TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Madis Kerner153158IASM

Hierarchical Temporal Memory Based Predictive Model and Anomaly Detection Component on FPGA

Master's thesis

Supervisor: Kalle Tammemäe
Associate Professor

TALLINN 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Madis Kerner153158IASM

Hierarhilise temporaalse mälu printsiibil toimiva ennustus- ja anomaalsustuvastuse komponent FPGA-l

magistritöö

Juhendaja:	Kalle Tammemäe
	professor

TALLINN 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Madis Kerner

23rd May 2017

Abstract

This thesis deals with the realization of one of the modern artificial neural networks on the system on chip (SoC). The underlying neural network for this work, hierarchical temporal memory (HTM), has been developed by the company Numenta and is functionally similar to the neocortex mini-columns.

The realization of the HTM algorithm is not computationally demanding. Despite this, the HTM evaluation and development platform Nupic, provided by the Numenta, is more suitable for executing on a personal computer.

One of the big obstacles for a hardware based realization is the algorithm's high demand for the system memory – neuron connectivity parameters are also stored by the Nupic platform, for example.

The main proposal presented in this work is to use the linear feedback shift register (LFSR) as the neuron address decoder. In addition to this, the resource friendly serial interface based physical layer is prosed for the inter mini-column communication.

In order to prove the usefulness and the feasibility of the proposals the algorithm's implementation on Xilinx ZYNC system on chip (SoC) platform is provided. Ideas and proposals for further development and modifications are provided in the conclusions.

This thesis is written in English and is 63 pages long, including 7 chapters, 36 figures and 16 tables.

Annotatioon

Hierarhilise temporaalse mälu printsiibil toimiva ennustus- ja anomaalsustuvastuse komponent FPGA-l

Käesolev lõputöö käsitleb ühe kaasaegse tehisnärvivõrgu realiseerimist kiipsüsteemil. Töö aluseks olev närvivõrk, hierarhiline temporaalne mälu (HTM), on välja töötatud firma Numenta poolt ja sarnaneb funktsionaalselt ajukoore närvitulbale.

HTM algoritmi realiseerimine on iseenesest arvutuslikult üsnagi vähenõudlik. Antud asjaolule vaatamata on Numenta poolt arendamiseks ja katsetamiseks mõeldud HTM tarkvaraline väljatöötlus Nupic sobilik pigem lauaarvutile.

Üheks peamiseks piiranguks HTMi riistvaraliseks realiseerimiseks näiteks väliprogrammeeritaval loogikal on suur mälu nõudlus – muuhulgas salvestatakse Numenta poolt pakutud näidis realisatsioonis mällu ka neuronite omavahelisi ühendusi kirjeldavad parameetrid.

Antud töös käsitletakse lineaarse tagasisidega nihkeregistri võimalikku kasutamist neuronite ühenduste aadresside dekodeerijana. Lisaks nimetatule vaadeldakse riistvaraliselt vähenõudliku jadaliidese kasutamist närvivõrgu tulpadevahelise kommunikatsiooni füüsilise liidesena.

Tõestamaks pakutud ideede kasulikkust ning teostatavust on esitatud ka Xilinx ZYNQ kiipsüsteemil teostatud käigukatsetuste tulemused. Lisaks esitatakse ka ideesid edasisteks uurimusteks ning parendusteks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 63-l leheküljel, 7 peatükki, 36 joonist, 16 tabelit.

List of abbreviations and terms

ANN	Artificial Neural Network
AXI4	Advanced eXtensible Interface 4
BSP	Board Support Package
CPS	Cyber Physical System
CPU	Central Processing Unit
CTS	Clear To Send
DMA	Direct Memory Access
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FWFT	First Word Fall Through
HTM	Hierarchical Temporal Memory
IoT	Internet Of Things
IP	Intellectual Property
LFSR	Linear Feedback Shift Register
NoC	Network On Chip
RTS	Request To Send
SDR	Sparse Distributed Representation
SoC	System On Chip
SP	Spatial Pooler
TP	Temporal Pooler

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

The topic of machine intelligence has received a lot of attention and different Artificial Neural Networks (ANNs) have been developed over the time [1]. Adding intelligence to the Cyber Physical Systems (CPSs), like emerging Internet Of Things (IoT) network nodes, adds another criteria to the system design – the hardware resources of the often battery powered devices are limited.

It has been suggested that the goals as a tools of volitional behavior guide the overall behavior of the being through the attention [2]. Therefore the system's ability to pay attention to only the most important environmental observations (especially anomalies and novelties), i.e. filter the sensory input data based on it's importance, can definitely be considered as an essential property for an intelligent system.

Distinguishing abnormal observations from the entire input data set brings us to the anomaly detection. But modeling the entire system's environment and extracting the abnormality threshold is often impossible task to perform. Therefore the system should incorporate learning capabilities – if the environmental action or reaction has been learn previously and the system knows how to cope with it there is no need to pay additional attention to it.

