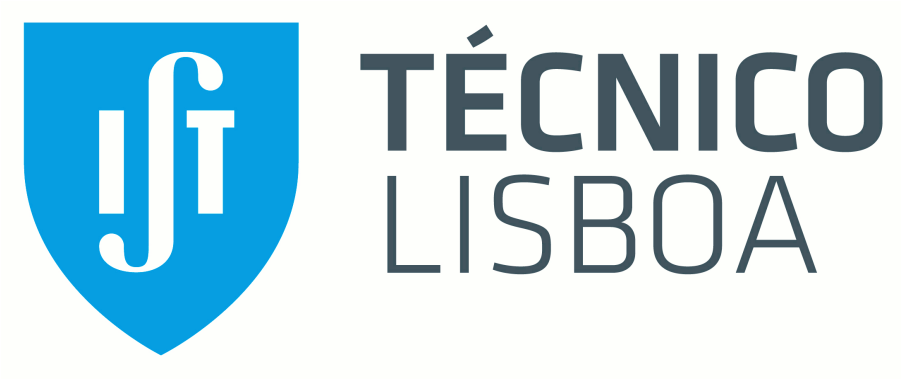

Computer Electronics

RISC-V Everything

Grupo 3

Francisco Mendes - Nº 84055

Gaspar Ribeiro - Nº 84059

January 13, 2020

Contents

1 Introduction

The RISC-V Everything project intents to develop a simple system of a chip - SoC - on a SP605 FPGA. On this project it is going to be used the PicoRV32I CPU core, an open-source implementation of the RISC-V RV32IMC Instruction Set. This implementation consists of a simple CPU with only integer support where peripherals such as a Boot ROM and a RAM and UART module and an LED module are going to be connected to the CPU.

The option for the this CPU Core brings added benefits and challenges. First, it is a Core based on the new and trendy RISC-V ISA, this allows for the exploration of this new technology that intents to quickly become one of the industry standards. Being based on the RISC-V ISA brings the benefit that a complete development tool-chain is widely available (*gcc*, binutils, etc...).

The objective of this project is to show the viability of having a small implementation of a versatile RISC-V based microcontroller used on regular FPGA performing regular tasks. The developed system will have two main memories, the boot memory and a RAM, allowing for on-the-fly application change. As a example application, a Convolution Neural Network (CNN) is devised. This neural network is capable of recognizing an hand written digit on a image of size 28 by 28 pixels. This application will also benefit from UART communication to a host computer, from which it will receive the network weights and the image to classify, responding with the prediction from the CNN.

Additionally a drawing and sending application is devised to be run on the host computer. This application built on Python receives mouse inputs to draw a image, and on signal, it sends a bitstream containing the drawn image to the FPGA CNN accelerator.

