SIP WS 2023
Project 2: *FPGA-based image classification*

1 Organization

- Group size: 2-4
- Deadline 2a: 12.12.2023, 23:59
- Deadline 2b: 23.01.2023, 23:59
- git repositories: `git.teaching.iaik.tugraz.at`
- Project material: `https://extgit.iaik.tugraz.at/sip/project2`

Where to ask questions

- In our weekly sessions
- Discord: `https://discord.gg/9KKGfndsD5`
- E-Mail: sip-team@iaik.tugraz.at

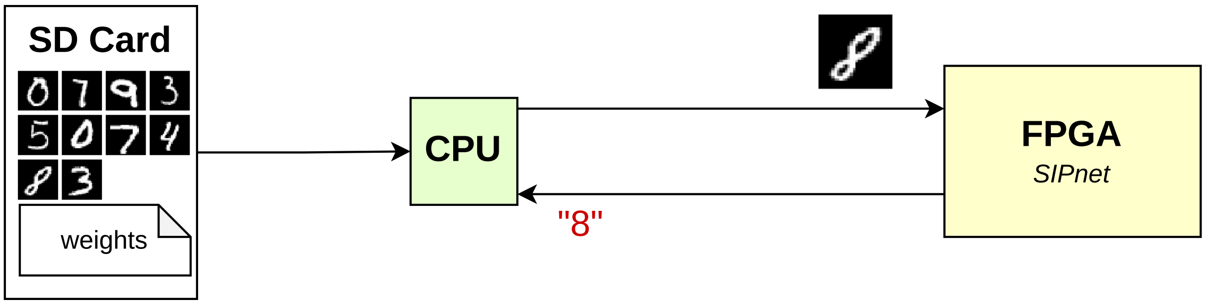

Figure 1: System overview

2 System specification

FPGAs play an increasingly important role in the artificial intelligence field. In this project we will explore hardware-accelerated image recognition by porting *SIPnet*, a simple neural network, to an FPGA. *SIPnet* reads an input image from the MNIST dataset [4, 5], which consists of images of handwritten digits, performs classification and outputs the digit which is visible in the image. *SIPnet* has already been trained, i.e., all the values for weights are already available, we will focus on the evaluation phase. In our system, the images and weights are stored on the SD card. To initialize *SIPnet*, the weights are first written into the FPGA's

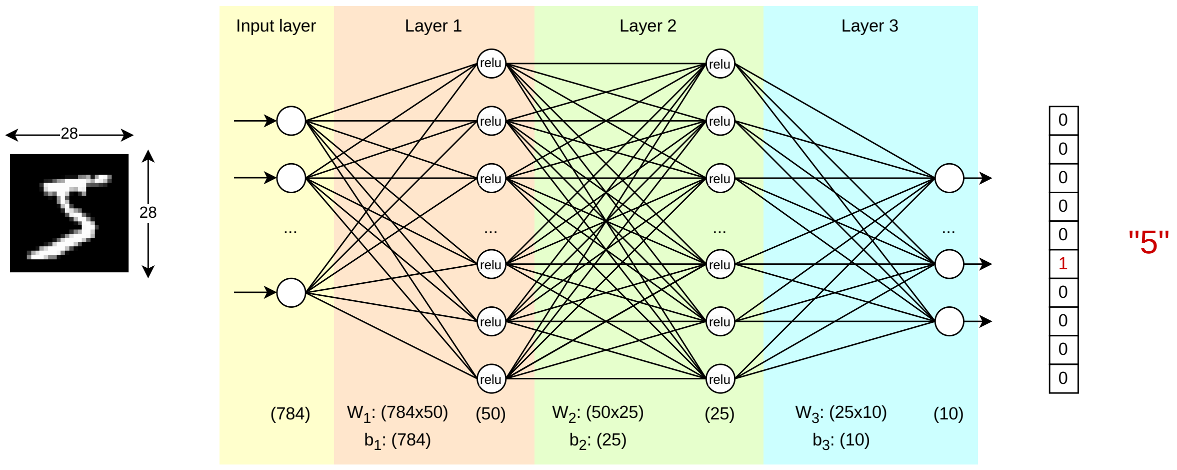

Figure 2: Overview of *SIPnet*

block RAM (BRAM) by the CPU. The BRAM is memory used to store large amounts of data which needs to be easily accessible by the FPGA and via the AXI bus by the CPU. In order to classify an image, the image is written to the BRAM by the CPU. During classification, *SIPnet* loads the weights and image data from the BRAM, performs the respective computations and outputs the correct digit.

