Synthese eines Asynchronen Multilayer Perceptron auf einem FPGA

Fabian Franz, Matr. Nr.: 644414

Juli 2021

Contents

1 Einführung

Dieses Projekt hat den Anspruch, ein effizientes neuronales Netzwerk auf einem FPGA zu entwerfen. Dazu geben die nächsten Abschnitte eine kurze Einführung in die grundlegenden Prinzipien, wie ein solches neuronales Netz modelliert werden kann.

1.1 Perceptron

Ein Perzeptron beschreibt ein analoges Modell einer biologischen menschlichen Zelle in Computerdomäne. Dieses Perzeptron kann mit der folgenden Grafik beschrieben und mathematisch formuliert werden:

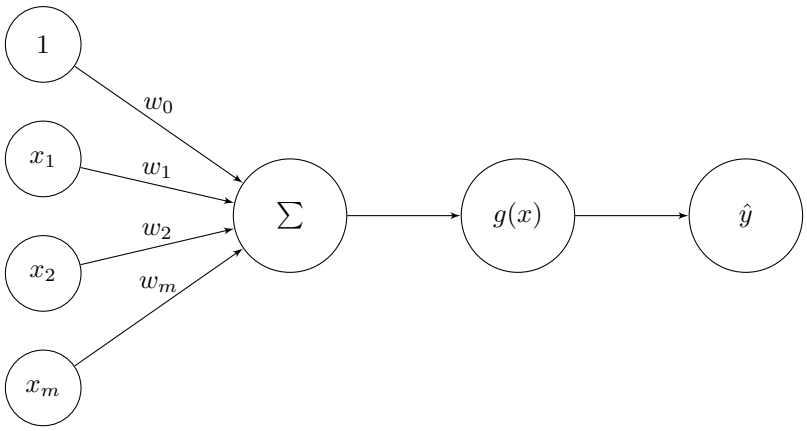


Figure 1: Model des Perzeptron

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right) \tag{1.1}$$

Mit:

g = Aktivierungsfunktion
w_o = Bias

In Vektor form:

$$\hat{y} = g(W_0 + X^T W) \tag{1.2}$$

Mit:

$$W = \begin{pmatrix} w_1 \\ \vdots \\ w_m \end{pmatrix}, \mathrm{X} = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}$$

Wie man sieht, wird in einem conventionellen Multilayer-Perzeptron jeder Eingang mit einer entsprechenden Gewichtung multipliziert. Es ist zu erkennen, dass die Umsetzung eines solchen Perzeptrons von einer gegebenen Hardwarearchitektur wie GPUs oder Matrix-Multiplizier-Einheiten abgearbeitet werden kann.

1.2 Aktivierungsfunktion

Die Aktivierungsfunktion eines Perzeptrons hat den Zweck, das Verhalten des Perzeptrons als Reaktion auf äußere Reize zu beeinflussen. Es gibt verschiedene Arten von Aktivierungsfunktionen, die für verschiedene Zwecke verwendet werden können.

2 Design Übersicht

2.1 Top Level

Die Hardware dieses Projekts besteht aus einem 32-Bit-Mikrocontroller namens SAM D21/DA1 und einem Zyklon 10 FPGA. Beide Chips sind auf einem Board platziert, dem sogenannten Arduino Vidor 4000. Dieses Board kommt mit einer USB-Schnittstelle, die die Kommunikation zwischen dem Frontend (PC) und der ausführenden Hardware realisiert. Die Abbildung 2 zeigt das abstrakte Hardwaremodell. Dieses Modell zeigt das erwähnte Frontend und die Hardware, aber auch den "Top Level Resolver", der die Daten entweder an den Speicher verteilt, oder die neuen Einstellungen für die einzelnen Perceptron-Aktivierungsfunktionen. Im konreten Modell werden insgesamt 16 Schichten mit 16 Perzeptronen pro Schicht realisiert. Der Einfachheit halber sind die Takt- und Reset-Signale für jedes einzelne Modul in der Abbildung nicht dargestellt.