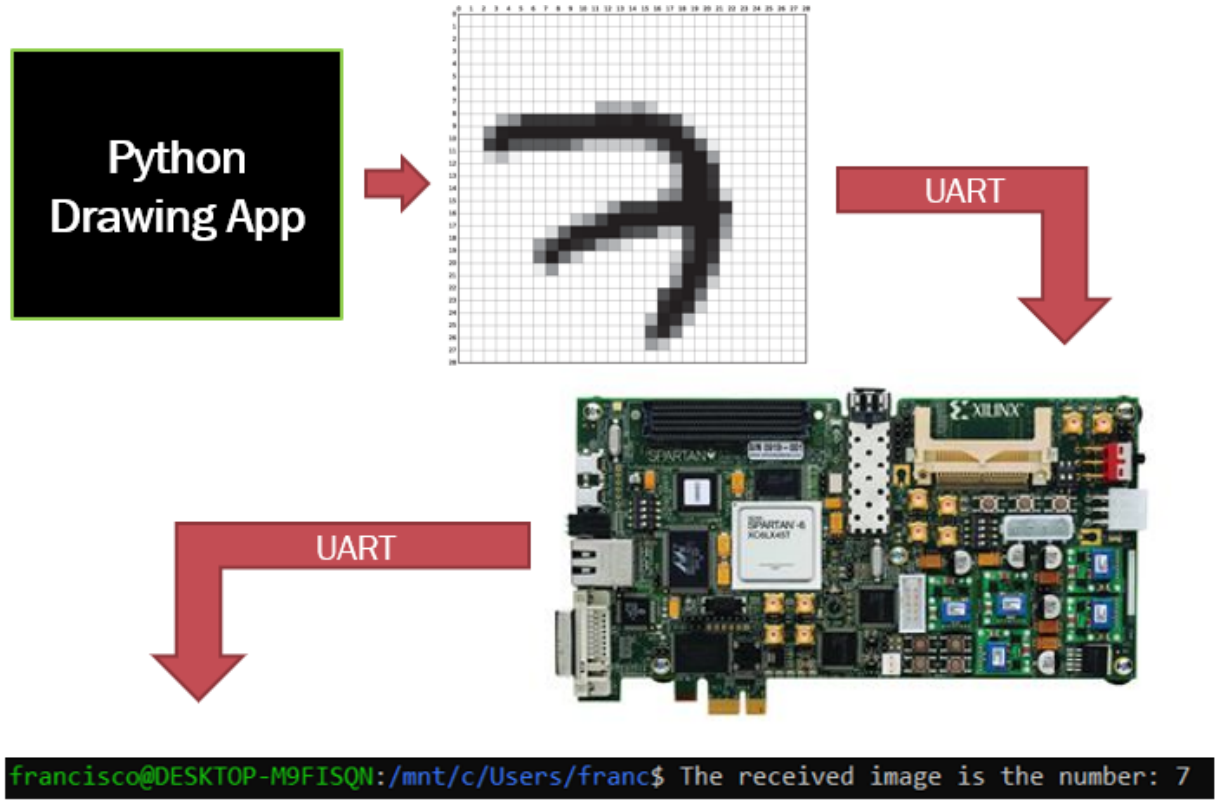

Figure 1: FPGA based Handwritten digit Machine Learning recognition Accelerator.

2 Block Diagram

The RISC-V Everything block diagram is shown in Fig. 8. RISC-V Everything contains two main modules, the CPU and the Peripherals. The CPU module is responsible for controlling the system, actuating on the peripherals modules and run the main program.

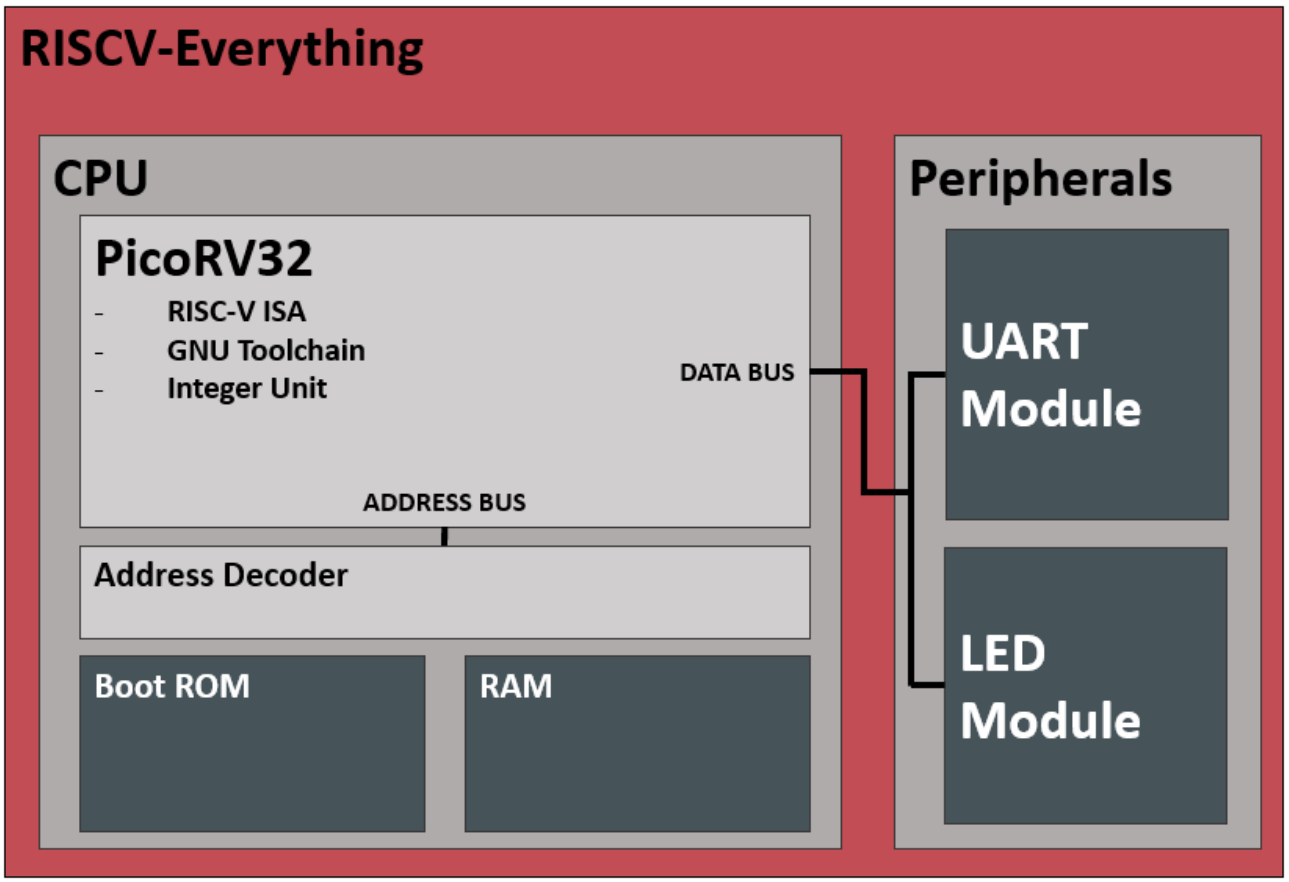

Figure 2: RISC- Everything - Block Diagram.

