NN@FPGA – A Machine Learning Framework

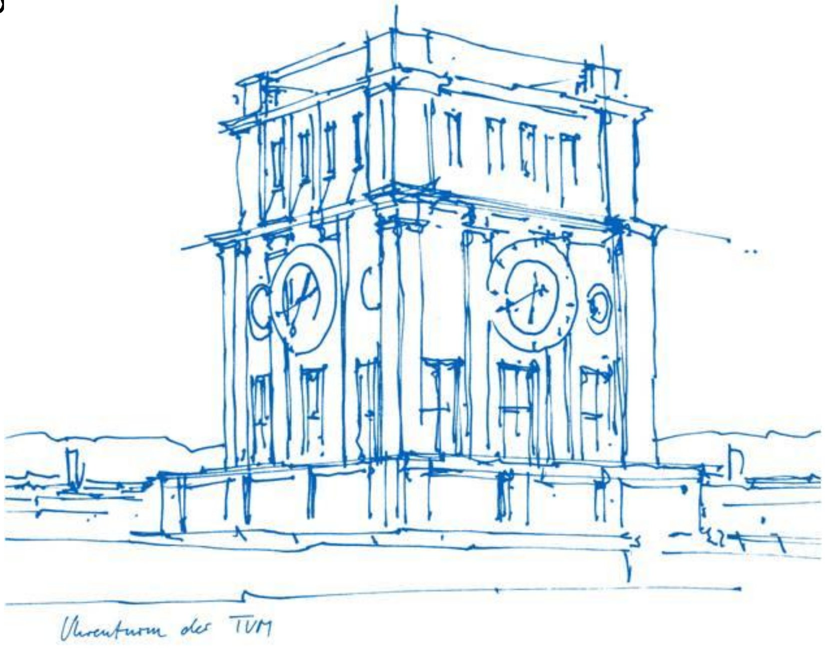
Šimon Růžička, Philipp Wondra

Practical Course – Creation of Deep Learning Methods

Technische Universität München

School of Computation, Information and Technology

Winter semester 2023/24

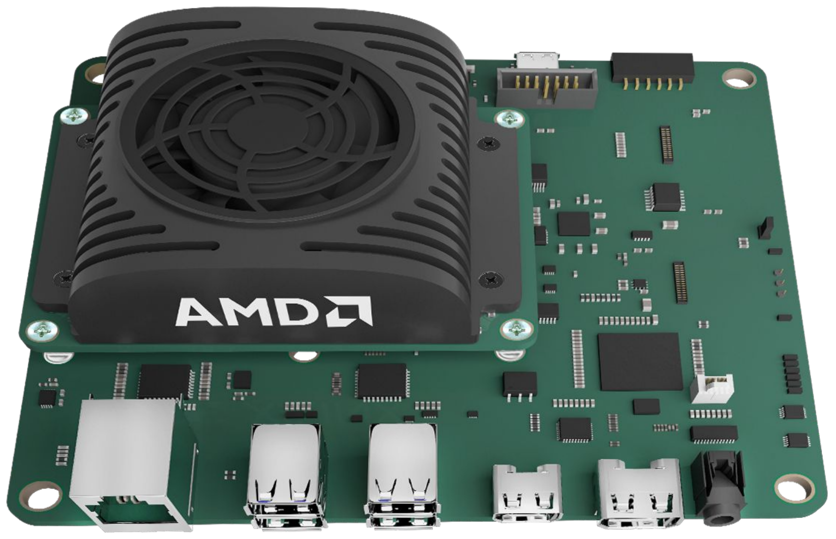


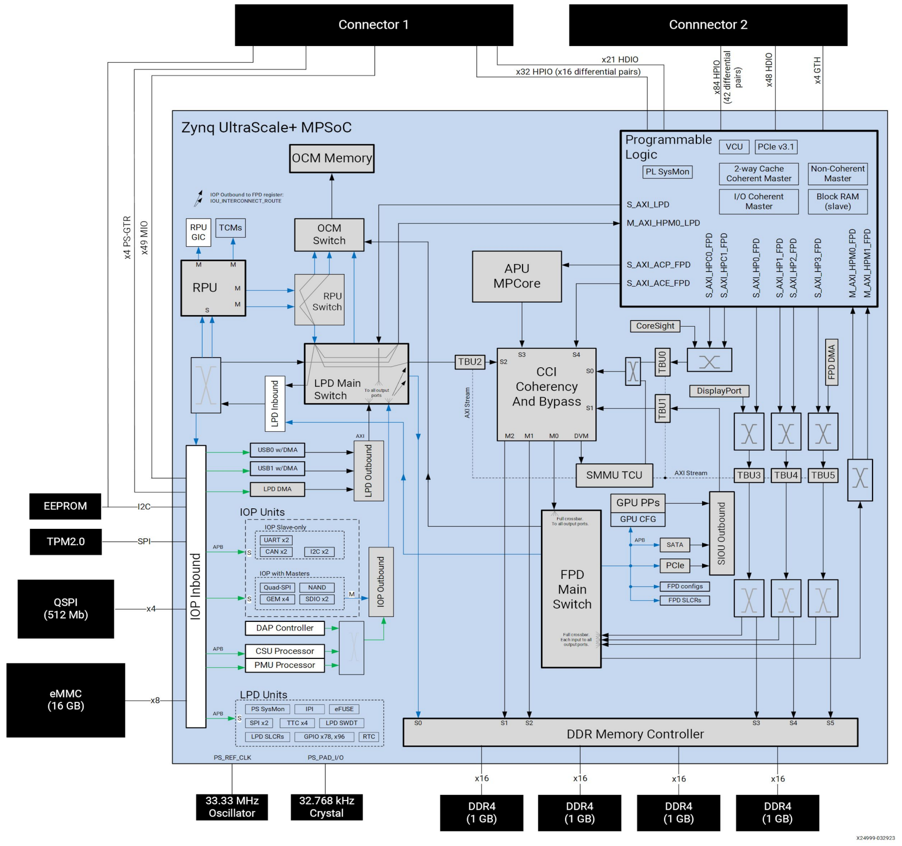
Goals

- Study the use of Hardware Definition Languages, such as Verilog, with special focus towards FPGAs.
- Understand the general layout of an FPGA and an MPSoC, such as Kria KV260 used in our work.
- Create a set of Verilog modules to perform computations with 32-bit floating point numbers.
- Combine these elementary modules into a multi-layer feed-forward neural network.
- Synthesize, implement, and run the HDL modules on the FPGA board.
- Communicate between the CPU and FPGA using instantiated Block RAM modules, mapped in memory.

About the board

Kria KV260 combines ARM Cortex A-53 CPU, conventional DDR4 RAM and interface (USB, Ethernet) with programmable FPGA. We ran standard Ubuntu 22.04 on the CPU.





Acknowledgements

All the listed work was created for the “Creation of Deep Learning Methods“ Practical Course at the Technical University Munich. All external sources are cited where relevant.

Special thanks belongs to Vladimir Golkov from the Computer Vision Group at TUM CIT for supervising our project. We also thank to Professorship Big Geospatial Data Management at TUM ED for letting us use their hardware (Kria board) and technical advice from Xuanshu Luo.

The results of our work will be shared publicly in their final form on https://github.com/ruzicka02/NN.FPGA.

Hardware design modules in Verilog

- Create a set of Verilog modules to perform computations with 32-bit floating point numbers.
- Combine these elementary modules into a multi-layer feed-forward neural network.

File structure

Our repository on [Github - ruzicka02/VerilogNN]() contains multiple directories with their own meaning. Following directories contain source files in Verilog:

- `src/`... Source files, mostly intended as synthesizable modules.
- `sim/`... Testbench files runnable using Icarus Verilog. These are mostly ignored when working with Vivado (replaced by their bd designs).
- `sim_vivado/`... Additional utility modules.

To extract data from program, bigger data sources (such as Neural Network weights) are stored in the `data/` directory.
In addition, following directories are created when running the make command:

- `run/`... Simulation files created by iverilog from simulation testbenches. These files are runnable by vvp simulator (part of iverilog package).
- `vcd/`... VCD waveform files, openable for example in GTKWave. Constructed whenever the files in `run/` are launched.

Brief Verilog introduction

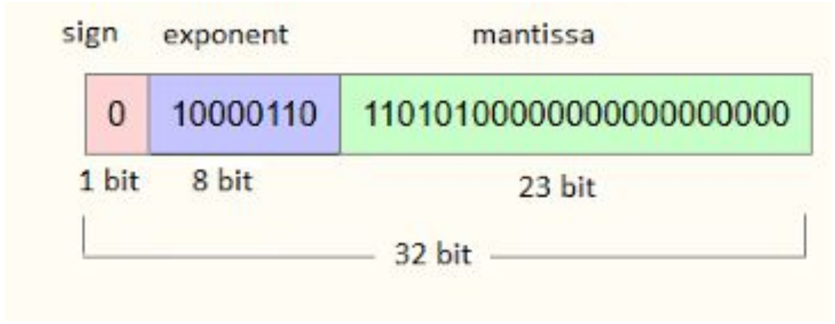
- Hardware Definition Language… unlike classical programming
- all code is executed in parallel, unless constraints are specified
- C-like syntax, unlike VHDL
- structure of Verilog codebase into **modules**… input, output, internal logic
- internal logic… combinatorial (without state) vs. sequential
- more complex tasks achieved by nesting modules into each other

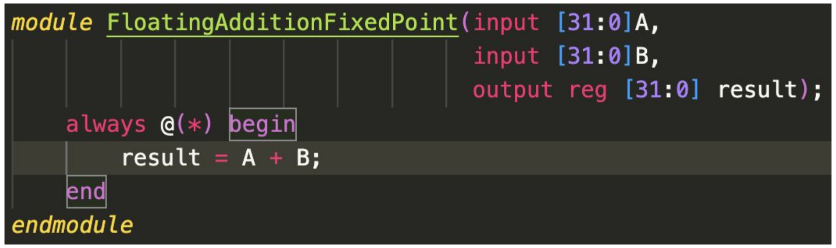
What can we do with a Verilog module?