2.1 SIPnet

As shown in Figure 2, *SIPnet* consists of three layers. The input layer transforms the 28x28-image into a vector of 784 entries. Every pixel is normalized to the range [0,1], i.e. divided by 256, yielding the vector I. Layer 1 is a fully connected layer working with the 784x50-weight matrix W_1, the bias vector b_1 which has 784 entries, and the ReLU activation function. The output of layer 1 o_1 is computed as follows:

$$o_1 = \text{ReLU}(I \times W_1 + b_1)$$

o_1 is a vector with 50 entries. It is given to layer 2, which is also fully connected. Layer 2 works with the 50x25-weight matrix W_2, the bias vector b_2 which has 25 entries, and the ReLU activation function. The output of layer 2 o_2 is computed as follows:

$$o_2 = \text{ReLU}(o_1 \times W_2 + b_2)$$

o_2 is a vector with 25 entries. It is given to layer 3, the output layer, which is also fully connected. Layer 3 works with the 25x10-weight matrix W_3, the bias vector b_3 which has 10 entries. The output of layer 3 o_3 is computed as follows:

$$o_3 = o_2 \times W_3 + b_3$$

Note that in this case no activation function is applied. The index of the entry of o_3 which has the highest value is (most likely) the digit shown in the input image. For this project, training of *SIPnet* was already done and you are given the values for weights and biases. The accuracy of *SIPnet* (when evaluated on unseen test data) is 95.58%.

Fixed-point arithmetic A big challenge when creating hardware implementations of neural networks is dealing with floating point numbers. One strategy is to use a special neural network architecture, such as a binarized neural network [3, 6], which restricts the values of the weights, biases and input values to $\pm$ 1. Another strategy is to use fixed-point arithmetic instead of floating point arithmetic. The idea of fixed-point arithmetic is to represent fractional numbers as an integer reserving a fixed number of bits for the fractional part [1, 7, 8]. For example, considering 8-bit integers, one could reserve 5 bits for the fractional part and the MSB for the sign bit. Each bit in the 8-bit integer represents the prefix of a power of two:

sign	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}

In this format, the decimal number $(6.325)_{10}$ is $(00110.011)_2$ because:

$$\begin{aligned}(6.325)_{10} &= 6 + 0.325 = 4 + 2 + 0.25 + 0.125\\ &= 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} +^1 *2^{-3}\\ &= (00110.011)_2 = (33)_{16}\end{aligned}$$

Similar to that, the hexadecimal number $(5f)_{16}$ would be $(11.875)_{10}$ because:

$$\begin{aligned}(5f)_{16} &= (01011.111)_2\\ &= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} +^1 *2^{-3}\\ &= (11.875)_{10}\end{aligned}$$

The resolution of the fraction is 0.125, meaning that the number after the comma can only be either .0, .125, .25, .325, .5, .625, .750 or .875.

To build *SIPnet*, we use 16-bit fixed point numbers. Let x be such a number, and let $x[15:0]$ be the bits of x. Then, $x[15]$ will be used to store the sign bit, i.e., whether the number is positive or negative. $x[14:9]$ is used to store the integer bits ("before comma"), and $x[8:0]$ is used to store the fractional bits ("after comma"). While the big advantage of fixed-point arithmetic is that it is easier to use in an hardware implementation, the disadvantage is the loss of accuracy. In case of *SIPnet*, the accuracy when using fixed-point arithmetic is 95.15%. Note that for the project, all the weights and images will be given to you already in (normalized) fixed-point format. If needed, you can find the respective conversion functions in the `fixed_point_arithmetic` directory.

