Neural Network Accelerator using SystemVerilog

(Integration with Efabless Caravel chip)

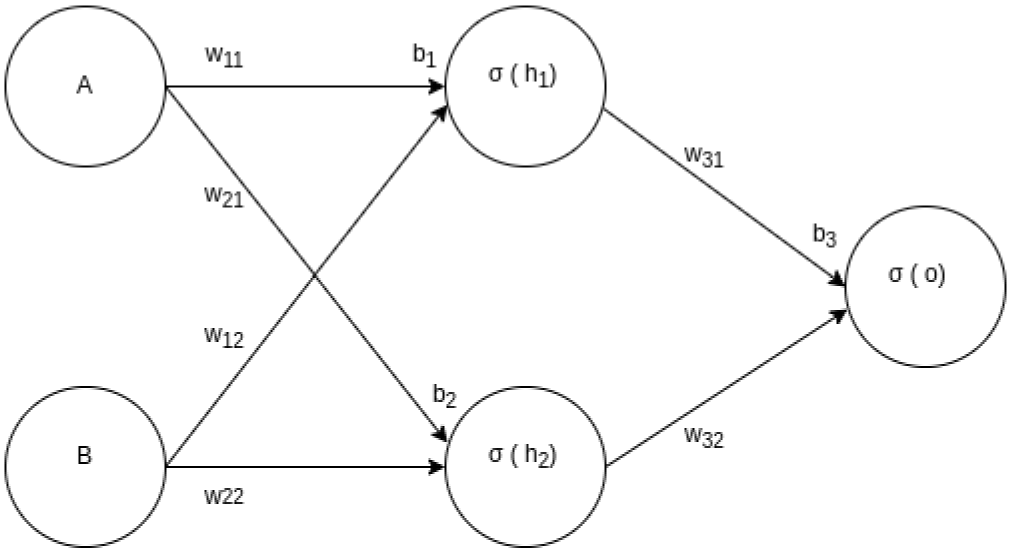

Figure 1: Neural Network Architecture to be implemented

Introduction

XOR Problem

The XOR problem refers to the inability of single-layer perceptrons to solve non-linearly separable functions, like XOR, highlighting the limitations of early neural networks. This issue contributed to skepticism about AI's potential, fueling the AI Winter in the 1970s. The XOR problem's resolution demonstrated the necessity of non-linear activation functions and multi-layer architectures, foundational for modern AI's success.

Figure 1 displays a simple neural network with 2-neuron hidden layer and sigmoid activation functions that solves the XOR Problem.

Mathematical Model

Modeling architecture from Figure 1 mathematically will provide the formal foundation for the SystemVerilog implementation.

$$\mathbf{h} = \mathbf{W_h} \cdot \mathbf{x} + \mathbf{b_h}$$

$$\mathbf{h} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

(EQ 1)

$$\mathbf{h'} = \begin{bmatrix} \sigma(h_1) \\ \sigma(h_2) \end{bmatrix}$$

$$o = \mathbf{W_o} \cdot \mathbf{h'} + b_3$$

$$o = \begin{bmatrix} w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} \sigma(h_1) \\ \sigma(h_2) \end{bmatrix} + b_3$$

$$o = w_{31}\sigma(h_1) + w_{32}\sigma(h_2) + b_3$$

(EQ 2)

$$o' = \sigma(o)$$

Figure 2: Mathematical operations of NN in figure 1

Goal Truth Table

A	B	O’	Y
1	1	< 0.5	0
0	1	> 0.5	1
1	0	> 0.5	1
0	0	<0.5	0

Values to achieve the Truth Table

var	value	var	value	var	value
w11	4	w22	-4	b1	-2
w12	4	w31	4	b2	6
w21	-4	w32	4	b3	-6

Verification: run a python script that uses those values `NN.py` inside the python folder.

```
(base) ➜ python git:(main) ✗ python NN.py
--------------------------
Input A: 0, Input B: 0
XOR_output: 0.19090909090909092 (0x3e437dac)
Final XOR Output: 0

--------------------------
Input A: 0, Input B: 1
XOR_output: 0.7 (0x3f333333)
Final XOR Output: 1

--------------------------
```

```
Input A: 1, Input B: 0
XOR_output: 0.7 (0x3f333333)
Final XOR Output: 1

--------------------------
Input A: 1, Input B: 1
XOR_output: 0.19090909090909092 (0x3e437dac)
Final XOR Output: 0
```

