UNIVERSITY OF SPLIT

FACULTY OF ELECTRICAL ENGINEERING, MECHANICAL ENGINEERING AND NAVAL ARCHITECTURE

PROJECT REPORT

SIMPLE NEURAL NETWORKS ON FPGA

Petar Kaselj

Marijan Šimundić Bendić

Split, July 2022

[1. ABSTRACT 1](#_Toc108572583)

[2. GENERAL ARCHITECTURE 2](#_Toc108572584)

[2.1. UART 2](#_Toc108572585)

[2.1.1. BASICS OF UART 2](#_Toc108572586)

[2.1.2. TRANSMITTER IMPLEMENTATION 3](#_Toc108572587)

[2.1.3. RECEIVER IMPLEMENTATION 4](#_Toc108572588)

[2.1.4. UART TOP MODULE 4](#_Toc108572589)

[2.2. NEURAL ACCELERATOR 6](#_Toc108572590)

[2.2.1 Multiply and Accumulate (MAC) Core 7](#_Toc108572591)

[2.2.2 Address Generator 9](#_Toc108572592)

[2.2.3 Control Unit 11](#_Toc108572593)

[2.2.4 SI UPSCALER 12](#_Toc108572594)

[2.2.5 SI MPY 13](#_Toc108572595)

[2.2.6 SI DOWNSCALER QUANT 14](#_Toc108572596)

[2.2.7 Bus Arbiter 16](#_Toc108572597)

[2.2.8 Neuron DP RAM 18](#_Toc108572598)

[2.2.9 Weight ROM 19](#_Toc108572599)

[2.2.10 Instruction RAM 20](#_Toc108572600)

[3. MODEL TRAINING 21](#_Toc108572601)

[4. QUANTIZATION 22](#_Toc108572602)

[5. USER GUIDE 24](#_Toc108572603)

[5.1. UPLOADING THE NEURAL NETWORK 24](#_Toc108572604)

[5.2. IMAGE PREPARATION AND TRANSMISION 25](#_Toc108572605)

[6. KNOWN BUGS 26](#_Toc108572606)

[7. TIPS 27](#_Toc108572607)

[SPECIFICATIONS 28](#_Toc108572608)

[NEURON DP RAM INITIALIZATION FILE 28](#_Toc108572609)

[INSTRUCTION RAM INITIALIZATION FILE 28](#_Toc108572610)

[WEIGHTS ROM INTIALIZATION FILE 29](#_Toc108572611)

[WEIGHTS CSV FILE 29](#_Toc108572612)

[ORIGINAL IMAGE FILE 29](#_Toc108572613)

[SERIALIZED IMAGE FORMAT 30](#_Toc108572614)

[RESOURCES 31](#_Toc108572615)

[SOURCE CODE 31](#_Toc108572616)

[EXTERNAL REFERENCES 31](#_Toc108572617)

# ABSTRACT

Field Programmable Gate Arrays (FPGAs) plays an increasingly important role in data sampling and processing industries due to its highly parallel architecture, low power consumption, and flexibility in custom algorithms. All this makes them perfect candidates for executing neural networks on certain low power devices, such as security cameras.

In this paper a basic neural network was implemented on the Xilinx Spartan3E. Its task is to identify a handwritten digit on an image of the resolution 28x28 pixels (classic MNIST dataset). Due to the resource constraints only the most basic of neural networks was implemented with only one fully connected layer.

# GENERAL ARCHITECTURE

The device is programmed to read grayscale image from host controller over UART, evaluate the image using OCR neural network (by the means of low precision general matrix multiplication) and send the probabilities for each digit (0-9) back to the host controller over UART.

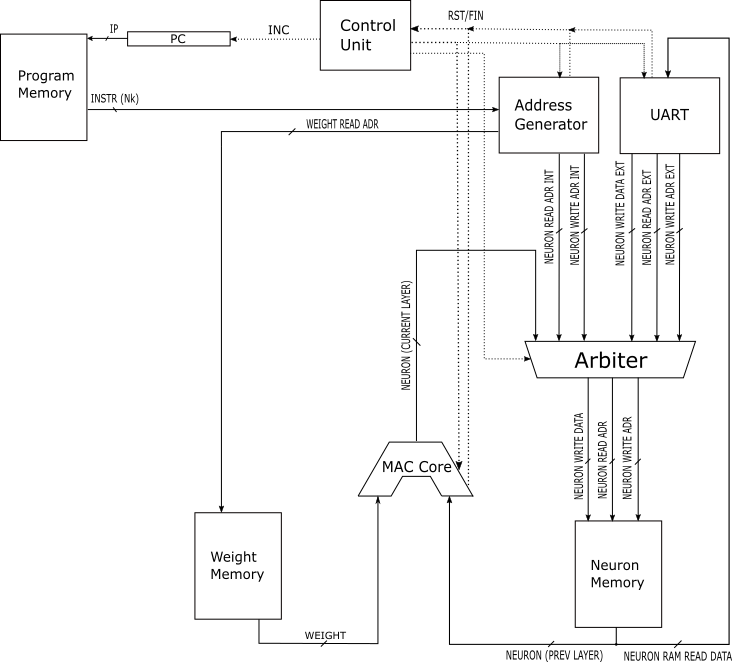


Image 2.1 General Schematic with External Interface

## UART

### 2.1.1. BASICS OF UART

UART (Universal Asynchronous Receiver Transmitter) is one of the most basic protocols. The protocol is full-duplex (a device can transmit and receive at the same time), asynchronous (the two devices don’t share a clock signal). It uses only two wires, RX (receive) and TX (transmit). The devices are connected as shown on image 2.1.1.1.