1. Run in simulator
   - Running on classical CPU, wires and gates are simulated using “variables” and “functions”
   - Discrete simulation time steps, manually set delays
2. Run on physical board
   - Synthesis -> Implementation -> Bitstream
   - Reprogram the FPGA gates to match the created design

Floating point numbers – IEEE-754

- we chose the IEEE-754 Format in order to store the floating point numbers
- this leads to a visibly more complex implementation of basic mathematical operators (such as addition, multiplication or Division)
- a fixed point implementation would store numbers in 32 bit registers and use the canonical operators provided by verilog
- The IEEE-754 requires to separately adjust the sign bit, mantissa and exponent. Nonetheless they can be stored in the same 32 bit register. Just interpretation needs to be adapted





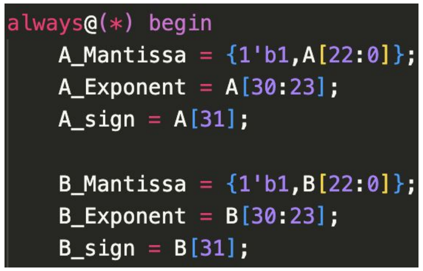
```
module FloatingAdditionFixedPoint(input [31:0]A,
                                  input [31:0]B,
                                  output reg [31:0] result);
    always @(*) begin
        result = A + B;
    end
endmodule
```

Basic floating point operations

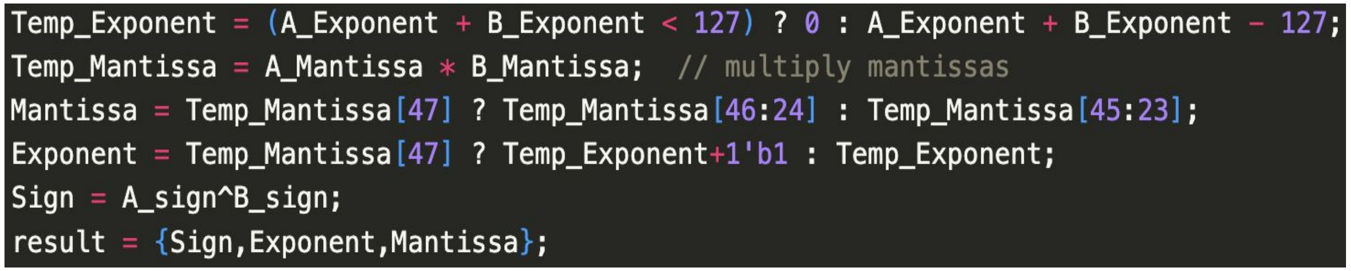
- Operating a neural network necessitates the presence of a correctly functioning floating-point arithmetic.
- although verilog provides a floating point data type (called “real”), this data type can only be used in simulation
- However, as described above, Verilog provides only basic arithmetic operations for integers and, consequently, fixed-point numbers.
- We utilized the operators available here (Github - akilm/FPU-IEEE-754) and addressed some of the existing errors within them (our Pull Request) to enable the required functionality.
- Implemented operations: addition, multiplication, and division.
- Subtraction can be accomplished by adding the second operand with the sign bit inverted.

Basic floating Point Operations - Floating Multiplication

- In the code snippet (left image), it is evident that the floating-point numbers passed in A and B are first decomposed into their components.
- Subsequently, based on these components, all calculations are performed (right image).
- Exponents are added unless overflow occurs.
- Mantissas are multiplied.
- New sign is calculated using XOR.
- Finally, in the variable "result," everything is reassembled in the correct order.



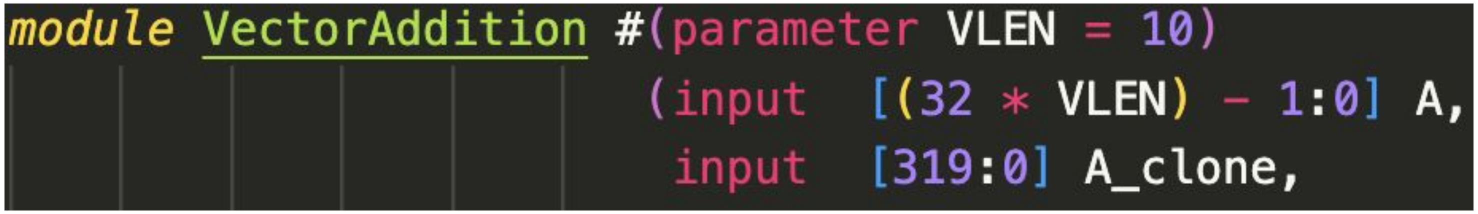
```
always@(*) begin
    A_Mantissa = {1'b1,A[22:0]};
    A_Exponent = A[30:23];
    A_sign = A[31];

    B_Mantissa = {1'b1,B[22:0]};
    B_Exponent = B[30:23];
    B_sign = B[31];
```



```
Temp_Exponent = (A_Exponent + B_Exponent < 127) ? 0 : A_Exponent + B_Exponent - 127;
Temp_Mantissa = A_Mantissa * B_Mantissa;  // multiply mantissas
Mantissa = Temp_Mantissa[47] ? Temp_Mantissa[46:24] : Temp_Mantissa[45:23];
Exponent = Temp_Mantissa[47] ? Temp_Exponent+1'b1 : Temp_Exponent;
Sign = A_sign^B_sign;
result = {Sign,Exponent,Mantissa};
```

Vectorization in Verilog

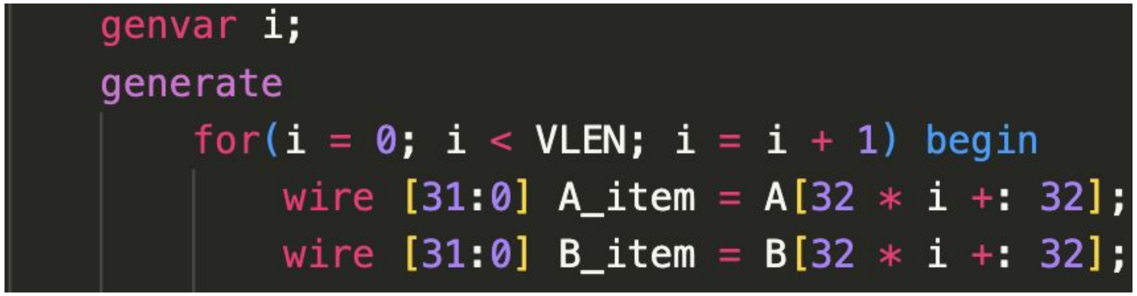
Although Verilog offers multidimensional registers (equivalent to arrays) it is more practical to use one-dimensional registers as often as possible.This is because Verilog only allows multidimensional registers within modules but does not permit them to be used as output or input registers. Therefore, using them always necessitates a highly intricate conversion. Thus, a register often takes the form:



```
module VectorAddition #(parameter VLEN = 10)
                       (input  [(32 * VLEN) - 1:0] A,
                        input  [319:0] A_clone,
```

We hereby use parameters (which are constant values) to define the length of the vector. This has the same semantic meaning as the explicit definition of `A_clone`.

indexing is achieved by using `[x +: 32]` syntax. This means indexing the next 32 bits starting from x

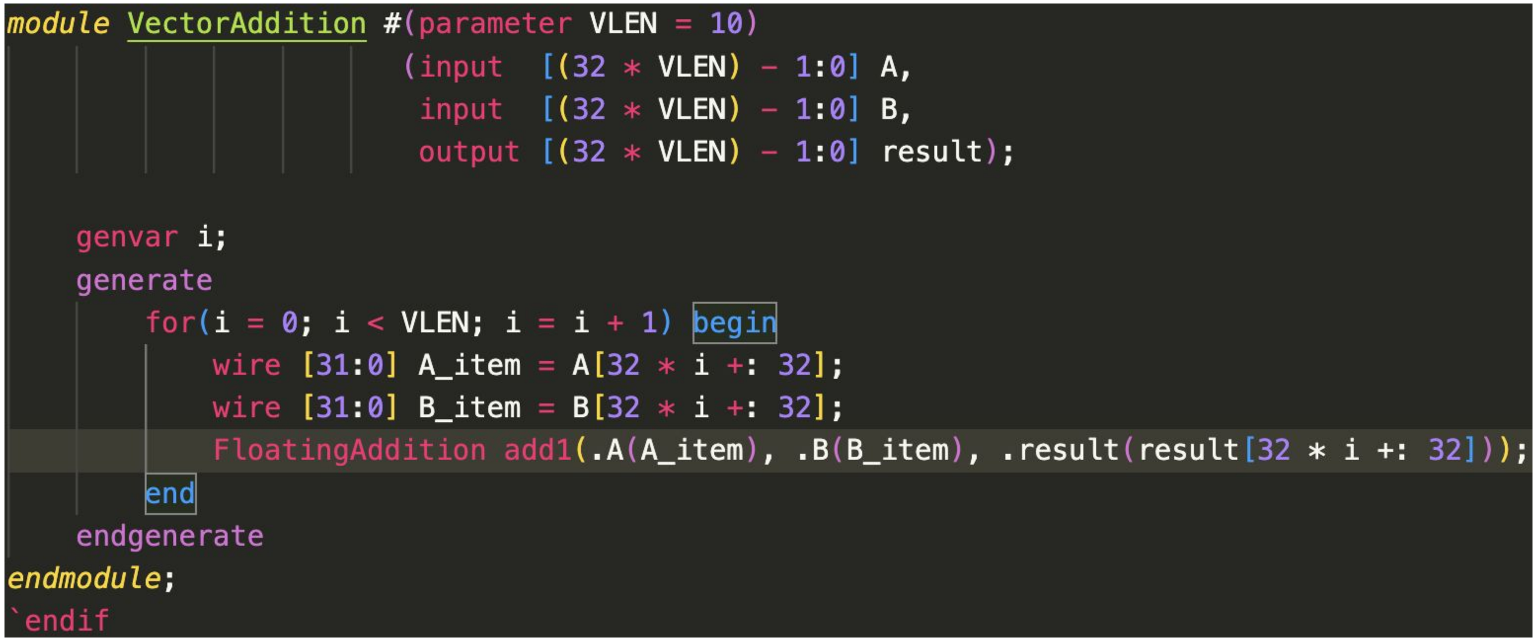


```
    genvar i;
    generate
        for(i = 0; i < VLEN; i = i + 1) begin
            wire [31:0] A_item = A[32 * i +: 32];
            wire [31:0] B_item = B[32 * i +: 32];
```

Vector Multiplication & Addition

Both vector multiplication and vector addition involve element-wise computation of corresponding 32-bit numbers in two vectors of the same length.

Therefore, their operation involves iterating over the number of elements in a vector, applying the appropriate floating-point operation explained in the previous slides to two elements, and storing the result in the output wire at the appropriate position.