1.1 Hierarchical Temporal Memory

Hierarchical Temporal Memory (HTM), one of the latest ANNs, is the model of the neocortex functionality, developed by Numenta, Inc. [3], originally described by Hawkins and Blakeslee [4].

The HTM is bio-inspired algorithm and has been developed based on the studies of the mammalians neocortex [5]. Although the entire functionality of the neocortex has not yet been implemented it has proven to be useful in the anomaly detection applications [6]. Geo-spatial tracking, natural language based predictions and server monitoring are another examples of the successful deployments [7]

The behavior of the algorithm can be tested using the Nupic software, provided by the Numenta [8].

1.2 HTM Building Blocks

HTM has the layered sructure, similar to the mammalian neocortex [3]. Every layer consist of possibly thousands of interconnected neural columns, with tens of cells each [9](Fig. 1.1).

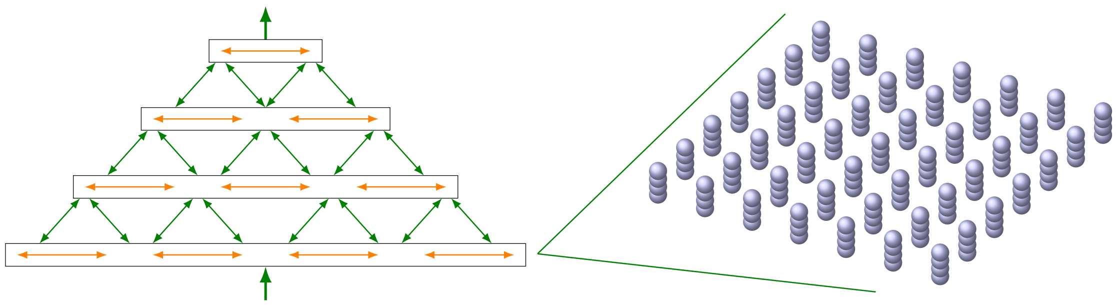


Figure 1.1: HTM has a layered structure. Every layer consists of possibly thousands of mini-columns and every mini-column consist of tens of cells. Every layer outputs the predictions to the higher hierarchical layers (green arrows). Orange arrows indicate the communication between cells of the mini-columns.

Every layer of the HTM constructs predictions based on the previously learned sequences. In case the received input data was unexpected in the current execution context the anomaly gets detected and reported. But if the same sequence is repeated often enough the HTM learns and remembers it and the sequence will be considered as a normal response of the environment in the future.

The data format HTM operates on is Sparse Distributed Representation (SDR) [10, 11, 12]. Sparsity means that only few bits of the entire data set are active concurrently and distributed means that the semantic meaning of the data is distributed across the bits, allowing sub-sampling of the data and providing fault tolerance.

Important aspect of the input data encoding is that semantically similar inputs should have similar encodings, i.e. semantically similar inputs should have common active bits in the SDR. HTM's ability to make generalizations is based on this feature.

The two main sub-tasks of the HTM algorithm are the Spatial Pooler (SP) and Temporal Pooler (TP).

1.2.1 Spatial Pooler

Algorithm execution starts from the SP receiving the feedforward input data. The input data is communicated to the mini-columns via the connections between the input data bits and a mini-column. Every mini-column is connected to approximately half of the input bits (Fig. 1.2).

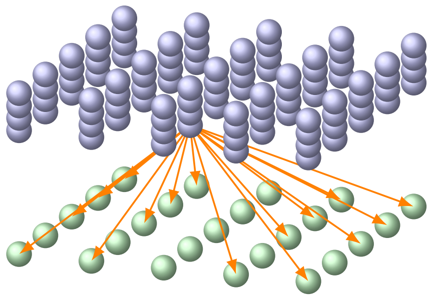


Figure 1.2: Every mini-column is connected to the $\sim$ half of the inputs. The set of connections is randomly selected for every column and does not change during the execution of the algorithm, but instead, every connection can be active or not active, based on it's associated permanence value.

Every connection between a mini-column and an input bit has a characteristic permanence value assigned to it. The connection is considered to be active, enabled, if the permanence value is exceeding the predefined threshold level (Eq. (1.1)).

$$W_{ij} = \begin{cases} 1 & if\ P_{ij} \geq Th \\ 0 & otherwise \end{cases} \tag{1.1}$$

Every mini-column calculates it's overlap count based on the applied input. The overlap count is the total sum of the active input bits connected to it via the enabled connections (Eq. (1.2)).

$$o_i = b_i \sum_j W_{ij} i_j \tag{1.2}$$

After the overlap calculation phase the required amount of mini-columns with the highest overlap count value will get activated. The required amount of active mini-columns is set by the required output sparsity of the SP process.