Chart, line chart

Description automatically generated

*Image 2.1.1. UART connections*

According to the UART standard each device must keep its TX line high when in idle. The data is sent in frames of ten bits as shown on image 2.1.1.2. START bit is always low, followed by eight data bits, and ends with a STOP bit which is always high.



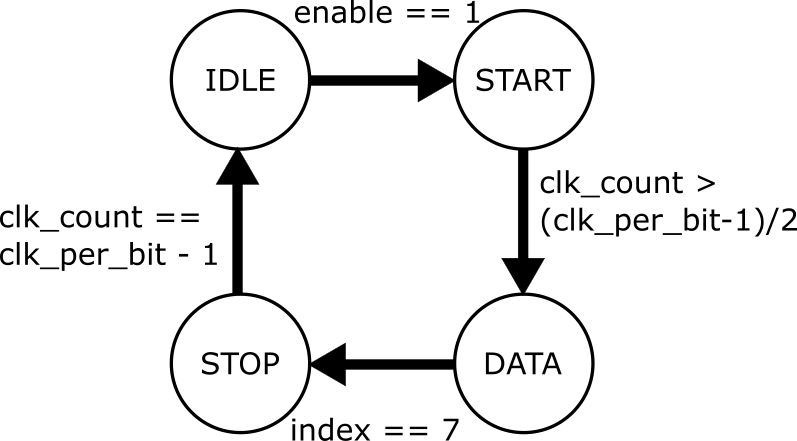
*Image 2.1.2. UART frame*

The transmission speed is defined as baud rate by a formula:

The clock speed of a UART device is usually much higher than the baud rate. The parameter clocks per bit is usually calculated after baud rate is chosen.

### 2.1.2. TRANSMITTER IMPLEMENTATION

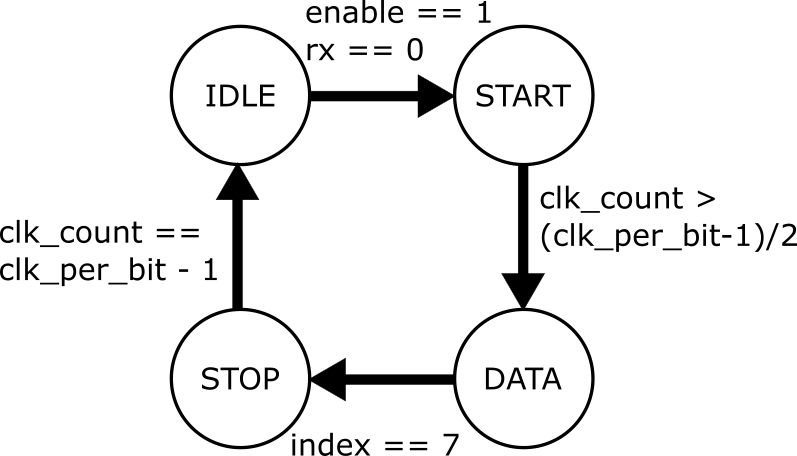
|  |  |  |
| --- | --- | --- |
| NAME | I/O | DESCRIPTION |
| clk | I | 50 MHz clock signal |
| enable | I | Set high when ready to transmit |
| data\_in | I | Data to send |
| tx | O | Connected to RS-232 port |
| done | O | High for one clock cycle when a frame is sent |



*Image 2.1.2.1. TX state machine diagram*

### 2.1.3. RECEIVER IMPLEMENTATION

|  |  |  |
| --- | --- | --- |
| NAME | I/O | DESCRIPTION |
| clk | I | 50 MHz clock signal |
| enable | I | Set high when ready to receive |
| data\_out | O | Data buffer |
| rx | I | Connected to RS-232 port |
| done | O | High for one clock cycle when a frame is received |



*Image 2.1.2.2. RX state machine diagram*

### 2.1.4. UART TOP MODULE

|  |  |  |
| --- | --- | --- |
| NAME | I/O | DESCRIPTION |
| clk | I | 50 MHz clock signal |
| rx | I | Connected to RS-232 port |
| tx | O | Connected to RS-232 port |

Top module rests in idle, awaiting RX activity. After receiving 784 bytes (size of image) it sets the reset signal low which tells the Accelerator module to start the calculation. When finished, Accelerator sets the finished signal high. The top module then proceeds to read the results from memory starting from address *result\_base\_address* and finishing at *result\_base\_address + result\_word\_count*.

## NEURAL ACCELERATOR

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| IP\_DATA\_BUS\_WIDTH |  | Determins max. nr. of neurons in a layer |
| IP\_ADDRESS\_BUS\_WIDTH |  | Determins max. nr. of layers |
| NEURON\_DATA\_BUS\_WIDTH |  | Determins max. value neuron can hold |
| NEURON\_ADDRESS\_BUS\_WIDTH |  | Determins max. nr. of neurons in a layer |
| WEIGHTS\_DATA\_BUS\_WIDTH |  | Determins max. value of weight |
| WEIGHTS\_ADDRESS\_BUS\_WIDTH |  | Determins max. nr. of weights in a model |
| neuron\_ram\_write\_adr\_ext | I | ADR where next image byte will be written |
| neuron\_ram\_read\_adr\_ext | I | ADR from where next result will be read |
| neuron\_ram\_write\_data\_ext | I | Image byte which will be written next |
| neuron\_ram\_wr\_en\_ext | I | Write enable signal |
| finished | O | Accelerator finished |
| neuron\_ram\_read\_data\_ext | O | Next result from accelerator (NN output) |
| result\_base\_address | O | Starting ADR of results (currently 900) |
| result\_word\_count | O | Number of results of NN (currently 10) |
| clk | I |  |
| reset | I |  |

The task of *NeuralAccelerator* module is to evaluate a given image through programmed neural network and pass the results to the external module.