2.1 CPU

The CPU Core used in this project is a RicoRV32i, open-source processor [https://github.com/cliffordwolf/picorv32]. PicoRV32 is a integer only CPU core that implements the RISC-V RV32IMC Instruction Set and has an attached address decoder, in order to communicate with the different peripherals. The implemented CPU contains two main memories, a Boot ROM responsible for holding the bootloader, and a RAM memory responsible for holding the received program. Both this memories are user configurable, being of size $2^{12} = 4096$ bytes and $2^{17} = 128KB$ on the final implementation.

2.2 Peripherals

The peripherals module is composed of a set of sub-modules each responsible to control - read or write - the corresponding physical peripheral that the system can connect to.

3 Program

The RISC-V Everything project consists of two programs to be run on the FPGA and a program to be run on the host computer.

3.1 FPGA

When electricity is supplied to the FPGA, the bootloader programs starts to run from the Boot ROM. When the bootloader program writes 1 to the `soft_reset` register, the FPGA resets and starts to run the program contained on the RAM.

3.1.1 Bootloader

The main loop of the bootloader is present figure 3. The program is always running when the system is connected to power and it consists in three different configuration of the UART communication, receiving the application program and perform a soft reset on the board.

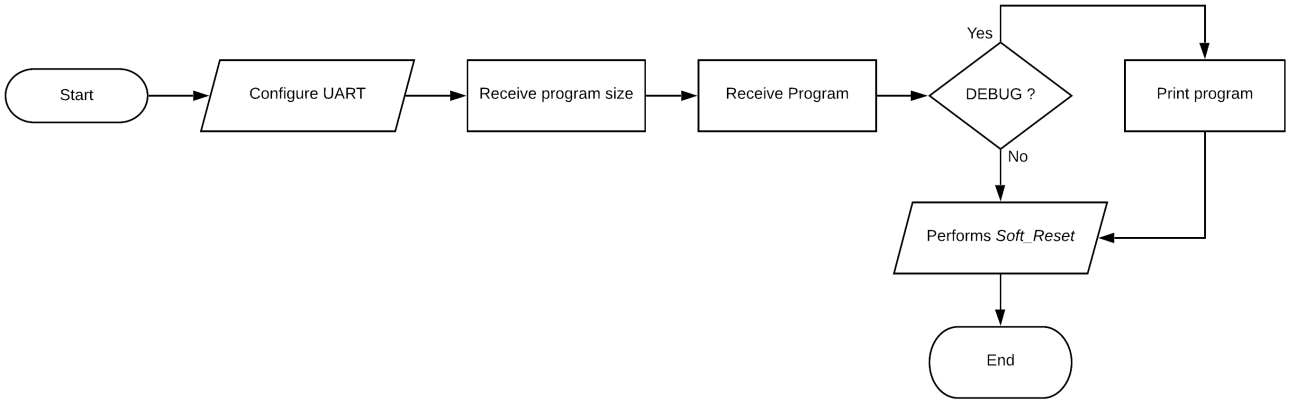

Figure 3: Bootloader Flowchart.

3.1.2 MNIST Classifier

The MNIST Classifier is an example application developed to test the capabilities of the PicoRV32i processor. It consists of a Convolution Neural Network (CNN) trained with the MNIST dataset [http://yann.lecun.com/exdb/mnist/]. This dataset consists of 60000 training and 10000 test images of size 28 by 28 pixels, with pixel values comprehended between [0,1[, containing the handwritten digits 0 to 9. The developed neural network is able to achieve an accuracy of 99.87% and 99.68% on the training and test datasets.

The neural network is composed of a convolutional layer, a max-pooling layer and a fully-connected layer. The convolutional layer is composed of 22 kernels of size 5 by 5 pixels and it is responsible to expose high-level characteristics that relate the images to the digit present in them. The max-pooling layer performs a domain reduction, by taking the pixel with highest value from each set of 4 pixels. This is equivalent to perform a feature enhancement. The last layer, conjugates the values of the 22 kernels into a 10 positions vector, corresponding the ten handwritten digits that can be present on the image. The CNN prediction corresponds to the vector position with the highest value.

One major difference and one of the most important challenges on the implementation of this network, is the lack of support for `floats` from the PicoRV32, meaning that the neural network weights and computations need to be performed in fixed-point notation. The development of the CNN started with the training procedure being done in 32 bits `floats`. After the end of the training session, each of the weights are converted to 16 bits integer with fixed point notation Q5.11. It was chosen to implement the weights in integer with 16 bits (vs the regular 32 bits) to reduce the program size. This decision did not invoke a change on the achieve accuracy. The notation was chosen by analyzing the highest weight value, since it is inferior to 32, 5 bits are reserved to the integer part. The first two layers have their results in notation Q5.11, however fot the fully connected layer, it was necessary to store the results in Q11.5. With 11 bits for the integer part, the maximum possible value for the layer output is guaranteed to have a correct representation.

After it, both the inference with floats as well in fixed point was implemented, in order to assess if a accuracy difference exists between the two implementation. It was verified that with the use of correct fixed point notation throughout the network inference, both implementations are able to achieve the same accuracy of 99.68% on the test set.