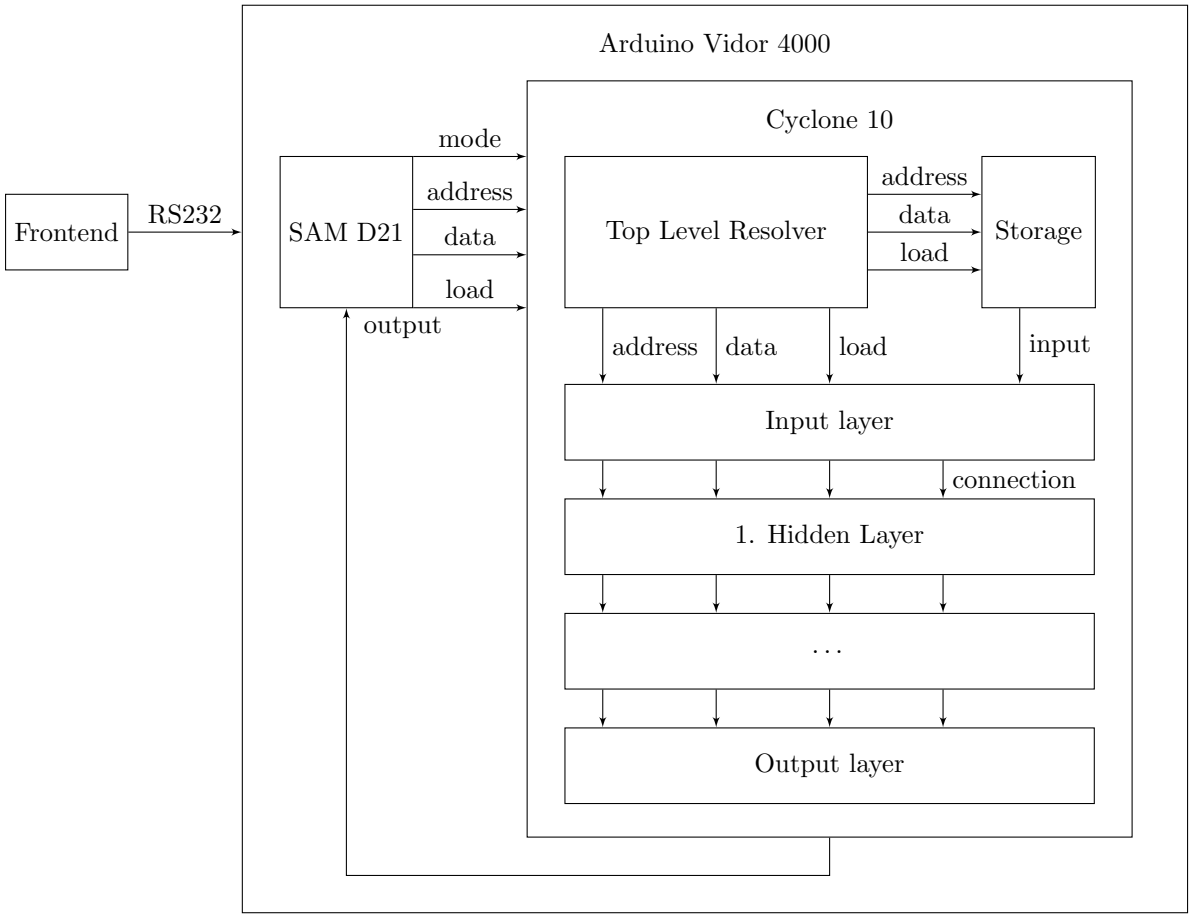


Figure 2: Top Level Hardware-Modell

2.2 Layer Level

Jede Schicht des Hardwaremodells besteht aus einem "Layer Resolver", der sich um die Signal-verteilung an jedes einzelne Perzeptron kümmert. Dies hat den Vorteil, dass die Datenleitungen insgesamt reduziert werden.

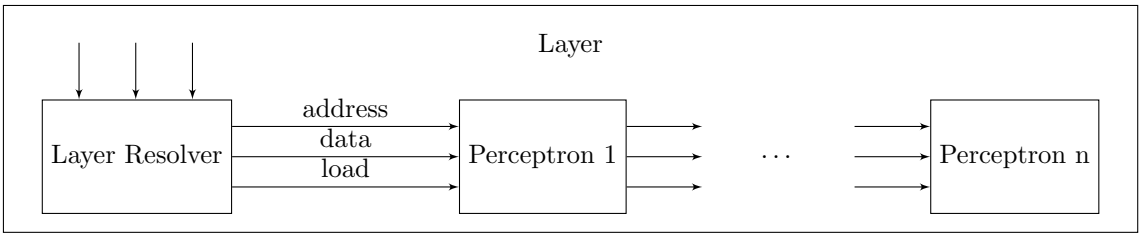


Figure 3: Layer Level Hardware Model

In abbildung 3 ist einer der 16 Layer grafisch dargestellt. Zu sehen ist, dass die Leitungen: "address", "data" und "load" jedes Perceptron miteinander verbindet und somit eine vollständige Adressierung und Datenbeaufschlagung garantiert wird.

3 Modul-Implementierung

Im folgenden wird die Implemetierung der einzelnen Module basierend auf der Hardware-übersicht beschrieben.

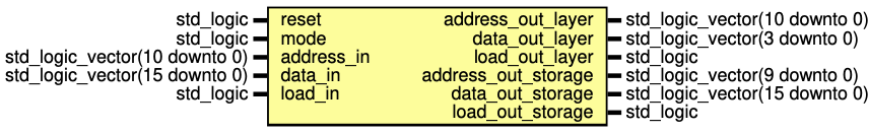
3.1 Top Level Resolver

Der "Top Level Resolver" realisiert eine Aufteilung und Skalierung der Signale von dem Eingang des Systems zu den Modulen "Storage" und "Layer Resolver". Konkret bedeutet das, dass wenn über die Signalleitung "mode" eine logische '1' anliegt die einzelnen schichten und somit die Perzeptronen adressiet werden. Liegt jedoch eine logische '0' an dem Eingang des "Top Level Resolver" an, so wird der Speicher, welcher die späteren Funktions-Werte des Systems enthält, adressiert.

Top Level Resolver

- **File:** top_level_resolver.vhdl
- **Author:** Fabian Franz (fabian.franz0596@gmail.com)
- **Version:** 0.1
- **Date:** 06.2021

Diagram



Description

This module implement the data and address handling switch between the layer modules and the storage module.