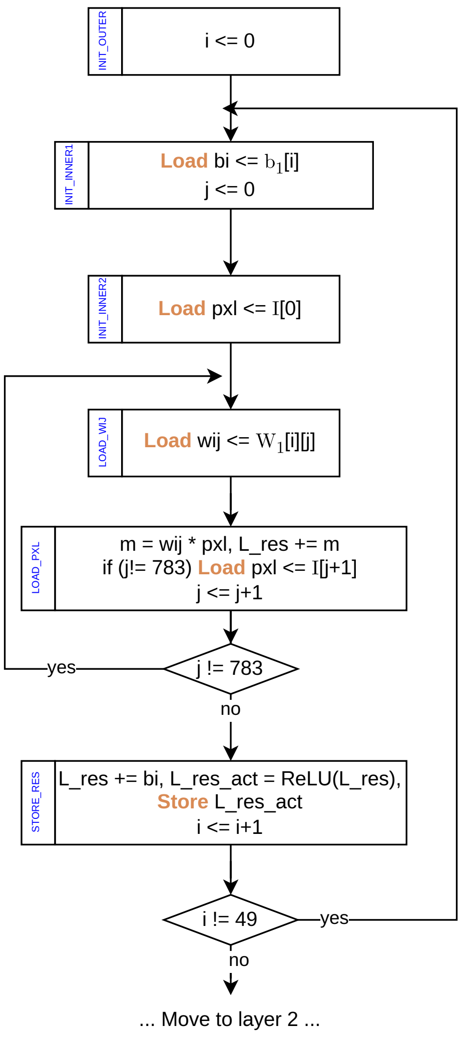

Figure 3: State machine used by *SIPnet*. States are only shown for the first layer.

2.2 BRAM

The BRAM is memory used to store large amounts of data which needs to be easily accessible by the FPGA, and via the AXI bus by the CPU [2]. In our project, the BRAM is used to store weights, biases and the image data itself, which are all 16-bit fixed-point numbers. Therefore, the used BRAM has a word size of 16 bit. The BRAM size is 41644 $\times$ 16 bit, which is used to store the values of $I, W_1, b_1, W_2, b_2, W_3$, b_3 and intermediate computation results. In order to instantiate BRAM, the Block Memory Generator from the IP Catalog can be used.

To initialize *SIPnet*, the CPU reads the values of W_1, b_1, W_2, b_2, W_3, b_3 from the SD card and writes it to BRAM. The connection to the BRAM is done over the AXI bus. One option would be to directly wrap the BRAM into an AXI IP core, using the BRAM controller IP core from the IP catalog. It is however not suitable for this project since it only supports 32-bit word sizes, and does not allow so large BRAM sizes because every BRAM cell is mapped to an AXI register. Instead, we use a custom IP core called *DataManager*, which maps a single 32-bit AXI register. Whenever the CPU writes something to it, a write is also triggered in the BRAM. The first 16 bit are used for the BRAM address, the second 16 bit are used for the value which should be written to BRAM. To classify an image, the CPU reads the image data from the SD card and writes it to BRAM. Initialization should only be done once in the beginning before classifying several images.

While the CPU only writes to the BRAM, *SIPnet* performs read and write accesses. Read accesses happen while loading the values of $I, W_1, b_1, W_2, b_2, W_3$ and b_3 from BRAM. Write accesses happen when the intermediate results (o_1 and o_2) are stored to BRAM, to be loaded again when computing the next layer. The state machine implemented by *SIPnet* is shown in Figure 3, with the loads and stores highlighted.

The memory map used by CPU and *SIPnet* is given as follows:

	Space needed [x 16 bit]	Begin Address	End Address
MNIST image	784	0	783
W_1	39200	784	399983
b_1	50	39984	40033
W_2	1250	40034	41283
b_2	25	41284	41308
W_3	250	41309	41558
b_3	10	41559	41568
o_1	50	41569	41618
o_2	25	41619	41643

For example, when storing the first pixel of the image, a write to address 0 is performed. Accessing the 28th value of b_1 would require to read from address 40012.

The BRAM uses the following interface:

- *addr* (16 bit, input): indicates where to write to or read from
- *din* (16 bit, input): data to be written to BRAM
- *dout* (16 bit, output): data to be read from BRAM
- *en* (1 bit, input): indicates that a read or write access happens
- *we* (1 bit, input): indicates that a write access happens

Note that write accesses can only happen if both *en* and *we* are high. Reads from BRAM have a latency of one cycle. For example, when setting the address in cycle t_0, the respective data is available in t_1.