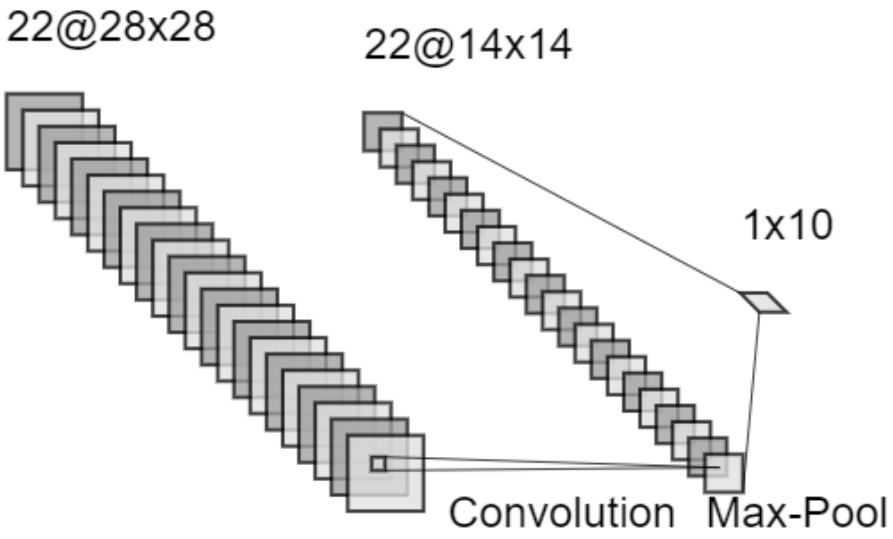

Figure 4: CNN Architecture with 192600 different weights.

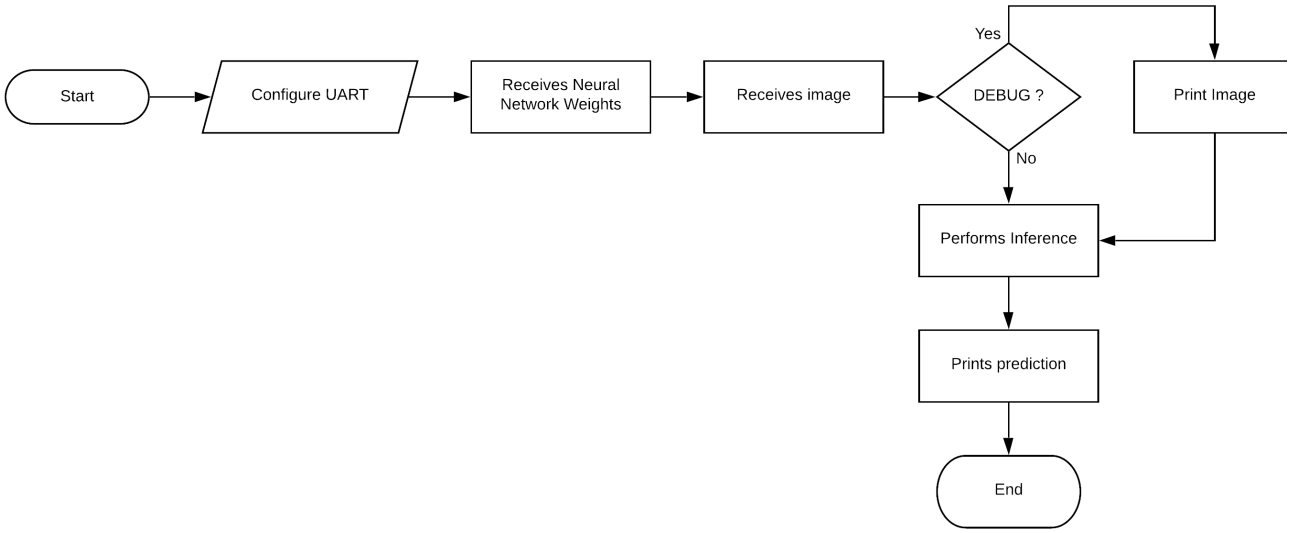

Figure 5: Inference Routine Flowchart.

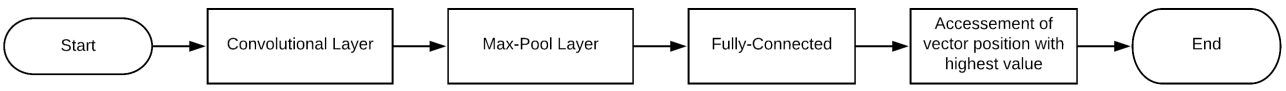

Figure 6: CNN Routine Flowchart.

3.2 Host Computer

3.2.1 Drawing App

The Python Drawing App is able to receive mouse inputs to drawn a number on the screen. When the user presses *Save*, it performs a image reduction to 28 by 28 pixels and sends the image to the MNIST Accelerator.