The image is written to the *Neural\_DP\_RAM* by the external module using *neuron\_ram\_write\_adr\_ext , neuron\_ram\_write\_data\_ext , neuron\_ram\_wr\_en\_ext*. The *NeuralAccelerator* evaluates the image through the neural network by means of low precision general matrix multiplication and writes the results back to the *Neural\_DP\_RAM* . The *NeuralAccelerator* then passes starting address of results in *Neural\_DP\_RAM* using *result\_base\_address* bus and the number of results (outputs of the neural network) using the *result\_word\_count* bus. Using current NN model the *result\_base\_address* always returns 900 (arbitrarily choosen) and the *result\_word\_count* always returns 10 (number of digits).

The external module then reads results one by one using the *neuron\_ram\_read\_adr\_ext* and *neuron\_ram\_read\_data\_ext* buses starting at address [*result\_base\_address*] and ending at address [*result\_base\_address* + *result\_word\_count* - 1].

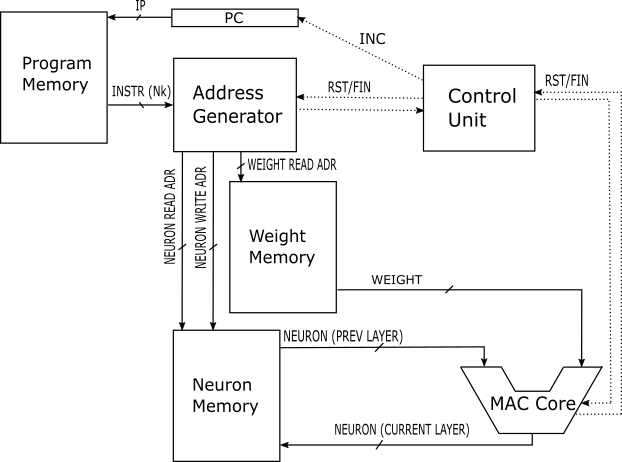


Image 2.2.1 The architecture of *NeuralAccelerator* (without external interface)

### 2.2.1 Multiply and Accumulate (MAC) Core

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| N |  | Bit width of input and output |
| N\_ACC |  | Bit width of accumulator |
| weight | I | Current weight value |
| in | I | Current neuron value |
| oe | I | Output enable (oe=0 sets out to 0) |
| reset | I |  |
| clk | I |  |
| forget | I | Clears accumulator on next clk |
| out | O | Value of output neuron |

The MAC core calculates value of a neuron in layer N+1 by iterating over all neurons from layer N and multiplying them with their respective weights, that connect them to the current neuron in layer N + 1.

The intermediate multiplication and accumulation result is stored in internal *accumulator* register which is passed to *out* when *oe* is active. When a value for a neuron in layer N + 1 is calculated, after all neurons from the layer N are iterated over, *oe* should be set active so that the output is written into *Neuron\_DP\_RAM* and the *forget* signal should be set to active so that internal *accumulator* resets to 0 and the process could be repeated for the next N+1 layer neuron.

Inputs (*in* and *weight*) are upscaled by padding them using module *SI\_UPSCALER* to the internal *accumulator*’s bit width. To downscale the *accumulator* to the *out*’s bit width the *SI\_DOWNSCALER\_QUANT* module is used which, not only downscales the bit width, but also dequantizes the result of matrix multiplication (which will be explained in subsequent chapters).

**NOTE:** Because of internal buffers, the output lags behind input by 2 clock cycles.

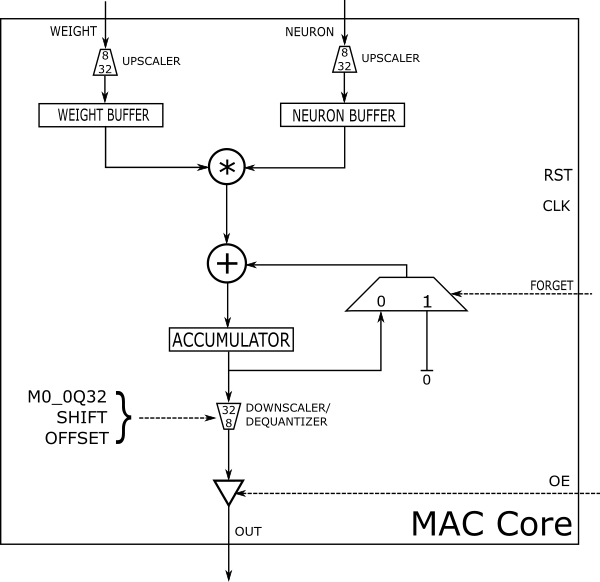


Image 2.2.1.1 MAC Core schematic

### 2.2.2 Address Generator

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| IP\_DATA\_BUS\_WIDTH |  | See table above |
| NEURON\_ADDRESS\_BUS\_WIDTH |  | See table above |
| WEIGHTS\_ADDRESS\_BUS\_WIDTH |  | See table above |
| Nk | I | Number of neurons in layer N + 1 |
| read\_weight\_base\_addr | I | Weights – base read address for layer N |
| read\_neuro\_base\_addr | I | Neurons – base read address for layer N |
| write\_neuro\_base\_addr | I | Neurons – base write address for layer N + 1 |
| clk | I |  |
| reset | I |  |
| read | I | Next layer signal (Read Nk) |
| finished | O | Finished generating ADR for layer N + 1 |
| neuron\_finished | O | Finished generating ADR for neuron in N + 1 |
| weight\_read\_addr | O | Current read address of weight |
| neuro\_read\_addr | O | Current read address of neuron in layer N |
| neuro\_write\_addr | O | Current write address of neuron in layer N+1 |
| current\_layer\_size | O | Layer N + 1 size |
| previous\_layer\_size | O | Layer N size |

The *AddressGenerator* module takes layer sizes as input and according to them generates all required addresses for matrix multiplication (forward propagation in neural network).