Verilog Implementation

Testing & verification

To test the Verilog implementation, navigate to **repo/verilog/cocotb** and write command `make`. This will run a testbench with the aforementioned weights on the four test cases for `A` and `B`, and print the results which successfully match those obtained from the python script.

```
  Testbench for the NN module.
-------------------------------------------------------------------------
|       A Value         |        B Value         |       XOR Output         | Output > 0.5 |
-------------------------------------------------------------------------
| 0x0                  | 0x0                   | 0x3e437dac               | 0             |
| 0x0                  | 0x3f800000            | 0x3f333334               | 1             |
| 0x3f800000           | 0x0                   | 0x3f333334               | 1             |
| 0x3f800000           | 0x3f800000            | 0x3e437dac               | 0             |
-------------------------------------------------------------------------
```

Figure 3: results from cocotb test for the NN module

Note: This test runs on commit ***. The cocotb test is currently invalid. Instead, use the following commands in `repo/verilog/tb`:

1. `iverilog -g2012 -I../rtl NN_tb.sv -Wall -o NN_tb`
2. `vvp NN_tb`"

```
➜ tb git:(main) ✗ iverilog -g2012 -I../rtl NN_tb.sv -Wall -o NN_tb
➜ tb git:(main) ✗ vvp NN_tb
VCD info: dumpfile ../sim/NN_tb.vcd opened for output.
Test Case 1: A=1.0, B=0.0
Final Result (A=1.0, B=0.0): 3f333334
Test Case 2: A=0.0, B=1.0
Final Result (A=0.0, B=1.0): 3f333334
Test Case 3: A=1.0, B=1.0
Final Result (A=1.0, B=1.0): 3e437dac
Test Case 4: A=0.0, B=0.0
Final Result (A=0.0, B=0.0): 3e437dac
```

Number format

The SystemVerilog NN module handles **IEEE 754 single-precision floating point** numbers. For example, the conversion of hexadecimal representation to decimal representation for numbers in figure 3 is as follows

Hex	Dec
0x3f800000	1.0
0x3e437dac	0.19090909
0x3f333334	0.70000005

Sigmoid Function

To make synthesizable SystemVerilog code and overcome the issue of the exponent in the Sigmoid Function $\frac{1}{1+e^{-x}}$ I used the following approximation.

$$g(x) = \begin{cases} 1 - 0.5\left(1 + \frac{-x}{1-x}\right) & : x < 0 \\ 0.5\left(1 + \frac{x}{1+x}\right) & : \text{otherwise} \end{cases}$$

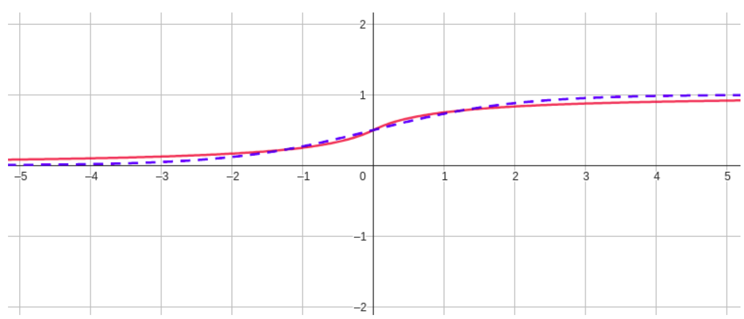

Figure 4: Sigmoid function approximation shown in solid red; sigmoid function shown in dotted purple.

To implement `sigmoid_approx` module in SystemVerilog I utilized the following modules:

- `add_sub`
- `multiplier`
- `divider`

Floating Point Unit

I used [Lampro-Mellon/Caravel_FPU] which is fully compliant with the IEEE-754 standard. Specifically I used the following modules

- `add_sub`

- `multiplier`
- `divider`

Matrix Multiplication

To utilize parallelism in hardware, I designed two custom matrix multiplication units to perform EQ 1 and EQ 2 from Figure 2.

- `matrix_multiply_1x2_2x1`
- `matrix_multiply_2x2_2x1`

Top Module

Top Module is `NN` in `NN.sv` that implements the mathematical model in figure 2 utilizing the following set of modules

- `add_sub`
- `multiplier`
- `divider`
- `matrix_multiply_1x2_2x1`
- `matrix_multiply_2x2_2x1`
- `sigmoid_approx`

In figure 5 For Neural Network calculations to be performed the NN module depends on NN registers for storing weights and biases and provision of NN inputs (A and B) that are affected by read/write instructions via the wishbone.

Only when `in_valid_user` is set to high will the NN Operations commence and the result will be calculated by a forward pass.