Generics and ports

Table 1.1 Generics

Table 1.2 Ports

Port name	Direction	Type	Description
reset	in	std_logic	global reset
mode	in	std_logic	select the direction
address_in	in	std_logic_vector(10 downto 0)	the address of data storage
data_in	in	std_logic_vector(15 downto 0)	the actual data to store
load_in	in	std_logic	triggers the storage
address_out_layer	out	std_logic_vector(10 downto 0)	address casting, mode='1'
data_out_layer	out	std_logic_vector(3 downto 0)	data forwarding
load_out_layer	out	std_logic	triggers storage
address_out_storage	out	std_logic_vector(9 downto 0)	address casting, mode='0'
data_out_storage	out	std_logic_vector(15 downto 0)	data forwarding
load_out_storage	out	std_logic	triggers storage

Processes

- **behaviour**: ***(reset, mode, address_in, data_in, load_in)***

Figure 4: Modulbeschreibung "Top Level Resolver"

In Abbildung 4 ist die Modulbeschreibung dargestellt. Daraus ist ersichtlich, dass die Datenleitung, welche an die einzelnen "Layer Resolver"-Module geleitet wird in eine 4Bit-Datenleitung gekürzt wird. Des weiteren wird die Adressleitung für das "Storage"-Modul in eine 10Bit-Datenleitung gekürzt. Daraus ergibt sich die Möglichkeit 1280 verschiedene 4Bit-Datenregister für die Perzeptron-Schichten und 1024 verschiedene 16Bit-Register für die Funktionseingabe in der erstem Perceptron-Schicht zu realisieren.

```
1  behaviour : process (reset, mode, address_in, data_in, load_in) is
2    begin
3      if reset = '1' then
4        ------------------------------------------
5        -- Reset all Outputs and internal Variables --
6        ------------------------------------------
7        address_out_layer   <= (others => '0');
8        data_out_layer      <= (others => '0');
9        load_out_layer      <= '0';
10       address_out_storage <= (others => '0');
11       data_out_storage    <= (others => '0');
12       load_out_storage    <= '0';
13     elsif reset = '0' then
14       ----------------
15       -- Layer Branch --
16       ----------------
17       if mode = '0' then
18         address_out_layer <= address_in;
19         data_out_layer    <= data_in(3 downto 0); -- least significant bits
20         load_out_layer    <= load_in;
21         ---------------------------
22         -- All others to default --
23         ---------------------------
24         address_out_storage <= (others => '0');
25         data_out_storage    <= (others => '0');
26         load_out_storage    <= '0';
27         ----------------
28         -- Storage Branch --
29         ----------------
30       elsif mode = '1' then
31         address_out_storage <= address_in(9 downto 0); -- least significant bits
32         data_out_storage    <= data_in;
33         load_out_storage    <= load_in;
34         ---------------------------
35         -- All others to default --
36         ---------------------------
37         address_out_layer <= (others => '0');
38         data_out_layer    <= (others => '0');
39         load_out_layer    <= '0';
40       end if;
41     end if;
42   end process;
```

Figure 5: Programmcode zu "Top Level Resolver"

In dem Programmcode aus 5 ist ersichtlich, dass es sich lediglich um rein kombinatorische Logik handelt, welche keinen externen Takt benötigt. Es handelt sich bei dem gezeigten Code lediglich um den "Process", die "Entity" und "Architecture" wurden im Folgenden näher erleutert.

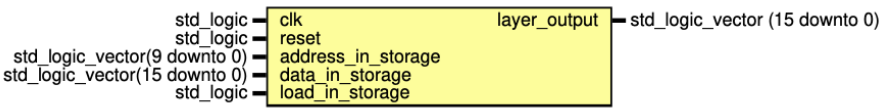
3.2 Storage

Das Modul "Storage" ist für die Speicherung der Eingangs-Funktionswerte zuständig, um das Modell des Multilayer-Perzeptrons mit Daten zu versorgen und optimierungen auf bestimmte Funktionen vor zu nehmen.

Perceptron Storage

- **File:** storage.vhdl
- **Author:** Fabian Franz (fabian.franz0596@gmail.com)
- **Version:** 0.1
- **Date:** 31.05.2021

Diagram



Description

This module implement the storage of 1024 16Bit Values and output them on each clock iteration.

Generics and ports

Table 1.1 Generics

Table 1.2 Ports

Port name	Direction	Type	Description
clk	in	std_logic	clock for iteration over storage values
reset	in	std_logic	reset to set default values
address_in_storage	in	std_logic_vector(9 downto 0)	address where tha value have to be stored
data_in_storage	in	std_logic_vector(15 downto 0)	actual data which have to be stored
load_in_storage	in	std_logic	trigger the storage
layer_output	out	std_logic_vector (15 downto 0)	output to the first layer

Signals, constants and types

Signals

Name	Type	Description
stored_value	arr_1024_times_16	

Types

Name	Type	Description
arr_1024_times_16		the one dimensional array of stored values

Processes

- **behaviour**: ***(load_in_storage, clk, reset)***

Figure 6: Modulbeschreibung "Layer Resolver"

In der Modulbeschreibung aus Abbildung 6 ist ersichtlich, dass der Speicher eine Adressweite von 10Bit besitzt und 16Bit Werte speichern kann. Des weiteren wird aus dem Code in Abbildung ersichtlich, dass bei anlegen eines Taktsignals "clk" die gespeicherten Werte iterativ am Ausgang "layer_output" ausgegeben werden. Der "layer_output" stellt die Verbindung zwischen dem Speichermodul und der ersten Perzeptron-Schicht dar.

```
1     behaviour : process (load_in_storage, clk, reset)
2     variable count : integer range 0 to 1023;
3   begin
4     ------------------------
5     -- reset handling --
6     ------------------------
7     if (reset = '1') then
8       -- Reset the 16Bit values of the whole 10Bit long array.
9       for i in 0 to 1023 loop
10        stored_value(i) <= (others => '0');
11        count := 0;
12      end loop;
13    end if;
14    ------------------------
15    -- input handling --
16    ------------------------
17    if (rising_edge(load_in_storage)) then
18      -- If the load is triggered, safe the value at the address.
19      stored_value(to_integer(unsigned(address_in_storage))) <= data_in_storage;
20    end if;
21    ------------------------
22    -- output handling --
23    ------------------------
24    if (rising_edge(clk)) then
25      layer_output <= stored_value(count);
26      count := count + 1;
27      if (count = 1023) then
28        count := 0;
29      end if;
30    end if;
31  end process;
```