**NOTE:** After reset, *AddressGenerator* needs to be fed sizes of first two layers: N and N + 1 respectively. After that it requires only the size of the next layer (N + 1) because it takes the current layer size and sets it as the previous layer size while the current layer size is populated by new layer size (*Nk*).

The active *read* flag signals the *AddressGenerator* to read next layer size from *Nk*, while switching the current layer size to previous layer size at next *clk* clock cycle. The *read* flag should be held active only for 1 clock cycle except in case of reset (see note above).

After loading layer sizes, the addresses will start being generated according to the table below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ADDRESS** | **INCREMENT**  **ON** | **RESET ON** | **START****[[1]](#footnote-1)** | **END1** | **OVERFLOW**  **FLAG** |
| weight\_read\_addr | clk | - | 0 | Nk\*Nk\_1 | - |
| neuro\_read\_addr | clk | neuron\_finished | 0 | Nk\_1 | neuron\_finished |
| neuro\_write\_addr | neuron\_finished | - | 0 | Nk | finished |

Table 2.2.2.1 Address generation scheme

For each neuron in layer N + 1 ( starting from address[*neuro\_write\_base\_addr*] and ending at [*neuro\_write\_base\_addr + Nk* - 1] ) address is generated to fetch every neuron in layer N (starting at address [*neuro\_read\_base\_addr*] and ending at [*neuro\_read\_base\_addr + Nk\_1* - 1] ) along with its weight ( starting from address[*weight\_read\_base\_addr*] and ending at [*weight\_read\_base\_addr + Nk\*Nk\_1* - 1] ) that connects it to the current neuron in layer N + 1.

After a neuron in layer N + 1 is finished (by generating addresses for all connected neurons in previous layer – layer N – and respective weights) the *neuron\_finished* flag is driven active for 1 clock cycle and the counters are incremented/reset according to table above (Table 2.2.2.1).

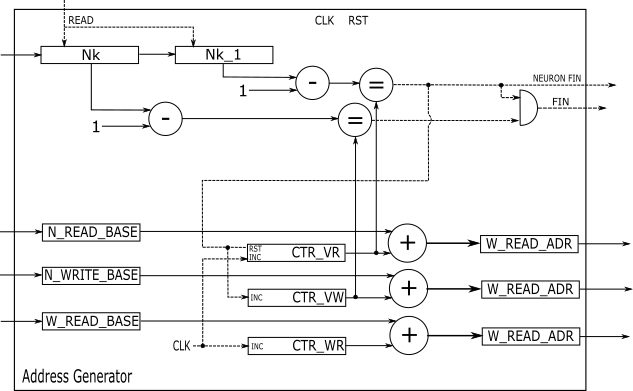


Image 2.2.2.1 *AddressGenerator* schematic

### 2.2.3 Control Unit

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| clk | I |  |
| reset | I | Global reset flag |
| forget | I | See *MAC\_Core* module |
| AG\_rst | O | *AddressGenerator reset* flag |
| AG\_read | O | *AddressGenerator* *read* flag |
| ALU\_rst | O | *MAC\_Core reset* flag |

*ControlUnit* module controls reset flag distribution according to the global state of the system. It takes care of *reset* timing and synchronization among different modules.

**NOTE:** Another (simpler) way the timing issues have been taken care of is by using additional registers in *NeuralAccelerator* suffixed with “\_1” and/or “\_2” etc. to delay signals by declared number of clock cycles e.g. *finished\_1*, *finished\_2*, etc.

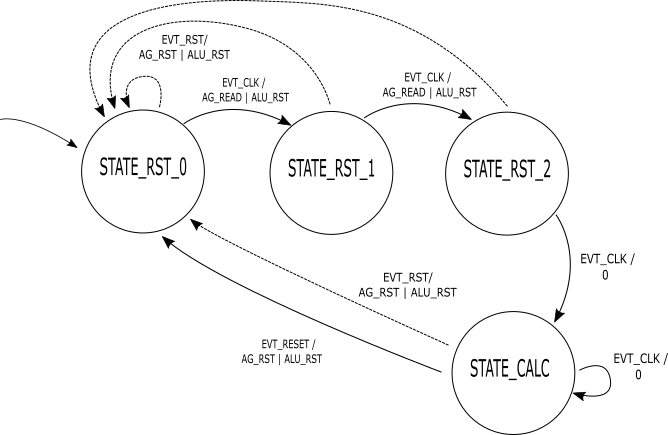


Image 2.2.3.1 *ControlUnit* FSM representation

### 2.2.4 SI UPSCALER

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| N\_IN |  | *in* bitwidth |
| N\_OUT |  | *out* bitwidth |
| in | I |  |
| out | O |  |

The *SI\_UPSCALER* module takes N\_IN bit wide input *in* and pads it with zeros up to N\_OUT bits on MSB side and relays it to the output *out*. The scaling is sign sensitive representing and expecting negative numbers to be coded using two’s complement.

