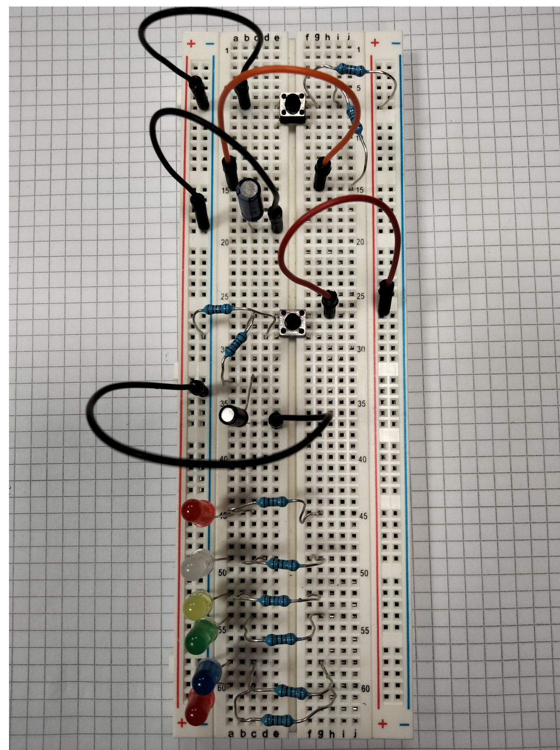


Figure 3.12: Breadboard connected to VirtLAB

The *GND* pin of the VirtLAB must be connected to the minus column on the left of the breadboard. On the other hand, Vdd must be plugged on the plus column on the right to feed the breadboard with the correct voltage.

On the top, there is a debounce circuit (pull-up) with a button for the *RST* signal. The pin connected to the RST IO on the VirtLAB should be put on the line where the two resistances and the button are connected.

In the middle, the second button and its debounce circuit (pull-down) are for the *CLK* signal. As for the RST signal, the CLK IO on the VirtLAB should be plugged into the line where the button and the two resistances meet.

Finally, on the bottom, there are six LEDs: one for each state of the control unit in the order (from *RESET* to *RESULTS*). One IO on the VirtLAB is dedicated to one state. They must be aligned with the associated led on the breadboard, at the resistance right side.

3.4.3 Debounce circuit

The debounce circuit for the RST and CLK signals is needed to compensate for the unperfect behavior of the button. The used button is a push button: when it is pressed, the value is unstable for a short period of time before being finally stable. This is due to the piece of metal that the push button moves while it is pressed: this piece of metal vibrates and needs time to stabilize.

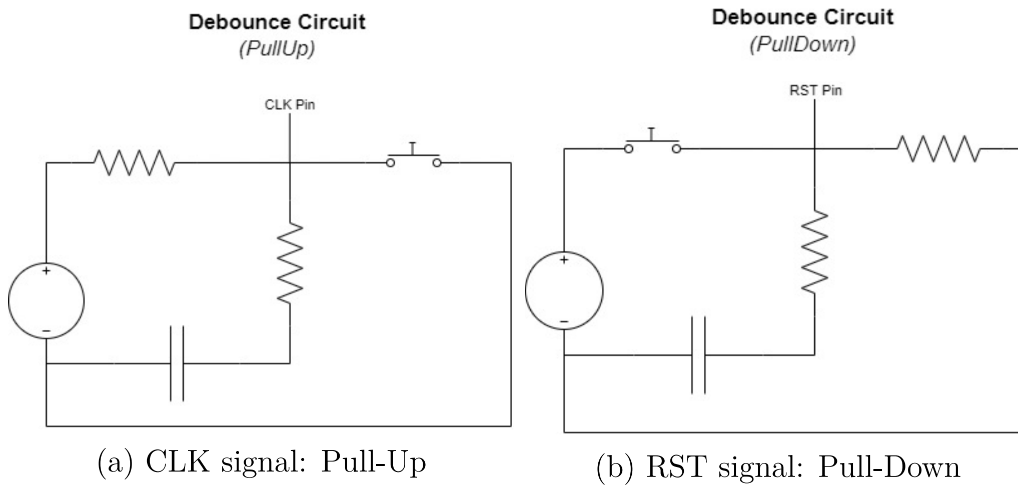
The debounce circuit is composed of:

- A push button;
- Two resistances;
- A capacitor.

The debounce circuit for the CLK signal is Pull-Up (Figure 3.13a). This means that the CLK signal is set to 1 when the push button is up and to 0 when it is pressed.

On the contrary, the debounce circuit for the RST signal is Pull-Down (Figure 3.13b): $RST = 0$ when the button is up; $RST = 1$ when the button is pressed.

Even with this hardware debounce circuit, the phenomenon has not been totally compensated, as shown on Figure 3.14. Ideally, the curve should be flat without the first peak.



(a) CLK signal: Pull-Up

(b) RST signal: Pull-Down

Figure 3.13: Debounce circuits on the breadboard to compensate the misbehavior of the push buttons

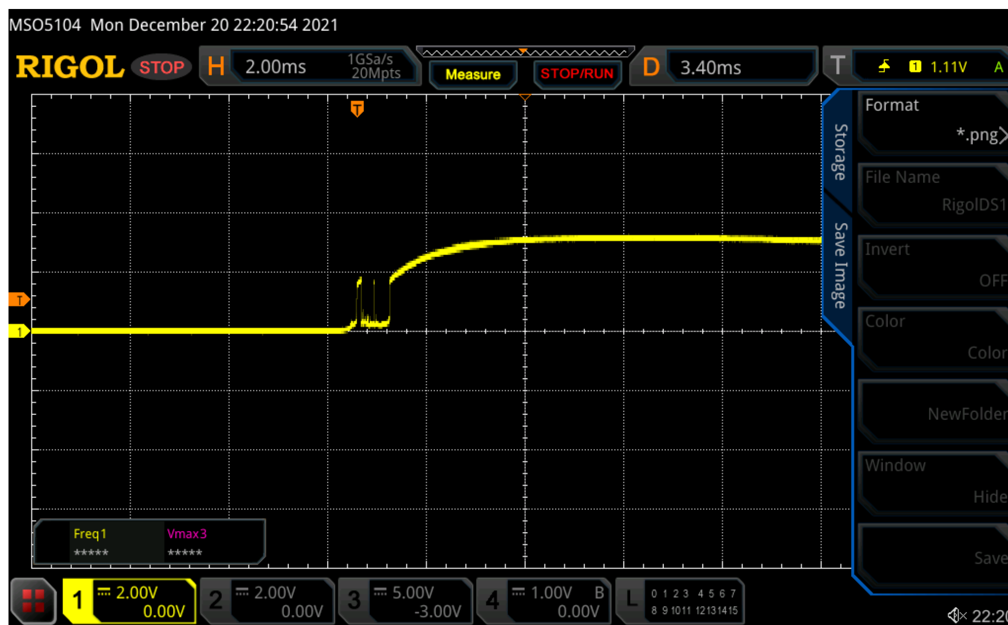


Figure 3.14: Debounce Circuit Anomaly

A possible solution to this problem could be a debounce software.

3.4.4 Debounce Software

The debounce software aims at stabilizing a signal due to misbehavior of a push button, as for the debounce circuit. Instead of a physical circuit, it is described with VHDL and integrated directly within the design.

Top entity

The debounce software detects when the button is pushed or not and waits for the signal to stabilize, as emphasized on Figure 3.15. If the button is activated when the value is equal to 1, then the debounce will wait for the button value to be stable at this value for a certain period of time.

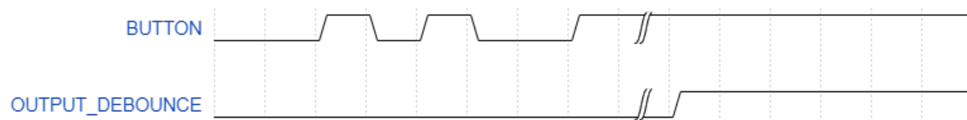


Figure 3.15: Debounce Software waveform: how does it react

The top entity of the debounce software is shown on Figure 3.16.

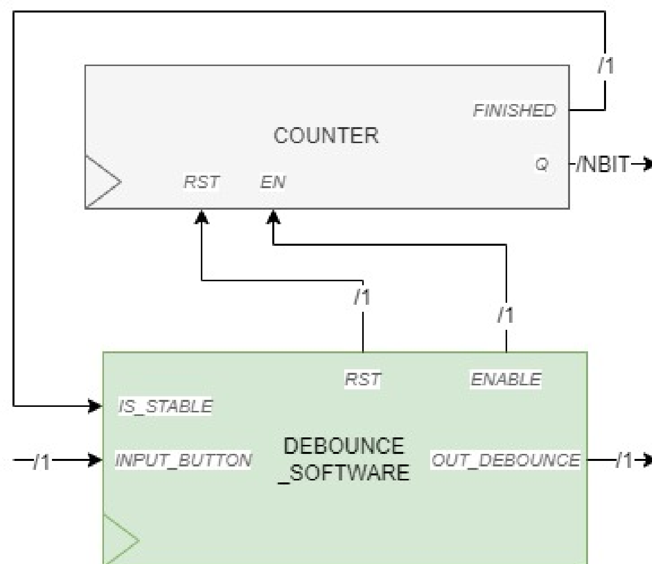


Figure 3.16: Debounce Software Top entity

The debounce software is described as a control unit that interacts with an external counter. This counter informs the control unit if the button value is stable or not.

Control Unit

The control unit of the debounce software is depicted on Figure 3.17.

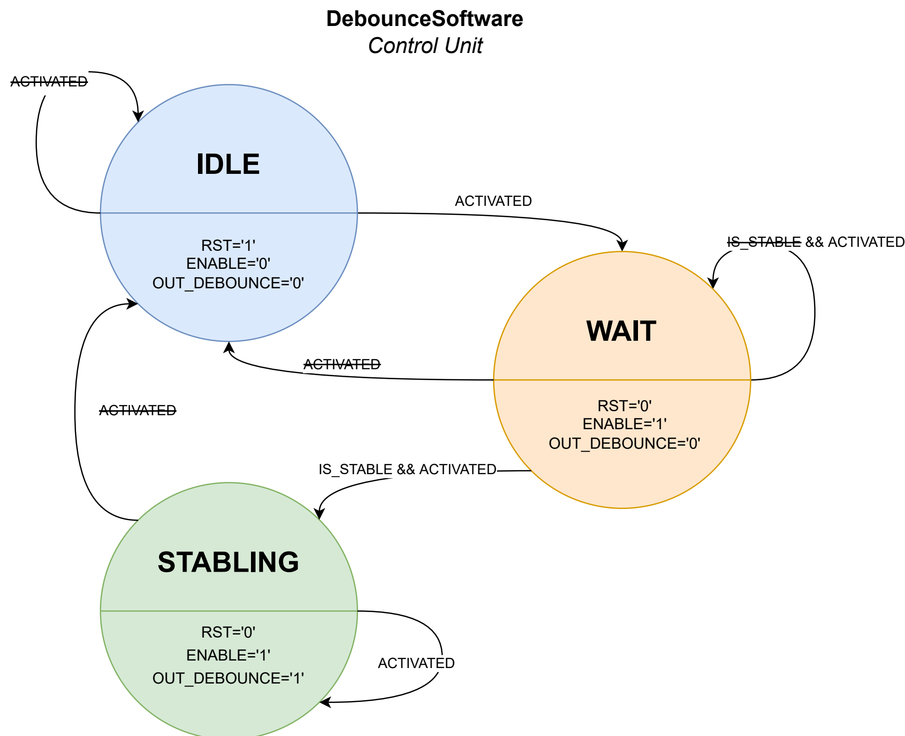


Figure 3.17: Debounce Software Control Unit

It is composed of three states:

- **IDLE**: The button is *NOT ACTIVATED*. The external counter is then reset, and the output is still at 0.
- **WAIT**: The button is *ACTIVATED*. The counter starts to measure the time in which the button value must be activated to be considered stable. If the value of the button changes during this state, the state IDLE is automatically reached.
- **STABLING**: The button is *ACTIVATED* and *IS_STABLE*. The counter finished its count, so the button value is considered as stable. As a consequence, the OUTPUT_DEBOUNCE is set to 1.

3.5 MCU-FPGA based

3.5.1 Overview

The MCU-FPGA-based version wants to introduce the LiM on FPGA as a co-processor and to feed the inputs in a more straightforward way than the FPGA-based version.

This means finally using the MCU and introducing a communication protocol between the MCU-User and the FPGA-User. On the VirtLAB User side, the only common point between those two components are the 32 IOs. The created protocol is described in subsection 3.5.4.

To allow the MCU to interact and control the LiM on the FPGA in a smoother way, the XNOR-Net implementation has been updated. The list and explanations of those changes are described in subsection 3.5.2.

3.5.2 Upgraded XNOR-net: changes overview

The previous VHDL description has been updated in order to have an external control more flexible, which allows asynchronous communication between MCU and FPGA.

Therefore, the following functionalities have been added:

- **Read operation as for a classical memory**: read enable with the target address
- **Write operation as for a classical memory**: write enable with the target address and input
- **Enable signal for computing** the convolution on the written inputs
- **Acknowledgement response from LiM** for the previous operations: read, write, compute

The overview of the XNOR-Net top entity is depicted Figure 3.18.
Five new inputs have been added:

- **WE_IFMAP**: Write enable signal for IFMAP input
- **WE_K**: Write enable signal for K input
- **RE**: Read enable signal for OFMAP
- **ADDR**: Target address for read and write

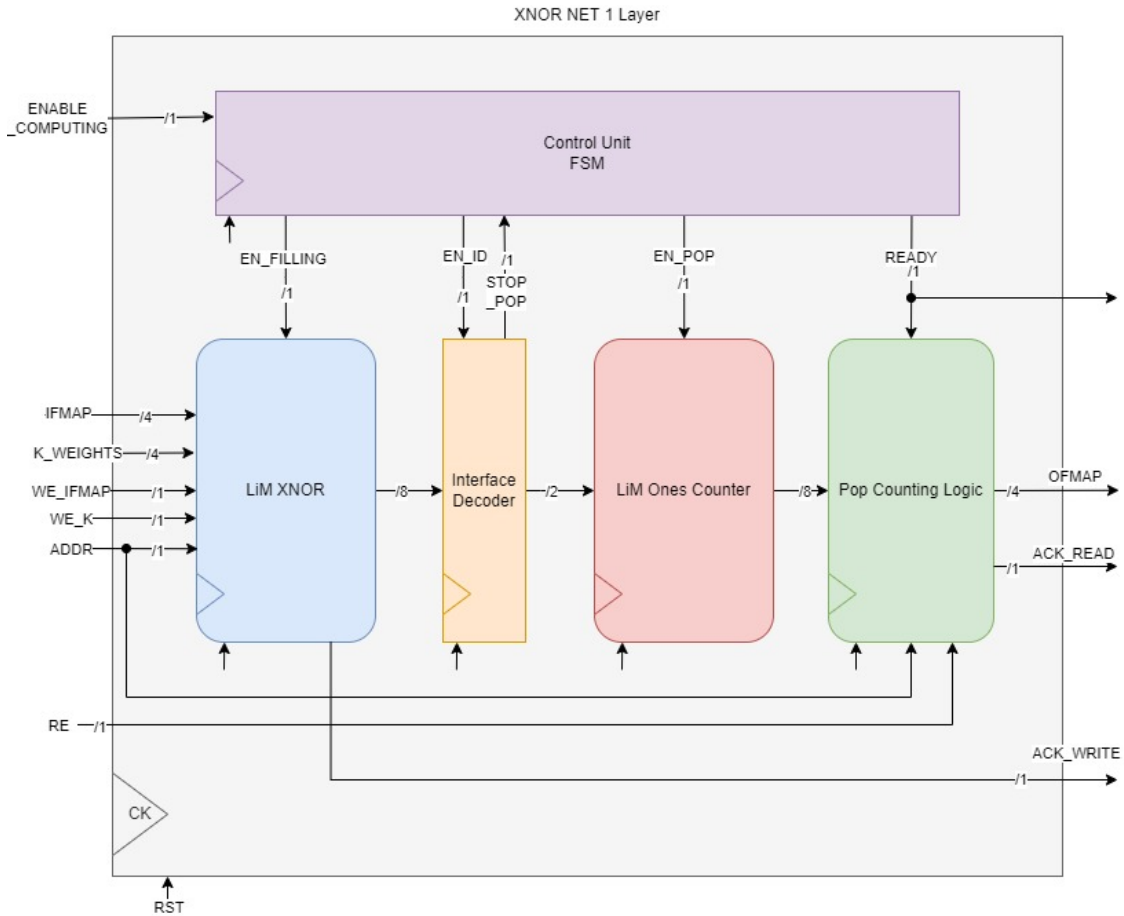


Figure 3.18: Updated XNOR-Net: Overview

- **ENABLE_COMPUTING**: Enable signal to compute the results on current values in memory

And also three outputs:

- **ACK_WRITE**: Acknowledge on write operation (K and IFMAP)
- **READY**: The computation is done and the LiM could now be red
- **ACK_READ**: Acknowledge on read operation (OFMAP can now be red)

3.5.3 Upgraded XNOR-net: Control Unit

To take into account those new input signals, the control unit experienced some changes: in particular on the states *IDLE*, *FILLING_XNOR*, and *RESULTS*. Its final version is depicted on Figure 3.19.

