Neural Net (NN) Perceptron Pattern Recognition Specification

Author: Jens Gutschmidt

opencores@vivare-services.com

Rev. [1.2]

July 28, 2022

This page has been intentionally left blank.

Revision History

Rev.	Date	Author	Description
0.1	07/10/22	[JDG]	First Draft
1.0	07/20/22	[JDG]	Last modifications and completed Appendix B – Sample Project section
1.1	07/22/22	[JDG]	Appendix B – Sample Project The value for Threshold is wrong in the test bench v04 (0x25 – changed to 0x20). In result of that, show results of training (p.44-48) are also wrong. "Table 13 – Targets" wrong. Values, tables and pictures corrected.
1.2	07/28/22	[JDG]	Remove bindings to "opencores.org"

Contents

Figures

Equations

Tables

1 Introduction

- Full synthesizable VHDL core for FPGA with on-chip memory
- Wishbone compatible (V.B4)
- User specific pre-configurable on-chip memory configuration
- On-the-fly memory windowing within pre-configured memory space
- No multiplications or DSP blocks - Only Add and Sub functions are used
- Auto-adjusting memory Wait State generator for Reading and Writing
- Enable/Disable Hardware Interrupt
- Perceptron type Neural Net for pattern recognition
- Bipolar s:t input data – t signed output
- Read from and write to any memory space (s, t, w matrix, bias and y)
- Built-in training module with user configurable *Max Epochs Counter*
- Threshold and Bias register for fine tuning of training
- Signed data types on the Wishbone data bus without masking

Technical Overview

This IP core for FPGA allows the user to add a function for pattern recognition to an electronic design with or without cpu support.

A Wishbone interface is included to control the core and read from/write to the register bank or on-chip memory space (s, t, w, bias or y).

Functions can be start by writing a dummy value to the appropriate function address. Most functions are completed within few cycles and can be observed by polling the status register. Function *training* is the only one which offers the option to generate a hardware interrupt after the training process has been finished. Training can be a time consuming process and is undeterminable.

The design was optimized *balanced* (60:40 – speed:area) on HDL level to fulfill most of daily work requirements. All paths between inputs and outputs of finite state machines are registered. Inputs of adders/subtractions are registered also.

After reset a built-in wait state generator test the available w-matrix memory to determine and adjust the best timing for reading and writing on the chosen target FPGA. The range is 0WS-7WS. Because of internal pipelining all memories lesser or equal to 2WSRD are handled as 3WSRD memories. Read latency include the preparation time of the address counter for x/y addresses. Write latency only measures the time for which data out is the same as written before without changing the address.

Because the IP core lacks any use of multiplications or DSP function blocks, any FPGA vendor and family can be used as long enough on-chip memory is available.

Signed data is used to connect the Wishbone data bus to the on-chip memory space which have a user specific data bus width. This helps to interpret the data in/out without having bus-width specific data masks to find out or isolate the sign-bit in software or hardware.

Memories

There are five memories (i/j -> x/y):

Name	Description
s	Hold the input components s (i direction)
t	Hold the input/output target t (neuron) (j direction)
w	Hold the weight matrix (i/j direction)
bias	Hold the bias matrix for each target t (neuron) (j direction)
y	Hold the temporary y values while training (j direction)

Table 1: Introduction - Memory Contents

The w-memory is two dimensional. All other memories are one dimensional.

All memories s, t, bias and y are organized in directions to form the w-matrix in the w-memory. This is the base for all calculations which are matrix related computations.

Before synthesizing the core, the user must write valid values of the maximum of the memory area x/y into a specific VHDL package file, holding all constants, definitions and declarations for the core.

After that, the memory area to use is programmable by the user on-the-fly while running the core.

Addressing Scheme

The memories are addressed row by row and column by column automatically by two address counters for x/y directions.

Both counters are able to increment or decrement their start values and are controlled by the current running function module.

They are organized internally as “For-Next-Loops”, named *i*- and *j-counter* and stop at their programmed end values automatically. These values must be programmed by the user before functions like *rd_wr_memory*, *training* or *testing* can be started.

i works as index for x-direction and j works as index for y-direction.

Every access to any type of memory (s, t, w, bias and y) must read/write from/to all addresses. Because the user only write the start and stop point to the start-/stop-register of *i-*/*j-counter*, it is not possible to address the memories randomly or directly.

For example, if the memory organization is 6x3, *s* is 6 *i* positions deep, t=3j, w=6i x 3j=18, bias=3j and y=3j and the user attempt to write to the s-memory, all six values must be written in correct order without any interruption. Pauses are allowed.

The tables below show the internal flow of data after the function “rd_wr_mem – write s” was started followed by the six necessary data words. Only the s-memory is affected by this operation, so other memories are shown with dotted border lines. Internally the address counter generate the next address automatically every time one data word was written (assumed start_i=0, stop_i=5). Read the appropriate status bit after each written data word lets the controlling process of “rd_wr_mem” know about the continuing or end of the function.

The address i-counter jump back to his start value automatically after reaches the stop value.

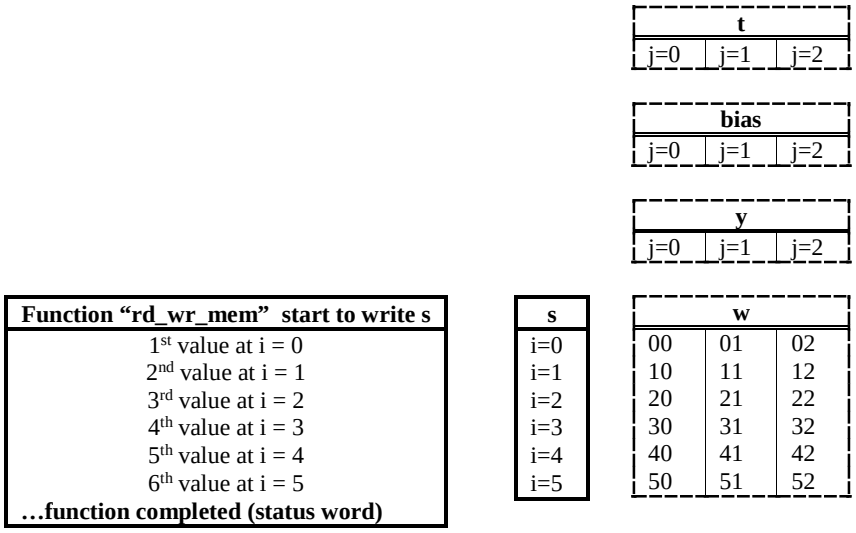


Figure 1: Introduction – Memory pointer while writing s memory

Data Types

All memories have the same size of data in/out. Although all data types are “signed” the range of *s* is only -1…1.

As result of that, the minimum data bus width of all memories is

2

Special care must be taken to fit the whole necessary number representation in all other memories for all computation scenarios.

s-memory

The input data *s* (components) is of data type “signed”, have the same data bus width as all other core related on-chip memories but is bipolar. There are three possible valid combinations of data values:

Orientation	Representation
$s_i < 0$	Representation of -1
$s_i = 0$	Representation of 0 (uncertain, noise)
$s_i > 0$	Representation of 1

Table 2: Introduction - Representation of *s*

The user is able to write other values into *s* but such values are always interpreted as -1, 0 or 1.

t-memory

The data *t* (targets) is of data type “signed” and is stored into the t-memory by the user for *training* or by *testing* module while testing the perceptron neural net against specific input pattern (components) *s*.

In contrast to the *s* value input, *t* is internally fully qualified as “signed” value and can be act as input or output.

As input, *t* represents the status of activation of all neural nodes while components *s* are present in s-memory for training (ONLY -1 and 1 as input values are recommended at this time).

Orientation	Representation
$t_j = -1$	Neuron should be NOT activated
$t_j = 0$	Leave w-matrix unchanged -> no learning
$t_j = 1$	Neuron should be activated

Table 3: Introduction - Representation of *t* as INPUT

As output, *t* represents the status of activation of all neural nodes while components *s* are present in s-memory for testing.

Orientation	Representation
$t_j <$ threshold	Neuron is NOT activated
$t_j >=$ threshold	Neuron is activated

Table 4: Introduction - Representation of t as OUTPUT

w-memory

The data *w* (weights) is of data type “signed” and is stored into the w-memory by the *training* process or by the user for working with pre-defined data sets.

It is used by *testing* or *training* module.

Perceptron Neural Net

The Perceptron Neural Net offers a very wide range of general purpose applications while bipolar input data (-1, 0, 1) for *s:t* (training) and *s* (testing) is the base for all further computations.

Writing the both dimensions of x/y of the memory space window to the specific registers initialize the core within the synthesized memory area. This method of dynamic memory allocation offers a maximum flexibility to the user to re-define the necessary memory matrix on-the-fly without having to re-synthesize the core after such change every time.

Training

Writing the components and the required answer *s:t* into the s-/t-memories prepare the training process.

Additional programmable parameters like *bias* and *threshold* values helps to obtain certain test results in specific different environments. As result of that, the kind of input data *s* can be noisy or exact.

Bias contains a bias value for each target neuron *t* individually. *Threshold* contains the net wide threshold value which must be reached to indicate an activation of any neuron *t*.

The *training* module process through the s:t values, generate the internal *y* matrix and calculate the *w* matrix while observing the *threshold* and *bias* for each target neuron *t*. This process iterates until the input components *s* activate all necessary target neurons *t* as required. If the *maxepochs* register was loaded with a value greater than zero, the training will end at this value.

Testing

Writing a complete set of components *s* into the s-memory prepares the test process for one pattern. It is not possible to change only parts of previously written pattern. Each time the complete set of components *s* must be written into s-memory.

After the test process have been finished, the t-memory contains the signed values for all *targets t*. All t-values must be than compared against the stored *threshold* value externally.

Datasets

A complete trained dataset can be read out from the bias-, w-memory and the *threshold* register respectively, to store it externally and to write it back at any time later to work with pre-trained data models.

2 Architecture

The design was divided into several modules and is “Wishbone Compatible”, v.B4. Each module contain a dedicated function like

- Wishbone interface (p0300_m00021_s_v03_wishbone_fsm)
- Calculation of *y* matrix (p0300_m00022_s_v02_cal_y_fsm)
- Calculation of *w* and *bias* matrix (p0300_m00023_s_v02_cal_w_fsm)
- Training (p0300_m00027_s_v01_train_fsm)
- Testing – determine the response of perceptron (p0300_m00024_s_v02_test_fsm)
- Read from and Write to memory (p0300_m00026_s_v02_rd_wr_fsm)

Some more modules are for internal use.

To fulfill the performance requirements of this IP core, all modules were designed as Finite State Machines (FSM) of type “Moore” (mainly).

On the next page the block diagram is shown.