Figure 7: Programmcode zu "Storage"

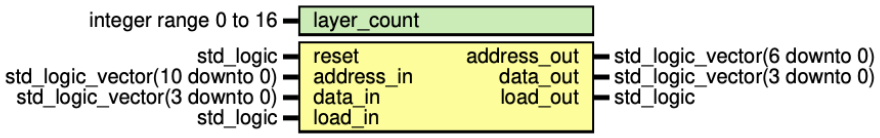
3.3 Layer Resolver

Der "Layer Resolver" übernimmt die Rolle des Bindegliedes zwischen dem "Top Level Resolver" und den einzelnen Schichten mit Perzeptronen.

layer_resolver

- **File:** pereptron.vhdl
- **Author:** Fabian Franz (fabian.franz0596@gmail.com)
- **Version:** 0.1
- **Date:** 18.05.2021

Diagram



Description

This module is designed for resolving the signals in every
single layer of the multilayer perceptron.
The address resolving is based on the "layer_count" variable, which
determine the number of every single layer in the whole multilayer perceptron.
Based on this number, the address is forwarded by a 7-Bit bus to the
single perceptrons in the layer.

Generics and ports

Table 1.1 Generics

Generic name	Type	Value	Description
layer_count	integer range 0 to 16		identifier for the current layer

Table 1.2 Ports

Port name	Direction	Type	Description
reset	in	std_logic	reset to default output values
address_in	in	std_logic_vector(10 downto 0)	input address from "Top Level Resolver"
data_in	in	std_logic_vector(3 downto 0)	input data from "Top Level resolver"
load_in	in	std_logic	load input from "Top Level Resolver"
address_out	out	std_logic_vector(6 downto 0)	addressing the sensitivity and activation value in the "Perceptron"
data_out	out	std_logic_vector(3 downto 0)	the actual value for sensitivity and activation in the "Perceptron"
load_out	out	std_logic	triggers the storage in the "Perceptron"

Processes

- **behaviour**: ***(address_in, reset, load_in, data_in)***

Figure 8: Modulbeschreibung "Layer Resolver"

In der Modulbeschreibung aus Abbildung 8 ist ersichtlich, dass der "Layer Resolver" bei seiner Instanziierung einen Parameter übergeben bekommt, welcher angibt in welcher Schicht sich der "Layer Resolver" befindet. Dies ist nötig, um die Adressen richtig an die Perzeptrone in jeder einzelnen Schicht weiter zu leiten. Die 10Bit Eingangs-Adresse wird durch den "Layer Resover" in eine 6Bit Ausgangs-Adresse gewandelt.

```
behaviour : process (address_in, reset, load_in, data_in) is
variable perceptron_address : integer range 0 to 15;
  begin
    if reset = '1' then
      address_out <= (others => '0');
      data_out    <= (others => '0');
      load_out    <= '0';
    else
      ------------------------------
      -- detect valid address --
      ------------------------------
      if (load_in = '1') then
        -- address = xxxx xxxx xxx
        -- (layer, perceptron in layer, value)
        if (shift_right((unsigned(address_in) and "11110000000"), 7) = layer_count) then
          ------------------------------
          -- forward address and data --
          ------------------------------
          address_out <= address_in(6 downto 0);
          data_out    <= data_in;
          load_out    <= load_in;
        else
          ------------------------------------
          -- reset address and data output --
          ------------------------------------
          address_out <= (others => '0');
          data_out    <= (others => '0');
          load_out    <= '0';
        end if;
      end if;
    end if;
  end process;
```

Figure 9: Programmcode zu "Layer Resolver"

Es handelt sich weider um einen rein kombinatorischen Prozess ohne Taktabhandlung.

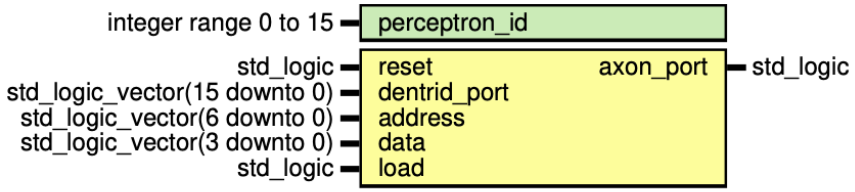
3.4 Perzeptron

Das Perzeptron selbst besteht aus einer binären Eingabesteuerung anstelle von Gleitkommaberechnungen. Das bedeutet, dass die Verbindung von einem Perzeptron-Ausgang zu einem anderen Perzeptron-Eingang nur aktiv oder nicht aktiv sein kann. Darüber hinaus ist der zweite Parameter innerhalb des Perzeptrons die Aktivierungsfunktion, die welche im Grunde die Summation über die "high"-Eingangssignale darstellt. Wenn die Summe größer ist als ein gegebener Wert wird der Ausgang des Perzeptrons auf "high" gesetzt. In dem Programmcode aus Abbildung 11 ist ersichtlich, dass die Adressierung des 16Bit-Vektors für die Gewichtung der Eingänge (aktiv oder nicht aktiv) auf insgesamt vier 4Bit-Vektoren aufgeteilt wird, welche von dem Perzeptron schließlich zu einem 16Bit-Vektor zusammengefasst werden. Die Adresierung des "activation_value", welcher eine Aussage darüber trifft, ab wie vielen logischen "high"- Werten der Ausgang eine "1" annimt wird auf die selbe Weise adressiert. Über die "data"-Verbindung wird schließlich den adressierten Speichern ein Wert zugewiesen. Zusammenfasend ist zu sagen, das die letzten 3-Bit in der Adresse immer die Adressierung der spezifischen Aktivierungsfunktionen darstellt.