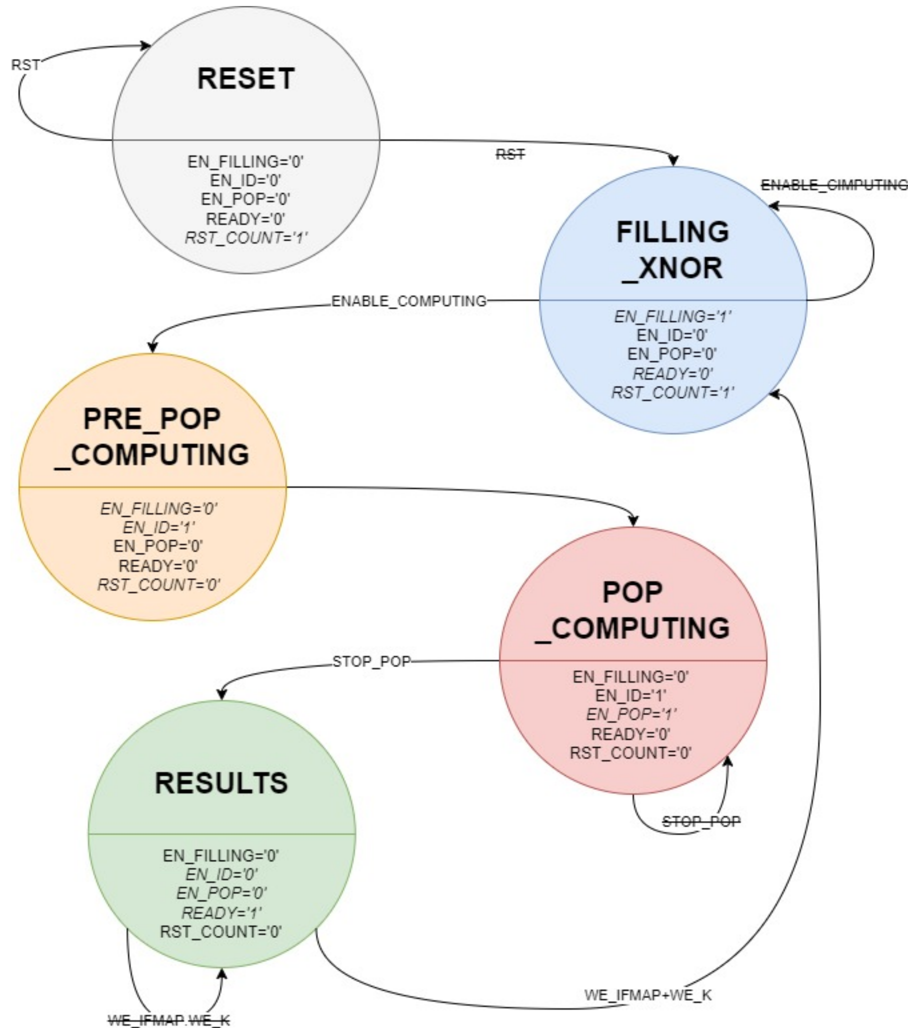


Figure 3.19: Updated XNOR-Net: Control Unit

In the previous version, the states *FILLING_XNOR* and *RESULTS* were not controlled by an external signal but by an internal one (**EN_FILLING** and **EN_RESULTS** respectively), coming from an internal counter. Therefore, the state *IDLE* is no longer useful: it was used to reset those internal counters. Since it no longer exists for *FILLING_XNOR*, the **RST_COUNT**

can now be done in that stage. That is why *IDLE* state has been definitively removed.

Now the MCU is able to choose when and how to read and write the LiM. As a consequence, the control unit stays in *FILLING_XNOR* state until the **ENABLE_COMPUTING** input is set. Once in state *RESULTS*, the control unit keeps that state until any write operation is requested (**WE_IFMAP** and **WE_K** are both unset).

Table 3.1 sums up which are the windows in which it is possible interacting with the XNOR-Net: in which states of the control unit is possible to request some operations from the LiM.

Operation	Associated states
Write IFMAP	FILLING_XNOR, RESULTS
Write K	FILLING_XNOR, RESULTS
Launch computation	FILLING_XNOR
Read OFMAP	RESULTS

Table 3.1: Constraints on operations given by external signals: states in which they are processed

3.5.4 Communication Protocol characteristics

The created communication protocol between the MCU-User and the FPGA-User uses the 32 IOs of the VirtLAB.

This is an **asynchronous** protocol, using **time multiplexing**.

The IOs are classified as followed (Figure 3.20):

- IO 0-5: MCU signals to be sent to FPGA
- IO 13-15: FPGA response signals
- IO 16-31: common databus
- IO 6-12: unused

This protocol is asynchronous because of the acknowledged response of the FPGA when its processing is done: thanks to those signals, the MCU is able to know when the FPGA is ready or has completed the requested operation, independently of the frequency of both components.

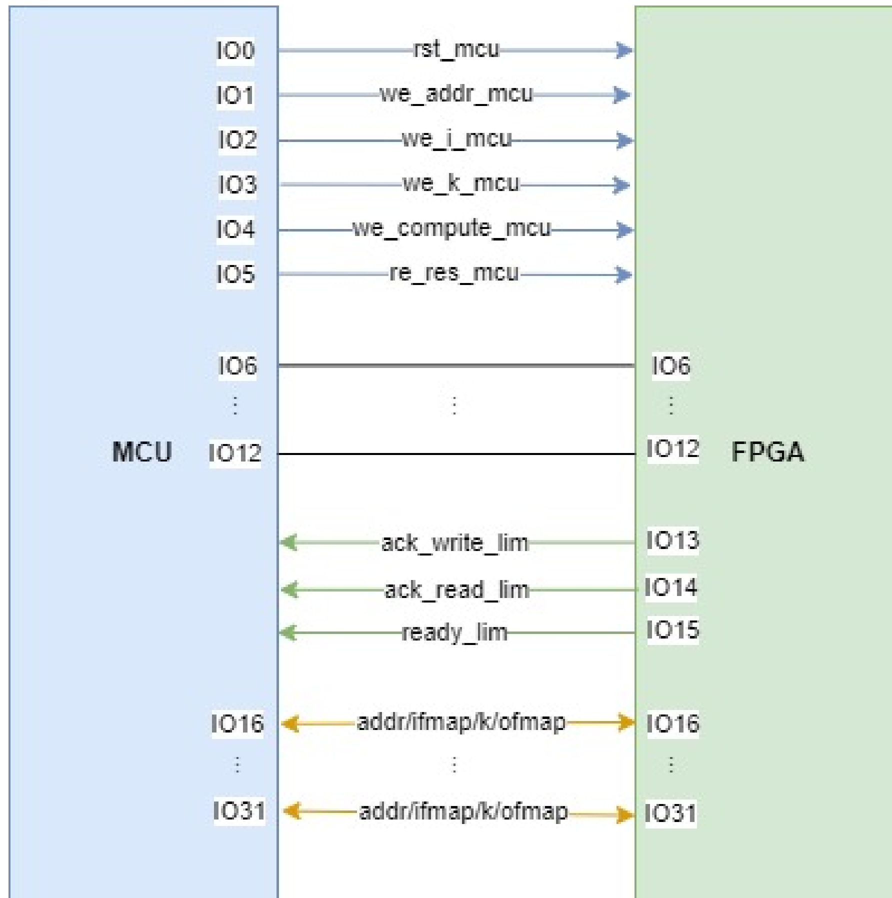


Figure 3.20: Protocol: Signals between MCU and FPGA

On the VirtLAB, the MCU is working at $80MHz$ and the FPGA at $10MHz$. The asynchronous aspect of the protocol allows those two entities to work at different frequencies.

Being aware that it still poses some problems: if the output of one component changes while the other one is sampling it (clock edge), it could create some glitches (because of violation of hold or setup time).

In addition, this protocol uses some time multiplexing because of its databus: those IOs do not have the same meaning in every step of the protocol.

First of all, the databus is common: both the MCU and FPGA can read and write on it, but they are not allowed to write simultaneously.

The MCU could write on the bus for:

- Write Operation: send the address, IFMAP or K value
- Read Operation: send the target address value

On the contrary, the FPGA is allowed to write on the bus only during the read operation: to send the requested OFMAP value.

3.5.5 Writing operation

The first operation to be done is to fill the LiM XNOR inside the XNOR-Net embedded on the FPGA with IFMAP and K. As highlighted in previous sections, the LiM offers a memory interface similar to the classical one.

To write, the MCU should use three signals to be sent to the LiM: **we_addr_mcu**, **we_i_mcu** or **we_k_mcu** and finally **databus**. On the other hand, the FPGA responds with **ack_addr_lim** and **ack_write_lim**.

The flow of the writing operation is depicted on Figure 3.21 and Figure 3.22: one for the IFMAP input and the other one for the K input. The mechanism is the same for both cases: the only difference is the write enable signal that are actually used by the MCU, i.e. **we_i_mcu** for IFMAP or **we_k_mcu** for K.

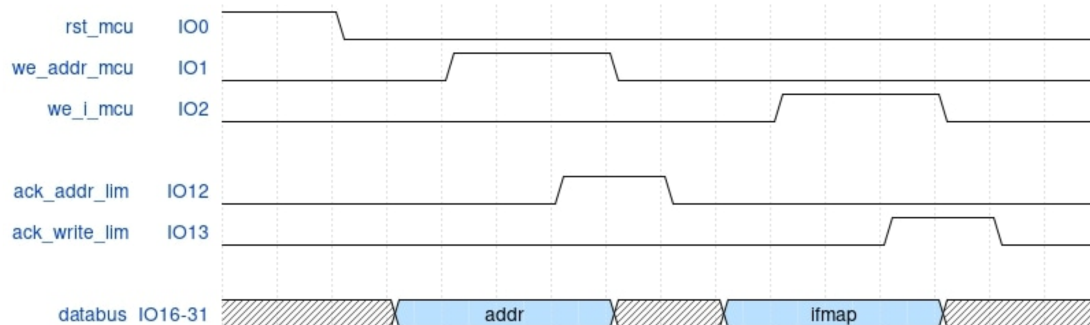


Figure 3.21: Protocol diagram: Write IFMAP

For instance, the steps to be followed, as depicted on Figure 3.21 are the followings:

1. Write textbfaddr value on databus
2. Set we_addr_mcu at 1
3. Wait for ack_addr_lim
4. Set we_addr_mcu at 0

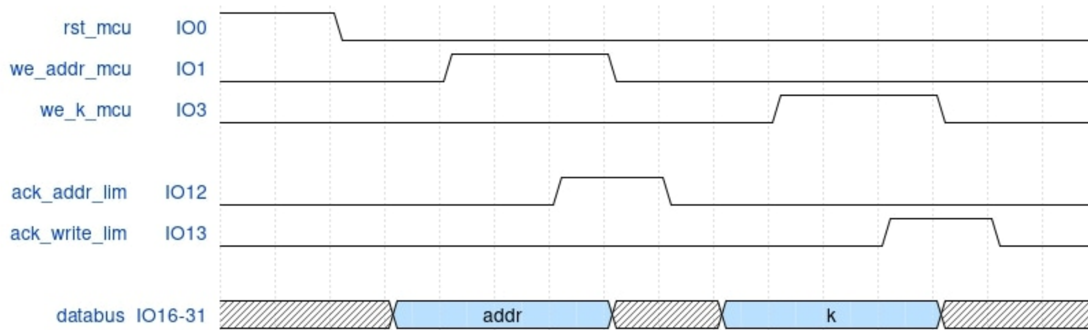


Figure 3.22: Protocol diagram: Write K

5. Write **IFMAP** value on databus
6. Set we_i_mcu at 1
7. Wait for ack_write_lim

This operation has some constraints, coming from the current state of the control unit. The LiM XNOR can be written only if the current stage is *RESULTS* or *FILLING_XNOR*. This means that the **rst_mcu** signal must be low and that the XNOR-Net is not currently computing, i.e. the **we_compute_mcu** is low.

3.5.6 Launch computation operation

Once the data is written inside the LiM, the computation is ready to be launched. This part of the protocol is described on Figure 3.23.

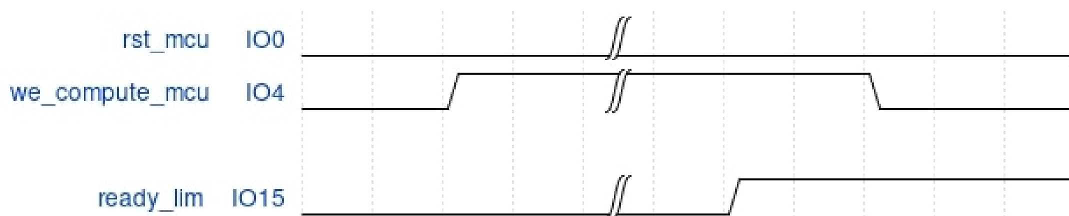


Figure 3.23: Protocol diagram: Compute

The steps to be performed are the followings:

1. Set we_compute_mcu at 1
2. Wait for ready_lim

3. Set we_compute_mcu at 0

The computation can be launched only if the current stage of the control unit is *FILLING_XNOR*.

The XNOR-Net has finished its computation only when **ready_lim** is up.

3.5.7 Read operation

While the computation is done, the results could finally be red, as described on Figure 3.24.

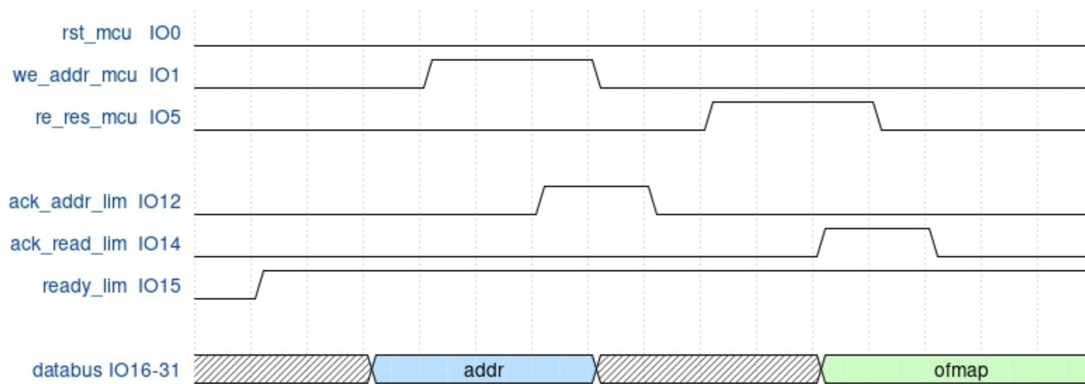


Figure 3.24: Protocol diagram: Read OFMAP

1. Write **addr** value on databus
2. Set we_addr_mcu at 1
3. Wait for ack_addr_lim
4. Set we_addr_mcu at 0
5. Set re_res_mcu at 1
6. Wait for ack_read_lim
7. Set re_res_mcu at 0
8. Read **ofmap** value on databus

For reading, the control unit must be in stage *RESULTS*, i.e. **ready_lim** is set.

Chapter 4

Measurements Methods

4.1 Timing measures

To estimate the performance of the XNOR-Net, the execution time for its different implementations is measured, exploring different methods.

The execution time that should be interesting to measure are:

- **Computation time** of the algorithm: when the memory is already full, in how much time the results are available
- **Time to write** a value: K and IFMAP
- **Time to read** a value: OFMAP

4.1.1 MCU

The timing measures could be done directly on the MCU-User. The following sections present the different methods.

DWT register

The DWT (Data Watchpoint Trigger) is a register of the ARM cortex that counts the number of clock cycles.

The process to be followed is depicted on Figure 4.1.

First, the use of the DWT register must be enable:

```
volatile uint32_t *DEMCR = (uint32_t *)0xE000EDFC;
```

Then, the cycle counter must be enable:

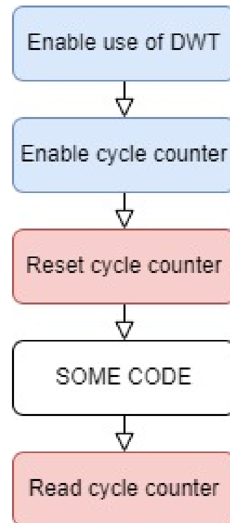


Figure 4.1: Process of use of the DWT register

```
volatile uint32_t *DWT_CONTROL = (uint32_t *)0xE0001000;
```

Finally, the cycle counter value is available at the following address:

```
volatile uint32_t *DWT_CYCCNT = (uint32_t *)0xE0001004;
```

As shown on Figure 4.1, it is recommended to reset the cycle counter value just before the start of the code to be measured. Then to read the value just after the end of the targeted execution.

Timers and GPIO Interrupts

This method is particularly efficient for the MCU-FPGA based implementation because of its communication protocol based on IOs.

The idea of this method is represented on Figure 4.2, showing an example for measuring the computation time.
The MCU asks the FPGA to compute by setting the signal *we_compute_mcu* on the IO4. The FPGA answers the MCU when it has completed the algorithm by setting *ready_lim* on IO15. Those IOs are then used as trigger by a GPIO interrupt.
During the GPIO interrupt on IO4, it stores the current value of the timer, and enables it. Then, during the GPIO interrupt on IO15, the timer is disabled. The elapsed time is performed by subtracting the actual time of the timer from the one stored before.

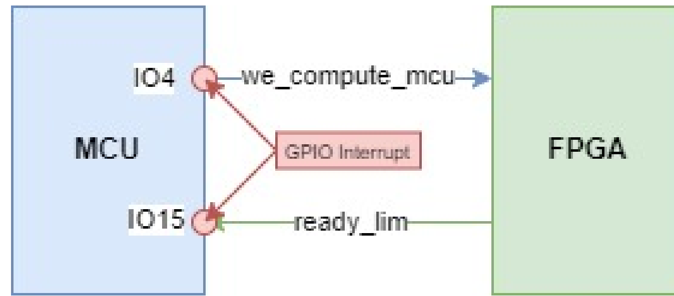


Figure 4.2: GPIO interrupts example: how to measure computation time in MCU-FPGA based implementation

The targeted timer is *TIM2* and works at the frequency of the FPGA multiplied by two: this is to avoid missed reading and improves the precision of the measure.

Input Capture

The Input Capture method is unfortunately not realisable on the VirtLAB: an IO cannot be assigned as Input Capture and GPIO at the same time. Since the measures, to be the most precise possible, are based on the IO values, this method is certainly not a good choice.

Timers only

This method is dedicated to MCU-based implementation, because it would not be precise enough for the MCU-FPGA based. The communication protocol introduces delays, because of the `while` loop that waits for the IO value to change. Most probably, checking the IO value implies going into some critical section and unexpectedly increases the measured execution time. Thus, it is much better to detect the variation of the IOs with an interrupt, as previously described in section 4.1.1.

By using the same timer as the method presented in section 4.1.1, this one is controlled to measure the computation time of the MCU-based implementation as shown on Figure 4.3.