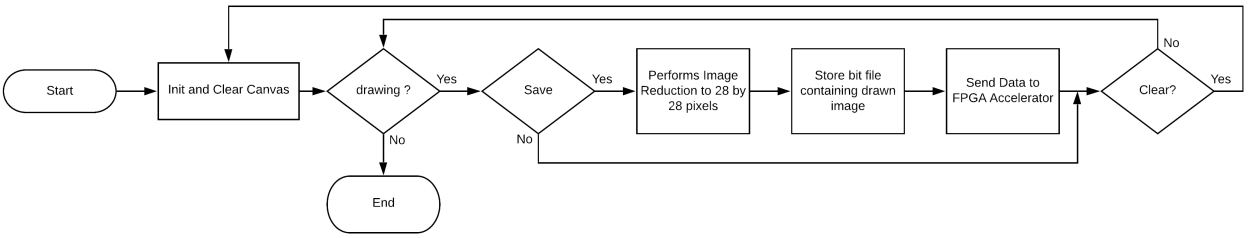

Figure 7: Drawing App Routine Flowchart.

4 Interface Signals

The interface signals of the RISC-V Everything is described in the Table 1.

Figure 8: RISC-V Everything - Block Diagram.

Name	Direction	Description
clk	Input	Clock signal
reset	Input	Reset signal, asynchronous, on high
RX	Input	Data input from UART to USB Chip
CTS	Input	Clear To Send - signal sent from other UART device
TX	Output	Data output to UART to USB Chip
RTS	Output	Request to Send - signal sent to other UART device
led [3:0]	Output	LEDs to bue turned on

Table 1: Interface signals.

5 Memory Map

The memory map of the system, as seen by RISC-V Everything programs, is given in Table 2. These values are defined in the `system.vh` verilog header file.

Mnemonic	Address	Read/Write	Read Latency	Description
BOOT_BASE	0	Read/Write	0	Boot Memory Base Address
UART_BASE	1	Read/Write	1	UART Peripheral
SOFT_RESET_BASE	2	Write	NA	Register holding current valid memory
RAM_BASE	3	Read/Write	0	RAM Base Address
LED_BASE	4	Write	NA	LED Peripheral

Table 2: Memory Base base

6 Peripherals

The SP605 development board contains peripherals responsible for the input/output of data to and from the board, that includes an array of LEDs, one push buttons and a UART to USB on-board chip which can be used to interact with the RISC-V Everything project implemented on the board.

6.1 CPU

The CPU responsibility is to enable the modules that interact with the physical peripherals and write/read to/from the common data-bus that connects the two sybsystems.

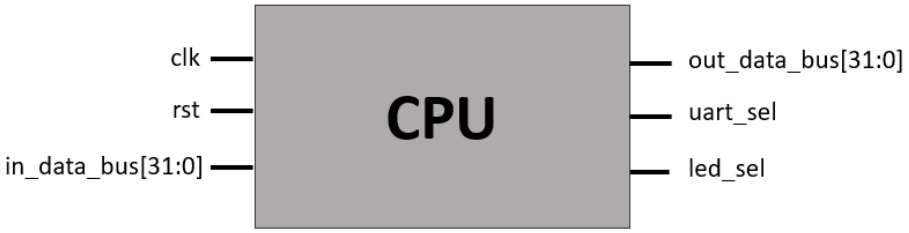

Figure 9: CPU module - symbol.

Name	Direction	Description
clk	Input	Clock signal
rst	Input	Reset signal, asynchronous, on high
in_data_bus [31:0]	Input	Data from the peripherals
out_data_bus [31:0]	Output	Data to the peripherals
uart_sel	Output	Activate the UART module
led_sel	Output	Activates the LED Module

Table 3: CPU Module signals.

6.1.1 PicoRV32I

The PicoRV32i symbol is shown in figure 10. PicoRV32i is a simple processor that implements the basic RISC-V Instruction Set Architecture. It includes the complete RISC-V tool-chain, allowing for the development of C programs and the guaranteed execution of the same. It has a common data-bus where all the peripherals are connected to.

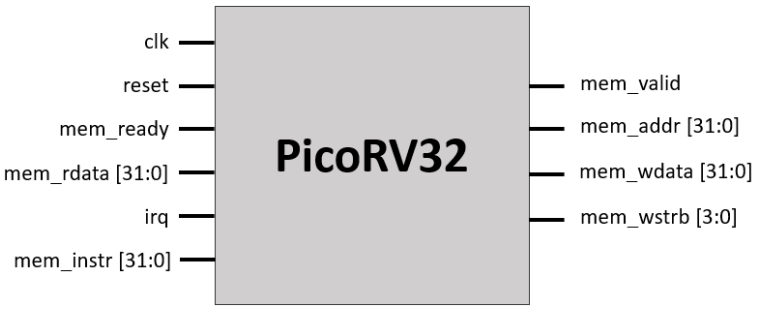

Figure 10: Picorv32i - symbol.

Name	Direction	Description
clk	Input	Clock signal
rst	Input	Reset Signal, asynchronous, on high
mem_ready	Input	Indicates that the memory is ready to be written/read
mem_rdata[[31:0]	Input	data read from the memory
irq	Input	Interruptions - Not in use
mem_instr[31:0]	Input	Current Instruction
mem_valid	Output	Enable of bootram module
mem_addr[31:0]	Output	Address position to be read or write
mem_wdata[31:0]	Output	Data to be writeen on memory
mem_wstrb[3:0]	Output	Enable of the memory block

Table 4: PicoRV32 module signals.