Perceptron

- **File:** pereptron.vhdl
- **Author:** Fabian Franz (fabian.franz0596@gmail.com)
- **Version:** 0.1
- **Date:** 18.05.2021

Diagram



Description

This module describes the actual behaviour of the combinatorical perceptron. The perceptron is capable of holding the values for input sensitivity and the output activation.

Generics and ports

Table 1.1 Generics

Generic name	Type	Value	Description
perceptron_id	integer range 0 to 15		the ID of the perceptron in the specific layer

Table 1.2 Ports

Port name	Direction	Type	Description
reset	in	std_logic	reset inputs and outputs of the entity to default values
dentrid_port	in	std_logic_vector(15 downto 0)	input from previous layer
axon_port	out	std_logic	output to next layer
address	in	std_logic_vector(6 downto 0)	current address for parameter manipulation
data	in	std_logic_vector(3 downto 0)	value of the addressed parameter
load	in	std_logic	signal to actually store the addressed parameter value

Signals, constants and types

Signals

Name	Type	Description
activation_value	unsigned (3 downto 0)	threshold parameter for input count until output is set to one
sensitivity_value	unsigned (15 downto 0)	determine whis inputs are activated and counted.

Figure 10: Modulbeschreibung ”Perzeptron”

```
behaviour : process (load, reset, dentrid_port, address, data) is
    variable count     : unsigned(3 downto 0)           := (others => '0');
    variable old_value : std_logic_vector (15 downto 0) := (others => '0');
  begin
    ------------------
    -- reset handling --
    ------------------
    if (reset = '1') then
      axon_port         <= '0';
      activation_value  <= (others => '1');
      sensitivity_value <= (others => '0');
      sens_1            <= (others => '0');
      sens_2            <= (others => '0');
      sens_3            <= (others => '0');
      sens_4            <= (others => '0');
      count     := (others         => '0');
      old_value := (others         => '0');
    else
      ---------------------------
      -- value storage handling --
      ---------------------------
      if load = '1' then
        -- check if the current perceptron is meant (first 4 Bit)
        if (shift_right((unsigned(address) and "1111000"), 3) = perceptron_id) then
          -- check the parameters (last 3 Bit)
          if ((unsigned(address) and "0000111") = 0) then
            activation_value <= unsigned(data);
          end if;
          if ((unsigned(address) and "0000111") = 1) then
            sens_1 <= unsigned(data);
          end if;
          if ((unsigned(address) and "0000111") = 2) then
            sens_2 <= unsigned(data);
          end if;
          if ((unsigned(address) and "0000111") = 3) then
            sens_3 <= unsigned(data);
          end if;
          if ((unsigned(address) and "0000111") = 4) then
            sens_4 <= unsigned(data);
          end if;
        end if;
        sensitivity_value <= sens_4 & sens_3 & sens_2 & sens_1;
      end if;
      ------------------
      -- output handling --
      ------------------
      if dentrid_port /= old_value then
        -- count over all Bits and proceed sensitivity and and output activation
        count := "0000";
        for i in 0 to 15 loop
          if ((dentrid_port(i) = '1') and (sensitivity_value(i) = '1')) then
            count := count + 1;
          end if;
        end loop; -- counter
        old_value := dentrid_port;
        -- update axon
        if (count = activation_value) then
          axon_port <= '1';
        else
          axon_port <= '0';
        end if;
      end if;
    end if;
  end process;
```

Figure 11: Programmcode zum "Perzeptron"

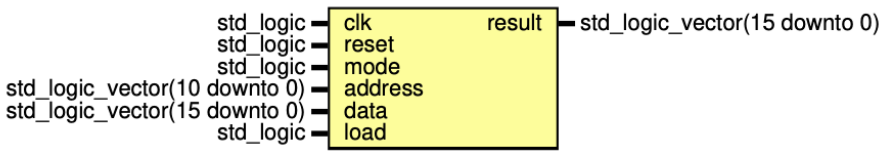
4 Gesamt-Implementierung

Im folgenden wird die Gesamtimplementierung basierend auf der Hardwareübersicht erläutert dabei wird lediglich auf die Instanziierung der Module eingegangen.

Top Level Entity

- **File:** top_level_entity.vhdl
- **Author:** Fabian Franz (fabian.franz0596@gmail.com)
- **Version:** 0.1
- **Date:** 14.06.2021

Diagram



Description

This module has the purpose to put all modules
of the perceptron project together.