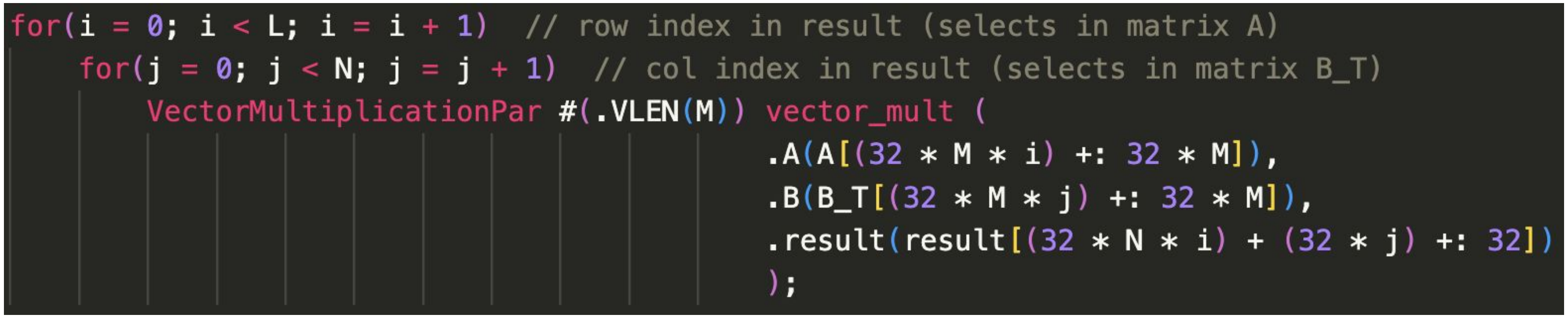


```
module VectorAddition #(parameter VLEN = 10)
                       (input  [(32 * VLEN) - 1:0] A,
                        input  [(32 * VLEN) - 1:0] B,
                        output [(32 * VLEN) - 1:0] result);

    genvar i;
    generate
        for(i = 0; i < VLEN; i = i + 1) begin
            wire [31:0] A_item = A[32 * i +: 32];
            wire [31:0] B_item = B[32 * i +: 32];
            FloatingAddition add1(.A(A_item), .B(B_item), .result(result[32 * i +: 32]));
        end
    endgenerate
endmodule;
`endif
```

Matrix Multiplication

In the implementation of matrix multiplication, iteration is performed over all rows and columns. For each combination of a row from the left matrix and a column from the right matrix, the dot product is applied.

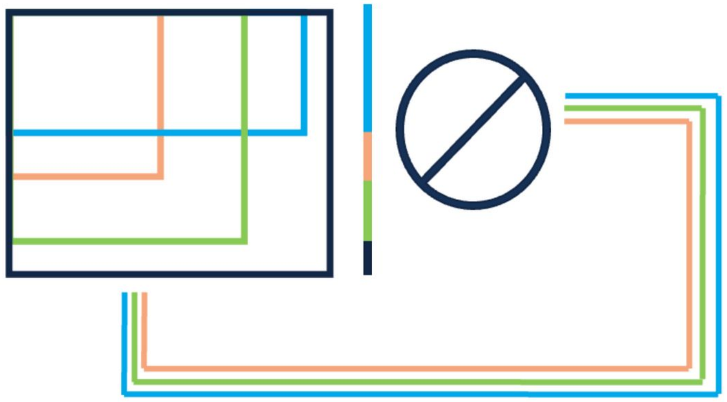
The parallel implementation below differs from the sequential one in that the sequential version instantiates only one module and fills it sequentially with the respective rows and columns. In contrast, the parallel implementation creates 15 modules for the multiplication of a 3x7 and a 7x5 matrix, each responsible for calculating a scalar product. Therefore, in case of hardware limitations, it is advisable to resort to the sequential variant.



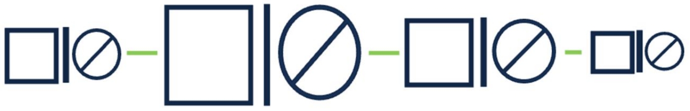
```
for(i = 0; i < L; i = i + 1)  // row index in result (selects in matrix A)
    for(j = 0; j < N; j = j + 1)  // col index in result (selects in matrix B_T)
        VectorMultiplicationPar #(.VLEN(M)) vector_mult (
                                             .A(A[(32 * M * i) +: 32 * M]),
                                             .B(B_T[(32 * M * j) +: 32 * M]),
                                             .result(result[(32 * N * i) + (32 * j) +: 32])
                                             );
```

Flexible modules I

- Basic design concept of Flex modules:
  - Create a black module with maximum dimensions.
  - All layers of the neural network perform calculations within this module.
  - calculation is then performed sequentially
  - at runtime the concrete dimensions are given hence the calculation can be sped up by not calculating all elements of black module
  - Synthesize only one large module.



- Contrasting parallel modules:
  - Parallel modules synthesize new circuits for each layer.
  - Require more resources, especially in deep neural networks.



Flexible modules II

Basic Differentiation between parallel, sequential and flex:

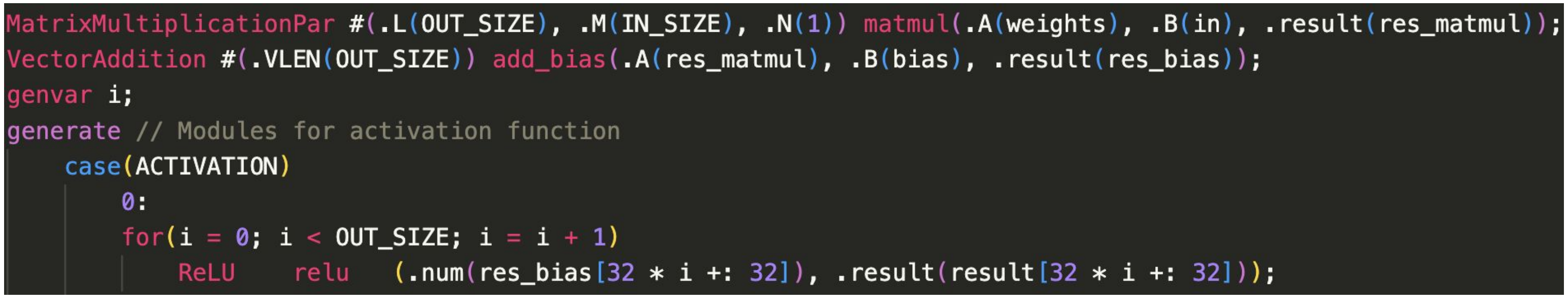
- parallel is most resource intensive, computes everything in a separate module
- parallel describes calculations inside a module (e.g: Vector Multiplication)

- sequential uses only one module of fixed size and sequentially runs every computation of exactly the same size inside of this module
- sequential needs more time but less resources
- sequential describes arrangement of calculations inside a module

- flex creates one maximum sized module as a place of calculation for all other modules of same type but not same size!
- flex modules need drastically less resources in most cases
- flex describes arrangement of multiple modules and layers (hence affects a different layer)

Neural network (layer)

- Fundamental structure of a Neural Network is a cycle.
- Process:
    - Input data stored as a vector.
    - Apply matrix multiplication, corresponding to multiplication of inputs with neuron weights. (line 1 of code snippet)
    - Add bias to result sums. (line 2 of code snippet)
    - Feed results into an activation function. (line 3-8 of code snippet)
    - Output becomes data for the next layer.
- Repeat this process until the last layer is reached.
- Applies to both general neural networks and our Verilog-implemented neural network.

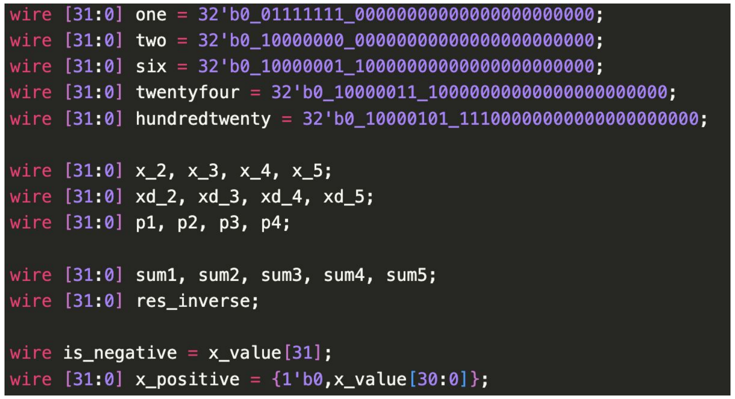


```
MatrixMultiplicationPar #(.L(OUT_SIZE), .M(IN_SIZE), .N(1)) matmul(.A(weights), .B(in), .result(res_matmul));
VectorAddition #(.VLEN(OUT_SIZE)) add_bias(.A(res_matmul), .B(bias), .result(res_bias));
genvar i;
generate // Modules for activation function
    case(ACTIVATION)
        0:
        for(i = 0; i < OUT_SIZE; i = i + 1)
            ReLU    relu    (.num(res_bias[32 * i +: 32]), .result(result[32 * i +: 32]));
```

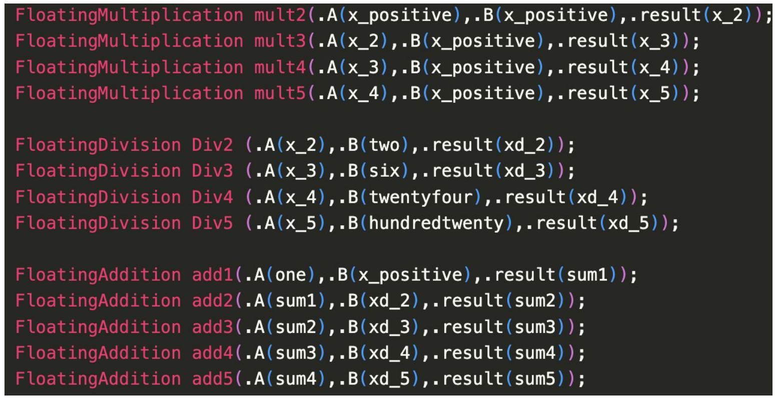
Activation functions I

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

- Activation functions often rely on the exponential function.
- Exact calculation of the exponential function is challenging.
- therefore an Approximation is necessary.
- the approximation utilizes the Taylor polynomial up to the fifth term for estimation (formula is shown above).
- The calculation requires the explicit definition of constants in the denominators (left image).
- Based on this, all terms are then computed and summed up (right image).



```
wire [31:0] one = 32'b0_01111111_00000000000000000000000;
wire [31:0] two = 32'b0_10000000_00000000000000000000000;
wire [31:0] six = 32'b0_10000001_10000000000000000000000;
wire [31:0] twentyfour = 32'b0_10000011_10000000000000000000000;
wire [31:0] hundredtwenty = 32'b0_10000101_11100000000000000000000;

wire [31:0] x_2, x_3, x_4, x_5;
wire [31:0] xd_2, xd_3, xd_4, xd_5;
wire [31:0] p1, p2, p3, p4;

wire [31:0] sum1, sum2, sum3, sum4, sum5;
wire [31:0] res_inverse;

wire is_negative = x_value[31];
wire [31:0] x_positive = {1'b0,x_value[30:0]};
```



```
FloatingMultiplication mult2(.A(x_positive),.B(x_positive),.result(x_2));
FloatingMultiplication mult3(.A(x_2),.B(x_positive),.result(x_3));
FloatingMultiplication mult4(.A(x_3),.B(x_positive),.result(x_4));
FloatingMultiplication mult5(.A(x_4),.B(x_positive),.result(x_5));

FloatingDivision Div2 (.A(x_2),.B(two),.result(xd_2));
FloatingDivision Div3 (.A(x_3),.B(six),.result(xd_3));
FloatingDivision Div4 (.A(x_4),.B(twentyfour),.result(xd_4));
FloatingDivision Div5 (.A(x_5),.B(hundredtwenty),.result(xd_5));

FloatingAddition add1(.A(one),.B(x_positive),.result(sum1));
FloatingAddition add2(.A(sum1),.B(xd_2),.result(sum2));
FloatingAddition add3(.A(sum2),.B(xd_3),.result(sum3));
FloatingAddition add4(.A(sum3),.B(xd_4),.result(sum4));
FloatingAddition add5(.A(sum4),.B(xd_5),.result(sum5));
```