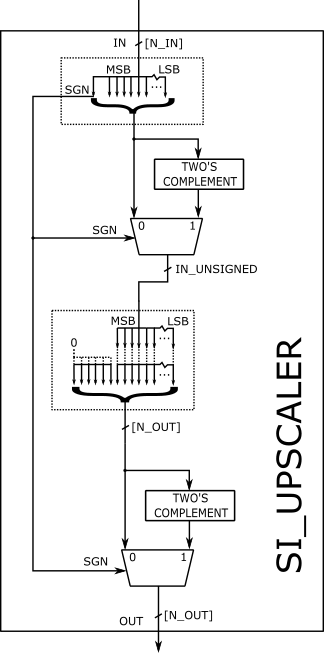


Image 2.2.4.1 *SI\_UPSCALER* schematic

### 2.2.5 SI MPY

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| N |  | bitwidth |
| A | I |  |
| B | I |  |
| A\_MPY\_B | O | = A \* B |

***SI\_MPY* module takes two N bit numbers *A* and *B* and performs** signed **multiplication on them** rounding the result to N bits **by discarding extra MSB bits.**

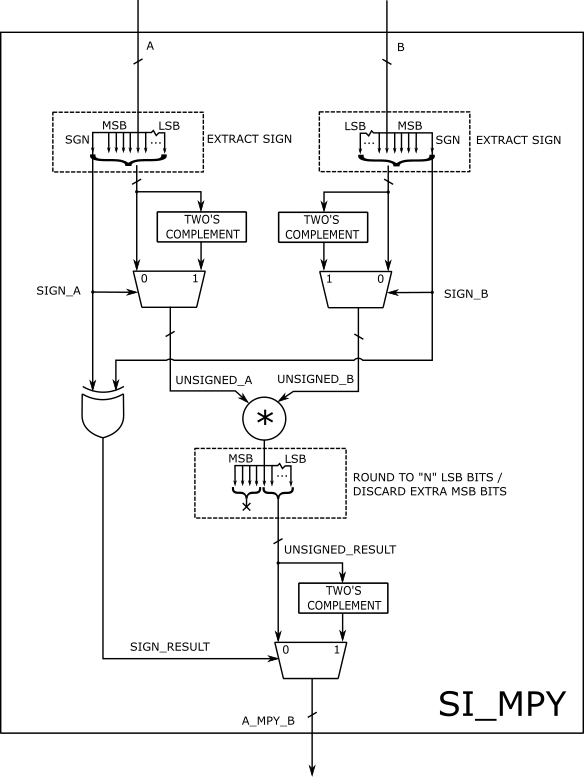


Image 2.2.5.1 *SI\_MPY* schematic

### 2.2.6 SI DOWNSCALER QUANT

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| N\_IN |  | *in* bitwidth |
| N\_OUT |  | *out* bitwidth |
| M0\_0Q32 |  | See „Quantization“ chapter |
| SHIFT |  | See „Quantization“ chapter |
| OFFSET |  | See „Quantization“ chapter |
| in | I |  |
| out | O |  |

*SI\_DOWNSCALER\_QUANT* module performs 2 functions :

* Dequantization
* Downscaling

The process of dequantization is performed by multiplying the *in* number with number

so that the N\_IN LSB bits of product *in \* M* are passed to the *out.*

Since M is generally smaller than 1, the multiplication is implemented using two’s complement integer multiplication and division by shifting.

The number M0 is coded in the parameters in 0Q32 format (M0\_0Q32) along with SHIFT parameter. The process of dequantization is implemented according to the following expression:

**NOTE:** In implementation the numbers are first scaled to 2 \* N\_IN bitwidth to accommodate for the full result of multiplication (so as not to overflow). Note that the result needs to be shifted to right (divided) additionally by 32 bits (to round to the nearest integer i.e. to discard 32 fractional bits from 0Q32 multiplication).

**NOTE:**The division (shifting) is broken down into two parts[[2]](#footnote-2): shifting by (SHIFT – 1) and then shifting by 1. This is used to take the first fractional bit from the result of the first shift-division. The fractional bit flag is then used later in rounding to avoid down-rounding.

After dequantization, (N\_IN – N\_OUT) MSB bits are discarded (taking care of signedness) to get the output with bitwidth of N\_OUT, in the process of downscaling.