2.3 Project 2a

Create a hardware design as shown in the sketch in Figure 4, connecting *SIPnet* to the BRAM and the CPU. Demonstrate that classification with *SIPnet* works by creating a bare-metal application and a Linux device driver.

- Create a custom IP core which runs *SIPnet*. It is connected to the AXI bus. The CPU can instruct *SIPnet* to start the classification (AXI Write). The CPU can also read

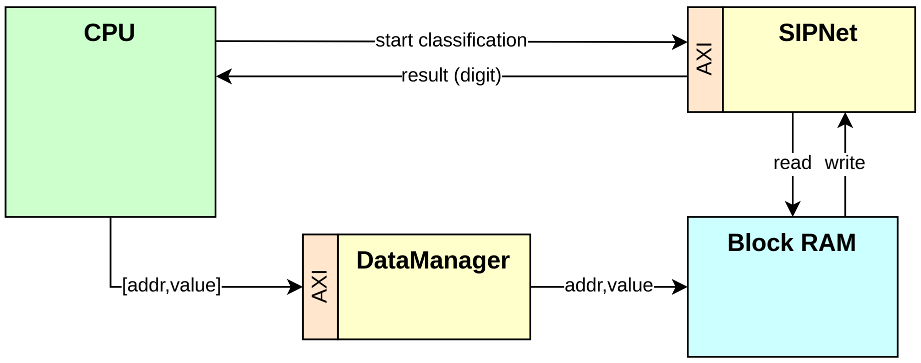

Figure 4: Sketch of block design

whether the classification is finished, and the classification result (AXI Read). Find the source code for *SIPnet* in the `sipnet`* directory. (7P)

- Create a BRAM with 41644 memory locations of 16 bit each. Use the Block Memory Generator from the IP Catalog. It must be in *Stand Alone* mode, of *True Dual Port RAM* type. Read and write widths are 16 bit. Make sure to untick the *Primitives Output Register* option! (3P)
- Create a custom IP core called *DataManager*. It is connected to the AXI bus. The CPU can write a 32-bit value to the *DataManager*. The first 16 bit are used for the BRAM address, the second 16 bit are used for the value which should be written to the BRAM. (5P)
- Connect the *SIPnet* to the BRAM (Port A). Connect the *DataManager* to the BRAM (Port B). Be aware that *SIPnet* performs read and write accesses, while the *DataManager* only performs write accesses (set enb = web). Create an AXI VIP test environment to demonstrate the resulting block design works. For testing purposes, perform some AXI Read and Write accesses. (5P)
- Write a bare-metal program which reads the values for weights, biases and pixels from the SD card, utilizes the *DataManager* to store the values to the BRAM, starts classification with *SIPnet* and reads back the digit once the classification is done. A small demo program using files on the SD card can be found in the `sddemo` directory. Find the weights, biases and images in fixed-point format in the `data` directory. Make sure that building xilffs as a Supported Library is enabled in the Board Support Package Settings. (5P)
- Write a device driver representing the *DataManager* and the *SIPnet* as a file in the procfs. Then, write a user program demonstrating reading/writing of the weights and classification (c.f. previous point). (5P)

* **Important note:** The directory includes the Verilog code of *SIPnet* in `rtl`, and a Makefile to build a Verilator testbench. The files `bram.v` and `SIPnet_top.sv` are only for simulation using Verilator, and must not be included in your custom IP core.