Name	Address offset	Read/Write	Description
mem_ready_reg	0	Write	Indicates that the memory is ready to be read
rdata_reg	4	Write	Content of the memory
mem_ready_reg	8	Read	Indicates that CPU is ready to write
wdata_reg	12	Read	Content to be written to the memory
wstr_reg	16	Read	Each 4 bits that wants to be written

Table 5: PicoRV32 module registers map.

6.1.2 Address Decoder

The address decoder symbol is shown in figure 11. It is responsible for decoding the address of the peripherals and create the enable signals that enable the peripherals modules. The address decoder is user configurable through the `system.vh` verilog header file.

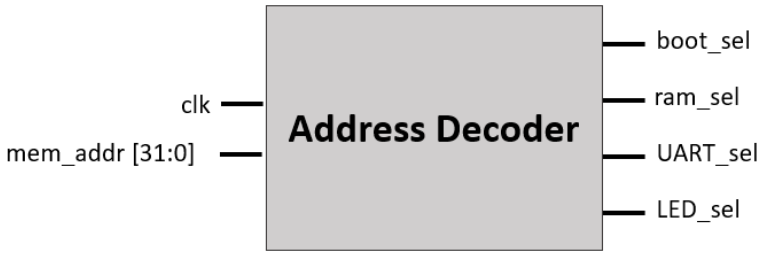

Figure 11: Address Decoder - symbol.

Name	Direction	Description
clk	Input	Clock signal
mem_addr [31:0]	Input	Address to be decoded
boot_sel	Output	Enable Boot ROM module
ram_sel	Output	Enable RAM module
UART_sel	Output	Enable UART module
LED_disp_sel	Output	Enable LED module

Table 6: Address decoder signals

Name	Address offset	Read/Write	Description
mem_addr_reg	0	Write	Address to be decoded

Table 7: Address Decoder module registers map.

6.1.3 Boot ROM/RAM

The Boot ROM/RAM symbol is shown in figure 12. The memory size is user configurable by selecting the number of digits allocated for addressing (changing the number of bits of the signal `mem_addr`, in the figure it was chosen to include the maximum possible value, with `mem_addr` being a 32 bits signal - a memory of 4 GBytes). In the current implementation, the Boot ROM has 4096kB (12 bits of addressing) and the RAM has 128 KBytes (17 bits of addressing). These values are configured through the `system.vh` verilog header file.

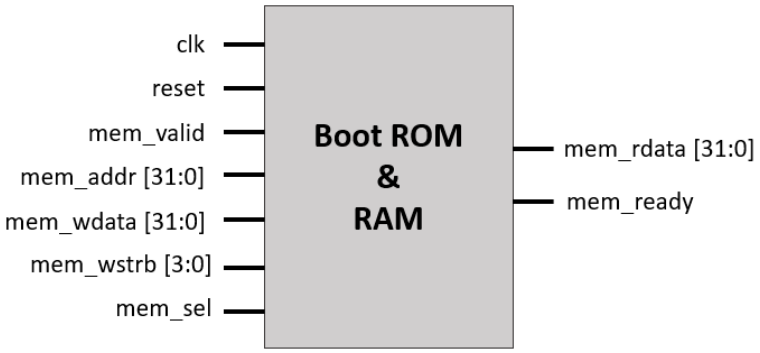

Figure 12: Boot ROM/RAM - symbol.

Name	Direction	Description
clk	Input	Clock signal
rst	Input	Reset Signal, asynchronous, on high
mem_valid	Input	Enable of bootram module
mem_addr[31:0]	Input	Address position to be read or write
mem_wdata[31:0]	Input	Data to be written on memory
mem_wstrb[3:0]	Input	Enable of the memory block
mem_rdata[31:0]	Output	Data read from the memory
mem_ready	Output	Indicates that the memory is ready

Table 8: bootram module signals.

Name	Address offset	Read/Write	Description
mem_ready_reg	0	Read	Reads if the memory is ready to be read
rdata_reg	4	Read	Content of the memory
mem_ready_reg	8	Write	Indicates that CPU is ready to write
wdata_reg	12	Write	Content to be written to the memory
wstr_reg	16	Write	Each 4 bits that wants to be written

Table 9: Boot ROM and RAM module registers map.

6.2 Peripherals

6.2.1 UART

The UART Module symbol is shown in Figure 13. This module is responsible to make the handshake between the FPGA and the host computer through the RTS and CTS signals. The module requires an initial configuration of the baud rate used by defining the division factor between the FPGA clock frequency and the desired baud rate.

Due to this module interact with an older UART chip that needed flow control and being relatively slow, it was necessary to include on the UART C-driver a debounce feature both on the read and write operation, that ensures that the RTS and CTS signals are stable before operation continues.