The computation is done by the step *XNOR bitwise* and *Pop-counting*. The timer is consequently surrounding those operations: the timer is read and enabled before the XNOR operation. Then, the timer is disabled and read again after the pop-counting operation.

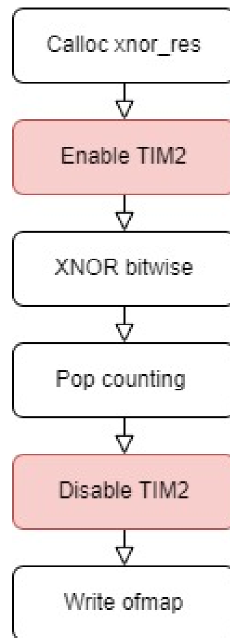


Figure 4.3: Example of use of timers to measure computation time on MCU-based implementation

By subtracting the read value, the elapsed time is obtained.

With the timer, it is recommended to check whether the max value has been reached or not: otherwise, the subtraction result to obtain the final execution time would be wrong. In this thesis, working on little dimensions of the XNOR-Net, this problem is not encountered.

STM32CubeMonitor: Graphical interface

STM32CubeMonitor (Figure 4.4) allows to visualize graphically the value of the *global* variables within the code of the MCU itself, while it is currently computing. The measures are then done in *real-time*.

In fact, the execution time value of the LiM and the MCU that are computing by the MCU are now visible in live.

For example, on Figure 4.4, with the help of a cursor, the read value of *compute_lim* is 64.

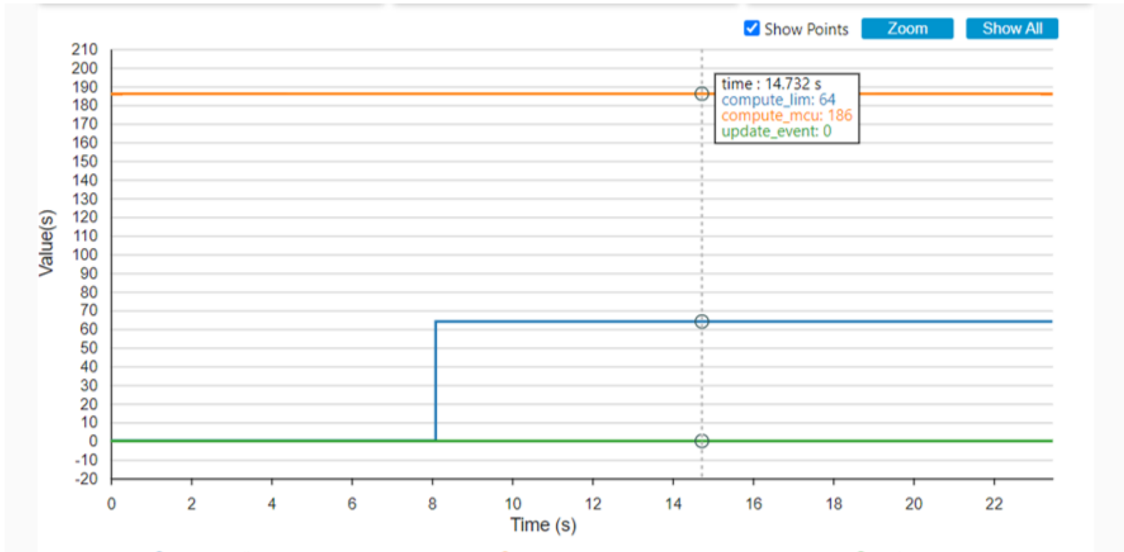


Figure 4.4: Measurement of execution time on ModelSim waves

4.1.2 VirtLAB Oscilloscope

The DSO on the master side of the VirtLAB has a digital trigger given by IO9. This means that the DSO is measured depending on the value of IO9.

As a consequence, IO9 could be linked to the trigger signals of the algorithm to measure, as the same time as power, the actual execution time.

4.1.3 ModelSim

This method is feasible only for the VHDL description of the LiM. This method is interesting for comparing the results of other methods and checking their precision.

ModelSim offers the functionality to observe the waves of the circuit over time. By creating two cursors, it measures the elapsed time between the two. On Figure 4.5, they are placed to measure the computation time: the response time of the LiM with the signal *READY* after sending the trigger signal *ENABLE_COMPUTING*.

Then the read elapsed time could be translated into a number of clock cycles. On the simulation depicted on Figure 4.5, one clock cycle is equal to $1ns$, which means that the LiM took 33.5 clock cycles to compute the algorithm on the given input

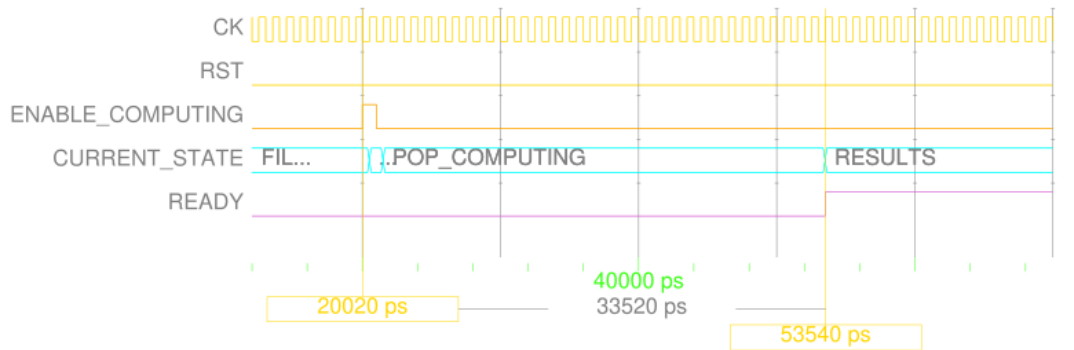


Figure 4.5: Measurement of execution time on ModelSim waves

4.2 Power measurements

For measuring the power, the only explored way is to exploit the DSO embedded on the Master side of the VirtLAB.

They are two options: the GUI and the CLI. The GUI is more intended for students and teaching purposes: in fact, it gives only an overview of what is happening on the user side. On the other hand, the CLI allows more precise measures and automates them by writing some scripts.

4.2.1 VirtLAB DSO: Observable channels

The DSO is able to observe the following channels:

- Channel 1
- Channel 2
- FPGA Icc 3.3V
- FPGA Icc 2.5V
- FPGA Icc 1.2V: interest for this thesis
- MCU Icc 3.3V: interest for this thesis

4.2.2 VirtLAB DSO: Digital trigger IO9

The DSO should use a channel as a trigger to perform its measures.

This trigger could be of type:

- Automatic;
- Normal;
- Single;
- Halted.

In addition, the trigger could be set on rising or falling edge.

The channel that could be used as a trigger input is the same as the ones observable plus a digital trigger: the IO9. This digital trigger permits to directly include within the software the slot in which the measures should be done. In other words, the IO9 must be set at the moment in which the MCU or the FPGA will start the computation to be studied.

4.2.3 VirtLAB: Oscilloscope GUI

Within the Java App of the VirtLAB, a menu is dedicated to the DSO, as shown on Figure 4.6.

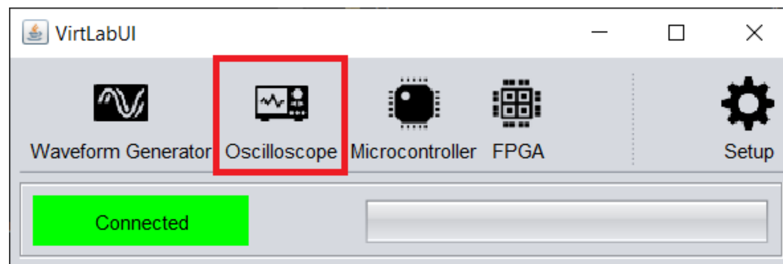


Figure 4.6: DSO menu in the Java Application of the VirtLAB

This menu allows to graphically view the measure done by the DSO, represented on Figure 4.7.

On the bottom part, the signal to be observed could be selected (listed in subsection 4.2.1), by clicking on the associated button **Show**. To deselect it, it is enough to click on the **Hide** button.

On the top right, two parameters are modifiable: the timebase and the *trigger*.

To start the DSO, click on the button **Stopped** in the *trigger* section on the top right.

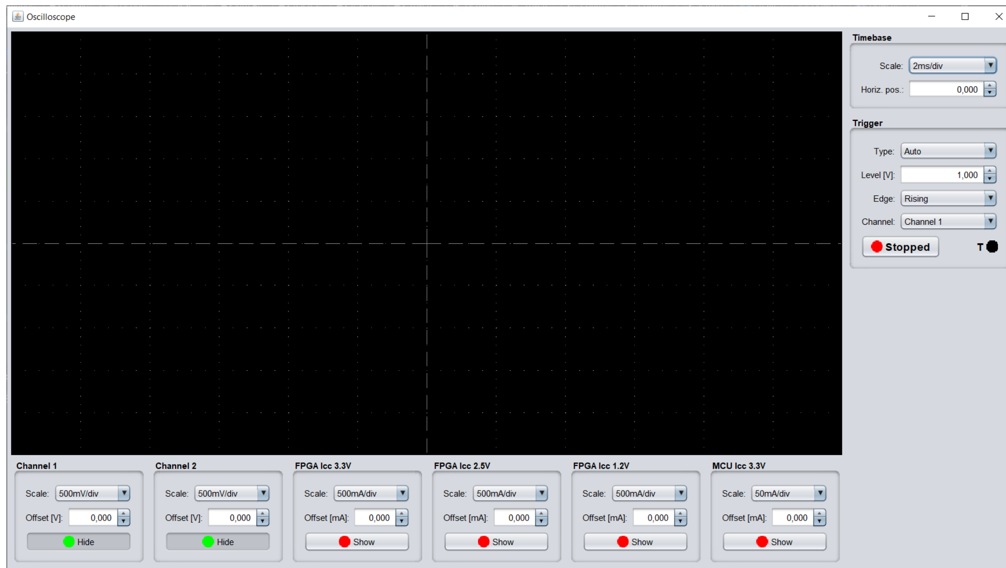


Figure 4.7: Java App: DSO home

4.2.4 VirtLAB: Oscilloscope CLI

The MCU Master also implements a Command Line Interface to interact with the DSO directly. The GUI (subsection 4.2.3) communicates with that CLI to render the provided answers.

Connection

This CLI is accessible by being connected directly on the serial: on Windows by using `putty` (configuration on Figure 4.8) or with `minicom` on Linux.

If the connection is successful, then the prompt `VirtLAB KO>` should appear after pressing any key, like on Figure 4.9. A command would then be submitted (for example `og` on Figure 4.9): the command is correctly processed if the prompt changes to `VirtLAB OK>`.

Commands

The commands available for the DSO are the followings:

- `os <period>`:
  - Set sampling time for DSO inputs
  - `<period>` : sampling time in nanoseconds, hex 32 bit number

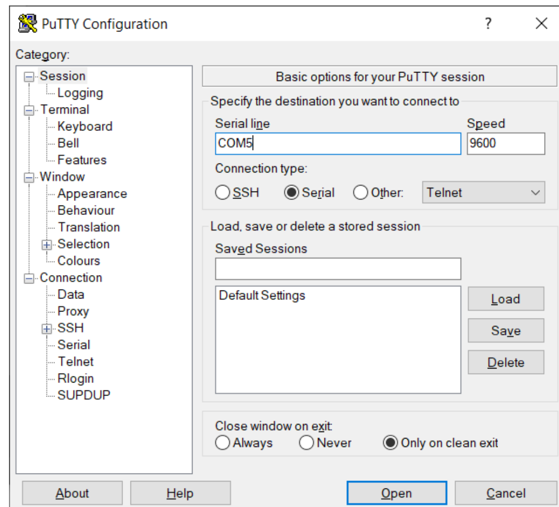


Figure 4.8: DSO CLI: connection with Putty on physical COM port

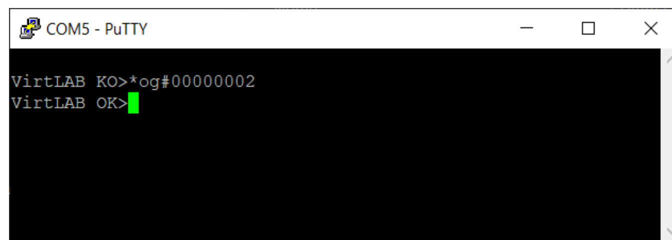


Figure 4.9: DSO CLI: connection successful, command *og* to verify the current status

- `ol <value>`:
  - Set input trigger level
  - `<value>` : trigger level, hex 16 bit number
- `op <edge>`:
  - Set input trigger edge
  - `<edge>` : single character describing the selected edge:
    - `r` : rising edge selected
    - `f` : falling edge selected
- `ot <type>`:
  - Set input trigger type

  - `<type>` : single character describing the selected type:
    - a : automatic
    - n : normal
    - s : single
    - h : halted

- `oi <channel>`:
  - Select channel used as trigger input
  - `<channel>` : ext. trigger (IO9), analog voltage channels (1 to 4), analog current channels (5 to 8)
    - `0` - Ext. digital trigger (IO9)
    - `1` - MSO in 1
    - `2` - MSO in 2
    - `3` - Reserved
    - `4` - Reserved
    - `5` - User FPGA 3.3V supply current (I/O)
    - `6` - User FPGA 2.5V supply current (PLL)
    - `7` - User FPGA 1.2V supply current (Core)
    - `8` - User MCU 3.3V supply current

- `od <channel> <start> <length>`:
  - Get DSO sampled data for selected channel
  - `<channel>` : input channel (1 to 8), hex 8bit number
  - `<start>` : index of first sample to retrieve, hex 16bit number
  - `<length>` : number of samples to retrieve, hex 16bit number

- `og`:
  - Get the DSO status (16 bit word), here the possible values:
    - `0` : stopped
    - `1` : filling
    - `2` : waiting
    - `3` : armed
    - `4` : triggered (dataValid flag)

Python script

A script in python automates the reading of the measures by:

1. Connecting to the serial port;
2. Setting the trigger;
3. For the channels *FPGA IO*, *FPGA Core* and *MCU* does 20 times:
   (a) Sends the command `od`;
   (b) Reads the current values and format them be readable;
   (c) Converts the values from hexadecimal to float (now the current is in `mA`)
   (d) Computes the average of the float values
4. Computes the current average (still in `mA`) of the 20 previous run for each channel;
5. Calculates the associated power (in `mW`).

This script performs averaging on the measured values to avoid as much as possible the impact of noise.

4.3 Area occupation

Quartus is the software used to synthesize the LiM and generate the ELF file to be put on the FPGA. This tool provides some reports on the area occupation of the current design. In particular, two of them are considered and studied:

- Resource Usage Summary: To understand the overall occupation of the design.
- Resource Utilization by Entity: To observe and analyse which part of the design needs more area.

Chapter 5

Results

5.1 Performance result approach

The followings sections state what has been measured to estimate the performance of the LiM XNOR-Net.

For each aspect of performance, different dimensions have been explored: the size of a word in bits and also the depth of the memory. The max dimension explored is the biggest one feasible on the given FPGA: 1Kbyte memory.

This way, the impact of the size is analysed and gives a first idea of the situation in which the LiM could be a good choice.

5.1.1 Current measurements on VirtLAB

What is actually measured

Measures in which we are interested:

- FPGA Core: current for the LiM computation
- FPGA IO: current for the databus
- MCU: current for the protocol communication (FPGA-MCU based) and the MCU-based computation

How the measurements are performed

The python script (described in section 4.2.4), which interacts with the DSO CLI through a serial connection, is the primary source of all the values written

in this thesis.

The FPGA frequency is set to *slowClk*, which is $32KHz$ so that the DSO is able to sample correctly the computation (which has a $2us$ resolution).

Reference: empty FPGA

As a reference, here are the current values of each channel when the MCU and the FPGA firmware are empty, i.e. are doing nothing special:

- FPGA Core: $1.85mA$
- FPGA IO: $1.59mA$
- MCU: $3.38mA$
- FPGA PLL: $5.79mA$

Additional measurements

In Appendix A can be found additional measurements, in particular current measurements in `mA` that are not commented in this part.

5.1.2 Timing measurements

The selected method for the FPGA-MCU based is to use the GPIO Interrupt and the timer TIM2, as explained in section 4.1.1.

On the contrary, for the MCU-based, the timer is used alone as described in section 4.1.1.

5.1.3 Area occupation measurements

Only the default synthesis options of Quartus are used to synthesise the LiM XNOR-Net.

As a reference, the FPGA-User on the VirtLAB has at its maximum 24624 logic elements.

5.2 MCU-FPGA based

The MCU-FPGA implementation requires taking care of both entities: the MCU-User and the FPGA-User.

They both perform some computation, and that is the reason why those components are considered as a whole: the analysis of performance concerned both the MCU and the FPGA.

5.2.1 Area occupation

Figure 5.1 summarizes the occupation area (in percentage) of the LiM on the FPGA, based on the dimensions of the memory in bytes.

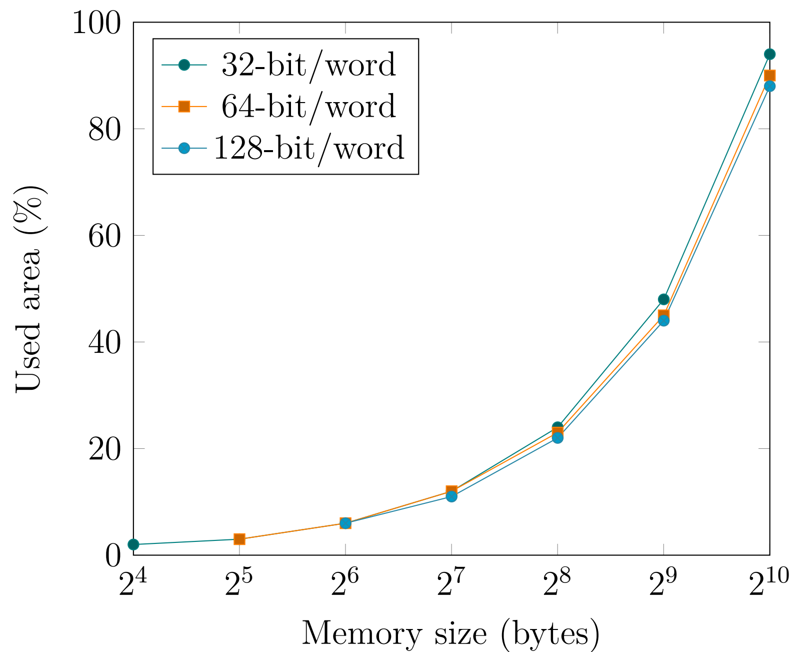


Figure 5.1: Used area by LiM on FPGA

The length of the word does not have much impact on the occupied area: the three curves with 32, 64, or 128 bit per word are almost merged together.