Block Diagram

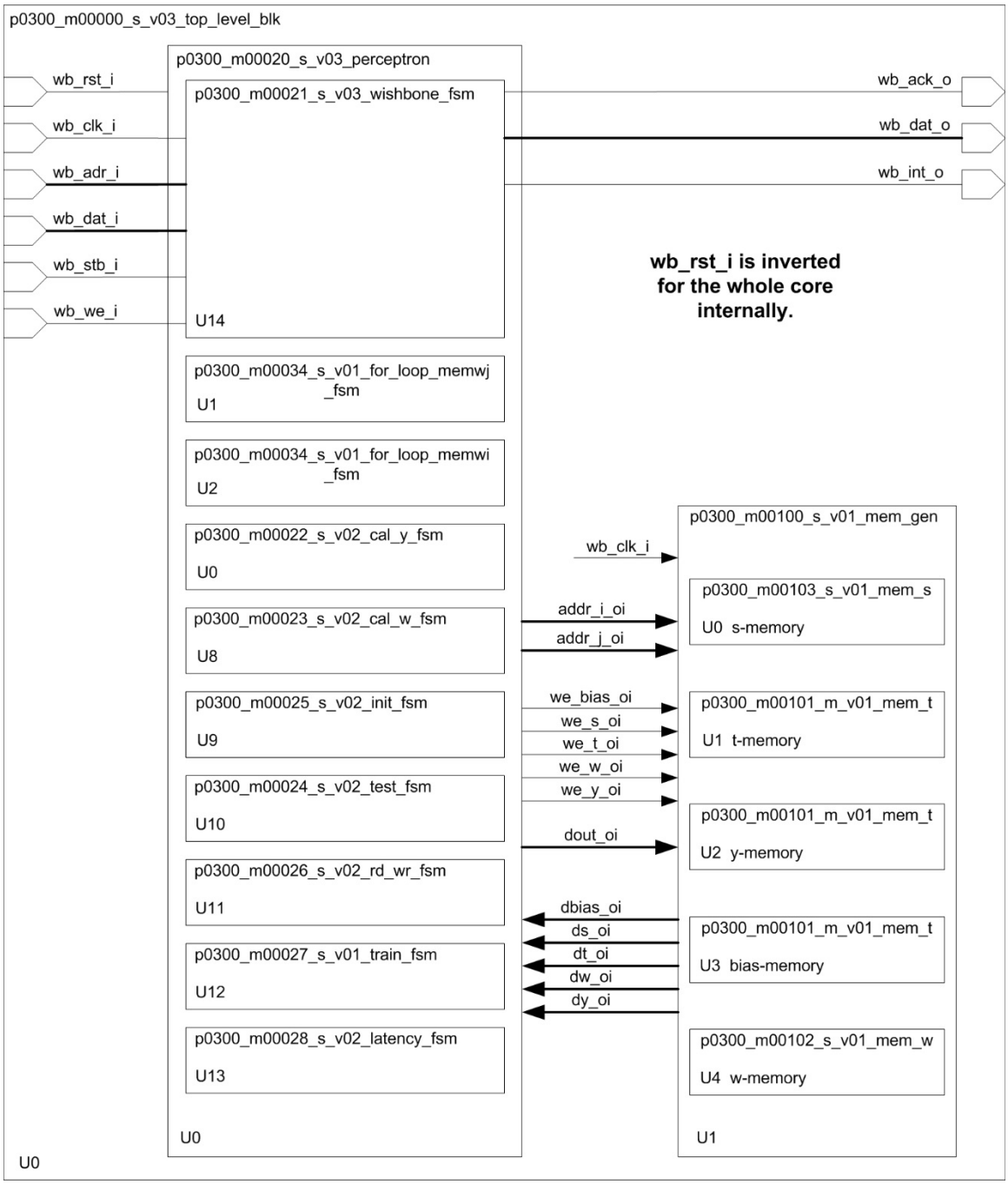


Figure 2: Architecture

Formulas

While training (p0300_m00027_s_v01_train_fsm) following formulas are used to build up and update the weight matrix *w* and *bias* by the given components *s*, targets *t* and threshold θ. The ranges for the row counter *i* and the column counter *j* are given by their pre-loaded start and stop values.

For each column index *j*, all row indexes *i* of *s* and *w* are processed to determine if *w* and *bias* must be updated to accomplish the desired threshold value for the requested output t_j activation.

$$y\_in = bias_j + \sum_{i=start}^{stop} s_i w_{ij}$$

Equation 1: Compute response of output unit

$$y_j = \begin{cases} 1, & if\ y\_in > \theta \\ 0, & if\ -\theta \le y\_in \le \theta \\ -1, & if\ y\_in < \theta \end{cases}$$

Equation 2: Translate y_in to y (1, 0, -1)

If y_j differs from t_j, then the *w* and *bias* matrix must be updated. Otherwise both left be unchanged.

$$w_{ij} = \begin{cases} w_{ij} + t_j s_i, & if\ y_j \neq t_j \\ w_{ij}, & if\ y_j = t_j \end{cases}$$

Equation 3: Compute w matrix

$$bias_j = \begin{cases} bias_j + t_j, & if\ y_j \neq t_j \\ bias_j, & if\ y_j = t_j \end{cases}$$

Equation 4: Compute bias matrix

The training have been finished if all columns were proceeded. Otherwise step to the next column and compute y_in, y_i, w_{ij} and $bias_j$ for all rows *i*.

A readable register *epochs* contain the number of epochs which were necessary to accomplish the required response from all target neurons *t* while the input pattern (components) *s* was trained.

3 Operation

WARNING:

DO NOT CHANGE ANY REGISTER'S CONTENT OR TRY TO START ANY MODULE OR TRY TO READ FROM/WRITE TO MEMORY WHILE THE CORE IS NOT READY. IT MIGHT INVALIDATE RESULTS OF RUNNING PROCESSES AND/OR CORRUPT ADDRESS COUNTER'S STATES.

HINT:

Following the subsections named "Program Level" the user is able to reproduce all the preparations of the sample application described in "Appendix B", page 42.

The core is controlled by reading from or writing to Wishbone's memory mapped addresses.

RESET for Wishbone (RST_I -> wb_rst_i) is defined as *active high* but is inverted internally to reset all connected registers and finite state machines (FSMs).

Memory reads/writes handled through one-address-style memory mapped channels with auto-increment/-decrement address counters. These counters controlled by start and stop values written by user to dedicated registers before any start of module operations.

Use only the auto-increment function of these counters (START < STOP). The auto-decrement feature is reserved for future releases.

Each register have its own memory mapped address for reading and/or writing. Some registers are only for reading from.

Reading from registers is possible at any time and at any state of the core.

Other operations are on user's responsibility and need preparation to registers, observing the core's state and synchronize data flows.

The Wishbone interface were developed and integrated for a smooth operation. As result of that, *wb_ack_o* becomes active after internal operations are finished for some functions like

- Initialize Memory Window content
- Testing of given components s

and

- Read/Write Memory Window content

So, these functions generate a delay of few clock cycles before the Wishbone interface is released by *wb_ack_o*.

The following subsections describe the necessary sequences for user's operation and important timing relations.

Configuration before Synthesis – VHDL Package File

Widths for address and data busses are configurable within the VHDL package file "memory_vhd_vxx_pkg.vhd". The "vxx" herein tells about the version number of the package file.

The following entries are the user settable values:

```
  -- ////////////////\\\\\\\\\\\\\\\\\\
  -- **********************************
  -- ***         User Settings        ***
  -- **********************************

   -- Wishbone Bus
   CONSTANT WB_DATA_WIDTH  : integer := 32; -- Wishbone Data Bus width
   CONSTANT WB_ADDR_WIDTH  : integer := 5; -- Wishbone Address Bus width

   CONSTANT VENDOR : string :=   "generic"; -- (generic, altera, xilinx)
                                --  NOT IMPLEMENTED YET: altera, xilinx

  -- Bus width (DATA_T) of all memories (all are equal)
  -- Chose a value high enough to hold all possible values cumulated in
  -- y_inj_reg,
  -- Memory t,
  -- Memory bias,
  -- Memory w
   CONSTANT DATA_WIDTH                  : integer := 8;

  -- Address width of s input vector memory (maximum number of
components/inputs = 2**MEM_S_ADDR_WIDTH)
   CONSTANT MEM_S_ADDR_WIDTH            : integer := 3;  -- = 8

  -- Address width of t output vector memory (maximum number of
neurons/outputs = 2**MEM_T_ADDR_WIDTH)
   CONSTANT MEM_T_ADDR_WIDTH            : integer := 2;  -- = 4
  -- ////////////////\\\\\\\\\\\\\\\\\\
  -- ////////////////\\\\\\\\\\\\\\\\\\
```

The both values for the Wishbone interface "`WB_DATA_WIDTH`" and "`WB_ADDR_WIDTH`" are dependent to the module *p0300_m00021_s_v03_wishbone_fsm*. Any changes to these both values must also be done within *p0300_m00021_s_v03_wishbone_fsm* and the other hierarchical levels lying above *p0300_m00021_s_v03_wishbone_fsm*.

"DATA_WIDTH", "MEM_S_ADDR_WIDTH" and "MEM_T_ADDR_WIDTH" are free configurable by the user and declare memory window related settings of the on chip memory areas. The given defaults are sufficient for the most general purpose perceptron related tasks.

There are 8x4 possible positions within the *w* matrix, eight entries for components *s* and four for targets *t*. If a larger memory window is required the user is allowed to increase these values as needed.

Within these memory area borders the user is able to shrink the memory window by setting the values of *start i*, *stop i*, *start j* and *stop j* as required for the given task dynamically after synthesis.

Look at section "Program Level - Initialization", page 17 for more detailed information.

Wire Level - Reset the core

RESET for Wishbone is defined as *active high* but is inverted internally for all reset related elements of the whole core. All registers resets synchronously internally at *active low* (*wb_rst_i* <= '1', external reset phase).

Because *wb_rst_i* is active *high*, care must be taken to generate a valid RESET signal to *wb_rst_i* while power is starting up.

Normally, if power comes up, the reset line is at *low* level and hold at this level by external reset logic since the required power levels reached their minimum values. After that, the reset level switched to *high* to start the device correctly.

Additionally the clocks may be stopped by a reset controller while the cold reset phase is active.

But with *active high* reset level, at cold start phase of the FPGA power levels may reach their required values while *wb_rst_i* is still hold at *low*. This may lead in malfunctions for a short time on Wishbone and core functions before the reset line switches *high* for signaling RESET regularly.

Most FPGA devices route reset signals through dedicated pins and offer specialized reset blocks to synchronize resets with associated clocks. Use such blocks within your FPGA design to simplify reset related paths and minimize power-up problems.

Figure 2 show the possible danger behind *active high* reset signals (DANGER at time slot 6 & 7).