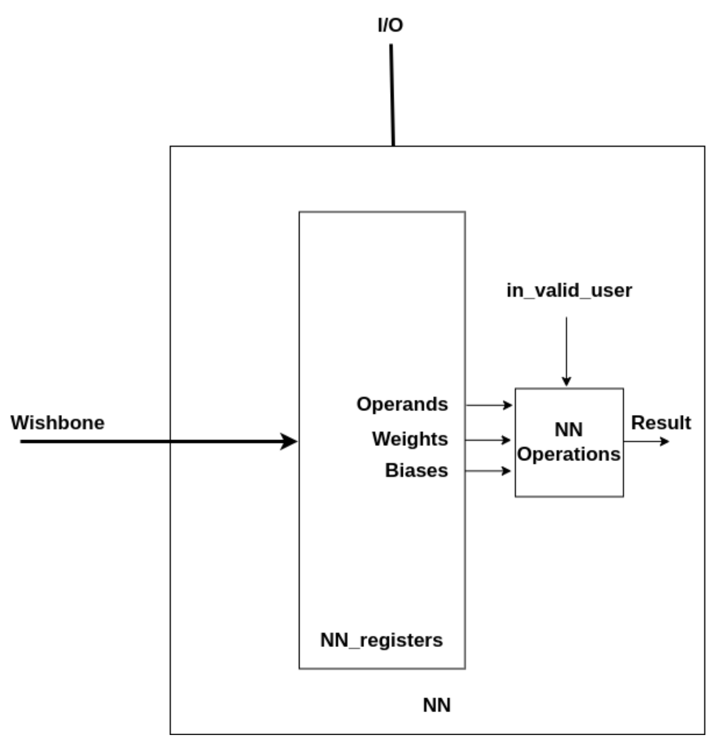

Figure5: NN module diagram

sigmoid_approx Module

In the figure `out_valid` is a flag that indicates the readiness of the result at the `out_sigmoid` signal. It takes on average about 26 cycles for the result to be ready as this module depends on the `divider` module for which you can find more information in Lampro-Mellon/Caravel_FPU .

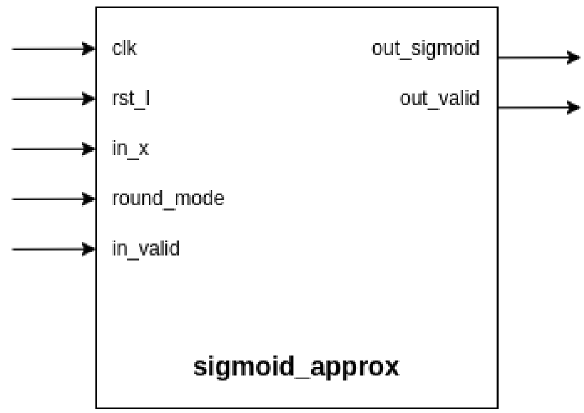

Figure 6: Sigmoid Module Diagram.

Caravel Chip Integration

Integration of NN as Memory Mapped Peripheral

A NN has a specific address range in memory that the core writes data to and reads data from. NN is integrated with the core using the Wishbone interface. The core acts as a master while IP acts as a slave. Write instructions are used at the beginning to write values of weights, biases, and inputs into the IP using the write control signals, then the IP sets acknowledge ack indicating that operation is completed. The unit is controlled via a set of registers.

NN range memory is illustrated in the following table

CSR	Access Type	offset
A	Read/Write	0x00
B	Read/Write	0x04
w11	Read	0x08
w12	Read	0x0C

w21	Read	0x10
w22	Read	0x18
w31	Read	0x1C
w32	Read	0x20
b1	Read	0x24
b2	Read	0x28
b3	Read	0x2C

Table 1

All the registers are located in the user design space with the base address of 0x3000_0000 + the offset described in table 1.