However, the area occupation drastically grows with the increasing memory dimensions.

In addition, the FPGA is already almost **full for 1Kbyte memory**!

5.2.2 Timing

Characteristics

Two frequencies are available for the FPGA-User:

- **slowClk** $= 32KHz$
- **mainClk** $= 10MHz$

However, the measures are taken thanks to the timer *TIM2* at $20MHz$. Those measures are then converted in *clock cycles* to be compared with the MCU-based implementation: indeed, the MCU and the FPGA do not work at the same frequency. Thinking in term of clock cycles make them more comparable.

Execution time: LiM is computing

Table 5.1 states the timing performance for the LiM while it is computing: i.e. from the instant it receives the enable signal of the computation to the moment it responds with the *ready* acknowledge signal.

During the measures, the FPGA was working at $10MHz$. The column *Compute TIM2 (cc)* is the value of the timer, which means the number of clock cycles that passed at a frequency of $20MHz$. *Compute TIM2 (us)* is the corresponding time to this number of cycles.

Bit/word	**#Words**	**Compute TIM2 (cc)**	**Compute TIM2 (us)**	**Compute slowClk (us)**
32	4-256	65	3.25	1015
64	4-128	128	6.4	2000
128	4-64	256	12.8	4000

Table 5.1: Timing performance of the LiM while computing

The LiM is a fully parallel algorithm. Therefore, the computation time does not depend at all on the number of words.

It only depends on the number of bits per word, i.e. the length of a single word: the number of clock cycles corresponds exactly to the operations done in stages *pre_pop_computing* and *pop_computing* of the XNOR-Net Control Unit.

Execution time: MCU is writing

On the contrary, the execution time to fill the LiM completely depends obviously on the number of words to be written.

With the small databus at disposition, the address and the value to be written are transmitted consecutively, one after the other. As a consequence, two consecutive write operations are performed behind writing one value into the LiM.

With the asynchronous protocol, writing one word could not be done in only one clock cycle and could be different each time, depending on some factors:

- The *write_enable* signal is not always set to write multiple values: between each write, it is set to 0 to reset the acknowledged answer of the LiM

- The instant when the FPGA samples

5.2.3 Power

MCU is writing and LiM is computing

#Words	FPGA IO (mW)	FPGA Core (mW)	MCU (mW)	Core + MCU (mW)
4	6.07	2.30	13.27	15.57
8	5.97	2.32	12.64	14.96
16	5.97	2.33	12.08	14.41
32	5.94	2.37	11.65	14.02
64	5.87	2.41	11.45	13.86
128	5.87	2.53	11.29	13.82
256	5.87	2.72	11.32	14.04

Table 5.2: MCU-FPGA: **power consumption** for a **32-bit word** during the *writing LiM + computation* of the LiM XNOR-Net

The current measures (mA) of the different channel while the MCU is writing inside the LiM and the LiM is computing are collected in Table A.4, Table A.5 and Table A.6, classified by the length of the word in bit (32, 64 and 128).

The associated values but for the power (mW) are written in Table 5.2, Table 5.3 and Table 5.4.

Figure 5.2 shows the correlation between the power consumption of the FPGA Core and the size of the memory.

#Words	FPGA IO (mW)	FPGA Core (mW)	MCU (mW)	Core + MCU (mW)
4	6.07	2.30	12.47	14.77
8	6.07	2.32	12.08	14.40
16	5.97	2.33	12.08	14.41
32	5.94	2.41	11.52	13.93
64	5.91	2.52	11.35	13.87
128	5.87	2.75	11.29	14.04

Table 5.3: MCU-FPGA: **power consumption** for a **64-bit word** during the *writing LiM + computation* of the LiM XNOR-Net

#Words	FPGA IO (mW)	FPGA Core (mW)	MCU (mW)	Core + MCU (mW)
4	6.11	2.34	13.20	15.54
8	6.04	2.36	11.65	14.01
16	5.97	2.42	11.48	13.90
32	5.94	2.52	11.35	13.87
64	5.91	2.75	11.39	14.14

Table 5.4: MCU-FPGA: **power consumption** for a **128-bit word** during the *writing LiM + computation* of the LiM XNOR-Net

The number of bits per word does not influence the power. However, from 256 bytes of memory, the power consumption increased, as it was stable for smaller dimensions. This makes sense since the area is growing, and so are the components within the LiM.

On the other hand, Figure 5.3 shows the overall power consumption of the FPGA-MCU implementation by summing the power of the FPGA Core and the MCU.

The LiM in this configuration seems to consume more for little dimensions and starts to stabilize around 14 mW from 256 bytes memories.

LiM is computing

As the execution time of the computation of the LiM has been measured, it is also interesting to do the same for the power: this way, the energy could be calculated as well.

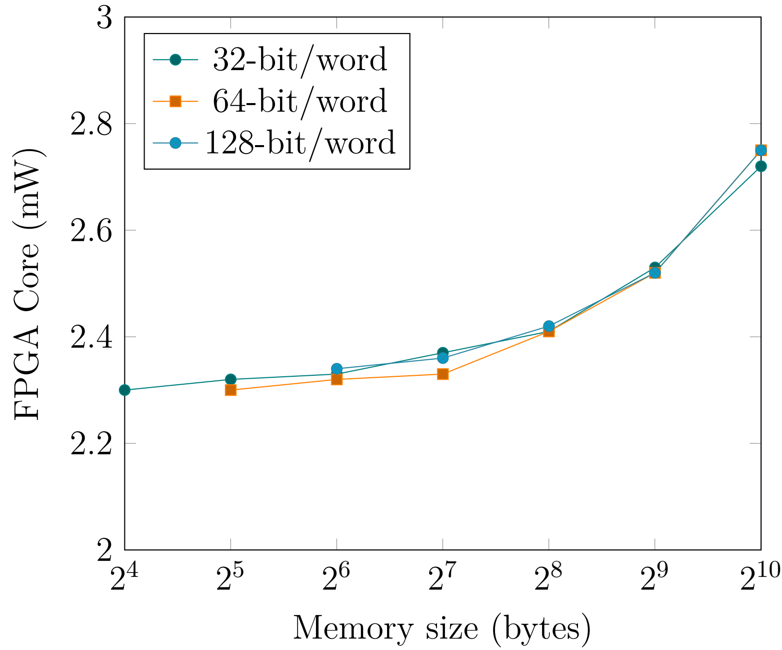


Figure 5.2: Power consumption of the FPGA Core while MCU is writing and LiM is computing

Table 5.5, Table 5.6 and Table 5.7 regroups the power value in `mW` of the channels FPGA IO, FPGA Core and MCU.

#Words	**FPGA IO (mW)**	**FPGA Core (mW)**	**MCU (mW)**	**Core + MCU (mW)**
4	6.59	2.36	47.38	49.74
8	6.60	2.42	47.36	49.78
16	5.96	2.48	23.61	26.09
32	6.28	2.69	35.57	38.26
64	6.28	3.07	35.58	38.65
128	7.60	3.96	82.16	86.12
256	6.91	4.79	58.91	63.70

Table 5.5: MCU-FPGA: **power consumption** for a **32-bit word** during the *computation only* of the LiM XNOR-Net

The value of the MCU channel is much bigger while the FPGA-MCU is computing than when the MCU is writing inside the LiM as well. It is also unstable and does not seem to be linked to memory size.

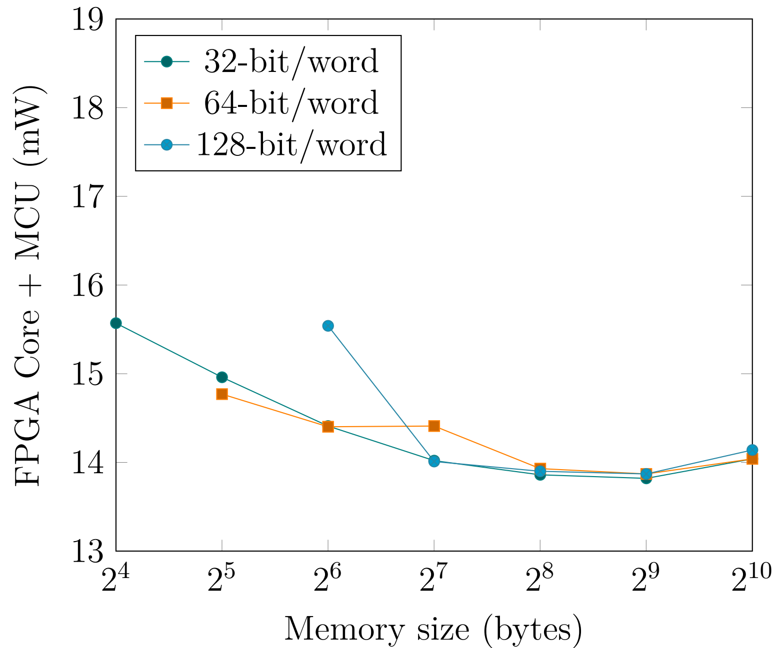


Figure 5.3: Power consumption FPGA Core and the MCU: while MCU is writing and LiM is computing

#Words	FPGA IO (mW)	FPGA Core (mW)	MCU (mW)	Core + MCU (mW)
4	6.91	2.44	58.95	61.39
8	7.01	2.68	70.76	73.44
16	7.25	2.78	70.70	73.48
32	6.91	3.11	59.00	62.11
64	6.91	3.73	59.00	62.73
128	6.59	4.18	47.35	51.53

Table 5.6: MCU-FPGA: **power consumption** for a **64-bit word** during the *computation only* of the LiM XNOR-Net

The same behavior appears for the channel FPGA IO, which is the databus.

These results are quite counter-intuitive. The MCU and FPGA IO channels should hold the same value for a given length of word because the number of words does not influence the execution time of the LiM (and so the FPGA): it should be transparent for the MCU and the databus.

This could be explained by the fact that the MCU is constantly waiting

#Words	FPGA IO (mW)	FPGA Core (mW)	MCU (mW)	Core + MCU (mW)
4	6.90	2.53	58.97	61.50
8	7.22	2.76	70.75	73.51
16	6.78	3.09	59.14	62.23
32	7.25	3.58	70.71	73.33
64	7.21	5.00	70.75	75.75

Table 5.7: MCU-FPGA: **power consumption** for a **128-bit word** during the *computation only* of the LiM XNOR-Net

for the FPGA to answer and does nothing else: the FPGA is so slow that the MCU gets stuck for a long time. This is done by the following code lines:

```
while (HAL_GPIO_ReadPin(GPIOE, IO15_Pin) == GPIO_PIN_RESET) {
        // Do nothing, waiting for ready
}
```

In the case that the HAL function `HAL_GPIO_ReadPin` requires some special computation and resources, it could exploit and stimulate in an stronger way the MCU and the databus.

On the other hand, when the MCU is writing inside the LiM, the communication between the two entities is more efficient and reactive: the FPGA should answer in less than two clock cycles, as slow as it is.

Figure 5.4 gives a graphical view of the power consumption of the FPGA core while it is only computing.

As expected, even if the execution time does not change, the power instead increases with the number of words: more area, so more components in order to process additional words, and that has a consequence on the power.

MCU is resetting the LiM

As a reference, the channels have been measured while the MCU is resetting the LiM for all dimensions.

Here what can be said about thoses values:

- FPGA IO and MCU channels hold the same value than the reference in section 5.1.1:

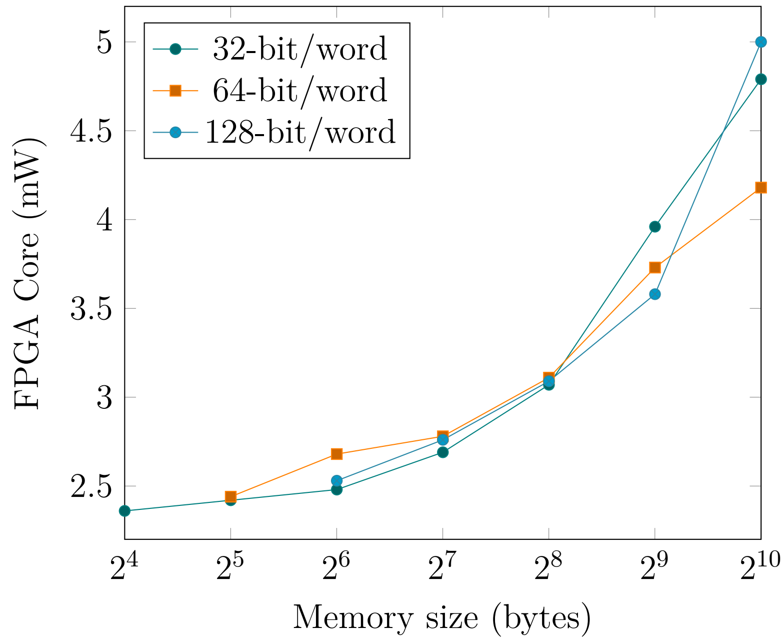


Figure 5.4: Power consumption of the FPGA while computing the LiM XNOR-Net

  – FPGA IO = 1.59mA;

  – MCU = 3.38mA.

- FPGA Core hold more or less the same value when it is resetting or computing.

5.2.4 Energy

The energy is calculated based on the results of the previous sections and is listed in Table 5.8.

Figure 5.5 emphasizes on the flatness of the energy regarding the memory size. However, it strongly depends on the length of the word.

This is due to the execution time that is correlated to the number of bits per word. Thus, for the same memory size, more energy is needed.

#Words	32-bit word Energy (uJ))	64-bit word Energy (uJ))	128-bit word Energy (uJ))
4	0.16	0.39	0.79
8	0.16	0.47	0.94
16	0.08	0.47	0.80
32	0.12	0.40	0.94
64	0.13	0.40	0.97
128	0.28	0.33	*NONE*
256	0.21	*NONE*	*NONE*

Table 5.8: MCU-FPGA: **energy** during the *computation only* of the LiM XNOR-Net

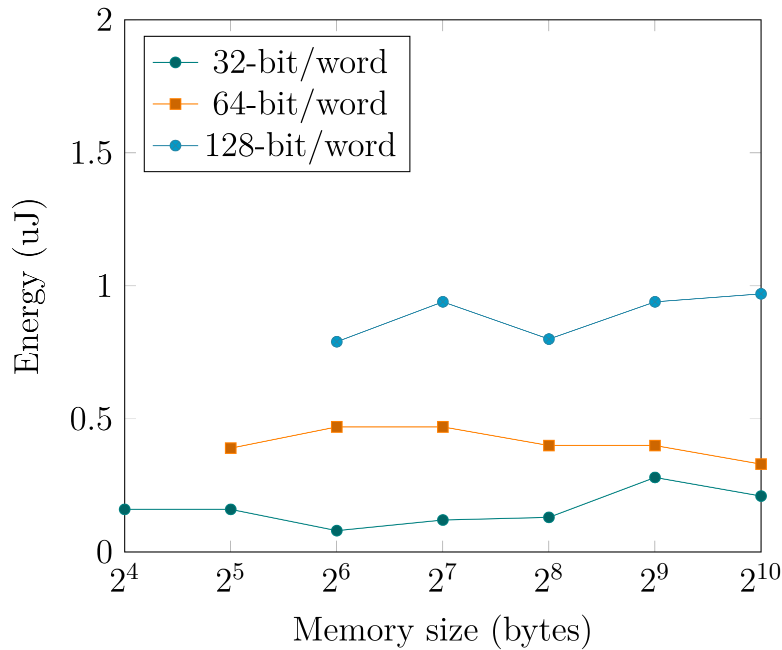


Figure 5.5: Energy of the LiM while computing

5.3 MCU-based

5.3.1 Timing

As for the FPGA-MCU implementation, the measures are done using the TIM2 of frequency $20MHz$, even if the MCU works at $80MHz$.

Table 5.9, Table 5.10 and Table 5.11 collect the results from the measures of TIM2.

The column *Compute TIM2 (cc)* is the value of the counter at the end of the process, so the number of samples / clock cycles that passed. The column *Compute TIM2 (us)* gives the associated execution time.

Bit/word	**#Words**	**Compute TIM2 (cc)**	**Compute TIM2 (us)**
32	4	186	9.3
32	8	337	16.85
32	16	639	31.95
32	32	1243	62.15
32	64	2451	122.55
32	128	4944	247.2
32	256	9698	484.9

Table 5.9: **32-bit word**: Timing performance of the MCU while computing

Bit/word	**#Words**	**Compute TIM2 (cc)**	**Compute TIM2 (us)**
64	4	321	16.05
64	8	607	30.35
64	16	1179	58.95
64	32	2323	116.15
64	64	4689	234.45
64	128	9186	459.3

Table 5.10: **64-bit word**: Timing performance of the MCU while computing

The MCU, on the opposite of the LiM, is sequential. Consequently, the execution depends on the number of bits per word and the number of words. For the same size, the execution time is still similar.

5.3.2 Power consumption

The Tables 5.12, 5.13 and 5.14 collects the measures done on the MCU-based implementation.

The MCU is programmed to execute in an infinite loop the function performing the XNOR-Net behavior. While the MCU is computing this loop, the current value of the MCU is measured.

Bit/word	#Words	Compute TIM2 (cc)	Compute TIM2 (us)
128	4	573	28.65
128	8	1111	55.55
128	16	2187	109.35
128	32	4417	220.85
128	64	8719	435.95

Table 5.11: **128-bit word**: Timing performance of the MCU while computing

Bit/word	#Words	MCU (mA)	MCU (mW)
32	4	3.34	11.02
32	8	3.30	10.89
32	16	3.28	10.82
32	32	3.27	10.79
32	64	3.26	10.76
32	128	3.26	10.76
32	256	3.26	10.76

Table 5.12: MCU: current and power measures for a **32-bit word** during the *write + computation* of the XNOR-Net

Figure 5.6 sums up the results from the Table 5.12, 5.13 and 5.14.