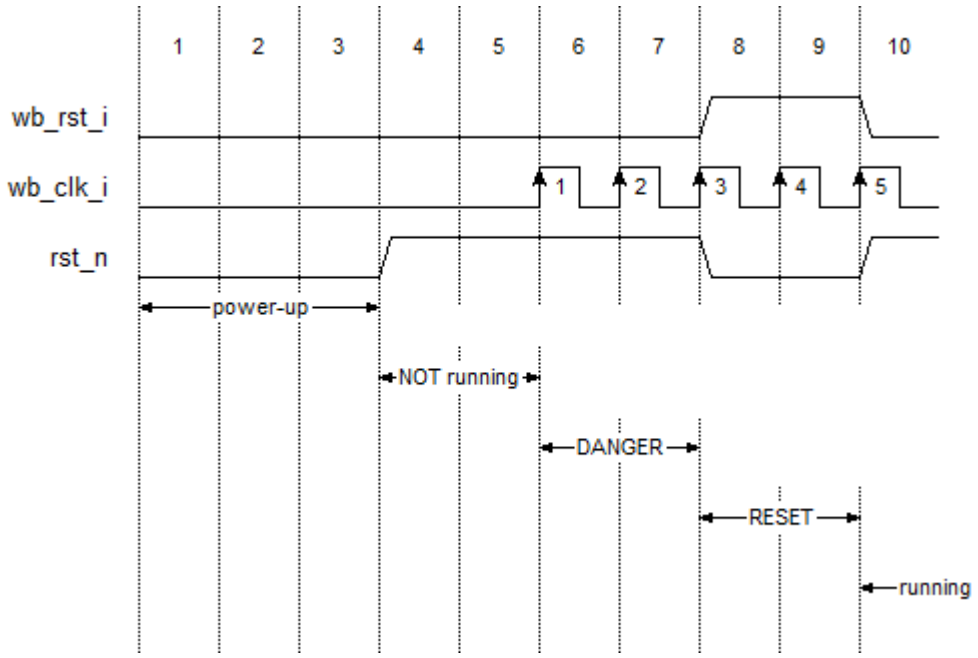


Figure 3: Operation - Active HIGH Reset with dangerous time slot

- Since time slots 1 to 3 the power comes up and both signals *wb_rst_i* and *rst_n* (the inverted internal reset) are *low*.
- At the end of 3 the power is up and *rst_n* is able to reflect the inverted *wb_rst_i* correctly. Even there is no active reset, the core is still not running because of the missing clock.
- DANGER: The clock starts at 6 without a valid reset signal. The core is running normally for one clock cycle (6 & 7) before the regular reset goes *high* (8 & 9).
- The core reset in 8 & 9 while regular reset is active.
- At 10 the core operates normally as indented.

Wire Level - Wishbone standard, synchronous read/write

All reads from and writes to the Wishbone interface are of type standard (pipelined not supported) and are synchronous.

Most register addresses delay *wb_ack_o* and *wb_dat_o* for one clock cycle (1WS – normal) while reading or writing. As result of that, at least

3 clock cycles

are need for a complete read/write sequence.

Other addresses generate more wait states to synchronize internal states to the Wishbone interface.

Fig. 3 shows the timing for slave-xWS, master-1WS/slave-normal and for master-0WS/slave-normal read sequences.

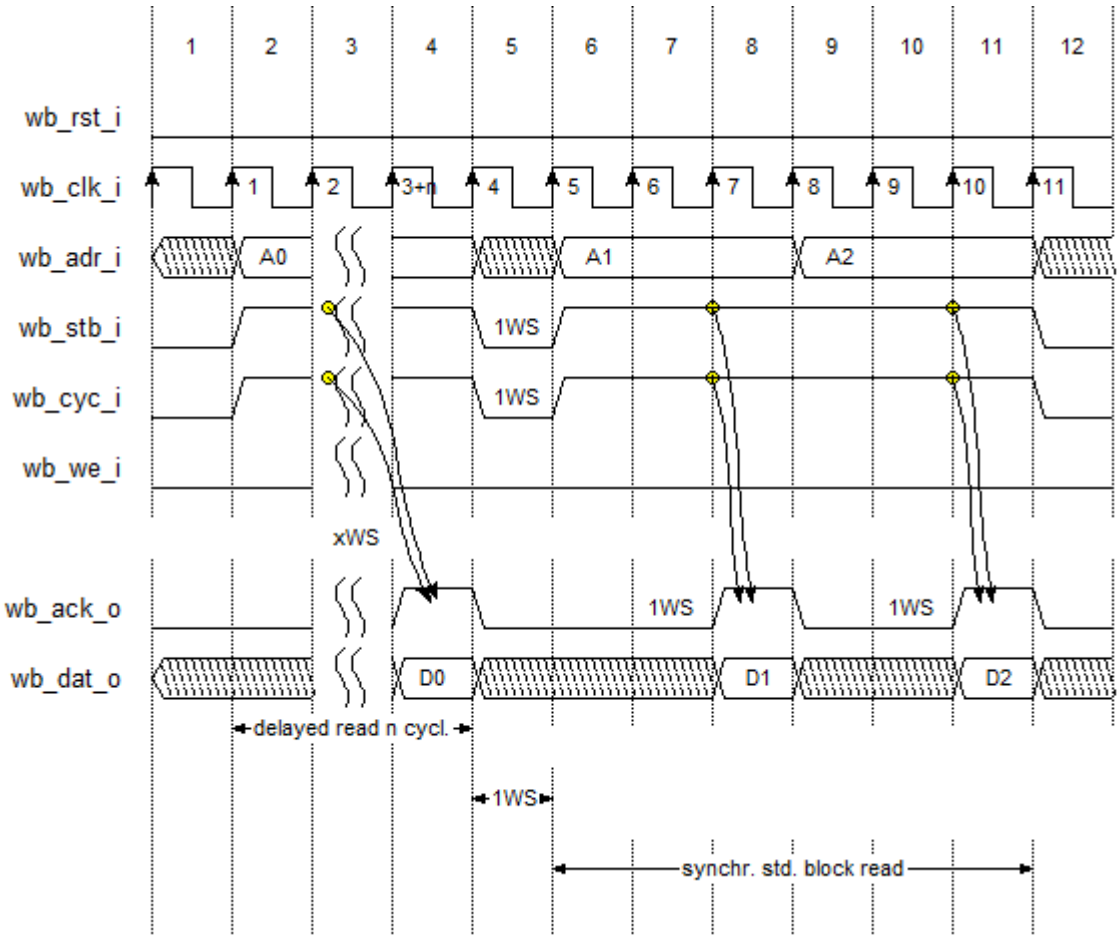


Figure 4: Operation - Wishbone standard, synchronous READ (delayed and normal)

Fig. 4 shows the timing for slave-xWS, master-1WS/slave-normal and for master-0WS/slave-normal write sequences.

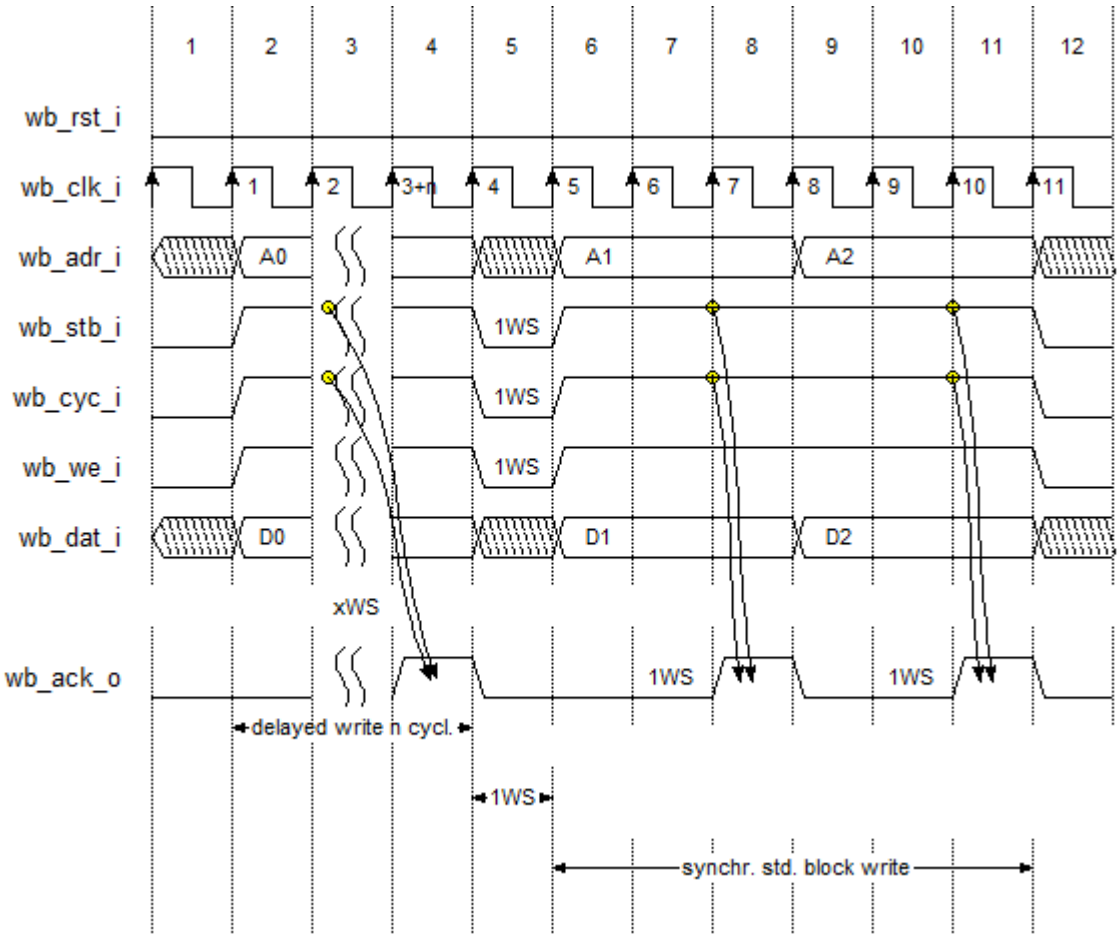


Figure 5: Operation - Wishbone standard, synchronous WRITE (delayed and normal)

The following table shows the number of wait states generated by each address.