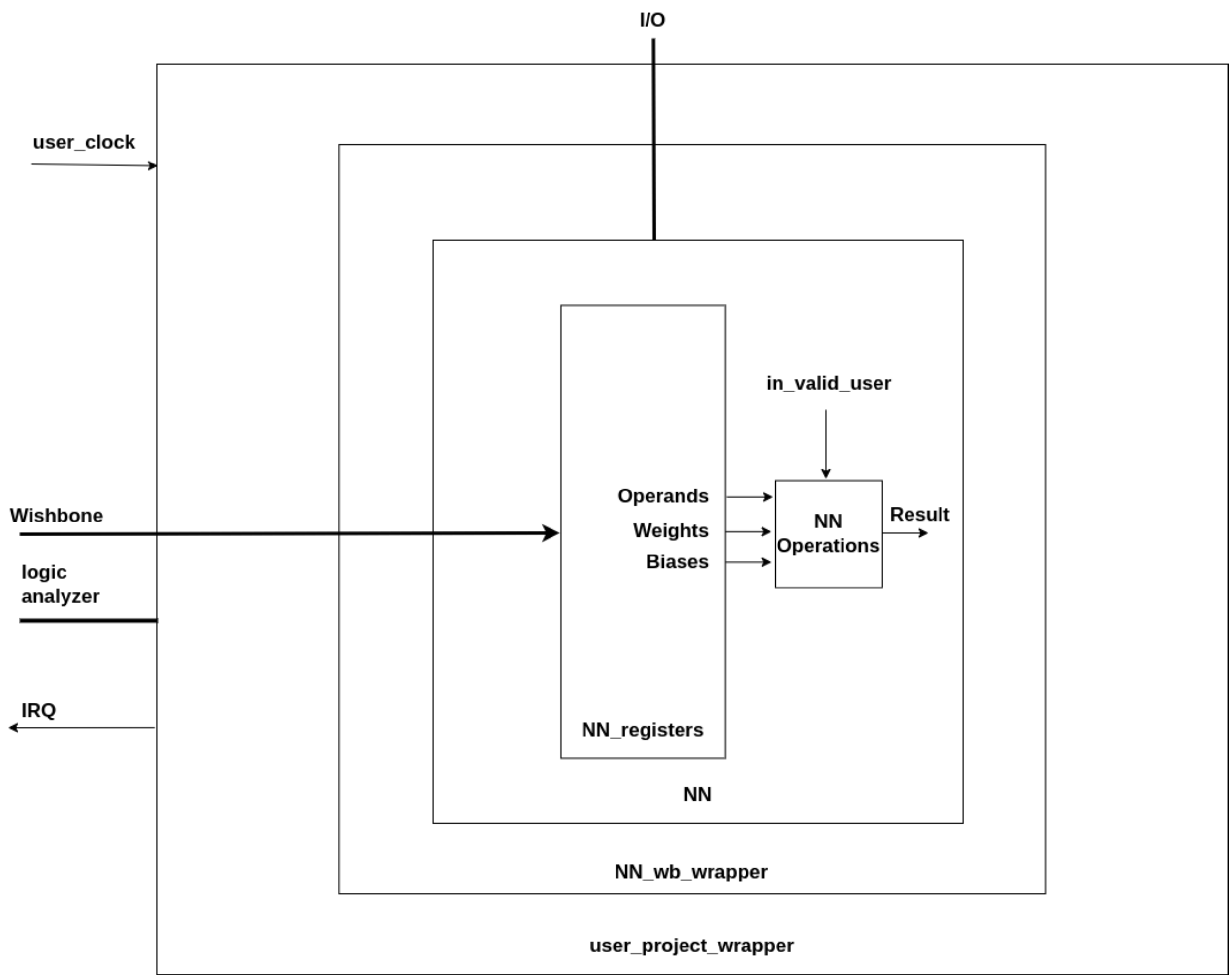

Figure 7: Hierarchy that illustrates how the NN top module is integrated into Caravel User Space via user_project_wrapper.

Flow of instructions

To perform a forward pass the following steps are required: 1. Write each operand with a store instruction to the core. 2. Once the operation is completed it can be accessed by the core on the GPIO pins.

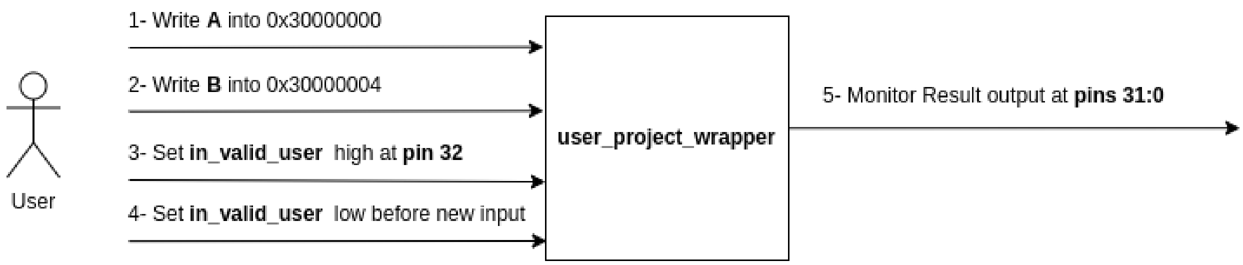

Figure: Flow of instructions