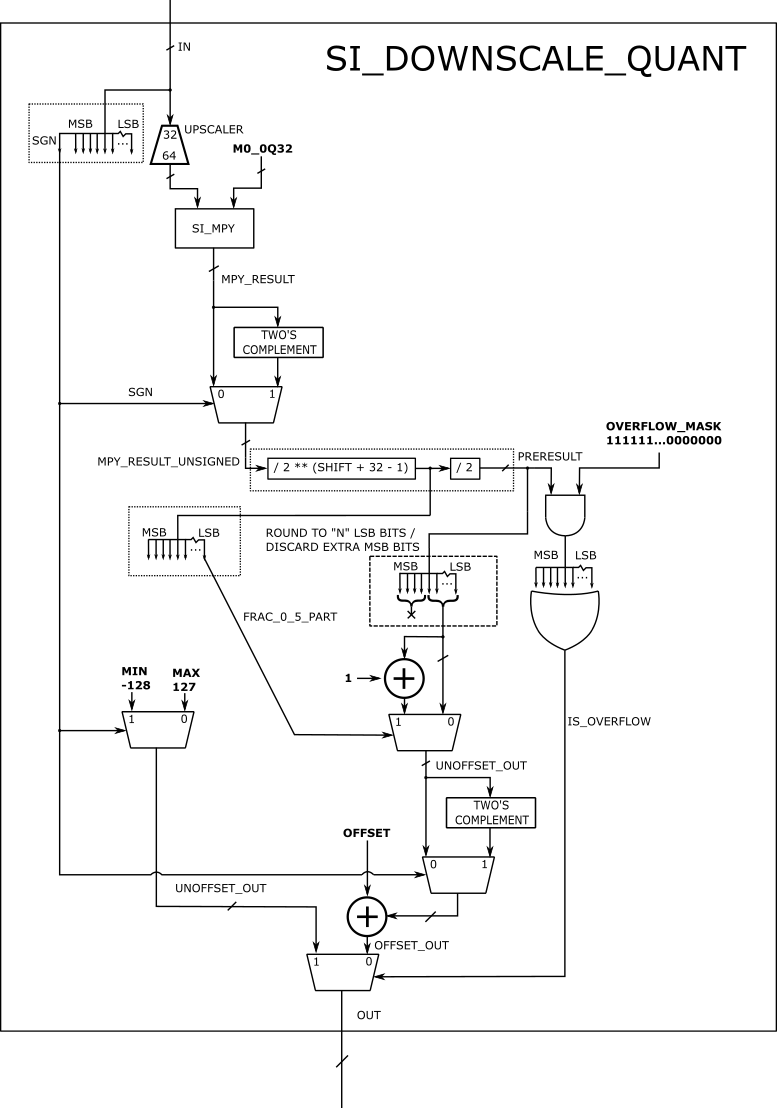


Image 2.2.6.1 *SI\_DOWNSCALER\_QUANT* implementation

### 2.2.7 Bus Arbiter

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| DATA\_BUS\_WIDTH |  | See table for *NeuralAcceletor* (NEURON...) |
| ADDRESS\_BUS\_WIDTH |  | See table for *NeuralAcceletor* (NEURON...) |
| neuron\_read\_address\_ext | I | External interface for read address bus |
| neuron\_read\_address\_int | I | Internal interface for read address bus |
| neuron\_write\_address\_ext | I | External interface for write address bus |
| neuron\_write\_address\_int | I | Internal interface for write address bus |
| neuron\_write\_data\_ext | I | External interface for write data bus |
| neuron\_write\_data\_int | I | Internal interface for write data bus |
| neuron\_write\_enable\_ext | I | External interface for write enable flag |
| neuron\_write\_enable\_int | I | Internal interface for write enable flag |
| select\_external | I |  |
| neuron\_read\_address | O | See *Neuron\_DP\_RAM* |
| neuron\_write\_address | O | See *Neuron\_DP\_RAM* |
| neuron\_write\_data | O | See *Neuron\_DP\_RAM* |
| neuron\_write\_enable | O | See *Neuron\_DP\_RAM* |

*BusArbiter* controls access to *Neuron\_DP\_RAM* ‘s input buses and signals among multiple sources. It has a set of outputs connected to *Neuron\_DP\_RAM*’s inputs and *external and internal* sets of input buses and signals.

To prevent clashes (electrical faults) when driving these buses, the *BusArbiter* multiplexes/arbitrates *internal* and *external* interface to *Neuron\_DP\_RAM.*

When *select\_external* flag is active, the buses and signals suffixed with “\_*ext*”(indicating *external* interface) are connected to the output buses. If *select\_external* is not active other set of buses and signals, the ones suffixed with “\_*int*” (indicating *internal* interface) are connected to the output set of buses.

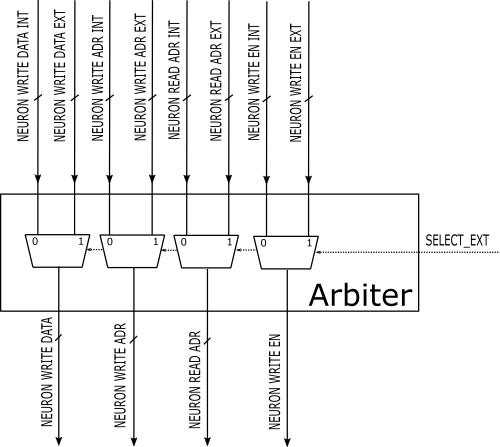


Image 2.2.7.1 *BusArbiter* schematic

### 2.2.8 Neuron DP RAM

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| DATA\_BUS\_WIDTH |  | See table for *NeuralAcceletor* (NEURON\_DATA\_...) |
| ADDRESS\_BUS\_WIDTH |  | See table for *NeuralAcceletor* (NEURON\_ADDR\_...) |
| read\_address | I |  |
| write\_address | I |  |
| write\_data | I |  |
| oe | I |  |
| wre | I | Write enable |
| clk | I |  |
| read\_data | O |  |

*Neuron\_DP\_RAM* is the module in which the image is written at the beginning. It is then used by *NeuralAccelerator* to store intermediate results for neural network forward-propagation including final results.It is a form of dual port RAM since it allows simultaneous read and write operations along with instantaneous propagation of data from *write\_data* to *read\_data* in case *read\_address* is same as *write\_address*. The access to *Neuron\_DP\_RAM* is controlled by *BusArbiter*.

DATA\_BUS\_WIDTH must be large enough for all values of neural network (the original one trained on host machine) to be stored properly.

ADDRESS\_BUS\_WIDTH must be large enough to address all neurons in **one layer**.

*Neuron\_DP\_RAM* is **divided into two parts** defined by NEURO\_RW\_BASE\_LOW and NEURO\_RW\_BASE\_HIGH in *NeuralAccelerator* module. The division is necessary so that the RAM can be used for storing intermediate results:

* The images are first loaded starting at NEURO\_RW\_BASE\_LOW
* Values of neurons of next layer are stored at NEURO\_RW\_BASE\_HIGH
* Values of neurons of the following layer are stored starting at NEURO\_RW\_BASE\_LOW overwriting the image (which is not needed anymore)
* Next layer is stored starting at NEURO\_RW\_BASE\_HIGH
* etc.

This way RAM only needs about double the size of the largest layer to compute and store all intermediate results.

For data format specification refer to the related chapter.

### 2.2.9 Weight ROM

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| DATA\_BUS\_WIDTH |  | See table for *NeuralAcceletor* (WEIGHT\_DATA\_...) |
| ADDRESS\_BUS\_WIDTH |  | See table for *NeuralAcceletor* (WEIGHT\_ADDR\_...) |
| address | I |  |
| data | O |  |
| enable | I |  |

*Weight\_ROM* is a module containing a form of Read-Only Memory storing all weights of a neural network.

DATA\_BUS\_WIDTH must be large enough for all weights of neural network (the original one trained on host machine) to be stored properly.