The Figure 5.6 highlights the flatness of the power consumption. To be comparable with the LiM, the exact same dimensions are tested. However, 1kbytes is relatively minor for the computing power of a MCU.

In other words, the dimensions are not huge enough to have a real impact on the power consumption of the MCU and to make conclusions.

5.3.3 Energy

Taking the results from the previous section, Table 5.15 sums up the energy used for the given dimensions.

The MCU working at a high frequency like $80MHz$ shows how low energy it needs. However, the energy needed increases considerably with the memory size.

This is due to the complexity of the sequential algorithm that takes care of every single value, one by one.

Bit/word	#Words	MCU (mA)	MCU (mW)
64	4	3.26	10.76
64	8	3.26	10.76
64	16	3.25	10.73
64	32	3.24	10.69
64	64	3.24	10.69
64	128	3.25	10.73

Table 5.13: MCU: current measures for a **64-bit word** during the *computation* of the XNOR-Net

Bit/word	#Words	MCU (mA)	MCU (mW)
128	4	3.26	10.76
128	8	3.24	10.69
128	16	3.23	10.66
128	32	3.23	10.66
128	64	3.24	10.69

Table 5.14: MCU: current measures for a **128-bit word** during the *computation* of the XNOR-Net

5.4 Comparison

5.4.1 Execution time

The speedup of the LiM over the MCU is shown on Figure 5.8. The MCU working at $80MHz$ and the FPGA at $10MHz$, the LiM allows a huge speedup over the MCU alone.

The speedup is given by the number of words because it emphasises on the parallel aspect of the LiM that the MCU does not have. Indeed, with only 256 words of 32-bit, the speedup is about 150.

The LiM takes advantage when it holds a really deep memory.

Nevertheless, the LiM is eight times slower than the MCU, but it still is drastically faster than the MCU, even for really small memories (less than 1kbytes).

With the clock of the FPGA sets at $32KHz$, there is no speedup: the MCU execution time is always lower than the one of the LiM.

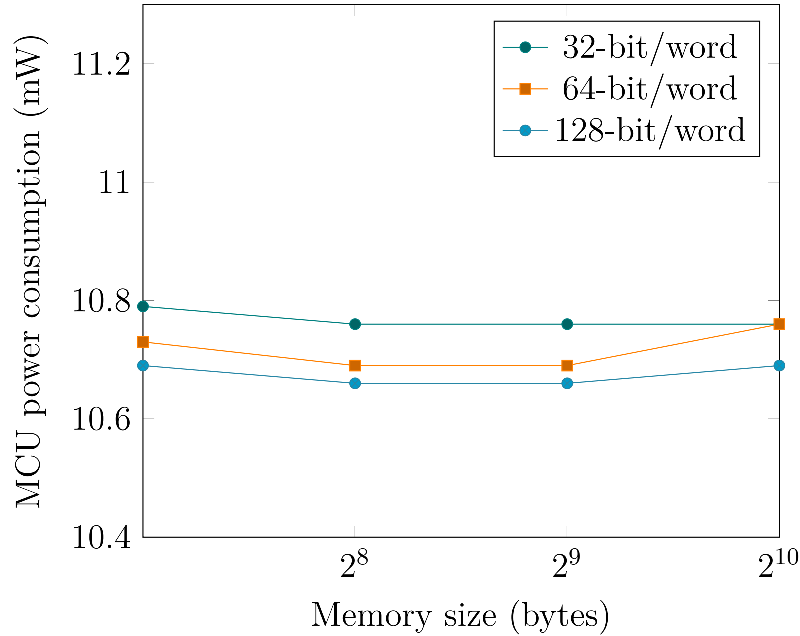


Figure 5.6: Power consumption of the MCU while computing the XNOR-Net

#Words	**32-bit word Energy (uJ))**	**64-bit word Energy (uJ))**	**128-bit word Energy (uJ))**
4	0.10	0.17	0.31
8	0.18	0.33	0.59
16	0.35	0.63	1.17
32	0.67	1.24	2.35
64	1.32	2.51	4.66
128	2.66	4.93	*NONE*
256	5.22	*NONE*	*NONE*

Table 5.15: MCU: **energy** during the *computation*

This shows how crucial it is to choose the correct frequency for the FPGA.

5.4.2 Energy

Figure 5.9 demonstrates the energy saved by the LiM over the MCU, by memory sizes.

The LiM, even with an FPGA working at 10MHz, is able to save energy for almost all dimensions of memories, being compared to a MCU working

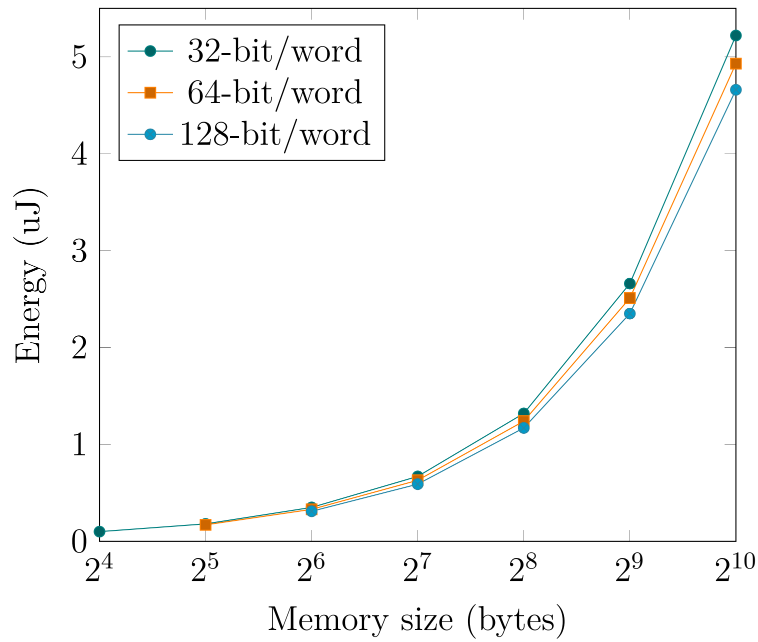


Figure 5.7: Energy of the MCU while computing

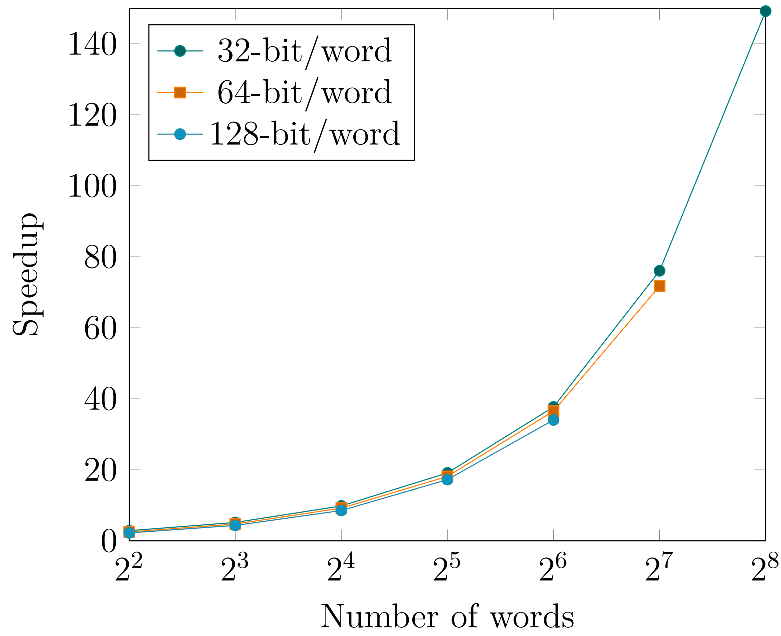


Figure 5.8: Timing performance comparison LiM and MCU: with FPGA at 10MHz

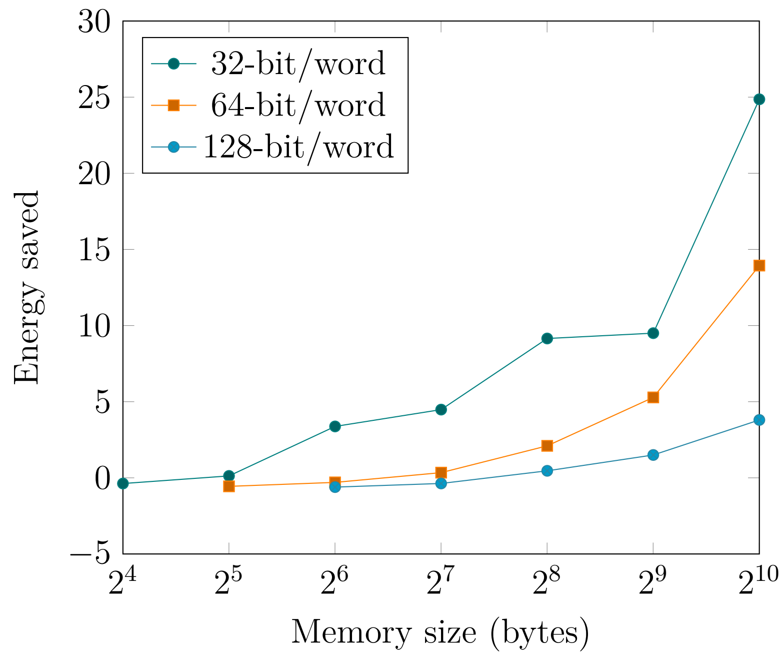


Figure 5.9: Energy performance: how much the LiM saves on MCU

at 80MHz.

The LiM is more energy saving when the size of a word is smaller: as explained in subsection 5.2.4, the execution time is linked to the number of bit per word and therefore increase the energy.

Chapter 6

Conclusions and Future works

This thesis aims at implementing on FPGA an architecture Logic-In-Memory so that to estimate the impact of this paradigm on a real board. The chosen application is the XNOR-Net, a BCNN, which is an embarrassingly parallel algorithm.

The used board for those implementations is the VirtLAB and has also been exploited for measuring the performance of the XNOR-Net on the board.

Three implementations have been done, from which two have been strongly studied with a deep analysis of performance: the MCU-based and the FPGA-MCU based. The first one implements the sequential behavior of the XNOR-Net. The second one takes advantage of the LiM: the FPGA is used as a co-processor to the MCU.

The results demonstrate that the LiM is drastically more efficient than the MCU in terms of energy and execution time, even with small memories. The advantages are even more substantial with a deep memory and a small word length (i.e. number of bits per word). Those results are even seen nevertheless how slow the FPGA in comparison to the MCU: the FPGA is working at 10MHz and the MCU 80MHz. This means that the impact of the LiM is visible even being eight times slower than the MCU.

Unfortunately, these speedup and saved energy are at the cost of area occupation: the FPGA is full with a 1kbytes memory.

As a future work, the implementation of other types of LiM on FPGA to continue exploring the impact of this paradigm and find a way to reduce the

area occupation.

Bibliography

[1] A. Coluccio and M. Vacca and G. Turvani, *Logic-in-Memory Computation: Is It Worth It? A Binary Neural Network Case Study*, Journal of a Low Power Electronics and Applications, 2020.

[2] Andrea Coluccio, *Master's Thesis: In-Memory Binary Neural Networks*, 10 April 2019

[3] Prof. Massimo Ruo Roch - DET Politenico di Torino, *VirtLab schematics*, 31 October 2020

[4] VLSI - Politecnico di Torino, *VirtLab 1.2 Overview*, 6 April 2021

[5] ST Microelectronics, *STM32L4x5 and STM32L4x6 advanced Arm®-based 32-bit MCUs*, Reference Manual RM0351 - Rev 6 - April 2018

[6] ST Microelectronics, *Description of STM32F4 HAL and low-layer drivers*, User Manual UM1725 - Rev 7 - June 2021

[7] ST Microelectronics, *Description of STM32L4/L4+ HAL and low-layer drivers*, User Manual UM1884 - Rev 9 - September 2021

[8] Prof. M. Zamboni, Prof. M. Martina, Dr. G. Turvani, ing. Y. Ardesi, ing. G.A. Cirillo, ing. A.Coluccio, Dott. U. Garlando, *Laboratori 7-8-9-10 di Elettronica dei Sistemi Digitali*, Politecnico di Torino

[9] *How to count cycles on ARM Cortex M*, http://embeddedb.blogspot.com/2013/10/how-to-count-cycles-on-arm-cortex-m.html

[10] *Measuring code execution time on ARM Cortex-M MCUs*, https://embeddedcomputing.com/technology/processing/measuring-code-execution-time-on-arm-cortex-m-mcus

[11] *Sources for the VirtLab given by prof. M. Ruo Roch*, https://www.dropbox.com/sh/wiq0b8y306ae3b2/AACzBp05XYW7v3jnoo3dLFZTa?dl=0

[12] Massimo Ruo Roch and Maurizio Martina, *VirtLAB: A Low-Cost Platform for Electronics Lab Experiments*, Sensors 2022

[13] Coralie Allioux, *Sources of the thesis: implementations, scripts and results*, https://git.vlsilab.polito.it/coralie.allioux/xnor-net,

2021-2022

Appendix A

Additional measures

A.1 MCU-FPGA based: FPGA a 32KHz

A.1.1 LiM is computing

#Words	FPGA IO (mW)	FPGA Core (mW)	MCU (mW)	Core + MCU (mW)
4	6.63	2.18	47.53	49.70
8	6.63	2.20	47.52	49.72
16	5.99	2.18	23.68	25.85
32	6.30	2.22	35.66	37.88
64	6.30	2.45	35.65	37.92
128	7.63	2.53	82.47	85.00
256	6.95	2.65	59.01	61.66

Table A.1: MCU-FPGA: **power consumption** for a **32-bit word** during the *computation only* of the LiM XNOR-Net

A.1.2 MCU is writing and LiM is computing

A.1.3 Energy with FPGA at 32KHz

#Words	FPGA IO (mW)	FPGA Core (mW)	MCU (mW)	Core + MCU (mW)
4	6.94	2.24	59.12	61.36
8	7.27	2.28	70.86	73.14
16	7.28	2.31	70.88	73.19
32	6.93	2.35	59.15	61.50
64	6.93	2.45	59.13	61.58
128	6.61	2.64	47.46	50.10

Table A.2: MCU-FPGA: **power consumption** for a **64-bit word** during the *computation only* of the LiM XNOR-Net

#Words	FPGA IO (mW)	FPGA Core (mW)	MCU (mW)	Core + MCU (mW)
4	6.93	2.27	59.12	61.37
8	7.24	2.32	70.91	73.23
16	6.91	2.36	59.29	61.65
32	7.28	2.46	70.87	73.33
64	7.24	2.71	11.39	70.92

Table A.3: MCU-FPGA: **power consumption** for a **128-bit word** during the *computation only* of the LiM XNOR-Net

Bit/word	#Words	FPGA IO (mA)	FPGA Core (mA)	MCU (mA)
32	4	1.84	1.92	4.02
32	8	1.81	1.93	3.83
32	16	1.81	1.94	3.66
32	32	1.80	1.97	3.53
32	64	1.78	2.01	3.47
32	128	1.78	2.11	3.42
32	256	1.78	2.27	3.43

Table A.4: MCU-FPGA: current measures for a **32-bit word** during the *writing LiM + computation* of the LiM XNOR-Net

Bit/word	#Words	FPGA IO (mA)	FPGA Core (mA)	MCU (mA)
64	4	1.84	1.92	3.78
64	8	1.84	1.93	3.66
64	16	1.81	1.94	3.66
64	32	1.80	2.01	3.49
64	64	1.79	2.10	3.44
64	128	1.78	2.29	3.42

Table A.5: MCU-FPGA: current measures for a **64-bit word** during the *writing LiM + computation* of the LiM XNOR-Net

Bit/word	#Words	FPGA IO (mA)	FPGA Core (mA)	MCU (mA)
128	4	1.85	1.95	4.00
128	8	1.83	1.97	3.53
128	16	1.81	2.02	3.48
128	32	1.80	2.10	3.44
128	64	1.79	2.29	3.45

Table A.6: MCU-FPGA: current measures for a **128-bit word** during the *writing LiM + computation* of the LiM XNOR-Net

#Words	32-bit word Energy (uJ))	64-bit word Energy (uJ))	128-bit word Energy (uJ))
4	50.45	122.72	245.48
8	50.47	146.28	292.92
16	26.24	146.38	246.60
32	38.45	123.00	293.32
64	38.49	123.16	283.68
128	86.28	100.20	*NONE*
256	62.58	*NONE*	*NONE*

Table A.7: MCU-FPGA: **energy** during the *computation only* of the LiM XNOR-Net

Appendix B

Tutorial VirtLab

Tutorial: Setting up and using VirtLab

Coralie Allioux
matricola 287268

July 14, 2022

B.1 Introduction

This tutorial aims at explaining **how to set up** the environment in order to use the VirtLab, which means: **how to connect** to the VirtLab and **how to program** it.

To do so, some software tools are needed in order to communicate with the VirtLab. When those communication ways are operational, programming the VirtLab becomes possible.

The main sources of the VirtLab are available on the following link: `https://www.dropbox.com/sh/wiq0b8y306ae3b2/AACzBp05XYW7v3jnoo3dLFZTa?dl=0`.

This tutorial has been tested on and is targeting *Windows 10/11* and *GNU/Linux*.

B.2 Connecting the VirtLab to a PC

The VirtLab needs to be connected to the PC, as shown on Figure B.1, via two cables:

- via microUSB (top left)
- and USB-C (bottom left)

The interrupter for turning ON/OFF the board is in red. To be more precise, on the picture, the board is actually connected to the PC with the interrupter turned OFF.