Generics and ports

Table 1.1 Generics

Table 1.2 Ports

Port name	Direction	Type	Description
clk	in	std_logic	global clock signal
reset	in	std_logic	global reset signal
mode	in	std_logic	select the direction '0' = layer, '1' = storage
address	in	std_logic_vector(10 downto 0)	the address of data storage in the layer or storage branch
data	in	std_logic_vector(15 downto 0)	the actual data to store
load	in	std_logic	triggers the storage
result	out	std_logic_vector(15 downto 0)	the result from the last layer

Figure 12: Modulbeschreibung "Top Level Entity"

In der Abbildung 12 ist die Beschreibung der oberen Instanz des Systems beschrieben. Diese generiert alle weiteren Module und stellt deren Verbindung untereinander dar. Wie diese Verbindung realisiert ist wird im folgenden erläutert.

```
1  ------------------------
2  -- Internal Signals --
3  ------------------------
4  -- connection top level resolver to layer
5  signal address_int_layer : std_logic_vector(10 downto 0) := (others => '0');
6  signal data_int_layer    : std_logic_vector(3 downto 0)  := (others => '0');
7  signal load_int_layer    : std_logic                     := '0';
8  -- connection top level resolver to storage
9  signal address_int_storage : std_logic_vector(9 downto 0)  := (others => '0');
10 signal data_int_storage    : std_logic_vector(15 downto 0) := (others => '0');
11 signal load_int_storage    : std_logic                     := '0';
12 -- connect storage to first layer
13 signal storage_to_first_layer : std_logic_vector(15 downto 0) := (others => '0');
14 -- layer connection arrays
15 -- axon ports to next layer
16 type arr_16_times_16 is array (0 to 14) of std_logic_vector(15 downto 0);
17 signal layer_axon_arr : arr_16_times_16;
18 -- address from layer resolver to layer
19 type arr_16_times_7 is array (0 to 15) of std_logic_vector(6 downto 0);
20 signal address_layer_arr : arr_16_times_7;
21 -- data from layer resolver to layer
22 type arr_16_times_4 is array (0 to 15) of std_logic_vector(3 downto 0);
23 signal data_layer_arr : arr_16_times_4;
24 -- load from layer resolver to layer
25 signal load_layer : std_logic_vector (15 downto 0);
```

Figure 13: Programmcode der internen Signale

Name	Type	Description
address_int_layer	std_logic_vector(10 downto 0)	connect adress line of "Top Level Resolver" and "Layer Resolver"
data_int_layer	std_logic_vector(3 downto 0)	connect data line of "Top Level Resolver" and "Layer Resolver"
load_int_layer	std_logic	connect load line of "Top Level Resolver" and "Layer Resolver"
address_int_storage	std_logic_vector(9 downto 0)	connect address line of "Top Level Resolver" and "Storage"
data_int_storage	std_logic_vector(15 downto 0)	connect data line of "Top Level Resolver" and "Storage"
load_int_storage	std_logic	connect load line of "Top Level Resolver" and "Storage"
storage_to_first_layer	std_logic_vector(15 downto 0)	connect the storage output with the first layer

Figure 14: Beschreibung der internen Signale der "Top Level Entity"

layer_axon_arr	arr_16_times_16	connect axon ports between layer and layer resolver
address_layer_arr	arr_16_times_7	connect address ports between layer and layer resolver
data_layer_arr	arr_16_times_4	connect data ports between layer and layer resolver
load_layer	std_logic_vector (15 downto 0)	connect load signal between layer and layer resolver

Figure 15: Fortführung der Beschreibung der internen Signale der "Top Level Entity"

In Abbildung 14 und 15 sind die internen Signale, welche zur Verbindung der Module nötig sind beschrieben. Alle internen Signale wurden kompatibel zu den Schnittstellen angelegt. Die jeweilige Verbindung einer Perzeptron-Schicht mit der nächsten im Bezug auf die Daten- und Adressleitung wird jeweils durch ein 2-dimensionales "Array" aus 16Bit Vektoren realisiert.

4.1 Instanziierung Der layer

Im folgenden wird lediglich auf die Instanziierung der Layer eingegeangen. Die Instanziierung der restlichen Module funktioniert ähnlich dem der einzelnen Layer.

```
1  -- instances of layer resolver
2  layer_resolvers : for i in 0 to 15 generate -- generate 16 layer resolvers
3    layer_resolver_inst : layer_resolver
4    generic map(
5      layer_count => i
6    )
7    port map(
8      reset       => reset,
9      address_in  => address_int_layer, -- tlr to resolvers
10     data_in     => data_int_layer, -- ""
11     load_in     => load_int_layer, -- ""
12     address_out => address_layer_arr(i), -- layer resolver to perceptron array
13     data_out    => data_layer_arr(i), -- ""
14     load_out    => load_layer(i)
15   );
16 end generate;
17 -- two dimensional layer array of perceptrons 14 x 16 (layer x perceptron)
18 layer_arr_2D : for i in 1 to 14 generate -- layer
19   layer_arr_1D : for j in 0 to 15 generate -- perceptron in layer
20     perceptron_arr_inst : perceptron
21     generic map(
22       perceptron_id => j
23     )
24     port map(
25       reset        => reset,
26       dentrid_port => layer_axon_arr(i - 1), -- prev layer to current layer
27       axon_port    => layer_axon_arr(i)(j), -- current to next layer
28       address      => address_layer_arr(i), -- layer resolver to perceptron array
29       data         => data_layer_arr(i), -- ""
30       load         => load_layer(i) -- ""
31     );
32   end generate;
33 end generate;
```

Figure 16: Programmcode der internen Signale

In der Abbildung 16 ist zu sehen, dass insgesamt 16 "layer_resolver" und 13 Schichten mit Perzeptronen instanziiert werden. In jeder Perzeptron-Schicht befinden sich wiederum jeweils 16 Perzeptronen. Wie bereits in Abschnitt 4 erwähnt, werden die erzeugten Pereptronen bei der Instanziierung mit den internen Signalen verbunden.

5 Validierung

Die Validierung der einzelnen Module, der Integration verschiedener Module und der des gesamten Systems wird mit Hilfe der Simulation auf Regeister-Transfer-Ebene in "Modelsim" realisiert. Alle Tests können durch hinzufügen der benötigten Module und das Ausführen der Modelsim-Skripte mit der Endung .do ausgeführt werden. Die erwähnten Skripte befinden sich im Ordner "SimulationConf".