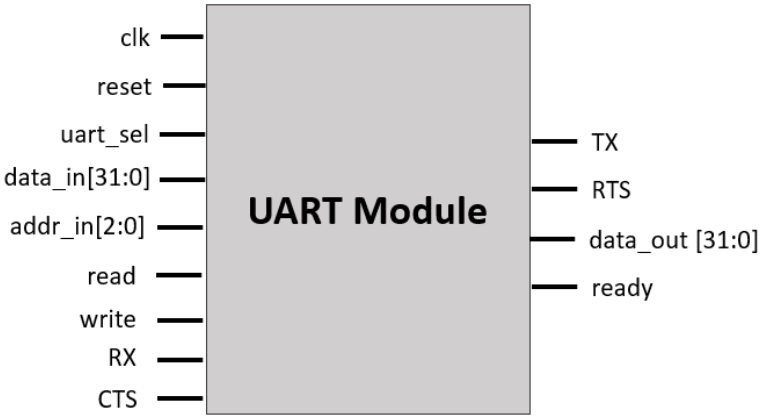

Figure 13: UART module - Block Diagram.

Name	Direction	Description
clk	Input	Clock
reset	Input	Reset Signal, asynchronous, on high
uart_sel	Input	Activate the module
data_in[31:0]	Input	Data to be write to the module
addr_in [2:0]	Input	Address to select to which register to write to
read	Input	Writes to a register to specify if the FPGA is ready to read the data
write	Input	Writes to a register to specify if the FPGA is ready to write data
RX	Input	Received data from the UART to USB chip
CTS	Input	Clear to Send
TX	Output	Data to be transmited
RTS	Output	Request to Send
data_out[31:0]	Output	Data received
ready	Output	Indicates when the device is ready

Table 10: UART module signals.

Name	Address offset	Read/Write	Description
div_factor	0	Write	UART division factor
data_in_out	4	Read/Write	Data input and output
write_wait	8	Read	Indicates if receiver is ready to receive
read_wait	12	Read	Indicates if there is incoming data

Table 11: UART module registers map.

6.2.2 LEDs

To turn on any of the 8 LEDs of the board, it's respective bit must be '1' and the led_sel signal also needs to be set to '1' by the CPU.

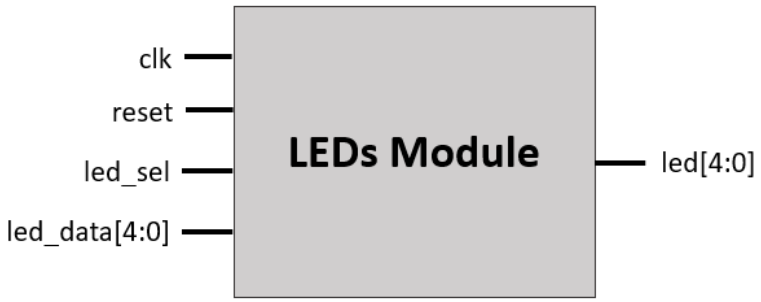

Figure 14: LEDs module - Block Diagram.

Name	Direction	Description
clk	Input	Clock
rst	Input	Reset Signal, asynchronous, on high
led_sel	Input	Activate the module
led_data[3:0]	Input	LEDs to be turned on if the module is active
led [3:0]	Output	LEDs to be turned on

Table 12: LEDs module signals.

Name	Address offset	Read/Write	Description
led_data	0	Write	Leds to be turned on

Table 13: LEDs module registers map.

7 Implementation Results

In this section it is present the results of implementing the project on a SP605 development board with the Spartan 6 FPGA and the performance of the CNN software.

7.1 Hardware

The project was implemented on the SP605 development board with the Spartan 6 FPGA. The chosen size for the RAM component of $2^{17} = 128KBytes$ is the maximum possible to be mapped to this FPGA. Adding more memory will result in an over utilization of the BRAM components. Overall, in this configuration the utilization of all resources is adequate for the FPGA, with all resources bellow the 70% mark (the limit used on the industry). Table 14 display the results of the implementation.

Resource Type	**DPS**	**FF**	**LUT**	**LUTRAM**	**BRAM**	**BUFG**
#	4	4391	3732	513	64	2
Utilization [%]	58	55	13	8	55	12

Table 14: RISC-V Everything Resources Utilization on SP605

The SP605 development Board with the Spartan 6 FPGA has two internal clock sources. The fastest with a frequency of 200 MHz and a slower of 27 MHz. One a first phase, the clock of 27 MHz was used, since it guarantees the direct use of a clock source. With the results of the implementation showing a minimum possible clock period of 13.577 ns, see Table 16, corresponding to a maximum frequency of 73.65 MHz, it was decided to instantiate a BUFG, together with a counter to 4, to divide the 200 MHz clock to 50 MHz.

Constraint	**Check**	**Worst Case Slack [ns]**	**Best Case Achievable [ns]**
TS_clkin = PERIOD TIMEGRP	SETUP	0.085	13.492
”clk” 13.577 ns HIGH 50%	HOLD	0.339	

Table 15: Timing Constraints - Best Value

Constraint	**Check**	**Worst Case Slack [ns]**	**Best Case Achievable [ns]**
TS_clkin = PERIOD TIMEGRP	SETUP	6.205	13.795
”clk” 20 ns HIGH 50%	HOLD	0.404	