Name	Address	# WS RD / WR
STATUS	0x00 / 00d	1 / 1
THRESHOLD	0x01 / 01d	1 / 1
BIAS	0x02 / 02d	1 / 1
OFFSET	0x03 / 03d	1 / 1
MAXEPOCHS	0x04 / 04d	1 / 1
-	0x05 / 05d	1 / 1
-	0x06 / 06d	1 / 1
START i	0x07 / 07d	1 / 1

Name	Address	# WS RD / WR
STOP i	0x08 / 08d	1 / 1
START j	0x09 / 09d	1 / 1
STOP j	0x0A / 10d	1 / 1
EPOCHS	0x0B / 11d	1 / 1
WR LATENCY	0x0C / 12d	1 / 1
RD LATENCY	0x0D / 13d	1 / 1
LATENCY	0x0E / 14d	1 / 1
INIT START	0x0F / 15d	1 / 3
TEST START	0x10 / 16d	1 / 3
SMEM RD/WR	0x11 / 17d	3 + LATENCY / 3 + LATENCY
TMEM RD/WR	0x12 / 18d	3 + LATENCY / 3 + LATENCY
WMEM RD/WR	0x13 / 19d	3 + LATENCY / 3 + LATENCY
YMEM RD/WR	0x14 / 20d	3 + LATENCY / 3 + LATENCY
BIASMEM RD/WR	0x15 / 21d	3 + LATENCY / 3 + LATENCY
TRAIN START	0x16 / 22d	1 / 1
MAX i	0x17 / 23d	1 / 1
MAX j	0x18 / 24d	1 / 1
MEMDBUSW	0x19 / 25d	1 / 1
-	0x1A / 26d – 0x1F / 31d	1 / 1

Table 5: Operation - Wishbone address map with associated wait states (read and write)

Although some addresses are reserved and read/write combinations are not valid, *wb_ack_o* is always generated to acknowledge every Wishbone's transfer/phase and bus activity within the core's address space correctly.

Wire Level - Controller Interrupt

If one of the processes *testing* or *training* have been finished, the corresponding interrupt flags *Pending Interrupt Testing* (bit D6=1) or *Pending Interrupt Training* (bit D7=1) within the status register will be set. If the *Enable Interrupts* bit within the status register is set (write bit D3 = 1 to status register to set D3) output *ctrl_int_o* goes also *high*.

The *Enable Interrupts* bit is not cleared automatically if an interrupt is pending because no more than one process can be started at the same time. Nested interrupts are currently not supported. So, disabling the interrupt generation itself is not necessary. The interrupt handler must be single-threaded.

Reading the interrupt flags D6 and D7 from status register allow the user to determine the source of the pending interrupt.

Accessing the status register for reading clear the flags

- Pending Interrupt Testing (D6=1 -> 0)
- Pending Interrupt Training (D7=1 -> 0)

As result of that, *ctrl_int_o* goes low if interrupts are enabled and one interrupt is pending because it's a function of

ctrl_int_o <= *ctrl_stat_a_reg*(3) AND (*ctrl_stat_ a_reg*(6) OR *ctrl_stat_ a_reg*(7))

Fig. 5 shows the timing relationship to *ctrl_int_o* if one interrupt is pending and *Enable Interrupts* bit D3=1.

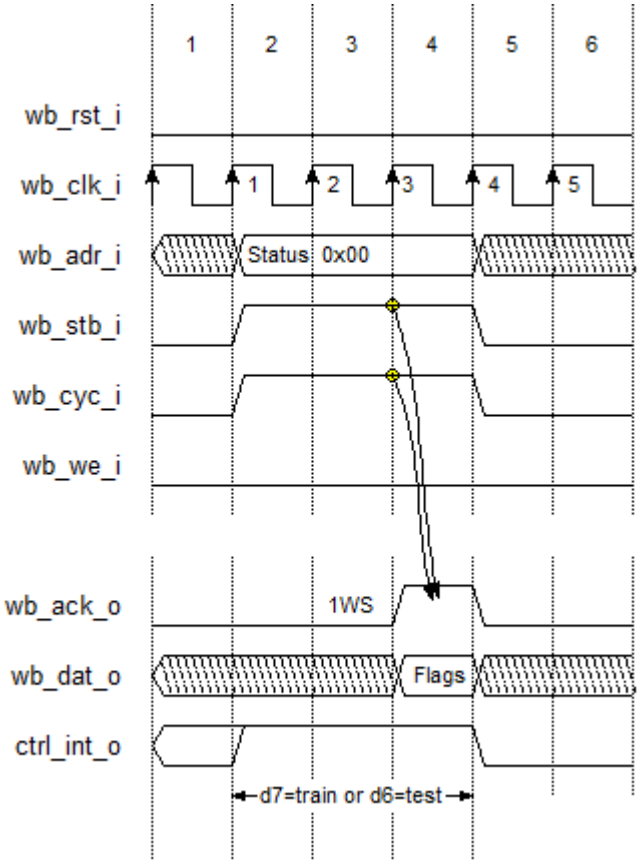


Figure 6: Operation - Reading from status register clear pending interrupt output ctrl_int_o

ctrl_int_o is 0 after the read sequence of the status register has been ended (cycle 4).

Program Level - Initialization

After reset of the core, the memory window is initialized to the minimum window size of 2x2 automatically. This means that one address line fed to the row and one address line fed to the column memory space.

The user must firstly set all parameters correctly to train or test the neural net or work with pre-trained data sets.

1. Set the memory window size
2. Set the bias value for bias memory initialization
3. Start the memory window initialization

Now the memory window is prepared for

- Training
- Testing
- Reading from and writing to memory window

All values named within the following subsections based on the sample application which is detailed described in “Appendix B”, page 42.

Memory Window Size

Following Fig. 6 will prepare the memory window size.

The order of register writes is not important.

Remember to define the stop values as <*necessary rows -1*> and <*necessary columns -1*>. The window size in this example is 6x3 (5x2 internally). As noted before, the smallest windows size of 2x2 is coded as 0, 1, 0, 1 (START i, STOP i, START j and STOP j). **Using values like START = STOP the address counters won’t be start.**

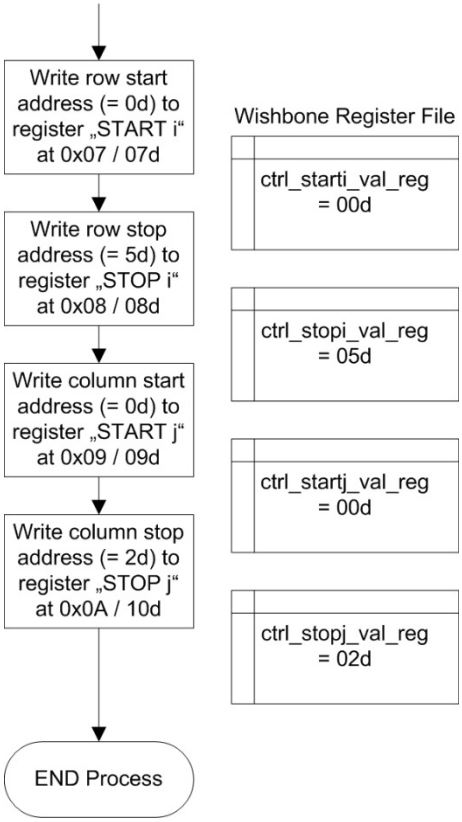


Figure 7: Operation - Initialize memory window size

Bias value

Following Fig. 7 will prepare the *bias* value to write to the bias matrix while INIT process.

For normal operation the *bias* value should be set to 1. This allow the *training* process to adjust the *w* matrix to accomplish the required *threshold* values.

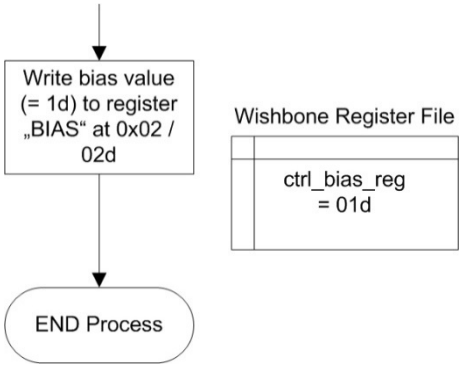


Figure 8: Operation - Initialize bias value

Start the memory window initialization

HINT:

Before start a process like *init start* check that the core is ready to start a process (D0=’1’ - STAT_RDY of the status register).

Following Fig. 8 will start and complete the *init start* process.

Bit 0 (STAT_RDY) of the status register reflects the ready state of the core and will be cleared after the start of the *init start* process. After bit 0 is ‘1’ again the *init start* process have been finished and the core is ready for another process to start.

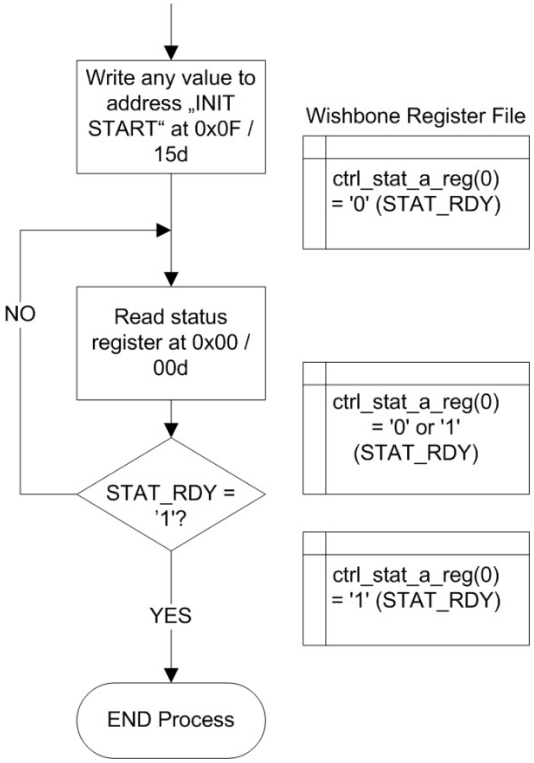


Figure 9: Operation - INIT START - Start the memory window initialization

Program Level – Interrupt enable/disable

Following Fig. 9 will enable or disable the generation of interrupt *ctrl_int_o*.

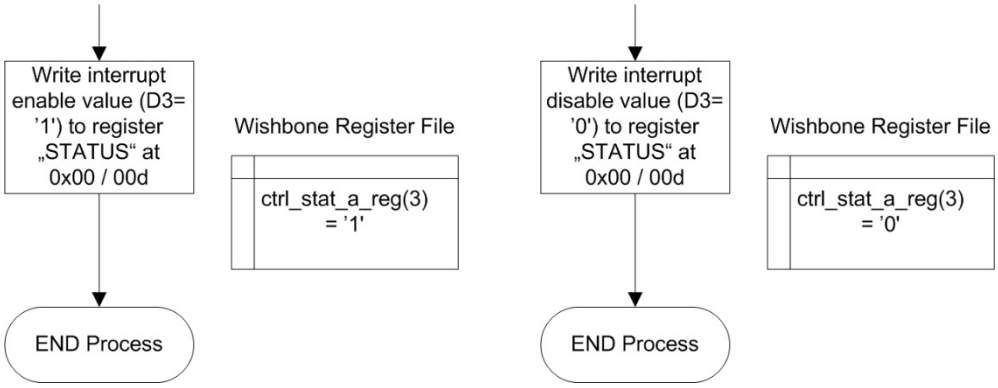


Figure 10: Operation - Interrupt enable or disable

Program Level - Training

Before the process *training* is able to train the perceptron neural net some more preparations are needed.