Note: *Those two cables uses an USB or USB-C port of the PC, depending on the PC characteristics.*

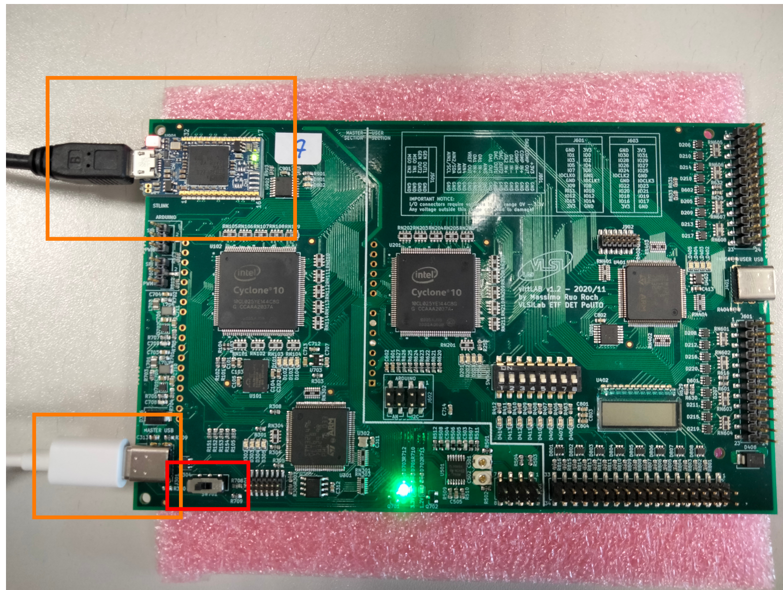


Figure B.1: VirtLab board conected to a PC

B.3 Needed software

This section exposes the list of needed software, in order to properly use the VirtLab. As a consequence, those programs must be installed on the PC physically connected to the VirtLab.

B.3.1 General overview: Interaction between software and VirtLab

The general schematic of the communication between software and the VirtLab is depicted on Figure B.2. The board is connected via two cables, as seen previously in section B.2. This schematic emphasizes on what are the purposes of those ports, which software tools are needed and how they interact together. A MCU ST-LINK programmer is given (top left), connected via microUSB. This programmer can download the compiled code (*ELF file*) on one of the MCUs. The ELF file can be created as explained in subsection B.3.6, using ST-Cube IDE, and can be downloaded on the MCU with the ST-Cube Programmer (subsection B.3.5). On the other hand, to program the FPGAs, the USB-C port is used to download a *RBF file*, which can be created with Quartus Prime (subsection B.3.7).

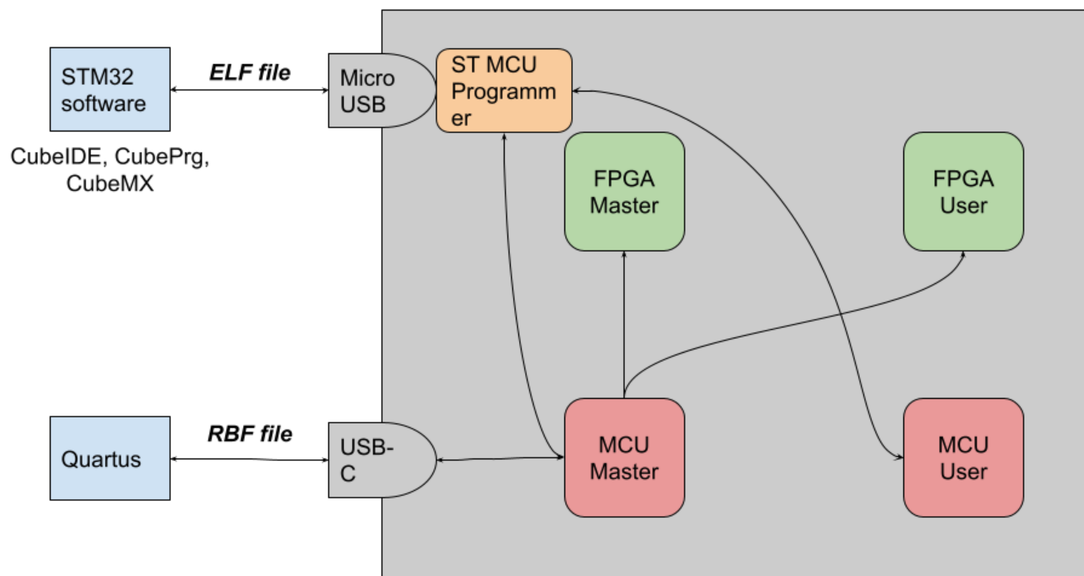


Figure B.2: Interaction between software and VirtLab

B.3.2 Stlinkbridge

`Stlinkbridge` is a `C++` application which uses the ST's API in order to recognize and connect to the MCU programmer. Since the VirtLab contains two MCUs, one of them needs to be chosen and specified to be programmed. The job of that application is allowing the user to select the desired MCU.

The sources (executable and library file) are available on the Dropbox folder:
`virtLAB/Software/stlinkbridge/`. Choose the subfolder corresponding to your operating system: `linux/` or `win64/`.

The following two commands are available to select the desired MCU:

```
stlinkbridge m        #choose Master MCU
stlinkbridge u        #choose User MCU
```

The VirtLab GUI (subsection B.3.4) uses those commands: hence, the `stlinkbridge` command must be accessible without having to specify the path to the `Stlinkbridge` application. The procedure to do so is explained in section B.4.

B.3.3 JDK 17

The JDK contains tools for developing and running Java programs. It is necessary for running the VirtLab GUI (subsection B.3.4).

If already installed, you can check the current version with the following command:

```
java -version
```

Otherwise, go to `https://www.oracle.com/java/technologies/downloads/`, download the files for your operating system and install the development kit. Then, check the current version running the `java -version` command.

Note: *The JDK 8 is obsolete for this setup. Not all version were tested. However, it is well known that the JDK 17 fits: hence, JDK 17 is the recommended version.*

B.3.4 VirtLab GUI

VirtLab GUI is a Graphical User Interface, located on the Dropbox folder (`dist/VirtLabUI-v1.0.jar`, unzipping `virtLAB/Software/VirtLabUI-v1.0.zip`).

This software is used for communicating with the VirtLab. It allows one to choose which MCU to program and to program the FPGAs.

It needs Stlinkbridge (called by the GUI) and JDK 17 (executes the GUI).

It can be launched on the command line by using JDK, with the following command:

```
java -jar VirtLabUI-v1.0.jar
```

Note: *This interface aims at simplifying the interaction with the VirtLab, even if in theory, it is not strictly necessary.*

B.3.5 STM32CubeProg

This ST's program is used for downloading the ELF file (created by Cube IDE) on the MCU. You only need to use STM32CubeProg to initialize the VirtLab by programming the Master MCU (the first time you use it or if a firmware update is released) because only the ELF file is provided. If the Master MCU already contains the firmware, you do not have to perform this step.

When you develop your own code, you can directly and *only* use STM32CubeIDE for *coding*, *creating the ELF file* and *downloading it* on the MCU.

The download link can be found bellow:
`https://www.st.com/en/development-tools/stm32cubeprog.html#overview`

Note: *You must register to download that software.*

B.3.6 STM32CubeIDE

This ST's software IDE allows one to program and compile some pieces of code, in order to create an ELF file, which will be then downloaded on the MCU.

The download link can be found bellow:
`https://www.st.com/en/development-tools/stm32cubeide.html`

Note: *No need for a particular tutorial: an ELF example file is given within the sources.*

B.3.7 Quartus

Quartus Prime is a software tool used to program the FPGA, *i.e.* compiling the VHDL design files and creating the associated `row binary file` (RBF) to be downloaded to the FPGA. This RBF file can be downloaded to the VirtLab through the VirtLab GUI.

Quartus Prime can be retrieved at the following download link: `https://fpgasoftware.intel.com/?edition=lite`
We suggest you to download the following individual files, according to your operating system:

- Quartus Prime Lite Edition (Free): Quartus Prime (includes Nios II EDS)
- Devices: Cyclone 10 LP device support

Once Quartus Prime has been successfully installed, the recommended procedure is to exploit the project available in the Dropbbox folder `virtLAB/Hardware/FPGA-templates/fpga-user-template` and modify (*i.e.* include the top entity of your design and do the proper port map) the `fpga-user.vhd` file. In this case, the target FPGA selection (family `Cyclone 10 LP`, device `10CL025YE144C8G`) and the pin assignment are already done. Once the project is compiled, an RBF file is automatically generated and can be downloaded to the FPGA. If the RBF is not generated click on `Assignments` ⟶ `Device` ⟶ `Device and Pin Options` ⟶ `Programming Files` and check `Raw Binary File (.rbf)`, as shown in Figure B.3. Then click on `Ok`.

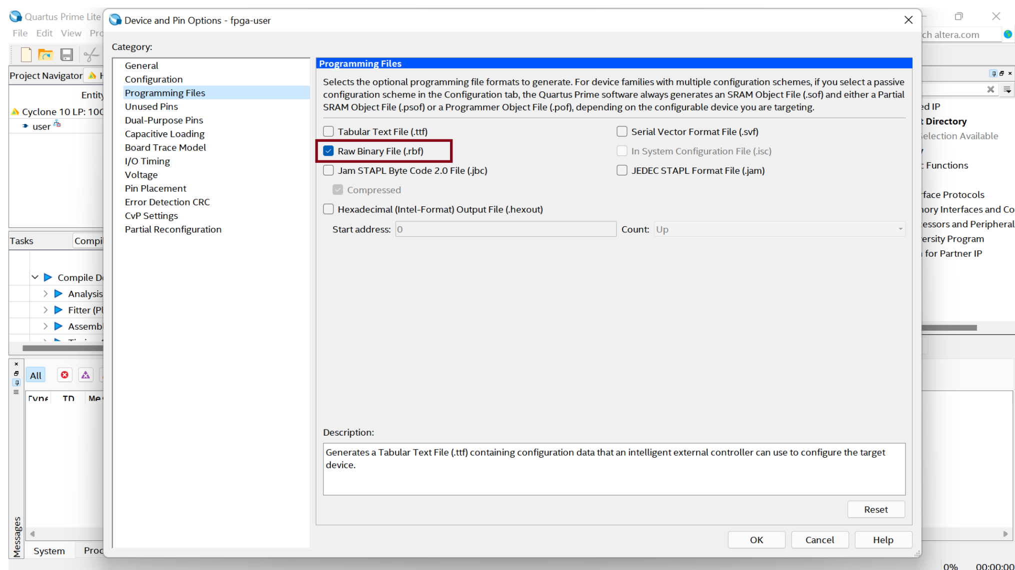


Figure B.3: Selection of RBF output file.

B.4 Setting up the environment

B.4.1 For Windows 10/11

As explained in the subsection B.3.2, the `Stlinkbridge` program should be run by the command line. To do so, it needs to be added in the environment variable called `PATH`.

Note: *In this tutorial, the language of the PC is set to Italian. In order to cover more languages, the text contains the English version.*

- STEP 1: Search for the environment variable control panel: Search for the control panel **Edit the system environment variables** through the Windows search (by typing *env* if English or *ambi* if Italian)
- STEP 2: Click on *Environment variables...*
- STEP 3: Select *Path* in *System Variables* and click *Modify*
- STEP 4: Create a new entry by clicking on *New*
- STEP 5: Click on *Browse...* and select the directory where *STLinkbridge* program (executable and library files) is
- STEP 6: Done! Dismiss all windows by clicking on *OK* (three times)

Note: *To verify if the* `Path` *variable is updated correctly, run the* `stlinkbridge` *command within the command line.*

B.4.2 For GNU/Linux

Alias for VirtLab GUI

This alias is recommended for using the GUI in a smoother way: through the command line, using a direct command like `VirtLab`. To do so, modify the file `~/.bashrc` by adding the following line:

```
alias VirtLab='java -jar DirectPath/VirtLabUI-v1.0.jar'
```

Note: *DO CHANGE* `DirectPath` *with the precise location of the program on your PC.*

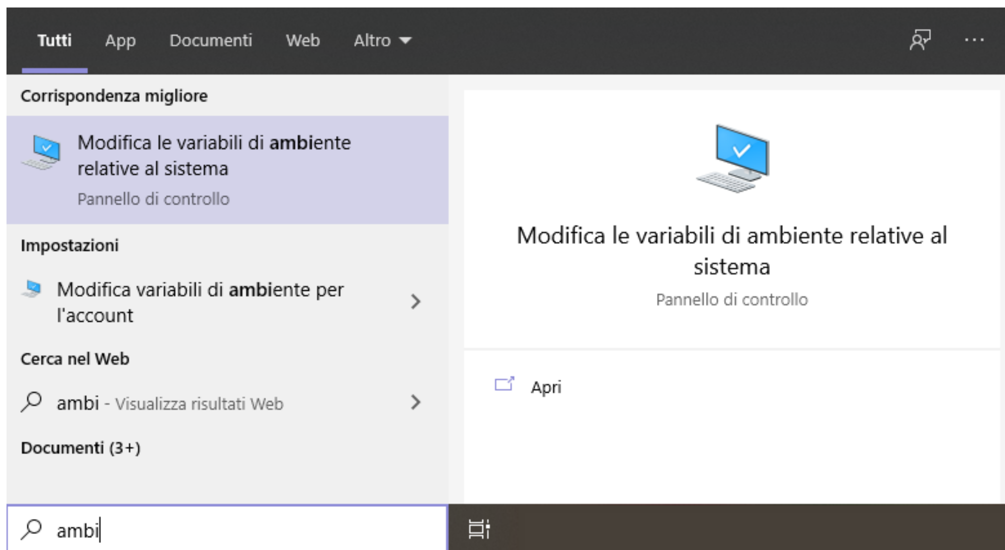


Figure B.4: STEP 1: Search for the environment variable control panel

Stlinkbridge configuration

Open the ~/.bashrc and add the following lines:

```
# Updating the PATH to use the direct command "stlinkbridge"
PATH="$PATH:/path/to/stlinkbridge/executable/"

# Updating the location of the needed libraries
export LD_LIBRARY_PATH="/path/to/stlinkbridge/library/"
```

The paths of the library and stlinkbridge executable depends on where you put the virtLAB Dropbox folder. Let's assume, for instance, that the virtLAB folder is located in /home/virtLAB/. The previous lines become:

```
# Updating the PATH to use the direct command "stlinkbridge"
PATH="$PATH:/home/virtLAB/Software/stlinkbridge/"

# Updating the location of the needed libraries
export LD_LIBRARY_PATH="/home/virtLAB/Software/stlinkbridge/"
```

To take into account those changes, execute the following command:

```
source ~/.bashrc
```

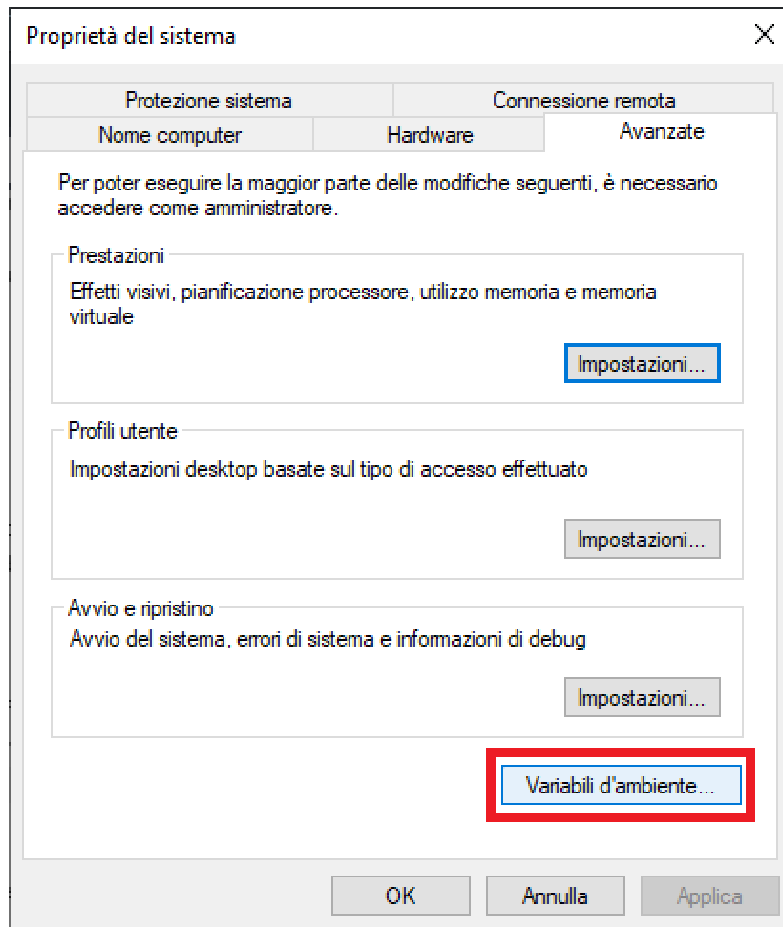


Figure B.5: STEP 2: Click on *Environment variables...*

Note: *«~» before* `/.bashrc` *is a shortcut for specifying your home. Therefore, you can use that command independently on your current location within the file tree, known with* `pwd`.

Update user groups

In order to execute everything, some permissions are needed that are resolved by adding some groups to the current user.

To do so, execute the following command:

```
sudo usermod -a -G dialout,plugdev $USERNAME
```

Reboot your PC.