Learning is the last phase of the SP process. Learning is based on the competitive Hebbian learning: only the activated mini-columns, i.e. the ones with the highest overlap to the input data, perform the learning phase. The learning process alters the feedforward intput permanence values – the permanence values associated width active inputs, i.e. the inputs contributing to the activation of the mini-column, are incremented while the others are decremented (Eq. (1.3)).

$$P_{ij} = \begin{cases} P_{ij} + P_{inc} & if\ i_j = 1 \\ P_{ij} - P_{dec} & otherwise \end{cases} \tag{1.3}$$

The total number of possible input patterns the HTM matrix can encode scales rapidly with the size of the matrix: possible encodings for $n = 2048$ mini-column HTM with 2%

output sparsity ($w = 40$ active mini-columns) can be calculated as Eq. (1.4):

$$\frac{n!}{w!\,(n-w)!} = 2.37 \times 10^{84} \tag{1.4}$$

The complete description and the mathematical formalization of the SP can be found in [13, 14].

1.2.2 Temporal Pooler

While the SP's task is to encode the input data and activate the required amount of mini-columns, TP gives the context to an encoding and performs the learning of of the input data sequences [15].

While a specific input is encoded using the same set of activated mini-columns, the cells within it define the preceding sequence – the context (Fig. 1.3).

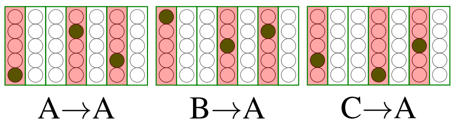


Figure 1.3: Temporal pooler gives an input pattern the context. While the same input is encoded using the same set of mini-columns, context is defined by the cells.

Every cell within a mini-column is connected to a set of cells in the other mini-columns in the HTM matrix. if a cell is connected to the enough amount of cells in an active state it changes it's state to *predictive*.

Switching the cells to the predictive state is the key for the anomaly detection. If an activated mini-column contains a cell which is predicting it means that the input sequence has been learned in the past.

Every connection between the two cells have similar permanence value assigned to it as the feedforward connections have. And the learning process is also similar: activated cells increment the permanence values connected to the contributing cells and decrement the rest.

If the input is encoded using total $w = 40$ number of mini-columns with $c = 10$ cells, the total amount of different contexts for an input can be calculated as Eq. (1.5).

$$c^w = 1 \times 10^{40} \tag{1.5}$$

2 OBJECTIVES

Objective of this master thesis is to evaluate the feasibility of running the HTM algorithm on a Field-Programmable Gate Array (FPGA) based System On Chip (SOC).

Although the algorithm is available [8] for the community for testing and further development, the implementation is more suitable for the conventional computer system. Smaller CPSs do not have enough resources available to meet the memory requirements, for example.

There have been several attempts to either boost the performance of the HTM algorithm [16, 17] or to provide the HTM building blocks for the FPGAs [18, 19, 20] but the functional model suitable for smaller SOCs still seems missing.

This thesis focuses on implementation of the SP process. The goal was to implement the mini-column capable of accepting the feedforward input and performing the overlap calculation and column activation with the minimal usage of the resources. The other goal set was to support the scalability, i.e. avoid cross-bar switches or similar.

3 PROPOSED METHODS

The first proposal by the author of this thesis is to use the Linear Feedback Shift Register (LFSR) for the SP address space decoder. The benefit of generating the connection addresses upon the need can be estimated from the amount of memory required for storing them into the memory. Lets suppose every mini-column is connected the approximately half of the bits of the feedforward input vector of length m. The minimum amount of bits required for storing every connection address then equals to $\log_2 m$ and the total amount of bits for storing all the required addresses for every mini-column can be calculated as Eq. (3.1).

$$n = \lceil \log_2 m \rceil * \frac{m}{2} \tag{3.1}$$

The second proposal is to use the minimal serial communication with simple flow control capabilities for the mini-column communication. Chaining the mini-columns of the HTM matrix does not provide any performance boost, but instead, reduces the cost of the hardware. This trade-off is analyzed in terms of the SP execution speed and is shown not to become the performance bottleneck.

These main ideas of this thesis have been presented and published in DDECS conference Apr.2017 Dresden, Germany (Appendix A).

3.1 Thesis Layout

The following paragraphs provide the bottom-up description of the realization and HDL simulations of the proposed methods. Hierarchically lower level building blocks like *serial interface* and data deserializer unit *data buffer* are described first, followed by the higher level components for SP processing and mini-column housekeeping related *mini-column controller*. The overall HTM controller and Intellectual Property (IP) implementation details conclude the following hardware description Chapter 4.

The theoretical performance of the SP process utilizing the proposed methods is defined in Chapter 5, followed by the chapter presenting the evaluation results of the algorithm executing in the real hardware.