It is assumed that following processes have been finished successfully before stepping further:

Program Level – Initialization

1. Memory Window Size
2. Bias value
3. Start the memory window initialization

and

Program Level – Interrupt enable/disable

Interrupts should be disabled.

The training procedure is shown in Fig. 10.

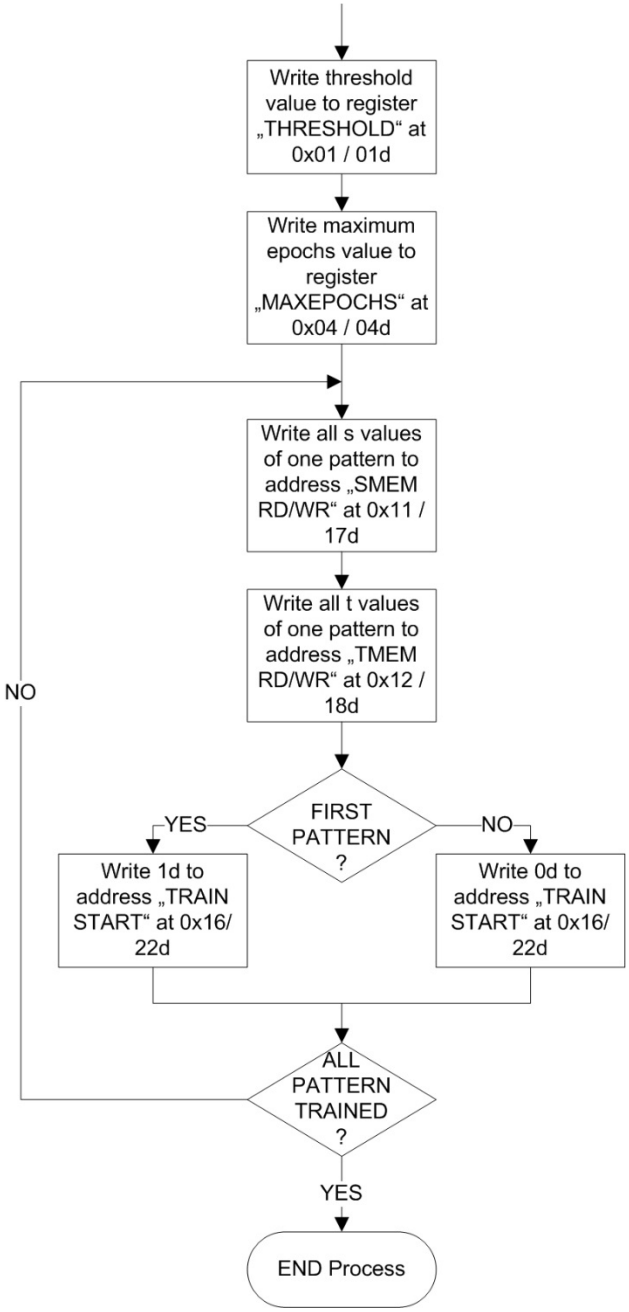


Figure 11: Operation - The training procedure

The blocks within the above figure are described more in detail in the following subsections.

Threshold value

Next, the *threshold* value should be defined. Using the settings of the sample application will produce clear results and recognitions when testing the perceptron neural net against known pattern in later steps.

So, use 32d for the *threshold*.

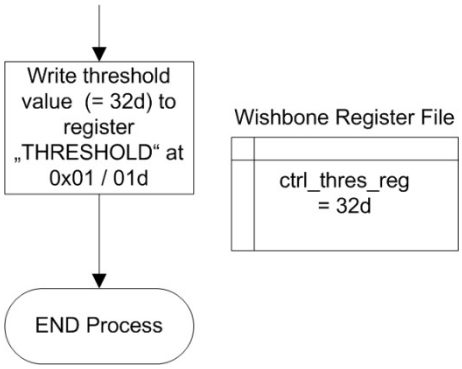


Figure 12: Operation - Initialize threshold value

Maximum Epochs value

Last, define how many epochs are allowed while training is progressing.

Set the *maxepochs* register to 0d to allow maximum precision.

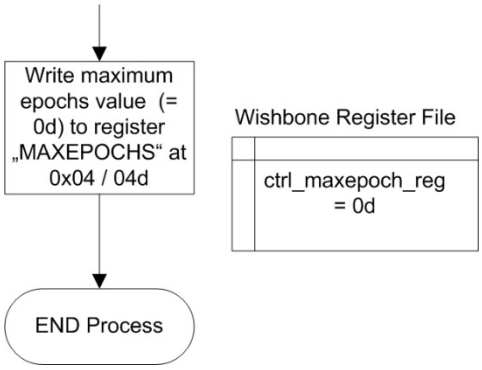


Figure 13: Operation - Initialize maxepochs value

Write a s:t pattern definition pair to memory window

HINT:

Before start a process like *smem rd/wr* check that the core is ready to start a process (D0=’1’ - STAT_RDY of the status register).

It is assumed, that the row and column address counters are in initialization state. If any Read or Write Memory process (*xmem rd/wr*) were started in the past, it must be completed (status register at 0x00 bit D5=’1’, READ/WRITE COMPLETED).

Because *xmem rd/wr* processes block the Wishbone bus until they have been finished, the user doesn't need to care about the run times.

The user is allowed to write *s* or *t* first to the memory window.

For every pattern pair *s*:*t* to train do the following:

1. Writing *s* values to *s* memory window
2. Writing *t* values to *t* memory window
3. Start the *training* process

The order to write the *s* and *t* values to memory is either LITTLE or BIG ENDIAN. For example, a value of 001010b can either translated to

$-1\ -1\ +1\ -1\ +1\ -1$

or

$-1\ +1\ -1\ +1\ -1\ -1$.

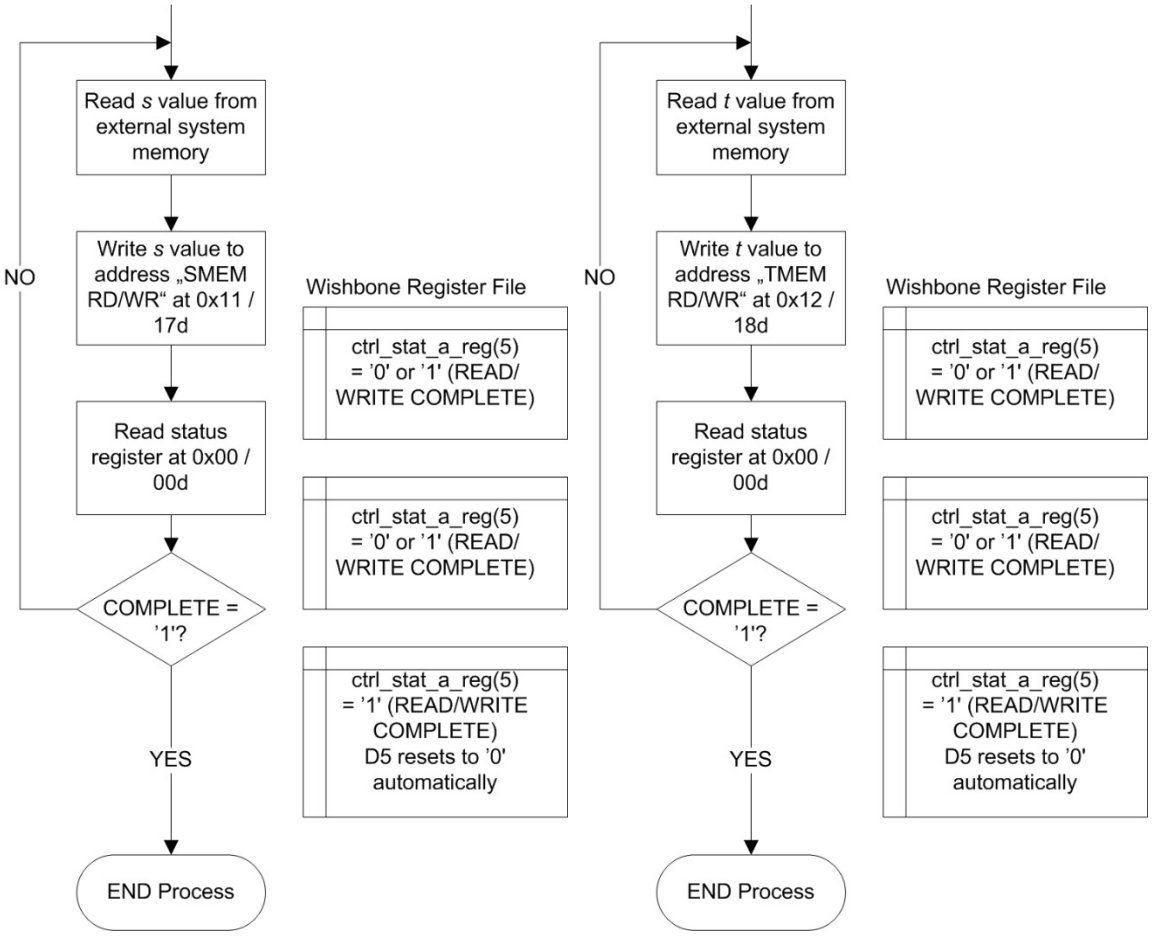


Figure 14: Operation - Writing a pattern pair s:t to *s* and *t* memory window

Start the training process

HINT:

Before start a process like *train start* check that the core is ready to start a process (D0=’1’ - STAT_RDY of the status register).

After the training pattern pair *s*:*t* has been written, the *train start* process can be start.

If this is the first training pattern pair to train it may be useful to reset the *epochs* counter to 0. To do this, write bit D0=’1’ to the address of *train start* 0x16/22d. Remaining pair trainings should be started with D0=’0’ (cumulate epochs to the counter *epochs* 0x0B/11d).

If the training of all pattern pairs *s*:*t* is complete, the user can read out the *epochs* register at 0x0B/11d to observe the number of epochs were needed to train the perceptron neural net.

Now, the neural net is ready for testing.