5.1 Modultest

Im Folgenden werden die Modultests zu allen Modulen erläutert.

5.1.1 Top Level Resolver

Der "Top Level Resolver" wird mit besonderem Augenmerk auf seine Reaktion auf die verschiedenen Modi untersucht. Er dient als Stellglied für die Weiterleitung und Anpassung der Länge der Signale. Um die Funktion dahingehend zu überprüfen wurde folgender Ablauf innerhalb einer Testbench realisiert:

```
1  beh_process : process
2    begin
3      reset       <= '1';
4      mode        <= '0';
5      address_in  <= "00000000000";
6      data_in     <= "0000000000000000";
7      load_in     <= '0';
8      wait for 2 * clk_period;
9      reset <= '0';
10     ------------------------------
11     mode        <= '0';
12     load_in     <= '1';
13     address_in  <= "00101001001";
14     data_in     <= "0000000000001101";
15     wait for 2 * clk_period;
16     ------------------------------
17     -- addressing the storage --
18     ------------------------------
19     mode        <= '1';
20     load_in     <= '1';
21     address_in  <= "01101100000";
22     data_in     <= "0000000000000011";
23     wait for 2 * clk_period;
24     report "Simulation Stop";
25     stop;
26   end process;
```

Figure 17: Programmcode der "Top Level Resolver" Testbench

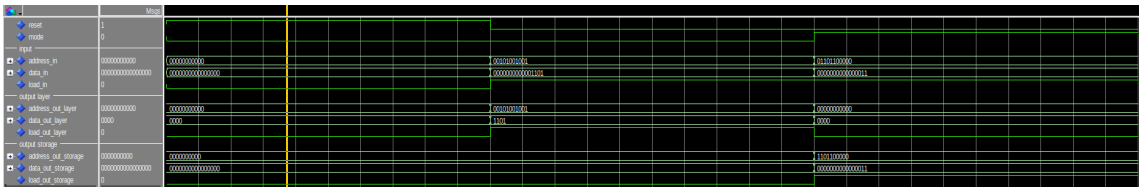


Figure 18: Simulationsergebnis "Top Level Resolver"

In dem Simulationsergebnis in Abbildung 18 ist zu sehen, dass der "Top Level Resolver" das erwartete Verhalten erfüllt und die Signale je nach gewähltem "mode" weiter leitet. Des weiteren ist auch die Funktionalität der Bearbeitung der Signale im Bezug auf deren Länge erfüllt.

5.1.2 Storage

Das Modul "Storage" wird auf seine Funktionalität im Bezug auf die Speicherung und Wiedergabe von 16Bit-Zahlen geprüft. Dafür werden in einer Testumgebung Zahlen erzeugt, in den Speicher geschrieben und danach wieder iterativ ausgegeben.

```
1  behaviour : process (load_in_storage, clk, reset)
2      variable count : integer range 0 to 1023;
3    begin
4      ------------------------
5      -- reset handling --
6      ------------------------
7      if (reset = '1') then
8        -- Reset the 16Bit values of the whole 10Bit long array.
9        for i in 0 to 1023 loop
10         stored_value(i) <= (others => '0');
11         count := 0;
12       end loop;
13     end if;
14     ------------------------
15     -- input handling --
16     ------------------------
17     if (rising_edge(load_in_storage)) then
18       -- If the load is triggered, safe the value at the address.
19       stored_value(to_integer(unsigned(address_in_storage))) <= data_in_storage;
20     end if;
21     ------------------------
22     -- output handling --
23     ------------------------
24     if (rising_edge(clk)) then
25       layer_output <= stored_value(count);
26       count := count + 1;
27       if (count = 1023) then
28         count := 0;
29       end if;
30     end if;
31   end process;
```

Figure 19: Programmcode der "Storage" Testbench

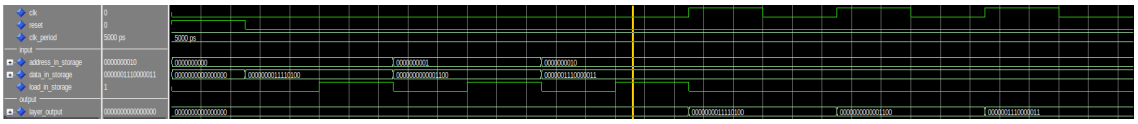


Figure 20: Simulationsergebnis "Storage"

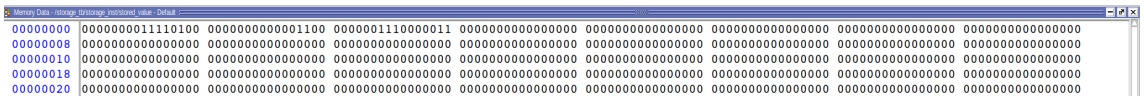


Figure 21: Speicherwerte "Storage"

In dem Simulationsergebnis aus Abbildung 20 ist zu sehen, dass die Funktionswerte korrekt am Ausgang nach dem Speichern ausgegeben werden. Außerdem wird der Speicher nach dem Zurücksetzen korrekt mit "0" gefüllt und die Werte ein weiteres mal durch das Prüfen mit der Funktion "Memory Data" in "Modelsim", wie in Abbildung 21 zu sehen, geprüft.