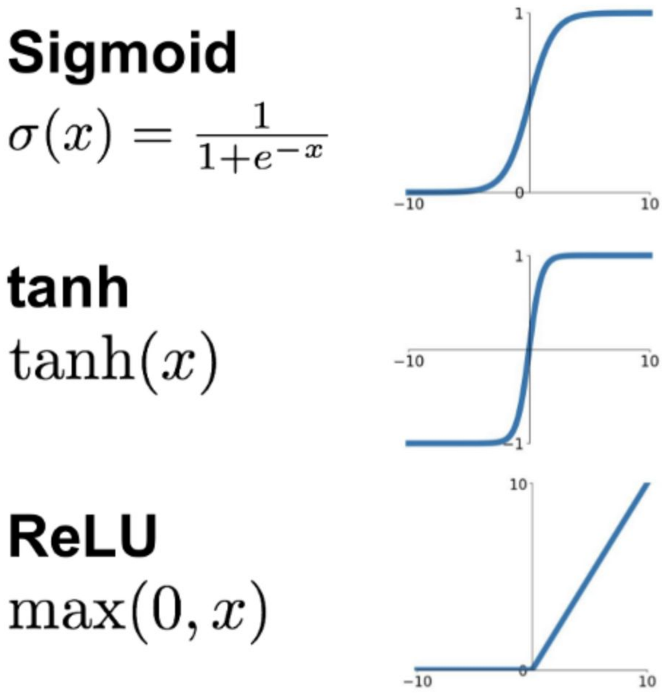
Activation functions II

- **Sigmoid Function:**
  - Approximated using the formula 1/(1+abs(x)).
- **Tanh Function:**
  - calculated using the formula mentioned below.
  - Relies on the exponential function approximation from the previous slide.

Tanh

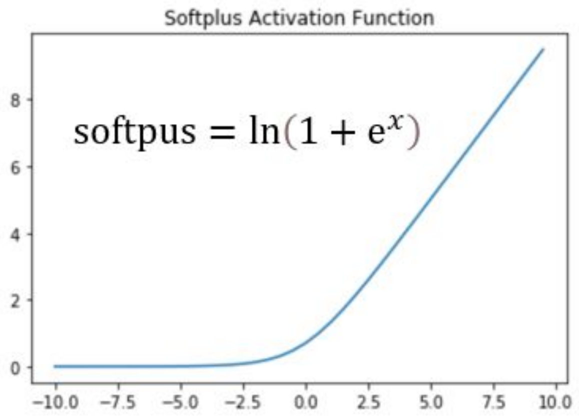
$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

- **ReLU Function:**
  - Implemented to check the sign bit.
  - Sets the floating-point value to 0 if the sign bit is set.



Activation functions III

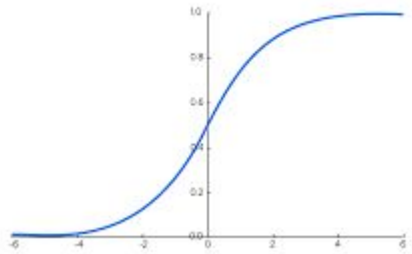
- **Softplus Activation Function:**
  - Implemented to calculate activation value using the formula ln(1+ex)ln(1+ex).
  - Approximated using a custom Taylor polynomial in the LogarithmApprox.v module.



- **Softmax Function:**
  - Specifically implemented for the output layer of the neural network.
  - Required for obtaining probabilities between 0 and 100% for digit predictions.
  - Implemented according to the formula mentioned below.

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Softmax Function



Backpropagation I

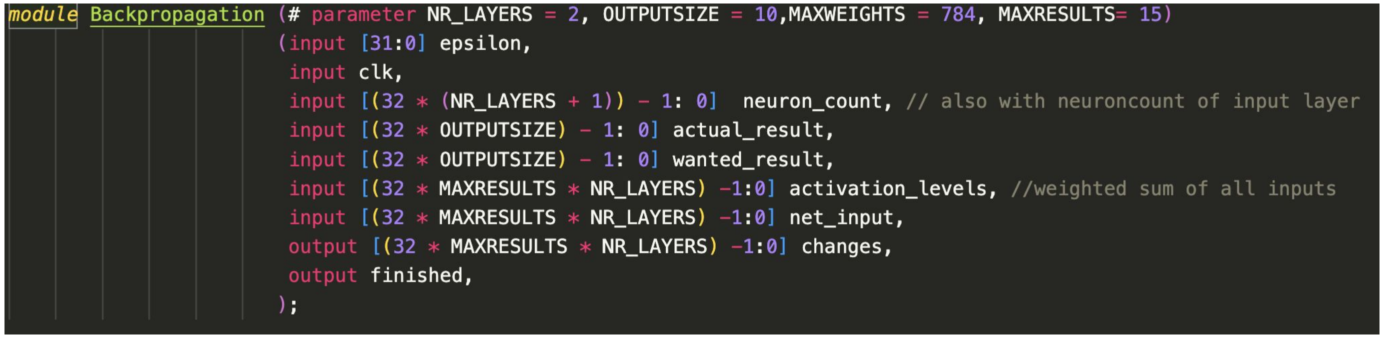
- Backpropagation implemented using the following formulas.
- The adjustment value for neuron weights varies based on whether the neuron is in the output (upper formula) or a hidden layer (lower formula).
- Epsilon represents a pre-defined learning rate (commonly around 0.01).
- The derivative of the activation function f′ is Implemented in the Sigmoid Derivative module.
- The last bracket in the formula for output neurons calculates the difference between the actual and observed values.
- The sum in the formula for hidden neurons is the sum of all weights multiplied by the adjustment values for the specific neurons to which the weights from the current neuron lead.

$$\delta_{\text{output}} = \varepsilon \cdot f'(\text{z}_{\text{output}}) \cdot (y_{\text{true}} - y_{\text{predicted}})$$

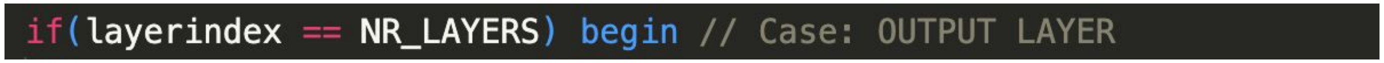
$$\delta_{\text{hidden}} = \varepsilon \cdot f'(\text{z}_{\text{hidden}}) \cdot \sum_k w_{\text{hidden\_output}} \cdot \delta_{\text{output}}$$

Backpropagation II

- The code snippet below reveals a complex and layered module.
- External supply of all parameters is crucial for calculations.
- Summation step in the formula for hidden neurons is performed externally.
- The second code snippet emphasizes that additional code and resources on the FPGA are required only for the differing part of the formula.
- Existing elements can be reused, minimizing resource usage.



```
module Backpropagation (# parameter NR_LAYERS = 2, OUTPUTSIZE = 10,MAXWEIGHTS = 784, MAXRESULTS= 15)
                      (input [31:0] epsilon,
                       input clk,
                       input [(32 * (NR_LAYERS + 1)) - 1: 0]  neuron_count, // also with neuroncount of input layer
                       input [(32 * OUTPUTSIZE) - 1: 0] actual_result,
                       input [(32 * OUTPUTSIZE) - 1: 0] wanted_result,
                       input [(32 * MAXRESULTS * NR_LAYERS) -1:0] activation_levels, //weighted sum of all inputs
                       input [(32 * MAXRESULTS * NR_LAYERS) -1:0] net_input,
                       output [(32 * MAXRESULTS * NR_LAYERS) -1:0] changes,
                       output finished,
                      );
```



```
if(layerindex == NR_LAYERS) begin // Case: OUTPUT LAYER
```

Implementation on the board

Verilog module differences

Verilog source modules performing computations will remain the same. However…

- Testbenches used for simulation are unusable
- Big part of Verilog syntax is not synthesizable
- Xilinx specification of Verilog != standard Verilog

Examples of these differences (contained in simulator, not in synthesis):

- “data type” real (64-bit float)
- most of the functions starting with $
- fixed time delays
- possible values on a wire (simulator enables Hi-Z state + undefined X)
- …

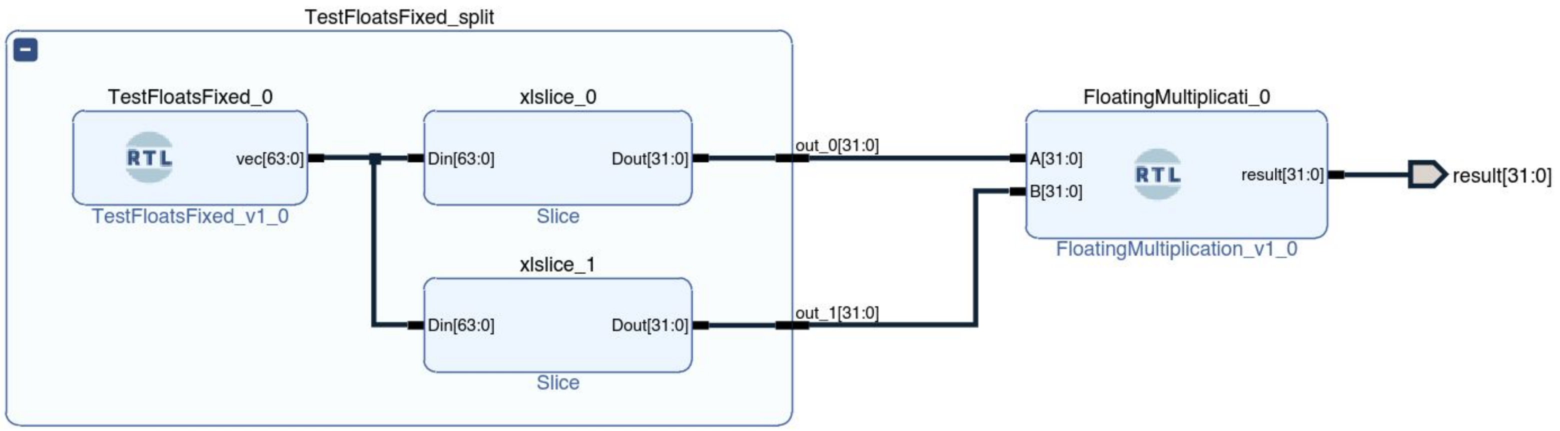
This means that writing Verilog modules with purely simulator testing is problematic. In multiple cases, we had to rewrite our code because of these incompatibilities.