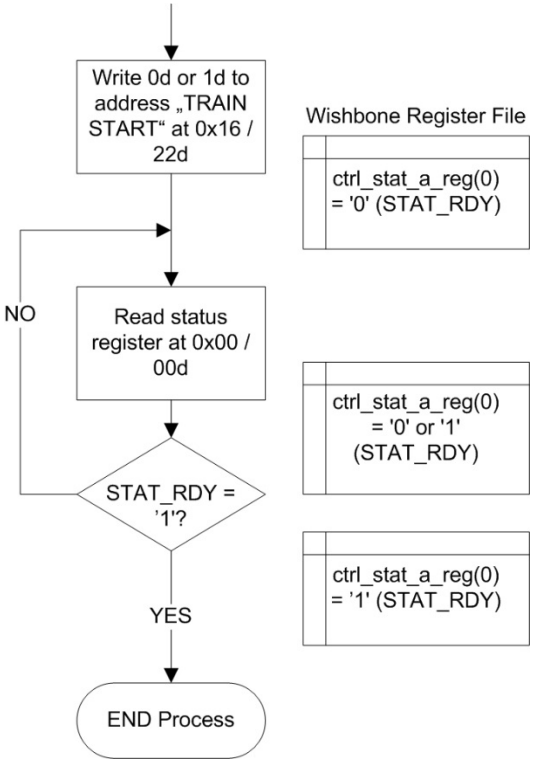


Figure 15: Operation - TRAIN START - Start the training process

Program Level - Testing

HINT:

Before start a process like *test start* check that the core is ready to start a process (D0='1' - STAT_RDY of the status register).

If the *bias* and *w* memory contains valid data the perceptron neural net is ready for testing pattern *s* and observe the generated answers *t* from the *test start* process.

Simply write the pattern *s* to test onto the *s* memory, start the *test* process and read out the *t* memory.

To move the whole range of results *t* numerically up or down, a specific value to the *offset* register 0x03/03d can be set before test start process is started.

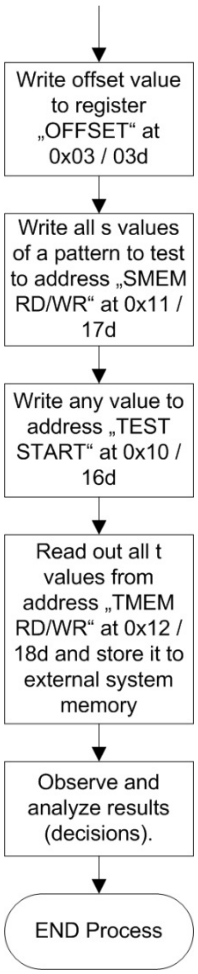


Figure 16: Operation - The testing procedure

Offset value

Following Fig. 16 to write an offset value to the *offset* register at 0x03/03d.

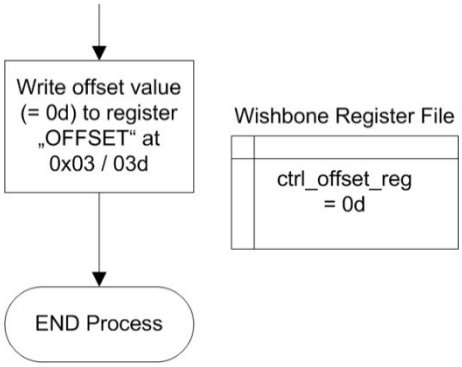


Figure 17: Operation - Initialize offset value

Write s values of test pattern to s memory

HINT:

Before start a process like *smem rd/wr* check that the core is ready to start a process (D0='1' - STAT_RDY of the status register).

The order to write the *s* values to memory is either LITTLE or BIG ENDIAN. For example, a *s* value of 001010b can either translated to

−1 −1 +1 −1 +1 −1

or

−1 +1 −1 +1 −1 −1.

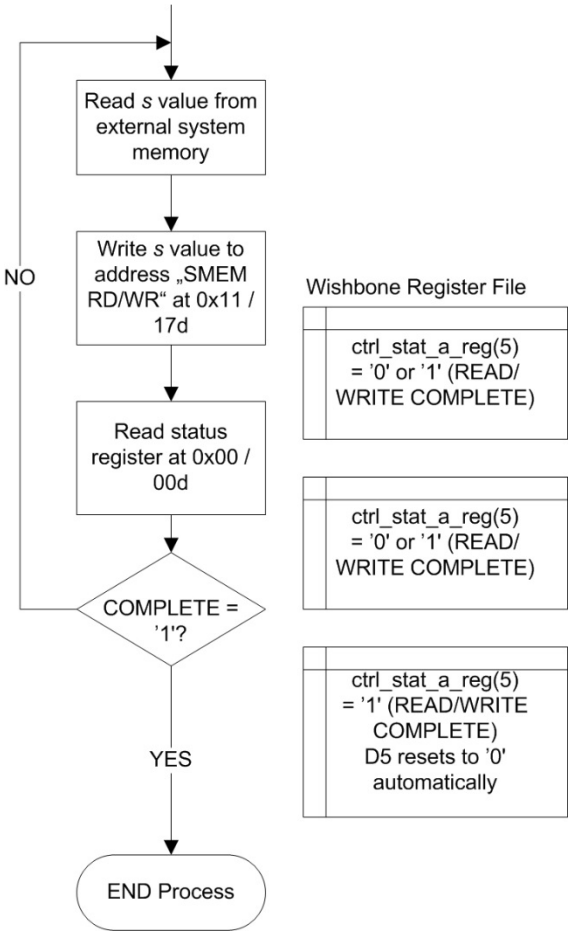


Figure 18: Operation - Writing a test pattern to s memory window

Start the testing process

HINT:

Before start a process like *test start* check that the core is ready to start a process (D0=’1’ - STAT_RDY of the status register).

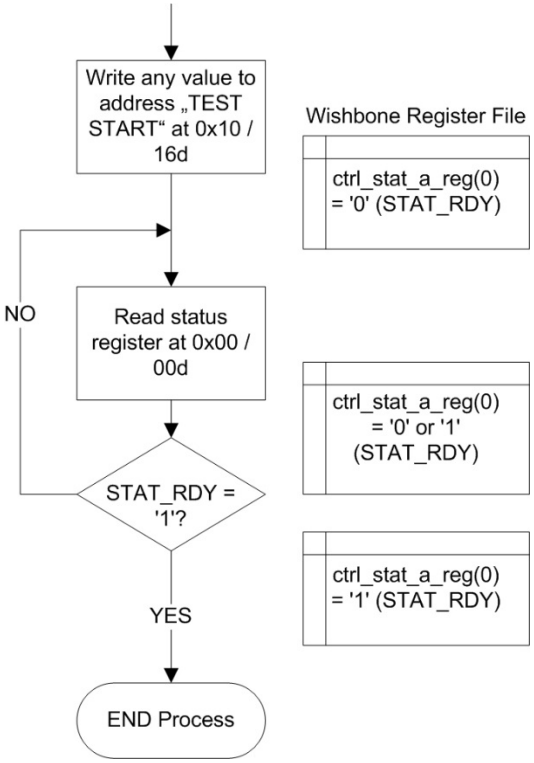


Figure 19: Operation - TEST START - Start the testing process

Read out t values from t memory

HINT:

Before start a process like *tmem rd/wr* check that the core is ready to start a process (D0=’1’ - STAT_RDY of the status register).

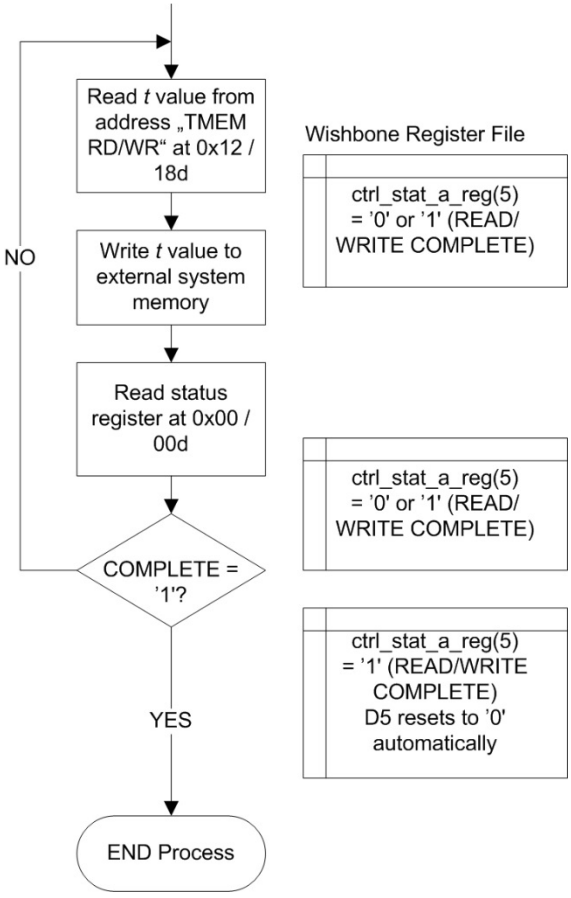


Figure 20: Operation - Reading target values from t memory window

Observe and analyze results

Working on own perceptron tasks and finding optimal values for threshold, bias, offset and a valid classification for noisy training pattern can be a time consuming process.

Studying the sample application described in “Appendix B”, page 42 can help to understand the different problems and the different solutions to accomplish a goal or trend.

4 Registers

List of Registers

Name	Address	Width	Access	Description
STATUS	0x00 / 00d	32	R/W	Status register Default: 0x01 – after reset phase is done, core is ready for receiving commands. Read: all status bits and reset bits RD_WR_COMPLETE, INT_TEST and INT_TRAIN. Write: D3=0 to disable interrupts – clear INT_EN bit, D3=1 to enable interrupts – set INT_EN bit. For further details look at “Status register – Description”.
THRESHOLD	0x01 / 01d	32	R/W	Threshold register Default: 0 Holds the value that must be accomplished while training for output nodes *t*.
BIAS	0x02 / 02d	32	R/W	Bias register Default: 0 Value to write into bias memory while initialization (Reset the core or INIT START).
OFFSET	0x03 / 03d	32	R/W	Offset register Default: 0 Value to adjust the results of the output nodes *t* while testing.
MAXEPOCHS	0x04 / 04d	32	R/W	Maximum Epochs register Default: 0

Name	Address	Width	Access	Description
				Its protect the core for endless loops or running for too long if the activation level for output nodes *t* cannot accomplish while training. Each time the weights are updated, the internal Epochs Counter is incremented by 1. If = 0, the training process ends at the point at which the activation level of an output node *t* reached "THRESHOLD" regardless of how many epochs will be needed. If > 0, the training process ends at the point at which the Epochs Counter (32 Bits wide) have been reached the value stored in "MAXEPOCHS" while training.
-	0x05 / 05d	-	-	Reserved – DO NOT USE
-	0x06 / 06d	-	-	Reserved – DO NOT USE
START i	0x07 / 07d	32	R/W	Start i register Default: 0 Holds the start value for the row address counter *i*. The counter is able to increment or decrement. In most cases set it to 0.
STOP i	0x08 / 08d	32	R/W	Stop i register Default: 1 Set it to a value <*necessary rows -1*>. The default value of 1 allows two rows 0-1. MUST differ from STOP i value.
START j	0x09 / 09d	32	R/W	Start j register Default: 0 Holds the start value for the column address counter *j*. The counter is able to increment or decrement. In most cases set it to 0.
STOP j	0x0A / 10d	32	R/W	Stop j register Default: 1 Set it to a value <*necessary columns -1*>. The default value