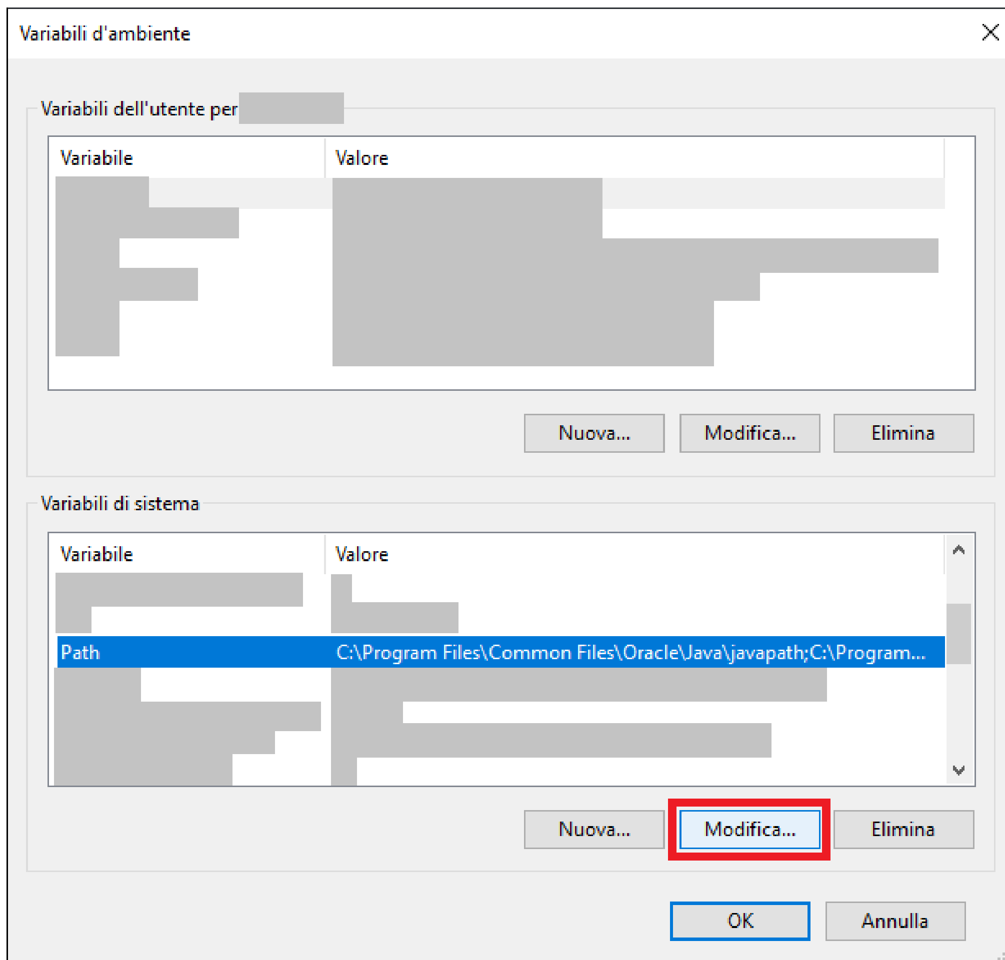


Figure B.6: STEP 3: Select *Path* in *System Variables* and click *Modify*

Note: *You can verifying if everything went well with the* `id` *command: observe for the groups* `plugdev` *and* `dialout` *after the reboot.*

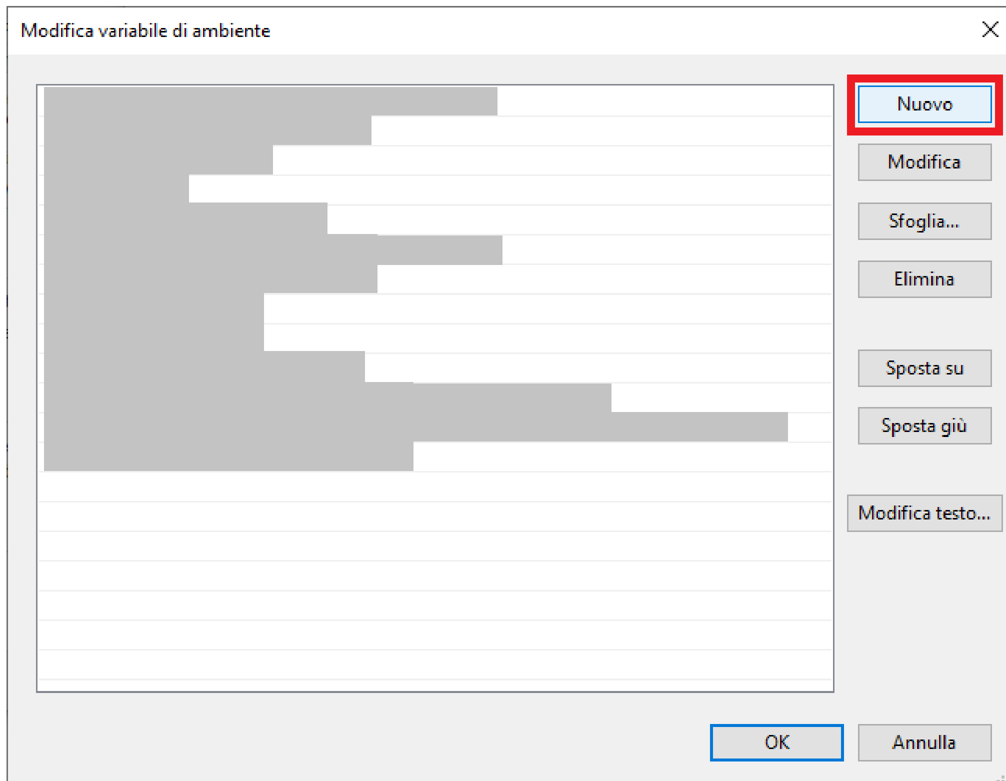


Figure B.7: STEP 4: Create a new entry by clicking on *New*

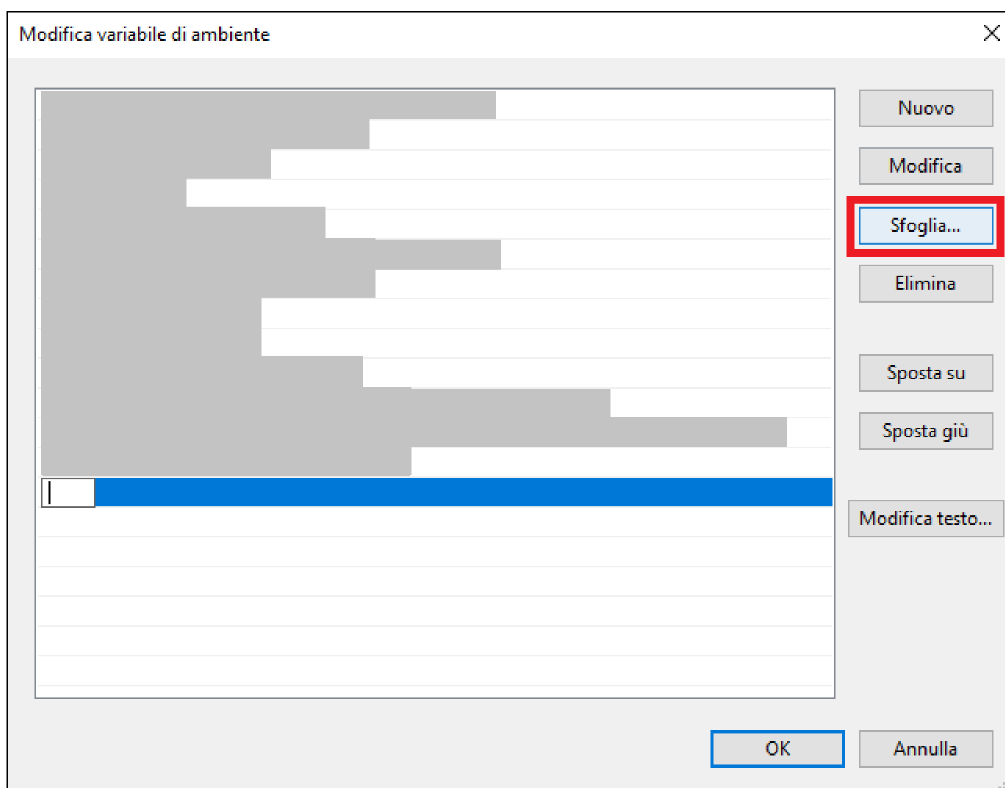


Figure B.8: STEP 5: Click on *Browse...* and select the directory where *STLinkbridge* program (executable and library files) is

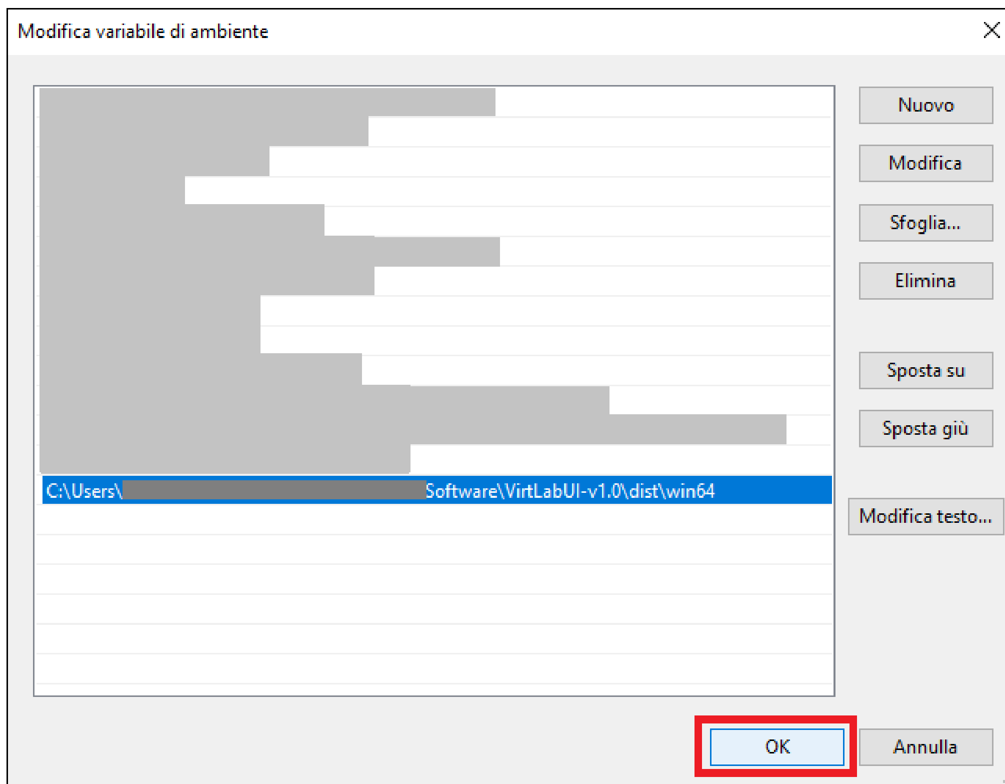


Figure B.9: STEP 6: Done! Dismiss all windows by clicking on *OK* (three times)

B.5 First use of VirtLab

For the first use of the VirtLab, neither the FPGAs nor MCU are programmed. Once the firmware available on Dropbox `virtLAB/Firmware/virtlab-master/virtlab-master.elf` is downloaded to the master MCU, one can program the FPGAs (master and user) through the VirtLab GUI. This section explains how to download this firmware on the MCU master.

- STEP 1: Connect the VirtLab to the PC via USB-C and microUSB, as described in section B.2.
- STEP 2: Switch on the board by turning on the interrupter at the bottom left of the board.
- STEP 3: Launch the Java application (by using the alias created before in section B.4.2 ; or on Windows by double clicking on the program file).
- STEP 4: Go to MCU tab
- STEP 5: Select *Master* and click *Apply*. The LED associated with the MCUs is now orange, as shown in Figure B.12. It confirms that the MCU-Master is the one selected.
- STEP 6: Open STM32CubeProg. Connect to the ST's programmer present on the board by clicking *Connect*. The top left LED of the Virtlab is now blinking Green/Red, which means that the programmer is well connected.
- STEP 7: Click on *Open file* and choose the firmware called *virtlab-master.elf*. Current code on the MCU is depicted in the middle.
- STEP 8: Click on *Download* to upload on the MCU the file previously selected.. Dismiss the successful pop-up by clicking `OK` and disconnect by clicking on `Disconnect`. The top-left LED is now fixed to Red.
- STEP 10: Switch off and then switch on the board (same interrupter as in step 2) to apply the changes. The uploaded firmware turns on the MCU-Master LEDs from right to left and vice versa (visual confirmation). Done!

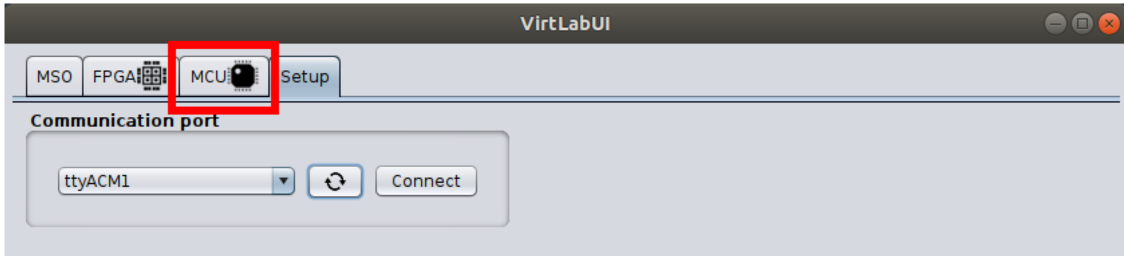


Figure B.10: STEP 4: Go to MCU tab

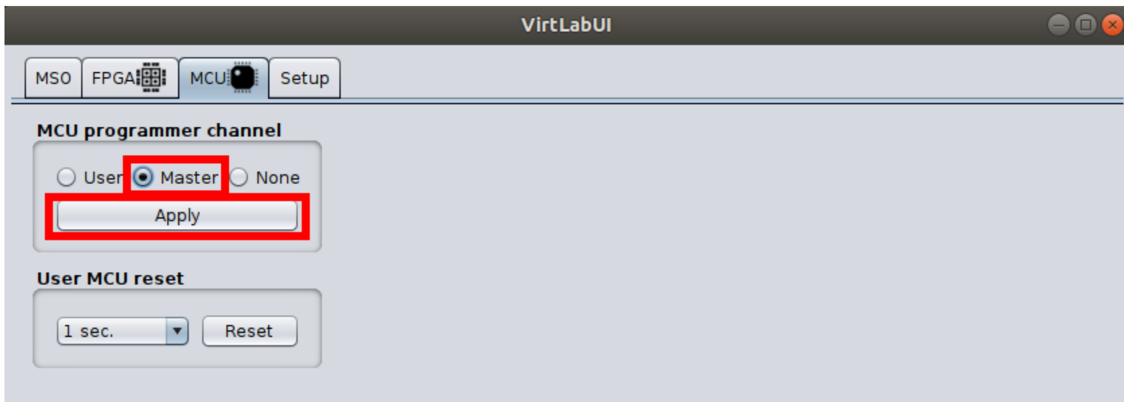


Figure B.11: STEP 5: Select *Master* and click *Apply*

With this firmware on the MCU master, you are now able to program the FPGA. Indeed, it implements the links between the MCU master and the FPGAs shown on Figure B.2.

Note: *Since that firmware is written on the Flash memory of the MCU, this step needs to be executed only the first time you use the VirtLab (or if a master MCU firmware update is released). The next time the board is shutting down and then turning on, the firmware is automatically loaded and executed.*

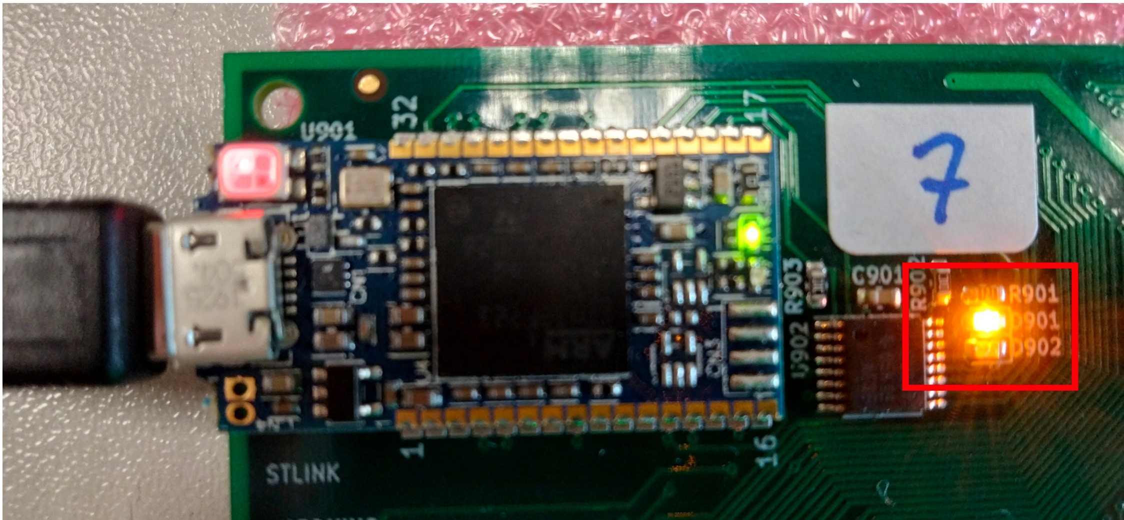


Figure B.12: VirtLab: MCU Master selected, ORANGE LED

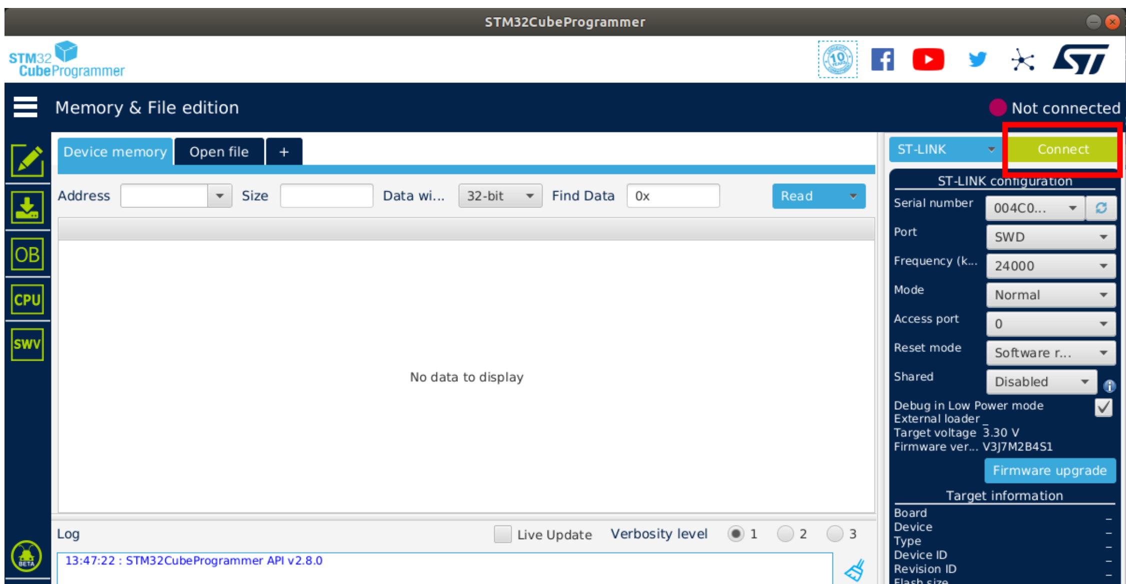


Figure B.13: Connect to the ST's programmer present on the board by clicking *Connect*