Design process differences

When working with simulator, we differentiate between source modules and testbenches. Tasks of testbenches are:

1. Include source module(s)
2. Provide them relevant inputs (either constant or changing)
3. Monitor outputs, either with `$display` or using GTKWave

Vivado provides a visual editor for these tasks. That is used to add multiple source modules into design and connect them with wires. Designs might have their own inputs/outputs, connected to board interface in Vivado.



Deployment process

Multiple steps are required between Verilog modules and a running application on the FPGA board.

1. Synthesis… translate from high-level structures into FPGA logical blocks
    - logical tables, FFs, LUTs etc. (see Wikipedia - Logic block)
2. Implementation… translation to board-specific design
    - optimizations, placement, routing, timing constraints
3. Bitstream generation… compile to “bytecode” that reprograms the FPGA
4. Bitstream upload and run
    - simple FPGA boards… one step (no on-board bitstream cache)
    - Kria KV260… store bitstreams on device, swap them using SSH connection
    - Vivado Connection Manager is normally used

Memory interaction

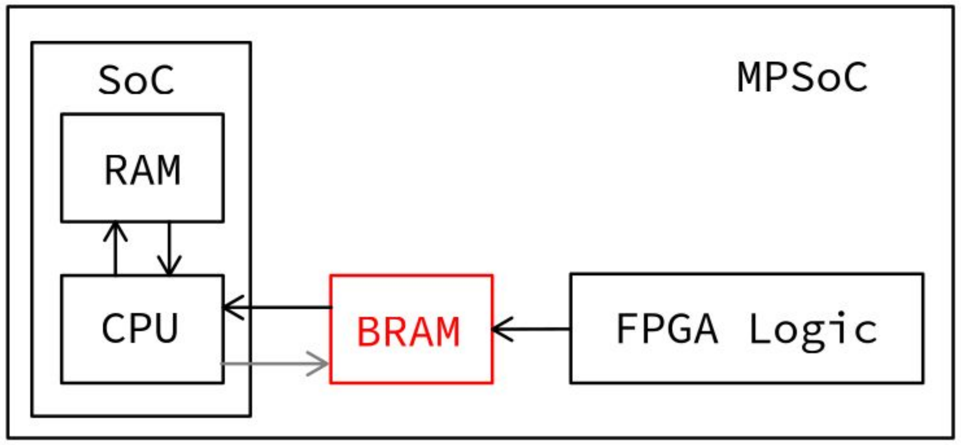
In order to see whether the FPGA computes correctly, form of displaying output is necessary. We settled on a technique of **BRAM memory mapping**:

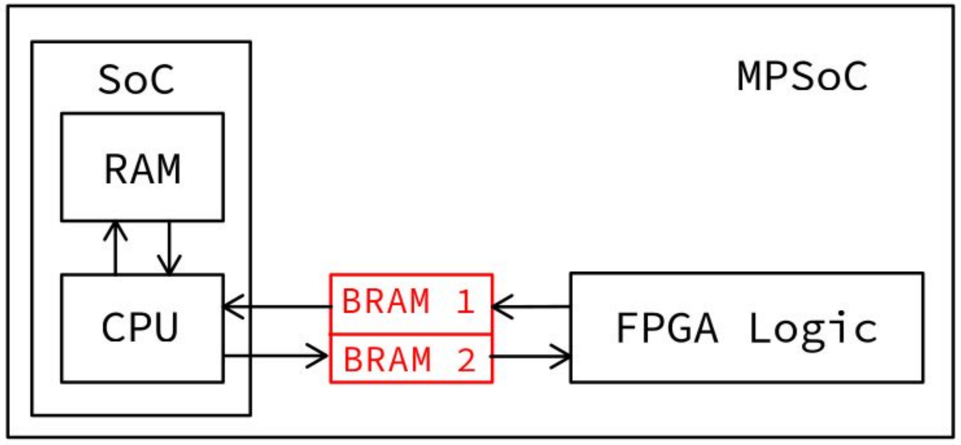
- Block RAM… one of the FPGA building blocks (read-write memory)
- Instantiate a Dual-port BRAM Generator
  - port A… rest of the board (CPU)
  - port B… our FPGA design performing computations
- In Vivado address editor, map the BRAM block to free address space
  - Note: Cortex CPU only has physical addresses, no address translation
- After running the bitstream, BRAM can be accessed via `/dev/mem` pseudofile
  - `devmem2` utility is an abstraction over this interaction

This method is also reversible, second BRAM can be instantiated on different address that serves as an input to the FPGA logic.

Memory interaction - diagrams

Output (read-only) design vs. Input/Output design





Memory interaction - AXI

Our original idea was to interact between the FPGA logic and BRAM using Advanced eXtensible Interface (AXI). This is a relatively standard protocol for on-chip communication. Therefore, we have implemented the `Vector2BRAM` Verilog module, that:

- Followed the AXI handshake protocol,
- iterated the different values on output, sending them within one burst,
- detected any error and repeated communication if necessary.

However, there were multiple problems with this module:

- It is significantly more complex than BRAM communication,
- no proper benefits were encountered in our simple usecase,
- we didn't manage to make our basic prototype work.

Overall, we found BRAM to be the more intuitive solution, so we abandoned this idea after some time.

Modified deployment process

Due to the memory mapping process, we cannot use the Connection Manager in Vivado. Different method of running the bitstream is used, inspired by Xilinx examples. This is also described in our README. Both Vivado and Vitis must be installed for this, which increases the install size significantly.

1. Generate `.bit.bin` bitstream in Vivado
2. Export `.xsa` project file in Vivado
3. Convert project file to `.dtbo` binary device tree
   - This step is automated with our custom script `xsa2dtbo.sh`
4. Create a `shell.json` configuration file
5. Upload all mentioned files to the board via SSH
6. use `sudo xmutil loadapp [app_name]`

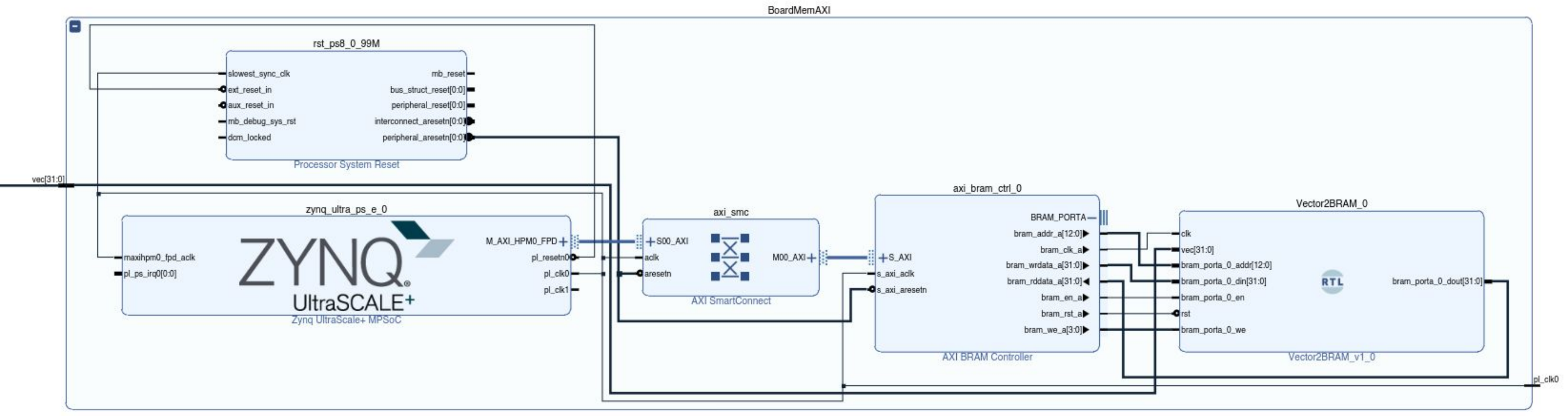
Other useful commands are `xmutil unloadapp`, `xmutil listapps`.

Basic output architecture

Realization of the output-only architecture in Vivado. Displayed block represents:

- SoC (Zynq UltraScale+, left)
- BRAM (`Vector2BRAM`, right)
- needed interconnects (AXI SmartConnect, AXI BRAM Controller)

Computing modules are connected externally, using the wire on the left. Data on the wire will be periodically written into the BRAM.