Name	Address	Width	Access	Description
				of 1 allows two columns 0-1 MUST differ from STOP j value.
EPOCHS	0x0B / 11d	32	R	Epochs register Default: 0 Count the events of weights matrix and bias matrix updates while training.
WR LATENCY	0x0C / 12d	32	R	Write Latency Holds the coded value for write latency for all memories determined by the *_latency_fsm module while core's reset phase. It represents the number of cycles to need to write a value successfully to memory before any address change is allowed. Format: …0000 => 0WS …0001 => 1WS …0011 => 2WS …0111 => 3WS …
RD LATENCY	0x0D / 13d	32	R	Read Latency Holds the coded value for read latency for all memories determined by the *_latency_fsm module while core's reset phase. It represents the number of cycles to need to read a value successfully from memory after any address change takes effect. Format: …0000 => 0WS …0001 => 1WS …0011 => 2WS …0111 => 3WS …
LATENCY	0x0E / 14d	32	R	Latency Holds the unsigned binary value for read latency for all memories determined by the *_latency_fsm module while core's reset phase. It represents the number of cycles to need to read a value

Name	Address	Width	Access	Description
				successfully from memory after any address change takes effect. Format: 0x0 => 0WS 0x1 => 1WS 0x2 => 2WS …
INIT START	0x0F / 15d	32	W	Start Initialization of memory's window Starts automatically after Reset. Write any value to this address to start the initialization of the previously defined memory window by start/stop values. The s, t, w and y memories will be initialized with zeros. The bias memory will be initialized with the value stored in register BIAS 0x02/02d. Depend on stored values in START i, STOP i, START j and STOP j.
TEST START	0x10 / 16d	32	W	Start Test process Write any value to this address to start the test of the loaded components *s*. The results of output nodes *t* will be stored in t memory after the test have been finished (see Status register). Depend on stored values in START i, STOP i, START j and STOP j.
SMEM RD/WR	0x11 / 17d	32	R/W	Read or Write s memory (signed, bipolar -1, 0, 1, extended to WB dbus width) Read: Complete content of s memory window while read-out is complete (see Status register). Write: Complete content of s memory windows while write-in is complete (see Status register). Depend on stored values in START i, STOP i, START j and STOP j.

Name	Address	Width	Access	Description
				Does not set RDY flag to NOT RDY=’0’.
TMEM RD/WR	0x12 / 18d	32	R/W	Read or Write t memory (signed, extended to WB dbus width) Read: Complete content of t memory window while read-out is complete (see Status register). Write: Complete content of t memory window while write-in is complete (see Status register). Depend on stored values in START i, STOP i, START j and STOP j. Does not set RDY flag to NOT RDY=’0’.
WMEM RD/WR	0x13 / 19d	32	R/W	Read or Write w memory (signed, extended to WB dbus width) Read: Complete content of w memory window while read-out is complete (see Status register). Write: Complete content of w memory windows while write-in is complete (see Status register). Depend on stored values in START i, STOP i, START j and STOP j. Does not set RDY flag to NOT RDY=’0’.
YMEM RD/WR	0x14 / 20d	32	R/W	Read or Write y memory (signed, bipolar -1, 0, 1, extended to WB dbus width) Read: Complete content of y memory window while read-out is complete (see Status register). Write: Complete content of y memory window while write-in is complete (see Status register). Depend on stored values in START i, STOP i, START j and STOP j. Does not set RDY flag to NOT RDY=’0’.

Name	Address	Width	Access	Description
BIASMEM RD/WR	0x15 / 21d	32	R/W	Read or Write bias memory (unsigned) Read: Complete content of bias memory window while read-out is complete (see Status register). Write: Complete content of bias memory window while write-in is complete (see Status register). Depend on stored values in START i, STOP i, START j and STOP j. Does not set RDY flag to NOT RDY='0'.
TRAIN START	0x16 / 22d	32	W	Start Training process Write 0 to cumulate epochs from previous runs and start the training. Write 1 to clear the Epochs Counter and start the training. Depend on stored values in START i, STOP i, START j and STOP j.
MAX i	0x17 / 23d	32	R	Maximum i Read the maximum of available memory *rows-1*. *rows* is the value of MEM_S_ADDR_WIDTH defined by user in the file "memory_vhd_v03_pkd.vhd". It can be used to determine the available memory area and to define STOP i and STOP j.
MAX j	0x18 / 24d	32	R	Maximum j Read the maximum of available memory *columns-1*. *columns* is the value of MEM_T_ADDR_WIDTH defined by user in the file "memory_vhd_v03_pkd.vhd". It can be used to determine the available memory area and to define STOP i and STOP j.
MEMDBUSW	0x19 / 25d	32	R	Memory Data Bus Width Read the memory data bus width.

Name	Address	Width	Access	Description
				It contain the value of DATA_WIDTH defined by user in the file "memory_vhd_v03_pkd.vhd".
-	0x1A / 26d – 0x1F / 31d	-	-	Reserved – DO NOT USE

Table 6: Registers - List of registers

Status register – Description

Bit #	Access	Description
0	R	RDY 0 => Core is NOT ready to receive or process commands. 1 => Core is ready to receive commands after following modules all become ready first: - p0300_m00022_s_v02_cal_y_fsm - p0300_m00023_s_v02_cal_w_fsm - p0300_m00025_s_v02_init_fsm - p0300_m00024_s_v02_test_fsm - p0300_m00026_s_v02_rd_wr_fsm - p0300_m00027_s_v01_train_fsm - p0300_m00028_s_v02_latency_fsm After reset the module *_latency_fsm measures w memory's latency. Then module *_init_fsm initialize the default memory window of 2 * 2 address space.
1	R	LATENCY MEASSUREMENT RUN 0 => Latency measurement have been finished. 1 => Latency measurement is currently running. Module *_latency_fsm measures w memory's latency to adjust the read and the write latency times for all memory blocks.
2	R	TRAIN RDY 0 => Module *_train_fsm is NOT ready – currently processing data for training. 1 => Module *_train_fsm is ready to process data for training. Check this bit before alter any memory's content.
3	R/W	ENABLE INTERRUPTS 0 => Interrupts are disabled. 1 => Interrupts are enabled. To enable interrupts, write D3=1 to status register's address. To disable interrupts, write D3=0 to status register's address. All other bits are handled as "*don't care*" while writing - masking of other bits is not necessary.

Bit #	Access	Description
4	R	MEMORY ERROR 0 => NO memory error occurred while latency measurement. 1 => Memory error occurred while latency measurement – core is not operational. If this bit is 1, status bit RDY becomes 0 to indicate that the core is not operational. In conjunction with status bit 1 LATENCY MEASSUREMENT RUN it helps to analyze the state of the core.
5	R	READ WRITE COMPLETE 0 => Memory windows start condition OR read from or write to a memory window is NOT complete – Address locations are remaining for read/write OR start condition of memory window reached after reset OR this bit was 1 before. 1 => Read from or write to a memory window is complete – Memory window is ready for a new read/write sequence. This bit is cleared automatically after reading the status register. To determine the exact state of a memory's read/write sequence observe and store any start and completion of memory read/write actions. If unsure, run "INIT START" to initialize the memory's window and reset the address counters.
6	R	PENDING INTERRUPT TESTING 0 => NO interrupt of process TESTING is pending. 1 => Interrupt of process TESTING is pending. This bit is cleared automatically after reading the status register. If 1, a previously started TESTING process have been finished and data is available to read from memory's window.
7	R	PENDING INTERRUPT TRAINING 0 => NO interrupt of process TRAINING is pending. 1 => Interrupt of process TRAINING is pending. This bit is cleared automatically after reading the status register. If 1, a previously started TRAINING process have been finished and data is available to read from memory's window.

Reset Value: 0x01 *after reset phase (latency measurement and memory initialization) is done.*

Reg_Address: 0x00

Table 7: Registers - Description of Status registers, 0x00

5 Clocks

Name	Source	Rates (MHz)			Remarks	Description
		Max	Min	Resolution		
wb_clk_i	PLL	-	-	-	Rate dependent on target imple-mentation.	System clock.

Table 8: Clocks - List of clocks

6 IO Ports

All inputs and outputs are HIGH ACTIVE.

Widths of address and data busses are configurable in VHDL package file.

Port	Width	Direction	Description
wb_clk_i	1	Input	Block's WISHBONE Clock Input
wb_rst_i	1	Input	Block's WISHBONE Reset Input Internally inverted to rst_n for synchronous active low
wb_adr_i	5	Input	Block's WISHBONE Address Inputs
wb_dat_i	32	Input	Block's WISHBONE Data Inputs
wb_stb_i	1	Input	Block's WISHBONE Strobe Input
wb_we_i	1	Input	Block's WISHBONE Write Enable Input
wb_dat_o	32	Output	Block's WISHBONE Data Outputs
wb_ack_o	1	Output	Block's WISHBONE Acknowledge Output
ctrl_int_o	1	Output	Block's CONTROLLER Interrupt Output

Table 9: IO Ports - List of IO ports

Appendix A

Wishbone Datasheet

General Description	Wishbone Register Bank and Perceptron Core Interconnect
Wishbone Version	B4
Type of Interface	Slave
Supported cycles	Slave, Read/Write, Standard, Synchronous Slave, Block, Standard, Synchronous Slave, RMW, Standard, Synchronous (pipelined cycles are not supported)
ERR_I handling	currently not supported
ERR_O handling	currently not supported
RTY_I handling	currently not supported
RTY_O handling	currently not supported
Tags	currently no tags supported
Data Port Size	Default: 32 Port size configurable with constant declaration in VHDL package file
Data Port granularity	currently no sub port granularity supported
Data Port Operand size	Dependent on addressed register and function (values of constants in VHDL package file before synthesis was made)
Data transfer ordering	Little Endian (port size=granularity)
Data transfer sequencing	Undefined
Clock Constraints	Frequency dependent on implementation

Table 10: Wishbone Datasheet – Datasheet

Signal Name	Wishbone Equivalent
wb_clk_i	CLK_I
wb_rst_i	RST_I
wb_stb_i	STB_I
wb_we_i	WE_I
wb_adr_i	ADR_I(0..n)
wb_dat_i	DAT_I(0..n)
wb_cyc_i	CYC_I
wb_ack_o	ACK_O
wb_dat_o	DAT_O(0..n)
ctrl_int_o	-

Table 11: Wishbone Datasheet - List of signals

Appendix B

Sample Application

This is a sample application to show up how to set up and use the perceptron neural network to solve given problems.