2.4 Project 2b

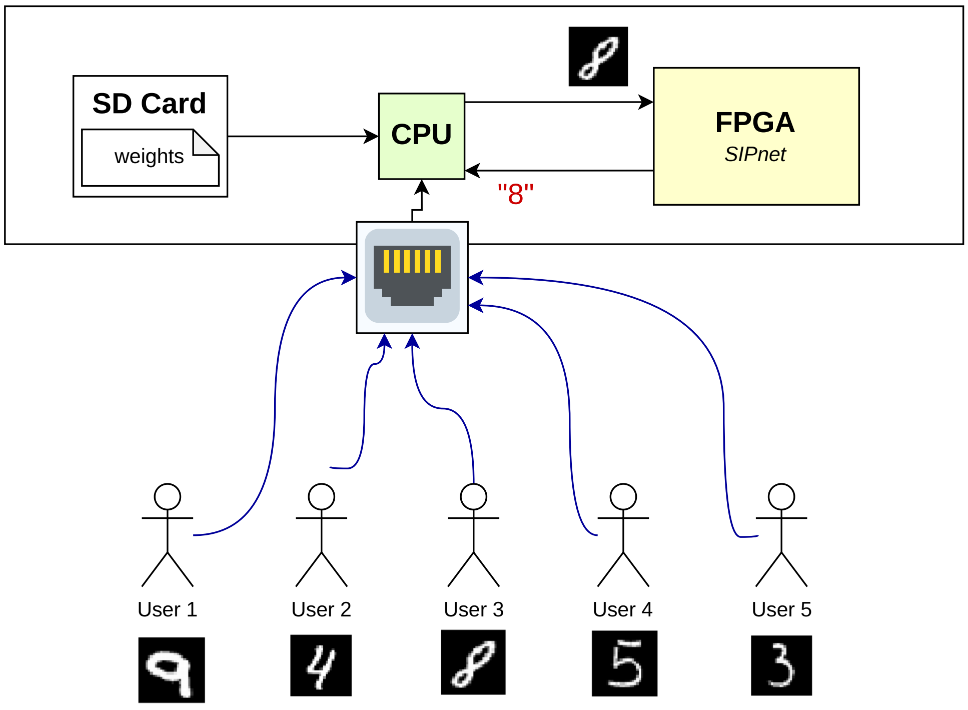

Figure 5: System overview

The FPGA which runs *SIPnet* will be offered to customers as a service on the web. Customers can submit an image from the MNIST dataset and retrieve the respective digit via ethernet. However, sometimes the system is very busy and a lot of customers might be submitting a lot of images at the same time. Create mechanisms to handle a lot of user requests, i.e., every user should get the classification result eventually.

- Create a web server which allows a single user to connect to the system and upload an MNIST image. Classify the image (as done in part 2a) and send the result back to the user. (10P)
- Extend the web server such that multiple users can submit MNIST images at the same time. Every user must eventually get their classification result. Create a test setup to demonstrate the functionality of the system in a multi-user environment. (7P)
- Is the resulting system secure? If no, which attacks would be possible? How can these attacks be prevented? Note down at least three ideas in a file called *security.md* (3P)

3 Submission

1. Export your block design within Vivado: write_bd_tcl −force bd.tcl
2. Commit bd.tcl and your constraints file (base.xdc)
3. Commit your custom IP cores.

4. Commit all the relevant software (device driver, bare metal program, ...)
5. Add a readme including any other relevant information.
6. Tag your submission:

```
git tag Project2[a|b]
git push --tags
```

7. Choose a time slot for the exercise interview (will be sent out via E-mail)

References

[1] Mokhtar Aboelaze. *Verilog Review and Fixed Point Arithmetics*. 2012. URL: https://www.eecs.yorku.ca/course_archive/2011-12/W/4210/L3.pdf.

[2] Inc. Advanced Micro Devices. *LogiCORE IP Block Memory Generator v7.1*. 2012. URL: https://docs.xilinx.com/v/u/en-US/blk_mem_gen_ds512.

[3] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized Neural Networks". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett. 2016, pp. 4107–4115.

[4] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proc. IEEE* 86.11 (1998), pp. 2278–2324.

[5] Yann LeCun, Corinna cortes, and Christopher J.C. Burges. 1998. URL: http://yann.lecun.com/exdb/mnist/.

[6] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit K. Mishra, Ganesh Venkatesh, and Debbie Marr. "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC". In: *2016 International Conference on Field-Programmable Technology, FPT 2016, Xi'an, China, December 7-9, 2016*. Ed. by Yuchen Song, Shaojun Wang, Brent Nelson, Junbao Li, and Yu Peng. IEEE, 2016, pp. 77–84.

[7] Hayden So. *Introduction to Fixed Point Number Representation*. 2006. URL: https://inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html.

[8] Randy Yates. *Fixed-Point Arithmetic: An Introduction*. 2007. URL: https://courses.cs.washington.edu/courses/cse467/08au/labs/l5/fp.pdf.