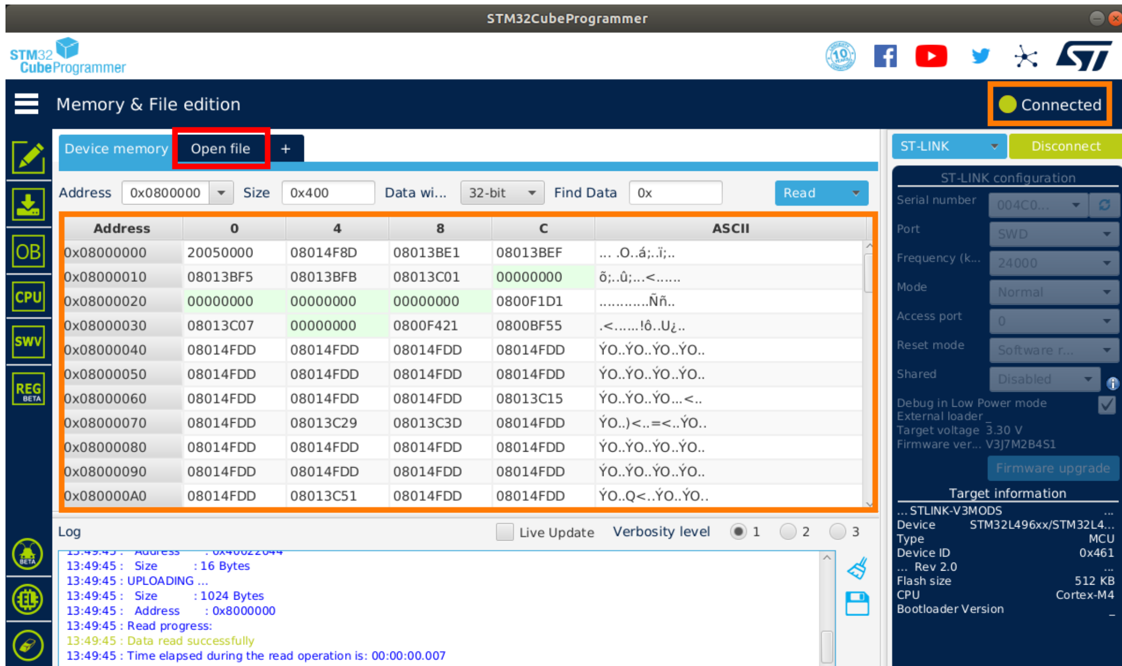


Figure B.14: STEP 7: Click on *Open file* and choose the firmware called *virtlab-master.elf*. Current code on the MCU is depicted in the middle.

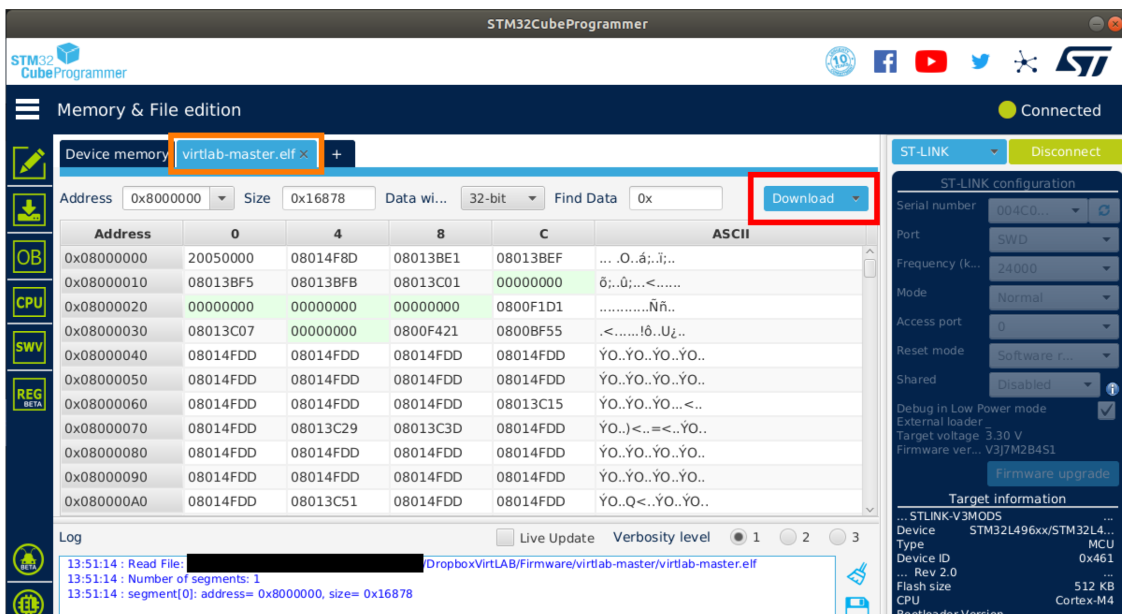


Figure B.15: STEP 8: Click on *Download* to upload on the MCU the file previously selected.

B.6 Program on VirtLab

You will need to use the MCU user and FPGA user for your laboratory experiences, whereas the FPGA master shall be reserved for specific applications. This section is devolved to explaining how to program these devices.

B.6.1 Program MCU User

The procedure is the same as the one explained for programming the MCU master in the section section B.5. You can use the given example file: `virtLAB/Firmware/virtLab-user/virtLab-user.elf`. The only difference is to **select the MCU user** on the MCU tab of the VirtLab GUI, as shown on Figure B.16. The LED associated with the MCU on the board will be turned green: Figure B.17.

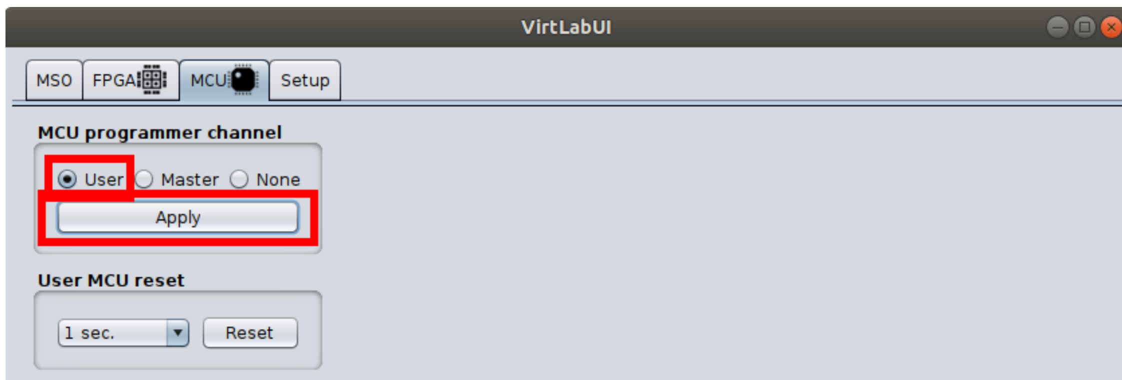


Figure B.16: VirtLab GUI MCU: choose MCU user and apply

B.6.2 Program FPGA User

You can use the following example file: `virtLAB/Hardware/FPGA-templates/fpga-user/fpga-user.rbf`. This program turns on the FPGA LEDs according to the status of the switches from n°5 to n°8 (switch ON $\longrightarrow$ LED ON and vice versa).

- STEP 1: Connect the VirtLab to the PC via USB-C and microUSB, as described in section section B.2.
- STEP 2: Turn on the interrupter at the bottom right of the board.

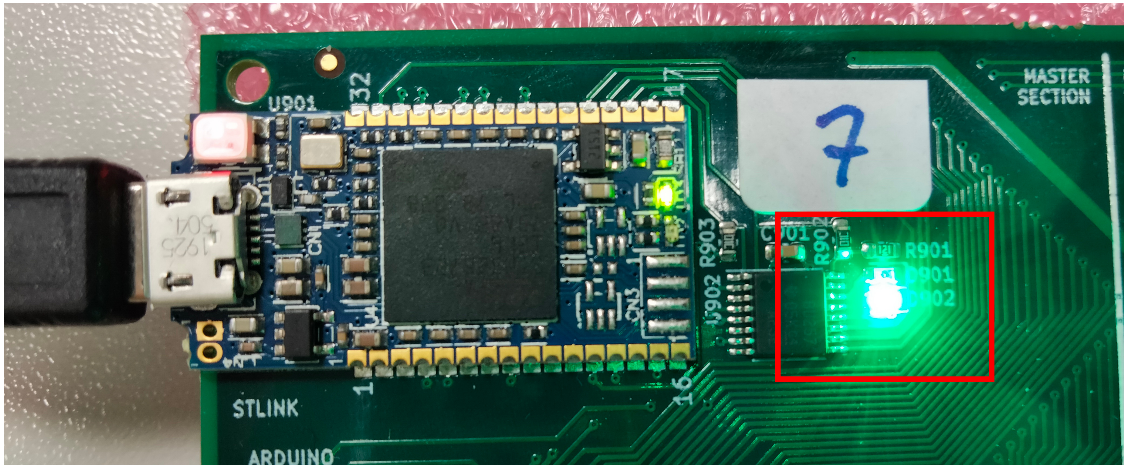


Figure B.17: VirtLab: MCU User selected, GREEN LED

- STEP 3: Launch the Java application (by using the alias created before in section B.4.2 ; or on Windows by double clicking on the program file).
- STEP 4: Connect to the communication port by clicking *Connect*. Then, Got to FPGA tab.. Generally, there will be only one port detected, which is the USB-C one of the board. The name of the port shall be `ttyACMX` in a Linux operating system and `COMX` under Windows, where `X` is a number. If under Windows you do not see the serial port, go to the *Device manager* ⟶ *Ports (COM and LPT)*. Here, Windows should enumerate two ports:
  - `Serial USB device (COMX)`.
  - `STMicroelectronics STLink Virtual COM port (COMY)`.

  The serial port to which you need to connect to program the FPGAs is the `Serial USB device (COMX)`. Under Linux, if you do not manage to connect to the serial port, you are recommended to install `minicom`. Then type

  ```
  minicom -o -D /dev/ttyACMX
  ```

  where `X` is usually either 0 or 1, and check if you manage to connect to the serial port.
- STEP 5: Select *File RBF* and ***User FPGA***. Then choose an RBF file by using *Browse* button. Finally click on the centered *arrow*, and wait for the status bar on the bottom right to be full.. Done!

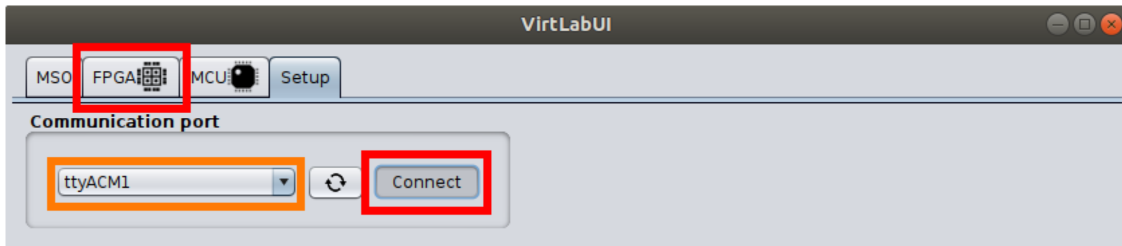


Figure B.18: STEP 4: Connect to the communication port by clicking *Connect*. Then, Got to FPGA tab.

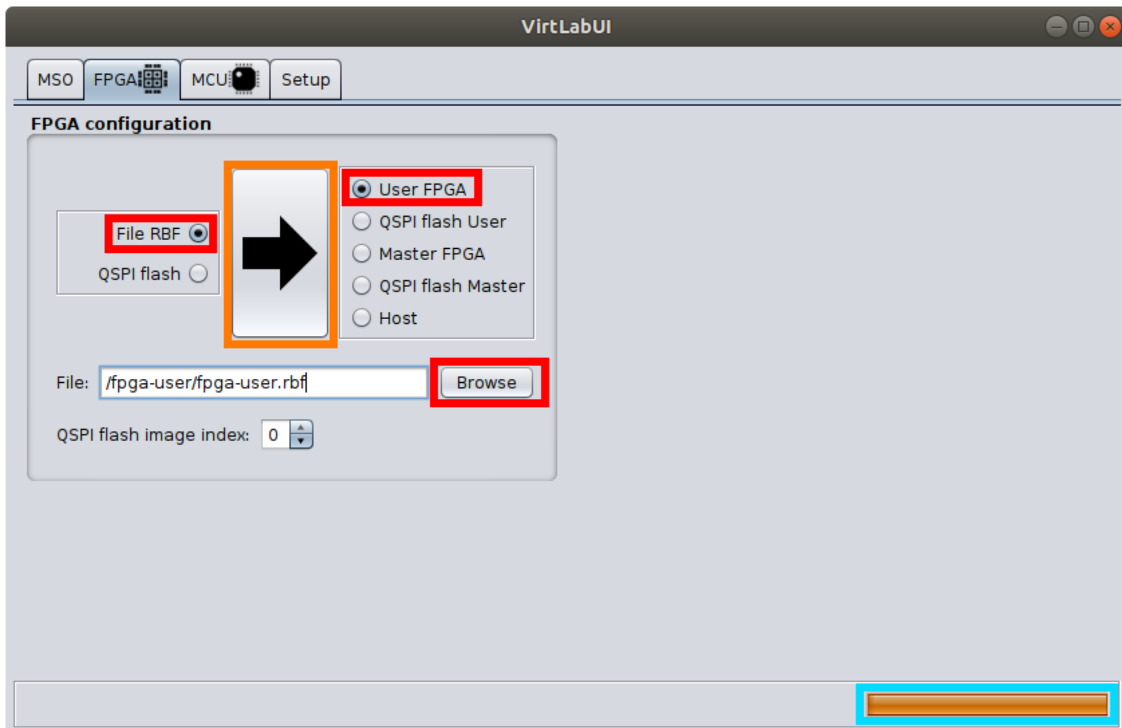


Figure B.19: STEP 5: Select *File RBF* and ***User FPGA***. Then choose an RBF file by using *Browse* button. Finally click on the centered *arrow*, and wait for the status bar on the bottom right to be full.

If you are using the example file, you can now check by using the switches 5 to 8 and observing the FPGA User LEDs. Keep in mind that if the MCU User is not programmed yet, that program can behave badly because of some active pull-up on the un-programmed components.

Note: *Since that firmware is written on the RAM of the FPGA, those steps need to be done each time the board is shutting down and then turning*

on.

B.6.3 Program FPGA Master

You can follow the same steps of subsection B.6.2. By changing the parameters on the FPGAs tab: VirtLabUI: Upload RBF file on FPGA Master. You can use the following example file instead:
`virtLAB/Hardware/FPGA-templates/fpga-master/output_files/fpga-master.rbf`.

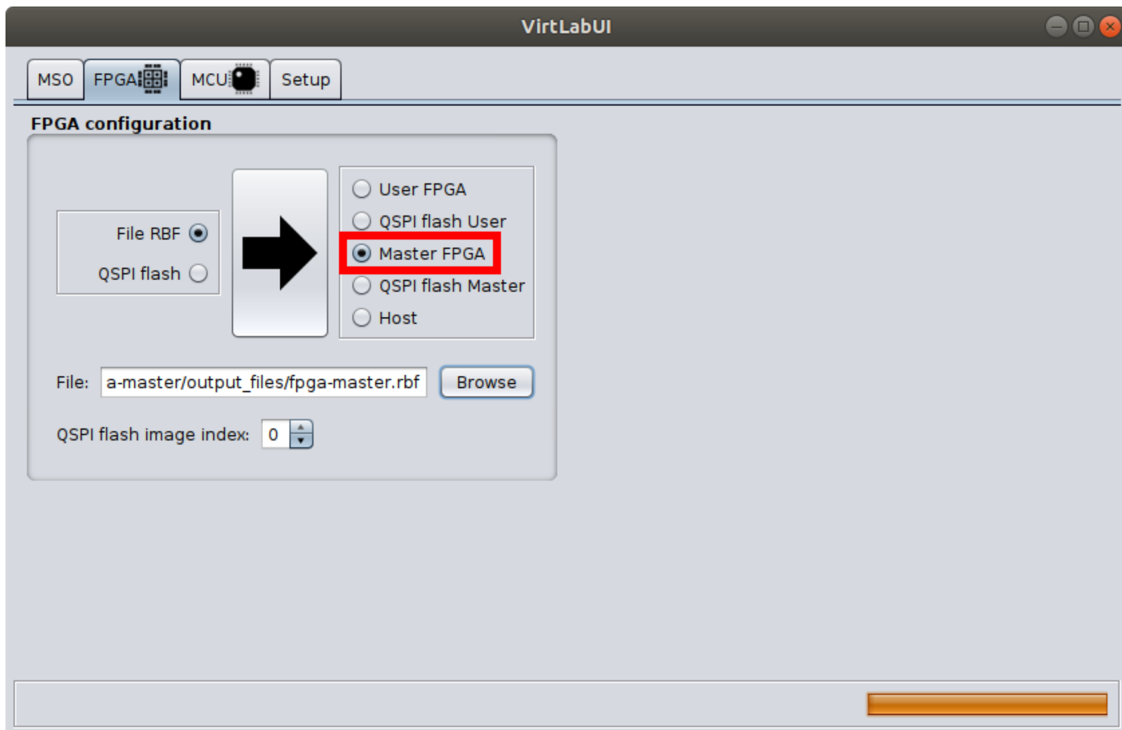


Figure B.20: VirtLabUI: Upload RBF file on FPGA Master

Note: DO IT ONLY IF NECESSARY and if explicitly required by the laboratory experience you are carrying out. *To test your VHDL designs, you must use the FPGA User.*