It is advised to read and understand all sections and subsections of this specification before trying to work on this sample application. “WHYs”, “DOs”, “DONTs” and backgrounds are not described any more in this section.

Problem to solve

The following problem is to solve:

- Three given symbols UP, DOWN and STOP must be recognized at rate 100%
- Symbol size is 3x2
- There is no noise on input pattern
- Three targets (neurons). One for every valid symbol UP, DOWN and STOP (100% recognition rate means: only one target *t* is activated at the time)
- All other possible symbols ($2^6 = 64$, $64 – 3 = 61$) must not activate any of the UP, DOWN or STOP targets

The order to write the *s* and *t* values to memory is either LITTLE or BIG ENDIAN. For example, a value of 001010b can either be translated to

−1 −1 +1 −1 +1 −1

or

−1 +1 −1 +1 −1 −1.

Definition and translating of symbols

Each symbol is defined and translated as follows:

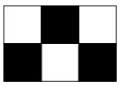
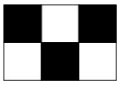
Symbol Name	Symbol picture	Binary representation (D5 – D0)	Bipolar Components s #0 - 5
UP		0 1 0 1 0 1 (21d)	+1 −1 +1 −1 +1 −1
DOWN		1 0 1 0 1 0 (42d)	−1 +1 −1 +1 −1 +1
STOP		1 1 1 0 1 1 (59d)	+1 +1 −1 +1 +1 +1
all others			Translated binary representations

Table 12: Sample Application - List of translated symbols

Definition and translating of targets

Each target is defined and translated as follows:

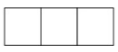
Target Name	Target picture	Binary representation (D2 – D0)	Bipolar Targets t #0 - 2
UP		0 0 1	+1 −1 −1
DOWN		0 1 0	−1 +1 −1
STOP		1 0 0	−1 −1 +1
all others		0 0 0	−1 −1 −1

Table 13: Sample Application - List of translated targets

Setup of memory and initializations

The memory window size is given by (*symbol size) x (# targets t).*

Window Size = (3x2 = 6) x 3 = 6x3 (size i x size j)

Setup values:

Disable interrupts
THRESHOLD = 32d

BIAS =	1d
OFFSET =	0d
MAXEPOCHS =	0d
START i =	0d
STOP i =	5d (6 - 1)
START j =	0d
STOP j =	2d (3 - 1)

Now, start *init start*.

Training

Because the given three symbols UP, DOWN and STOP must be recognized 100% and without any noise all possible 64 pattern must be trained.

The remaining 61 pattern pairs *s:t* must be trained as "no activation of any target *t*" (t = −1 −1 −1).

To simplify the process loop, the *s* pattern are generated from the binary representation of the outer loop index i.

If the loop index i is equal to one of the three symbol values 21d, 42d or 59d, the required target value is written to *t* memory instead of write "−1 −1 −1" globally for all other not target activating *s* pattern.

After the training of all 64 pattern pairs *s:t* is completed, the *epochs* register contain the value 0x13/19d.

This mean:

- 6x3 epochs were regularly needed to train the three symbols at 21d, 42d and 59d.
- 1 additional epochs were needed for symbol "1 1 0 1 0 1" (53d, t = −1 −1 −1) to separate its similarity to symbol UP "0 1 0 1 0 1" (21d, t = +1 −1 −1)
- All other symbols does not require training.

At the end only four symbols were trained:

1. UP, 010101, 21d
2. DOWN, 101010, 42d
3. >*unknown*<, 110101, 53d very similar to UP
4. STOP, 111011, 59d

Testing all 64 pattern pairs and analyzing the results

It is suggested that the user is able to write some helper programs or scripts to analyze and interpret the results of *training* and *testing* processes rightly.

This sample application was written on VHDL test bench level to be not depend to programming languages or some operating system's restrictions. As result of that, the test bench write the contents of s, t and bias memory out to separate files while simulation runs.

Converting the resulted values from binary and hexadecimal types to signed decimal type is also a task to be done.

Additionally, the content of these generates files imported to a Windows Excel sheet for interpreting and analyzing the results more precisely.

The next table shows the numerical results *t* (UP, DOWN, STOP) of all 64 test pattern *s*. GREEN marked fields represents activated targets *t* and are positioned exactly as expected. The three results of activated targets are greater or equal to the given *threshold* value of 32d.

Number	UP	DOWN	STOP
0	2	0	-24
1	12	-12	-12
2	-8	12	-12
3	2	0	0
4	12	-12	-36
5	22	-24	-24
6	2	0	-24
7	12	-12	-12
8	-8	12	-12
9	2	0	0
10	-18	24	0
11	-8	12	12
12	2	0	-24
13	12	-12	-12
14	-8	12	-12
15	2	0	0
16	12	-12	-12
17	22	-24	0
18	2	0	0
19	12	-12	12
20	22	-24	-24
21 (UP)	32	-36	-12
22	12	-12	-12
23	22	-24	0
24	2	0	0
25	12	-12	12
26	-8	12	12
27	2	0	24
28	12	-12	-12
29	22	-24	0
30	2	0	0
31	12	-12	12

Number	UP	DOWN	STOP
32	-12	12	-12
33	-2	0	0
34	-22	24	0
35	-12	12	12
36	-2	0	-24
37	8	-12	-12
38	-12	12	-12
39	-2	0	0
40	-22	24	0
41	-12	12	12
42 (DOWN)	-32	36	12
43	-22	24	24
44	-12	12	-12
45	-2	0	0
46	-22	24	0
47	-12	12	12
48	-2	0	0
49	8	-12	12
50	-12	12	12
51	-2	0	24
52	8	-12	-12
53	18	-24	0
54	-2	0	0
55	8	-12	12
56	-12	12	12
57	-2	0	24
58	-22	24	24
59 (STOP)	-12	12	36
60	-2	0	0
61	8	-12	12
62	-12	12	12
63	-2	0	24

Table 14: Sample Application - Results t of 64 test pattern

This is a good example to set *offset* to a value above 0 to get better coverage results for pattern testing.

Following picture shows the results of all 64 tested pattern. The *threshold* is marked as RED dotted line and the three activated targets *t* are marked as RED circles.

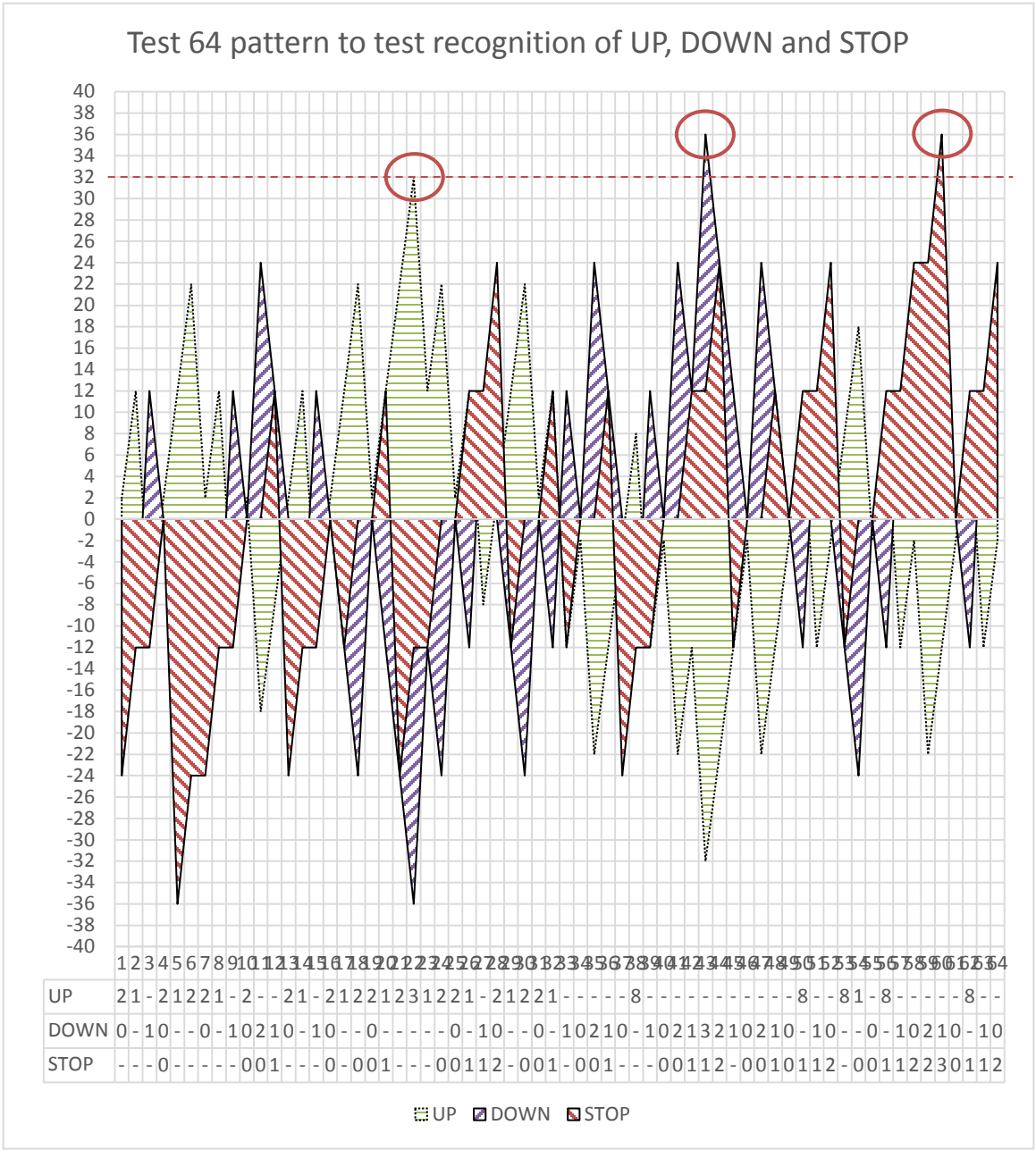


Figure 21: Sample Application – Results of t of 64 tested pattern

Content of w matrix

The content of *w* matrix at *threshold* = 32d is (i = 0…5 top to down, j = 0…2 left to right)

$$w_{ij} = \begin{bmatrix} 5 & -6 & 6 \\ -5 & 6 & 6 \\ 5 & -6 & -6 \\ -5 & 6 & 6 \\ 5 & -6 & 6 \\ -7 & 6 & 6 \end{bmatrix}$$

Content of bias matrix

The content of *bias* matrix at *threshold* = 32d is (j = 0…2 left to right)

$$bias_j = [7 \quad 7 \quad 6]$$

7 References

FAUSETT, L.V. (year unknown), “Fundamentals of Neural Networks”, Sec. 2.3, Perceptron, p.p. 59-80