ADDRESS\_BUS\_WIDTH must be large enough to address all neurons **GLOBALLY** (all neurons of a neural network).

For data format specification refer to the related chapter.

### 2.2.10 Instruction RAM

|  |  |  |
| --- | --- | --- |
| **NAME** | **I/O** | **DESCRIPTION** |
| DATA\_BUS\_WIDTH |  | See table for *NeuralAcceletor* (IP\_DATA\_...) |
| ADDRESS\_BUS\_WIDTH |  | See table for *NeuralAcceletor* (IP\_ADDR\_...) |
| address | I |  |
| data | O |  |
| enable | I |  |

*Instruction\_RAM* is a module containing a form of **Read-Only Memory** (misleading module name, TODO) storing all sizes of layers in a neural network.

DATA\_BUS\_WIDTH must be large enough to represent size of the **largest layer**.

ADDRESS\_BUS\_WIDTH must be large enough to address **all layers**.

For data format specification refer to the related chapter.

# MODEL TRAINING

The Optical Character Recognition network was trained in python using the Keras and TensorFlow libraries. The topology of the network is extremely simple, input and output layer with no hidden layers (Image 3.1.). The reason for this is limited resources on the FPGA chip.

Background pattern

Description automatically generated

*Image 3.1. Neural network*

After training, in order to simplify the Verilog code, the network had to be quantized to only use signed 8-bit integers.

# QUANTIZATION

The default numerical datatype used in neural networks in *Tensorflow* and *Keras* is *float32.* Floating point arithmetic is more flexible but computationally more expensive than integer arithmetic. To improve performance in low-end embedded devices, the processes of quantization and dequantization are introduced to neural networks.

The forward propagation in neural network can be decomposed into series of multiple matrix multiplications and additions[[3]](#footnote-3). One iteration of forward-propagation (matrix multiplication) can be represented as follows:

where *n* represents value of a neuron (in layer N or N+1) and *w* represents value of weight connecting those two neurons. All values are floating-point values.

If we decompose each value (for *n* and *w*) as follows:

choosing arbitrary **floating-point** value for *S* and arbitrary **integer** value for Z such that *q* results in (or rounds nearly enough to) an **integer** value. We can then rewrite the original equation as:

further evolving into[[4]](#footnote-4):

where:

We can see that the matrix multiplication algorithm (forward propagation) decomposed into **integer** multiplication and addition with one **floating-point** multiplication **per matrix multiplication** (which can be reduced to one per whole forward propagation by choosing correct *S* parameters).

Additionally, the sole floating-point multiplication can be decomposed into **fixed-point multiplication and division by shifting** by choosing arbitrary parameters *M0* and *SHIFT* such that:

For additional details refer to the “*SI DOWNSCALER QUANT”* chapter and to listed references.

# USER GUIDE

* 1. **UPLOADING THE NEURAL NETWORK**

After training the network in *ModelTrainer\_MNIST.py* one must open the **quantized** network in *Netron* (Image 5.1.1.), available at:

<https://github.com/lutzroeder/netron>

Graphical user interface

Description automatically generated with medium confidence

*Image 5.1.1. Netron user interface*

From Netron one must save the weights of the neural network (in .*npy* format).

After saving the network weights as *.npy* one must process the *.npy* file into a .csv file using *npyToCSV.py* script. The *.csv* file is the processed through *readmem\_parser.py* in order to get the files ready to upload to the FPGA *(.txt* files, which are programmed using *readmemh*).

All the processed files are then copied to the Xilinx project directory. Upon programming one can expect the FPGA chip to send a byte over UART with the value 0x00 (this happens for unknown reasons).

* 1. **IMAGE PREPARATION AND TRANSMISION**

Input images must be in .bmp monochrome format and have a resolution of exactly 28x28 pixels. The image is then processed through *Evalutor.py* which gives the output of the **non-quantized** network as a reference and saves the image in *.bin* format which is needed to send the image over UART.

The *.bin.* image is then sent over an UART to USB cable to the FPGA using *Realterm* (Image 5.2.1.), available at:

<https://realterm.sourceforge.io/>

Graphical user interface, text

Description automatically generated with medium confidence

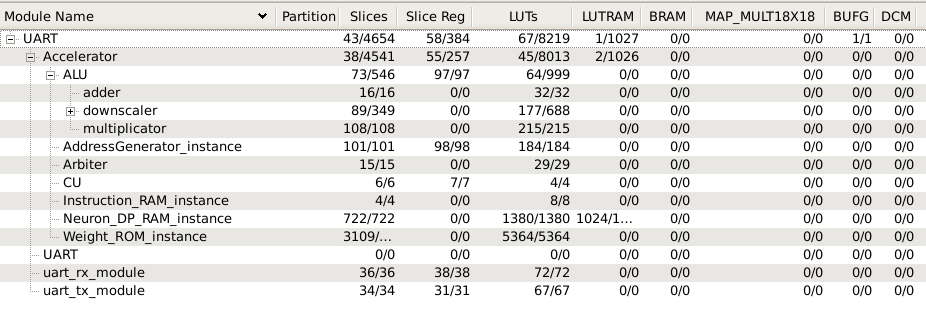
*Image 5.1.1. Realterm*

In *Realterm*, the correct port must be selected, *baud rate* set at 9600 (unless the UART modules are modified), *Display as* set as int8, and the correct file chosen in the *Dump File to Port* box.

After transmission, when the FPGA calculates the results, eleven bytes are received. The first byte can be ignored (always the same as the second byte). The following ten bytes correspond to digits 0-9. Each number represents the certainty that the sent image contains that digit. The greater the number, the more likely it is the corresponding digit.