5.1.3 Layer Resolver

Der "Layer Resolver" hat die Funktion, die eingehenden Adress- und Datensignale auf die einzelnen Perzeptronen in der jeweiligen Schicht zu verteilen. Dies geschieht, wenn die richtige Adresse der jeweiligen Schicht codiert ist. In der folgenden Testbench in Abbildung 22 wird dieses Verhalten geprüft.

```
1  check_beh : process
2    begin
3      ------------------------------
4      -- assign default values --
5      ------------------------------
6      address_in <= (others => '0');
7      data_in    <= (others => '0');
8      load_in    <= '0';
9      ------------------------
10     -- trigger reset --
11     ------------------------
12     reset <= '1';
13     wait for 1.5 * clk_period;
14     reset <= '0';
15     ---------------------------
16     -- set first address --
17     ---------------------------
18     -- adress format: xxxx xxxx xxx (layer, perceptron, value)
19     load_in    <= '1';
20     address_in <= "00000011010"; -- (0, 3, 2)
21     data_in    <= std_logic_vector(to_unsigned(4, data_in'length));
22     wait for clk_period;
23     ----------------------------
24     -- set second address --
25     ----------------------------
26     load_in    <= '1';
27     address_in <= "00010011010"; -- (1, 3, 2)
28     data_in    <= std_logic_vector(to_unsigned(5, data_in'length));
29     wait for clk_period;
30     ------------------------------------
31     -- check the reset functionality --
32     ------------------------------------
33     reset <= '1';
34     wait for 2 * clk_period;
35     reset <= '0';
36     stop;
37   end process;
```

Figure 22: Programmcode der "Layer Resolver" Testbench

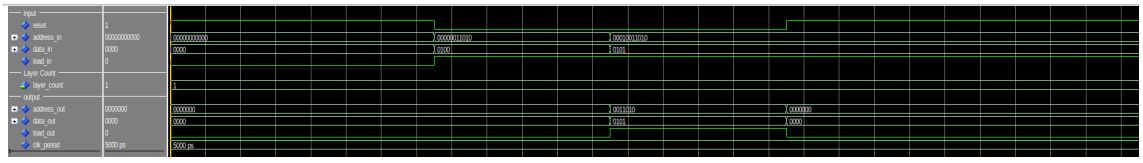


Figure 23: Simulationsergebnis "Layer Resolver"

In dem Simulationsergebnis aus Abbildung 23 ist zu erkennen, dass die Daten- und Adressignale lediglich dann weiter geleitet und im Fall der Adressleitung gekürzt wird, sofern die richtige Schicht, in dem Fall die Schicht 1 adressiert wurde.

5.1.4 Perzeptron

Das Modul "Perzeptron" realisiert die eigentliche Funktion des am Ende in der "Top Level Entity" gebildeten Netzes dieser. Die Testbench in Abbildung 24 und 25 prüft, ob die Werte für die Eingangssensitivität und den Schwellwert der Aktivierung gespeichert werden und schließlich auf Eingaben basierend auf den zuvor eingestellten Werten entsprechend reagiert wird.

```
1  check_beh : process
2    begin
3      ------------------------
4      -- default values --
5      ------------------------
6      reset        <= '0';
7      dentrid_port <= (others => '0');
8      address      <= (others => '0');
9      data         <= (others => '0');
10     load         <= '0';
11     wait for clk_period;
12     reset <= '1';
13     wait for clk_period;
14     reset <= '0';
15     wait for clk_period;
16     ------------------------------
17     -- set activation value --
18     ------------------------------
19     address <= "0000000";
20     wait for clk_period;
21     data <= "1111";
22     wait for clk_period;
23     load <= '1';
24     wait for clk_period;
25     load <= '0';
26     wait for clk_period;
27     ---------------------------------
28     -- set a sensitivity value --
29     ---------------------------------
30     -- sens_1
31     address <= "0000001";
32     wait for clk_period;
33     data <= "1111";
34     wait for clk_period;
35     load <= '1';
36     wait for clk_period;
37     load <= '0';
38     wait for clk_period;
39     -- sens_2
40     address <= "0000010";
41     wait for clk_period;
42     data <= "1111";
43     wait for clk_period;
44     load <= '1';
45     wait for clk_period;
46     load <= '0';
47     wait for clk_period;
```

Figure 24: Programmcode der "Perzeptron" Testbench

```
    -- sens_3
    address <= "0000011";
    wait for clk_period;
    data <= "1111";
    wait for clk_period;
    load <= '1';
    wait for clk_period;
    load <= '0';
    wait for clk_period;
    -- sens_4
    address <= "0000100";
    wait for clk_period;
    data <= "1111";
    wait for clk_period;
    load <= '1';
    wait for clk_period;
    load <= '0';
    wait for clk_period;
    -- final load to set the sens value
    load <= '1';
    wait for clk_period;
    load <= '0';
    wait for clk_period;

    ------------------------------------------------------------
    -- check the programmed values and output behaviour --
    ------------------------------------------------------------

    dentrid_port <= "1111111111111111";
    wait for 2 * clk_period;
    load <= '1';
    wait for clk_period;
    load <= '0';
    wait for clk_period;
    dentrid_port <= "1111111111111110";
    wait for 2 * clk_period;
    load <= '1';
    wait for clk_period;
    load <= '0';
    wait for clk_period;
    dentrid_port <= "1111111111111111";
    wait for 2 * clk_period;
    load <= '1';
    wait for clk_period;
    load <= '0';
    wait for clk_period;
    report "Simulation Stop";
    stop;
    end process;
```

Figure 25: Fortsetzung des Programmcode der "Perzeptron" Testbench

5.2 Integrationstest

5.3 Systemtest

5.4 Timing-Analyse