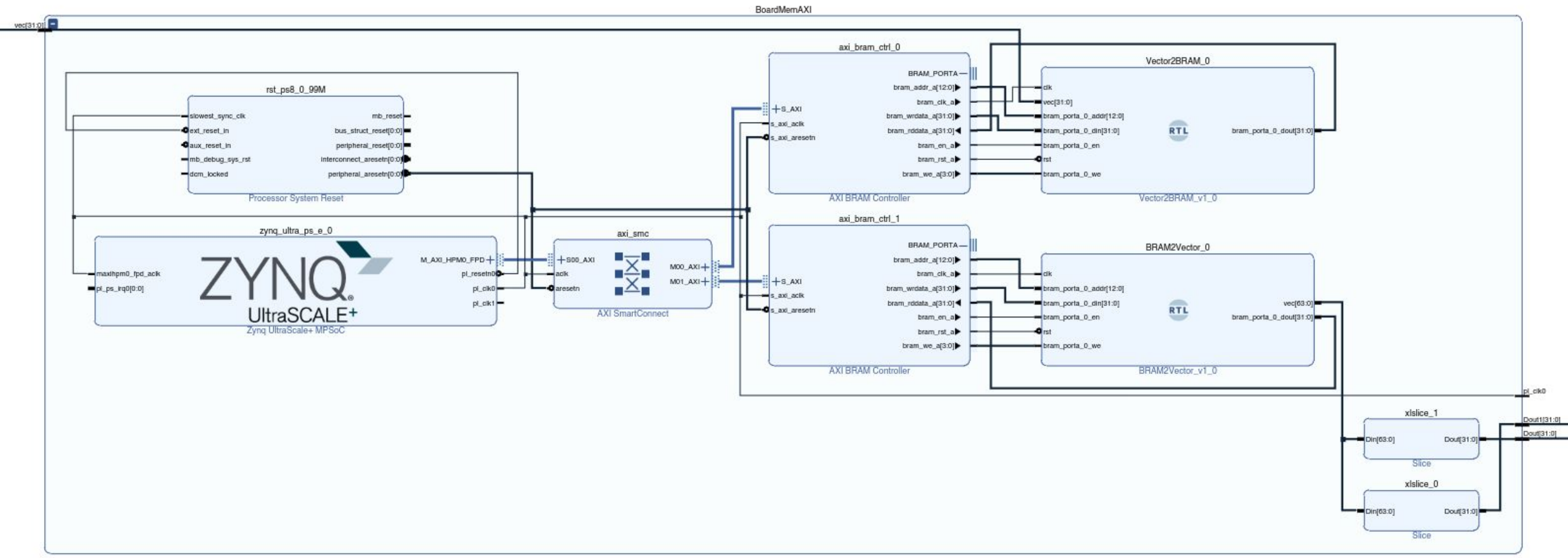


Basic output architecture

```
always @(posedge clk)
   begin
       if (rst) begin  // manual reset
           ...
       end else begin
                                       2 ** 11 numbers stored, 4 B each => 8 KiB memory
           if (bram_portb_0_addr < 11'h7ff) begin
               1. shift address by 1
               bram_portb_0_addr <= bram_portb_0_addr + 1;
               2. write from vector position to BRAM
                  BRAM bigger than vector... write DEAD_FEED to the rest
               // address change is one clock cycle behind (therefore, add 1)
               bram_portb_0_din <= (bram_portb_0_addr < VLEN - 1) ?
                                   vec[32 * (bram_portb_0_addr + 1) +: 32] : 32'hdead_feed;
               ...
           end else begin
               3. disable memory writing when finished
               bram_portb_0_we <= 1'b0;
           end
       end
   end
```

Input/Output architecture

Compared to the output-only architecture, one element was added – `BRAM2Vector`. This instantiates new BRAM which serves for inputs. Output is split into multiple buses using Slicers.



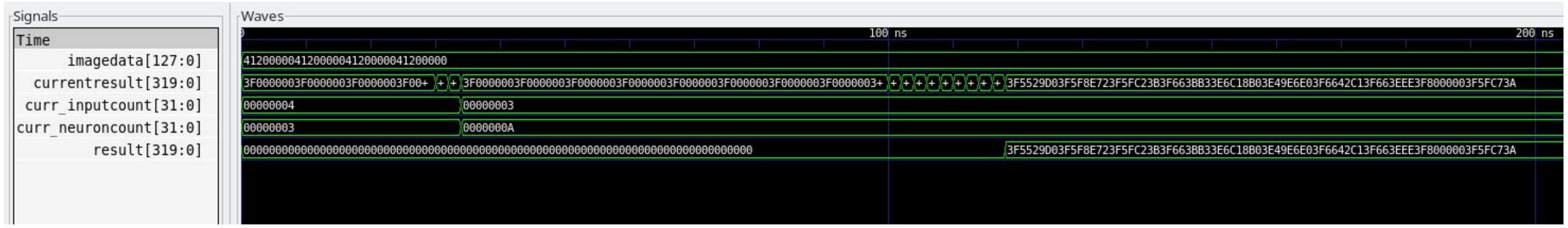
Input/Output architecture

```
always @(posedge clk)
   begin
       if (rst) begin  // reset address and turn off the memory access (en)
           ...
       perform a periodic reset, so that changes propagate over time
       end else if (&bram_portb_0_addr == 1'b1) begin  // unary reduction - all 1 in address
           ...
       end else begin
           1. increase the address by 1
           bram_portb_0_addr <= bram_portb_0_addr + 1;  // also serves as a periodic reset
           if (bram_portb_0_addr < VLEN + 1) begin
                2. read the BRAM output into vector
                vec[32 * (bram_portb_0_addr - 1) +: 32] <= bram_portb_0_dout;
                bram_portb_0_en <= 1'b1;
           end else begin
                3. turn off the memory access
                bram_portb_0_en <= 1'b0;
           end
       end
   end
```

Result showcase

Simulation results I

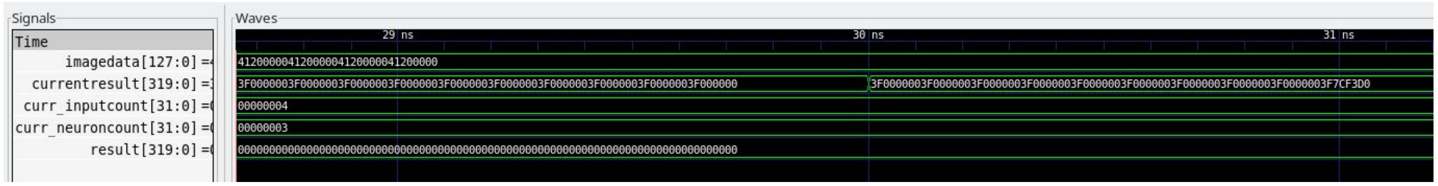
- The described implementation of the neural network in Verilog was tested using simulation and analyzed results using GTKwave.
- Due to hardware limitations, MNIST data couldn't be used as input, so the network was reduced to 4 input neurons, 3 hidden neurons, and 10 output neurons.
- Despite the reduction, it serves as a proof of concept.
- The input data is depicted in the following image's first row, where the float value of 10 is inserted four times.
- The result can be observed at the end of the second row, which, upon verification, corresponds to the correct values.



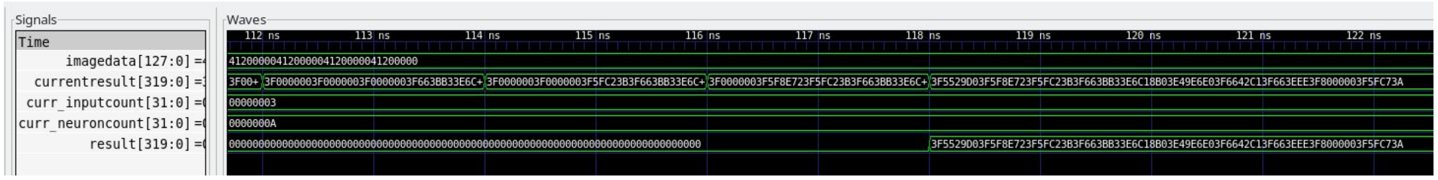
```
Signals                 Waves
Time                    0                                                   100 ns                                                  200 ns
imagedata[127:0]        41200000412000004120000041200000
currentresult[319:0]    3F0000003F0000003F0000003F00+ + + 3F0000003F0000003F0000003F0000003F0000003F0000003F0000003F0000003F0000003+ + + + + + + + + + 3F5529D03F5F8E723F5FC23B3F663BB33E6C18B03E49E6E03F6642C13F663EEE3F8000003F5FC73A
curr_inputcount[31:0]   00000004    00000003
curr_neuroncount[31:0]  00000003    0000000A
result[319:0]           00000000000000000000000000000000000000000000000000000000000000000000000000000000    3F5529D03F5F8E723F5FC23B3F663BB33E6C18B03E49E6E03F6642C13F663EEE3F8000003F5FC73A
```

Simulation results II

- An interesting observation is that in the second row of the image on the previous slide, the implementation calculates the solution for all neurons in a layer one after the other.
- This is evident in the two short blocks for the hidden layer with 3 neurons and the subsequent nine short blocks for the output layer with 10 neurons.
- It is noticeable in the two images below that, for example, in the first image, only the last 8 hexadecimal digits and thus only the last stored float value are changing.
- This corresponds to the calculation of the value for the first neuron in the layer.



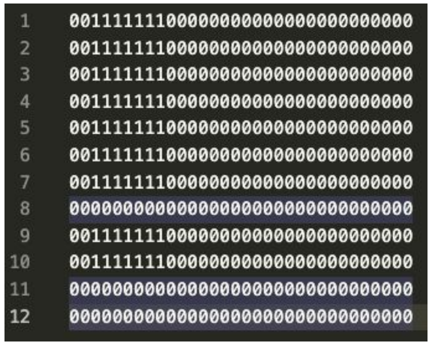
```
Signals                      Waves
Time                                 29 ns                                        30 ns                                        31 ns
imagedata[127:0] =           41200000412000004120000041200000
currentresult[319:0] =       3F0000003F0000003F0000003F0000003F0000003F0000003F0000003F0000003F0000003F000000 | 3F0000003F0000003F0000003F0000003F0000003F0000003F0000003F0000003F0000003F7CF3D0
curr_inputcount[31:0] =      00000004
curr_neuroncount[31:0] =     00000003
result[319:0] =              00000000000000000000000000000000000000000000000000000000000000000000000000000000
```



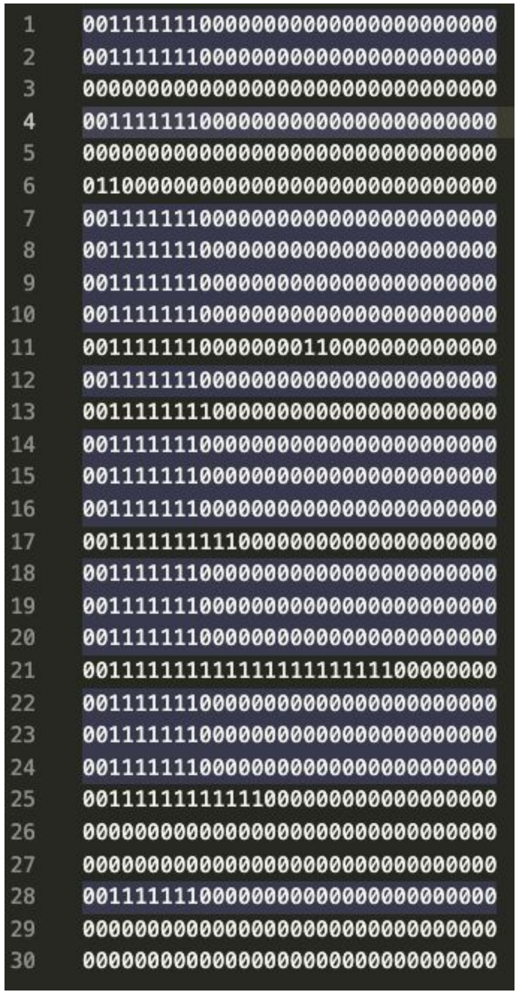
```
Signals                      Waves
Time                         112 ns  113 ns  114 ns  115 ns  116 ns  117 ns  118 ns  119 ns  120 ns  121 ns  122 ns
imagedata[127:0] =           41200000412000004120000041200000
currentresult[319:0] =       3F00+ | 3F0000003F0000003F0000003F663BB33E6C+ | 3F0000003F0000003F5FC23B3F663BB33E6C+ | 3F0000003F5F8E723F5FC23B3F663BB33E6C+ | 3F5529D03F5F8E723F5FC23B3F663BB33E6C18B03E49E6E03F6642C13F663EEE3F8000003F5FC73A
curr_inputcount[31:0] =      00000003
curr_neuroncount[31:0] =     0000000A
result[319:0] =              00000000000000000000000000000000000000000000000000000000000000000000000000000000 | 3F5529D03F5F8E723F5FC23B3F663BB33E6C18B03E49E6E03F6642C13F663EEE3F8000003F5FC73A
```