1. **KNOWN BUGS**

* Upon programming the FPGA, it sends 0x00
  + Probably transient due to electrical changes during programming
* The FPGA sends the number corresponding to zero twice
  + Bug in logic of *UART* module, probably counter *off-by-one* error
* After each image is sent the FPGA must be reprogrammed
  + Not all values are reset properly in *NeuralAccelerator* on second *reset*
  + Specifically high and low base write addresses in *NeuralAccelerator*
* In rare cases some probabilities calculated by the network running on FPGA don’t exactly match the expected values from *neural\_network\_simulator.py*
  + Possibly a bug in *SI\_DOWNSCALER\_QUANT* module in last always block when following conditions are met:
    - is\_overflow = 0
    - preresult = near 127 or -128 (under/overflowing or at lower/higher end)
  + So that addition of OFFSET overflows the result
* Unexpectedly high utilization of LUTs (Image 6.1.)
  + Memory was implemented using logic blocks instead of built-in memory modules



*Image 6.1. Resource utilization*

1. **TIPS**

For easier debugging look up some of the following:

* Chipscope – use the official resources available and the resources available at your faculty
* Simulations
  + Add additional signals to the waveform viewer (internal variables, not just input and output signals)
  + By right clicking the variable in waveform view in category *radix* change the display type (binary, hexadecimal, int, unit, etc.)
  + Memory view – for inspecting memory

# SPECIFICATIONS

## NEURON DP RAM INITIALIZATION FILE

**NOTE:** Unused!!! Was used in early stages of debugging. Only for informational purposes when inspecting memory in debugger.

Used by *Neuron\_DP\_RAM* in initialization block during synthesis to initialize RAM.

|  |  |
| --- | --- |
| **NAME** | **VALUE** |
| File extension | .txt |
| File count | 1 |
| Data order | Each RAM location at separate line stored sequentially |
| Data type | Integer ( bitwidth = NEURON\_DATA\_BUS\_WIDTH) |
| Data encoding | Binary (negatives represented using two's complement) |
| Padding value | 0 |
| Data count | 2 ^ (NEURON\_ADDRESS\_BUS\_WIDTH) |

## INSTRUCTION RAM INITIALIZATION FILE

Used by *Instruction\_RAM* to initialize memory in initial block. Represents sizes of layers in neural network.

|  |  |
| --- | --- |
| **NAME** | **VALUE** |
| File extension | .txt |
| File count | 1 |
| Data order | Each RAM location at separate line stored sequentially |
| Data type | Unsigned integer ( bitwidth = IP\_DATA\_BUS\_WIDTH) |
| Data encoding | Hexadecimal |
| Padding value | Max integer (all ones i.e. 2 ^ (Data count) - 1) |
| Data count | 2 ^ (IP\_ADDRESS\_BUS\_WIDTH) |

## WEIGHTS ROM INTIALIZATION FILE

Used by *Weight\_ROM* to initialize memory in initial block. Represents weights of neural network.

Weights are stored in **ROW MAJOR** order(relative to WEIGHTS CSV FILE)

|  |  |
| --- | --- |
| **NAME** | **VALUE** |
| File extension | .txt |
| File count | 1 |
| Data order | Each RAM location at separate line stored sequentially |
| Data type | signed integer (width = WEIGHT\_DATA\_BUS\_WIDTH) |
| Data encoding | Binary (negatives represented using two's complement) |
| Padding value | 0 |
| Data count | 2 ^ (WEIGHT\_ADDRESS\_BUS\_WIDTH) |

## WEIGHTS CSV FILE

Used by *readmem\_parser.py* to create weight ROM initialization file.

|  |  |
| --- | --- |
| **NAME** | **VALUE** |
| File extension | .csv |
| File count | One file for weights of two connected layers |
| Data order | Column = layer N, Row = layer N + 1 |
| Data type | Signed integer |
| Data encoding | Decimal |
| Padding value | - |
| Data count | Size of layer N \* Size of layer (N + 1) |

## ORIGINAL IMAGE FILE

Starting image file from which the serialized image format is created.

|  |  |
| --- | --- |
| **NAME** | **VALUE** |
| File extension | Any bitmap format |
| File count | One for each image of a digit |
| Data order | - |
| Data type | Unsigned 8bit integer (Grayscale image) |
| Data encoding | - |
| Padding value | - |
| Data count | 28 \* 28 = 784 Bytes |

## SERIALIZED IMAGE FORMAT

Image format which the FPGA accepts.

|  |  |
| --- | --- |
| **NAME** | **VALUE** |
| File extension | Any, preferrably .bin |
| File count | One for each image of a digit |
| Data order | Sequential bytes |
| Data type | Signed 8 bit integer (NEURON\_DATA\_BUS\_WIDTH) |
| Data encoding | Bytes |
| Padding value | - |
| Data count | 28 \* 28 = 784 Bytes |

# RESOURCES

## SOURCE CODE

* Python Utilities (<https://github.com/MSimundic/Verilog_NN_utilities>)
* Verilog Source (<https://github.com/pkaselj/Verilog_NeuralNetwork>)

## EXTERNAL REFERENCES

* *TensorFlow* Post-Training Quantization (<https://www.tensorflow.org/lite/performance/post_training_quantization>)
* *TensorFlow* Quantization Specification (<https://www.tensorflow.org/lite/performance/quantization_spec>)
* Quantization Paper [Bo Chen] (<https://doi.org/10.48550/arXiv.1712.05877>)
* General matrix multiplication library used by *TFLite* (<https://github.com/google/gemmlowp>)

1. Relative to respective base address [↑](#footnote-ref-1)
2. See note above. The 32 bit shift is omitted for brevity. [↑](#footnote-ref-2)
3. The neural network in this project doesn’t use biases to further improve performance and resource usage [↑](#footnote-ref-3)
4. The neural network in this project has constants *Z1* and *Z2* set to 0 further simplifying the expression [↑](#footnote-ref-4)