Table 16: Timing Constraints - 50 MHz

```
# Clock
NET "clk" TNM_NET = "clk";

# 200 MHz clock - used with bufg to 50 Mhz
TIMESPEC "TS_clkin" = PERIOD "clk" 20 ns HIGH 50 %;
NET "clk"  LOC = "K22";

# 27 MHz clock - used for debug
#TIMESPEC "TS_clkin" = PERIOD "clk" 37.037 ns HIGH 50 %;
```

```
#NET "clk"   LOC = "AB13";

# BUTTON
NET "reset"   LOC = "H8";

# LED TRAP
NET "trap"   LOC = "D17";
#NET "led<0>"   LOC   = "AB4";   ## LED DS4
#NET "led<1>"   LOC   = "D21";   ## LED DS5
#NET "led<2>"   LOC   = "W15";   ## LED DS6

# UART
NET "uart_rxd"      LOC = "H17";
NET "uart_txd"      LOC = "B21";
NET "uart_rts_n"    LOC = "F18";
NET "uart_cts_n"    LOC = "F19";
```

Listing 1: top_system.ucf

7.2 Software

The execution of the program starts by sending the CNN firmware to the FPGA through the UART. This action takes roughly 93.7 seconds (the program has a total size of 101.7 KBytes and using a baud rate of 115200). To measure the throughput of the CNN accelerator, it was chosen to hard-code an input image with the digit 3 on the code (instead of sending it through the UART). In this way, we can measure only the computation time that the accelerator takes to perform the inference of the digit. It was measured that in 1 minute, the FPGA could predict 53 digits, achieving an average time taken to perform one inference of 1.12 seconds. After it, the 1000 images from the MNIST testing dataset were tested on the FPGA to guarantee the accuracy of the implementation. In this run, the code implemented on the FPGA managed to achieve the same number of correct predictions as the original code run on the host computer.

After it, the complete system was assessed, measuring the time it takes to send an image (of 28 by 28 pixels, each represented as a 8 bits char), process it and receive the prediction through the UART. This process takes around 2.6 seconds. Figure 15 shows the output from the UART at the host computer, after sending an image containing the digit 3. As it is possible to observe, the CNN accelerator was able to correctly predict the digit on the image.

To notice that in the tested implementation, the accelerator was running at a frequency of 50 MHz, if the maximum frequency of 73.65MHz was instead being used, it was possible to reduce the computation, roughly by 1.5 times, with the inference taking less than 1 second to be performed.

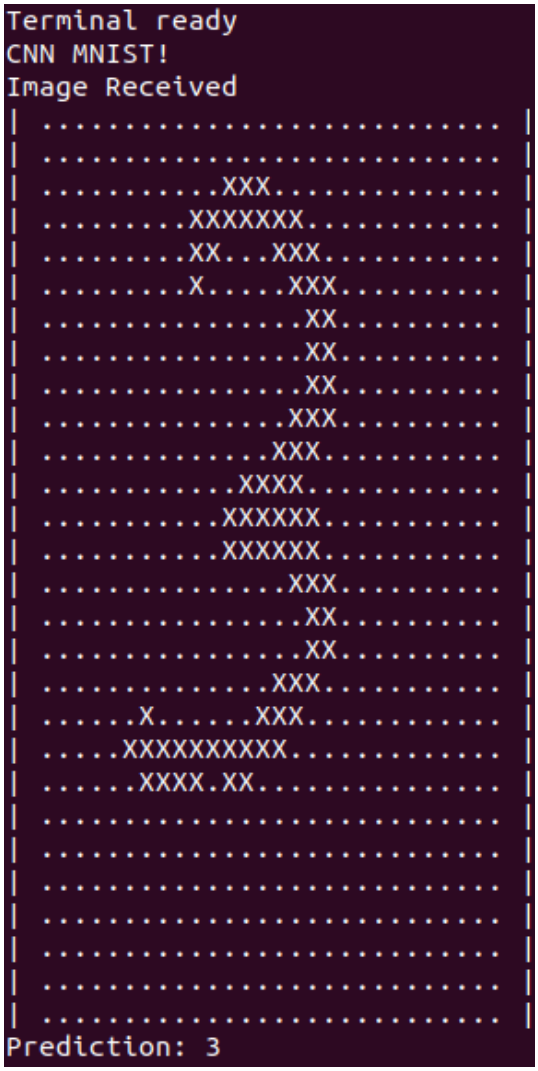

Figure 15: Result of the picocom for the inference of the number 3.

8 Conclusions

RISC-V Everything provides a RISC-V microcontroller virtual implementation and a UART communication module are made available. This implementation allows for flexibility and easiness of programming, by providing a way to interact with the processor in real-time through UART and the ability to change the program on the fly. The main goal of the project was to port this implementation to an older FPGA that is still widely used and capable, nowadays.

The develop example application was MNIST classifier. This application was made to illustrate the capabilities of both the UART and the RISC-V processor. The generated images are transfered images through UART to the FPGA, where all the processing takes place. The MNIST classifier uses a convolutional neural network to predict the digit on the image. This type of application is a category of deep learning algorithms, widely used and implement nowadays in FPGAs to allow for edge computing.

The development of the many components that are included in this project (software - MNIST Classifier and python drawing app, and the hardware) are made in such way, that allows for standalone products to be made of them. This is an important characteristic to allow for easy interchange of modules and easy integration of the components in other systems.