Simulation results III

- For the calculation, the weights from the left image were used for the hidden layer, and those from the right image for the output layer.
- It is evident in the images that there are 12 weights for the hidden layer since each of the three hidden neurons has a connection to the 4 input neurons, requiring a total of 12 weights.
- Similarly, the format of biases is analogous, although it is not shown here for the sake of clarity.



```
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00000000000000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```



```
00111111100000000000000000000000
00111111100000000000000000000000
00000000000000000000000000000000
00111111100000000000000000000000
00000000000000000000000000000000
01100000000000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000110000000000000
00111111100000000000000000000000
00111111110000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111111100000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111111111111111111100000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111100000000000000000000000
00111111111111000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00111111100000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Vivado designs

Designs were made to be **modular** – attempt to separate logic and memory.
Therefore, for most of the computing modules, we have created following versions:

- No I/O, only for simulator usage (no suffix)
- Fixed input, output displayed on BRAM (`ToMemory`)
- I/O using 2 BRAMs (`_IO`)

We have created following modules:

- Multiplication of 2 floats (SimpleFP)
- Addition of 2 vectors of floats (VectorFP)
- Single neural network layer (NeuralLayer)
- 2-layer neural network on MNIST inputs (NeuralNetwork)*

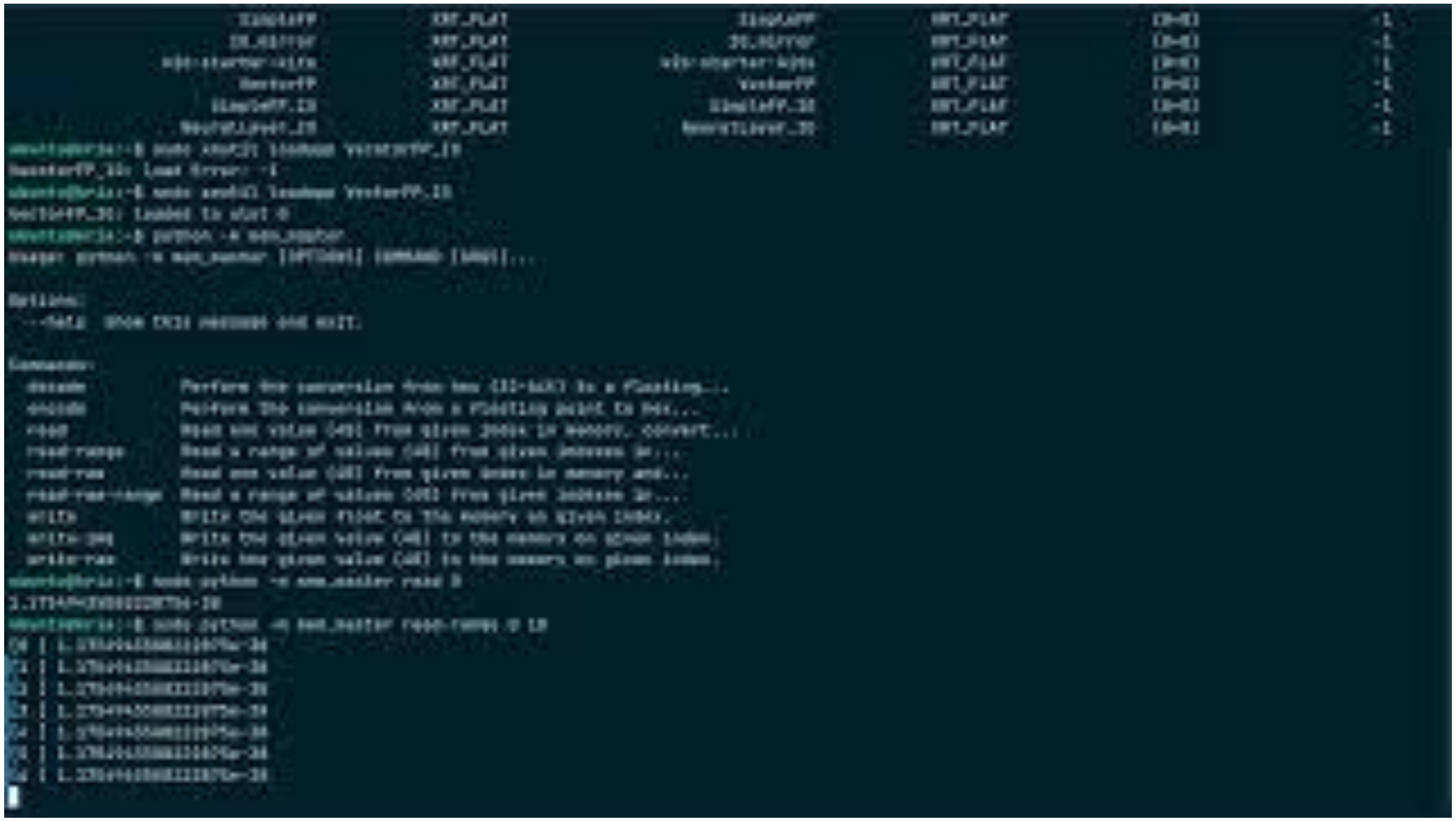
*NeuralNetwork module is prepared, but the lack of memory on provided hardware was causing repeated Vivado crashes.

Due to these limitations on the side of Vivado processing, we didn’t attempt to construct more complex structures.

Video Showcase

https://www.youtube.com/watch?v=UUmsJQ_za34

Video embed:



```
                SimpleFP          XRT_FLAT                  SimpleFP          XRT_FLAT          [0-0]          -1
               IO_Mirror          XRT_FLAT                 IO_Mirror          XRT_FLAT          [0-0]          -1
       xdp-starter-kits          XRT_FLAT          xdp-starter-kits          XRT_FLAT          [0-0]          -1
                VectorFP          XRT_FLAT                  VectorFP          XRT_FLAT          [0-0]          -1
             SimpleFP_IO          XRT_FLAT               SimpleFP_IO          XRT_FLAT          [0-0]          -1
          NeuralLayer_IO          XRT_FLAT            NeuralLayer_IO          XRT_FLAT          [0-0]          -1
ubuntu@kria:~$ sudo xmutil loadapp VectorFP_IO
VectorFP_IO: Load Error: -1
ubuntu@kria:~$ sudo xmutil loadapp VectorFP_IO
VectorFP_IO: loaded to slot 0
ubuntu@kria:~$ python -m mem_master
Usage: python -m mem_master [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  decode          Perform the conversion from hex (32-bit) to a floating...
  encode          Perform the conversion from a floating point to hex...
  read            Read one value (48) from given index in memory, convert...
  read-range      Read a range of values (48) from given indexes in...
  read-raw        Read one value (48) from given index in memory and...
  read-raw-range  Read a range of values (48) from given indexes in...
  write           Write the given float to the memory on given index.
  write-[illegible]       Write the given value (48) to the memory on given index.
  write-raw       Write the given value (48) to the memory on given index.
ubuntu@kria:~$ sudo python -m mem_master read 0
1.1754943508222875e-38
ubuntu@kria:~$ sudo python -m mem_master read-range 0 10
[0 | 1.1754943508222875e-38
[1 | 1.1754943508222875e-38
[2 | 1.1754943508222875e-38
[3 | 1.1754943508222875e-38
[4 | 1.1754943508222875e-38
[5 | 1.1754943508222875e-38
[6 | 1.1754943508222875e-38
```

Python helper scripts

mem_master – FPGA memory interaction

Python wrapper for memory interactions using devmem2 command.

Features:

- Automatic conversions between bytes and floating point numbers
- `devmem2` subprocesses in order to interact with memory
- Read/write in bulk, write from file etc.

Interface… CLI, using Python package Click (Click Documentation)

Repository URL https://github.com/ruzicka02/mem_master

mem_master – Usage

```
$ python mem_master
Usage: mem_master [OPTIONS] COMMAND [ARGS]...

Options:
 --help  Show this message and exit.

Commands:
 decode          Perform the conversion from hex (32-bit) to a floating...
 encode          Perform the conversion from a floating point to hex...
 ...
$ python mem_master encode 3.14
4048f5c3
$ python mem_master decode 4048f5c3
3.140000104904175
$ sudo python mem_master write 2 3.14
Success!
$ sudo python mem_master read-range 0 10
[0 ] 2.0
[1 ] 1.1754943508222875e-38
[2 ] 3.140000104904175
[3 ] 1.1754943508222875e-38
[4 ] 1.1754943508222875e-38
[5 ] 1.1754943508222875e-38
[6 ] 1.1754943508222875e-38
[7 ] 1.1754943508222875e-38
[8 ] 1.1754943508222875e-38
[9 ] 1.1754943508222875e-38
```

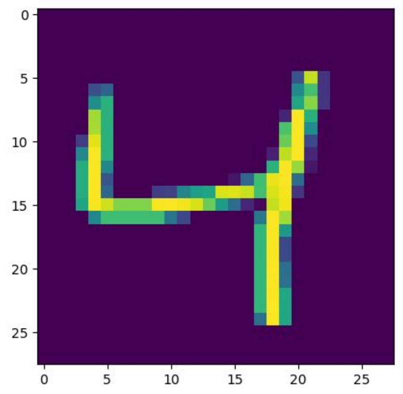
MNIST-Data-Transformation script I (float to binary)

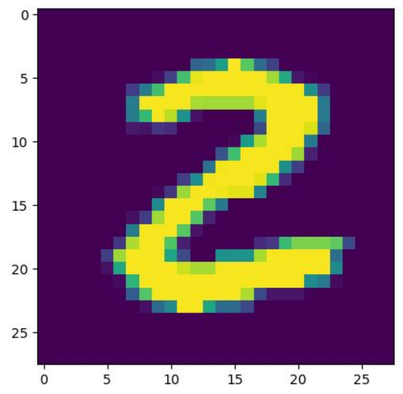
MNIST Dataset:

- contains pairs of feature data and label
- feature data consists of the grayscale values of the 784 pixels (28x28 pixel images)
- label is the correct digit, ranging from 0 to 9

Goals of the Data-Transformation script:

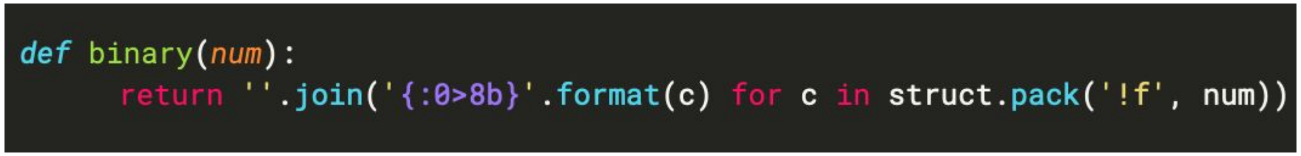
- provide an understanding of the data
- transform data from conventional 4 Byte float format to a binary string format requiring 33 Byte and explicitly storing a concatenation of “0” and “1” ascii chars
- 187.5 => “01000011001110111000000000000000\n”
- store the result in a dedicated file





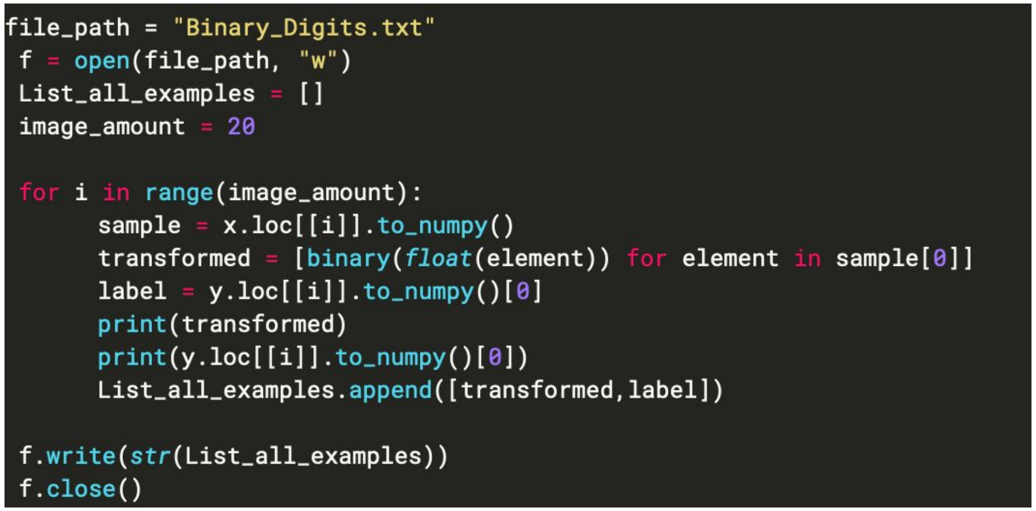
MNIST-Data-Transformation script II

The following code snippet shows the function, which converts a number from float format to binary format.



```python
def binary(num):
    return ''.join('{:0>8b}'.format(c) for c in struct.pack('!f', num))
```

- first four lines open file for storage purposes and define amount of images needed
- For loop iterates over images and transforms them
- Last part stores all images with the correct label

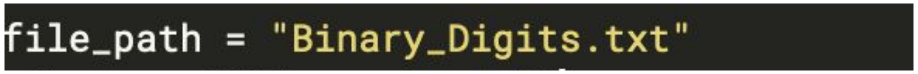


```python
file_path = "Binary_Digits.txt"
 f = open(file_path, "w")
 List_all_examples = []
 image_amount = 20

 for i in range(image_amount):
        sample = x.loc[[i]].to_numpy()
        transformed = [binary(float(element)) for element in sample[0]]
        label = y.loc[[i]].to_numpy()[0]
        print(transformed)
        print(y.loc[[i]].to_numpy()[0])
        List_all_examples.append([transformed,label])

 f.write(str(List_all_examples))
 f.close()
```

Usage MNIST-Data-Transformation

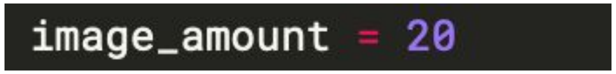
To ensure proper use of the script, the following variables must be set correctly:

- The desired file path for the final file must be specified in the variable `file_path`.



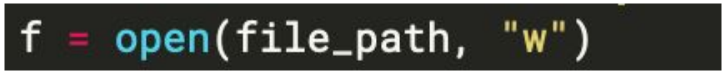
```
file_path = "Binary_Digits.txt"
```

- The correct number of desired images for the training or testing process must be set in the variable `image_amount`.



```
image_amount = 20
```

- The resulting file will overwrite any file with the same name. If this is not desired, the `w` in the open function should be changed to `a`.



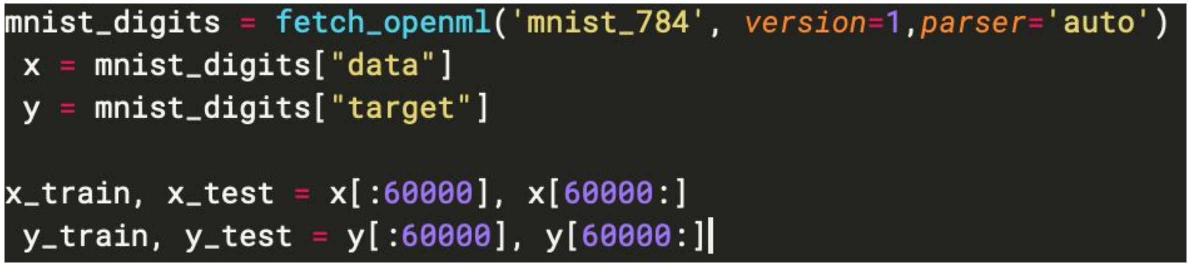
```
f = open(file_path, "w")
```

- The resulting Data Format is: `[[[784 pixels binary], label], next images…]`

Training NN script I

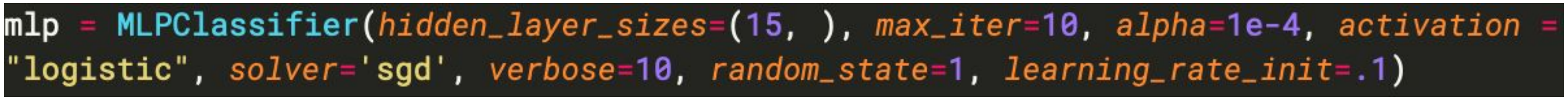
Purpose of this script is to calculate the weights and biases needed for the Neuronal Net in Verilog without the backpropagation mechanism.

First part of this script loads the training data from the mnist-dataset (not binary since we only require binary format for the NN in verilog itself)



```
mnist_digits = fetch_openml('mnist_784', version=1,parser='auto')
 x = mnist_digits["data"]
 y = mnist_digits["target"]

x_train, x_test = x[:60000], x[60000:]
 y_train, y_test = y[:60000], y[60000:]
```

Second part initializes a MLPClassifier from SKlearn. Details regarding unmentioned parameters can be found here: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html



```
mlp = MLPClassifier(hidden_layer_sizes=(15, ), max_iter=10, alpha=1e-4, activation =
"logistic", solver='sgd', verbose=10, random_state=1, learning_rate_init=.1)
```

Training NN Script II

The activation function of all neurons in hidden neural layers is set to `logistic` which represents the sigmoid activation function. The activation function of the output layer is set to `softmax` since we require probabilities as an output metric.

```
mlp.out_activation_ = "softmax"
```

The last part of the script stores the calculated weights and biases in the corresponding files. The file names can be adapted if wanted.

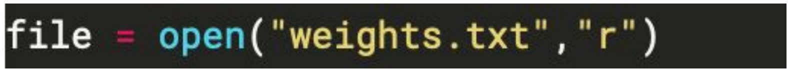
```
f1,f2 = open("weights.txt", "w"), open("biases.txt", "w")

for layers in mlp.coefs_:
    for neurons in layers:
        f1.write(str(neurons)+"\n")
f1.close()

for layers in mlp.intercepts_:
    for neurons in layers:
        f2.write(str(neurons)+"\n")
f2.close()
```

Weights Biases Formating script

purpose of the last script:

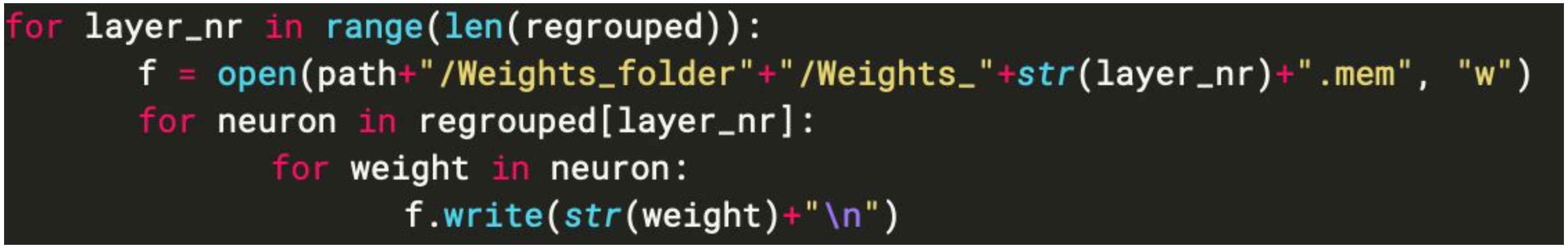
- load weights and biases from file specified in script 1
- select needed data based on current layer of the NN
- convert data into binary format
- create the folder structure required for the NN
- store weights (and biases) of each layer in separate file

The following part is the only one that should be adjusted accordingly



```python
file = open("weights.txt","r")
```

The following snippet shows the code segment responsible for the storage process



```python
for layer_nr in range(len(regrouped)):
        f = open(path+"/Weights_folder"+"/Weights_"+str(layer_nr)+".mem", "w")
        for neuron in regrouped[layer_nr]:
                for weight in neuron:
                        f.write(str(weight)+"\n")
```

Sources:

IEEE-754 image: https://numeral-systems.com/ieee-754-add/
Activation functions image:
https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092
tanh formula:
https://www.nomidl.com/deep-learning/what-is-tanh-activation-function/
Softplus Activation function image:
https://medium.com/geekculture/different-activation-functions-for-deep-neural-networks-you-should-know-ea5e86f51e84
Softmax function:
https://botpenguin.com/glossary/softmax-function
softmax function formula:
https://developer.apple.com/documentation/accelerate/bnnsactivationfunction/2915301-softmax