4 HARDWARE DESCRIPTION

This chapter will discuss the implementation of the hardware capable of executing the SP process. Simulation results of the HDL components are provided together with the description of the implementation while the details about the syntheses are collected to the Sections 4.6.2 and 4.6.3, describing the IP design.

The main target platform selected for testing the implementation on the real hardware was Xilinx ZYNQ-7020 SoC [21]. The main criteria for the selection was to have the conventional Central Processing Unit (CPU) resource closely coupled to the programmable logic, allowing easier analysis of the hardware execution results and possible partitioning of the algorithm between the software and the hardware.

The selected target platform provides:

- Dual-Core ARM Cortex-A9 processing system.
- 140 36Kbit dual port block RAMs. Each memory block can be split into two single port RAMs.
- 53200 LUTs.
- 106400 Flip-Flops.

Assuming that the implementation of a mini-column does not use more than 18KBit of the RAM the selected SoC could fit over 200 mini-columns, depending on the usage of the logic cells.

In addition to the main target the HTM matrix was synthesized targeting the Altera's MAX10M50 FPGA [22]. The main reason for this was just to demonstrate the compatibility of the HTM controller with the First In First Out (FIFO) IPs and host bus protocols from the different vendors.

Having all the mini-columns chained to the the serial communication scheme with the host bus interface attached leads to the overall system design as per Fig. 4.1.

4.1 Serial Interface

All the mini-columns in the HTM have to be connected to the set of input bits and all the cells within the mini-columns have to be connected to the set of other cells, i.e. the re-

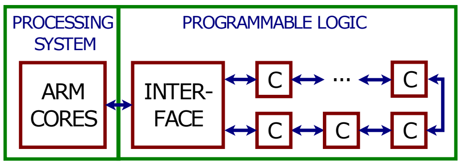


Figure 4.1: All the mini-columns are serially connected and interface to the CPU using the bus controller suitable to the processing core(s) on the SoC. Bus interface controller can provide optional buffering of the data and interrupt signals for service requests.

quired amount of interconnections is huge. Expanding the size of the HTM matrix makes things even worse, adding one extra mini-column requires adding $L_{in}/2$ connections for connecting the mini-column to the required amount of the inputs plus the set of connections for every cell for the SP operation, where the L_{in} stands for the length of the input pattern.

Serial interface in between adjacent mini-columns can efficiently solve this problem. The approach presented in this thesis uses the interface consisting of three wires, serial data and flow control signals (Fig. 4.2). Flow control involves two signals: Request To Send (RTS) and Clear To Send (CTS). In case a mini-column has any new data to send it asserts it's RTS signal to high and if there is buffer available for a new data it asserts it's CTS signal. This kind of minimal handshake avoids the risk of possible buffer over- or underflows.

Using the serial interface and communication protocol instead of the hard-wired connections reduces the maximum update rate of the HTM matrix. In order to minimize this side effect the delay introduced by every additional interface in the communication chain has to be as short as possible – 1 clock cycle, i.e. every instance of the interface should forward the received data bit on the next clock cycle.

Requiring each interface to forward the received data in the next clock cycle can yield to the very long CTS signal path if the proper counter measures are not considered. This effect can easily be imagined considering the chain of the mini-columns all forwarding data and de-assertion of the CTS at the end of the chain. As the result, the last mini-column can not forward it's data and therefore can not accept any new data either. But provided that the long CTS chain has to be avoided then every mini-column should implement a shadow buffer for an additional data storage. Having the shadow buffers available effectively limits the required CTS signal propagation to one mini-column per clock cycle. However, efficient operation of the HTM requires more capabilities than simple data reception and forwarding from the serial interface (Fig. 4.3). E.g. determining the winning mini-columns during the SP phase requires arbitration of the mini-column's locally calculated overlap count with the value serially received from the interface. As again, the arbitration process should not include any additional delay – if the mini-column's overlap

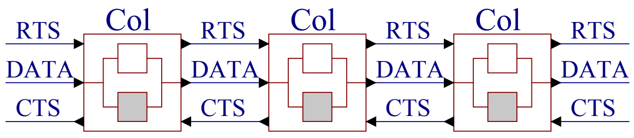


Figure 4.2: All the mini-columns are connected it's neighbors using the simple serial interface. Implemented shadow buffers (gray) help to avoid the long CTS signal path.

count is higher than the value contained in the frame transmitted, it's local overlap count should be transmitted to the upstream port instead.

Yet another feature added to the serial interface is data replace mode. The name of this mode is quite self explanatory, the interface instance just replaces the received data with the data locally available to it via the TxBit signal line. This mode is used for reading the state of the HTM mini-column or contents of it's block RAM, for example.

Operating modes of the serial interface are explained in the Sections 4.1.1 to 4.1.3.

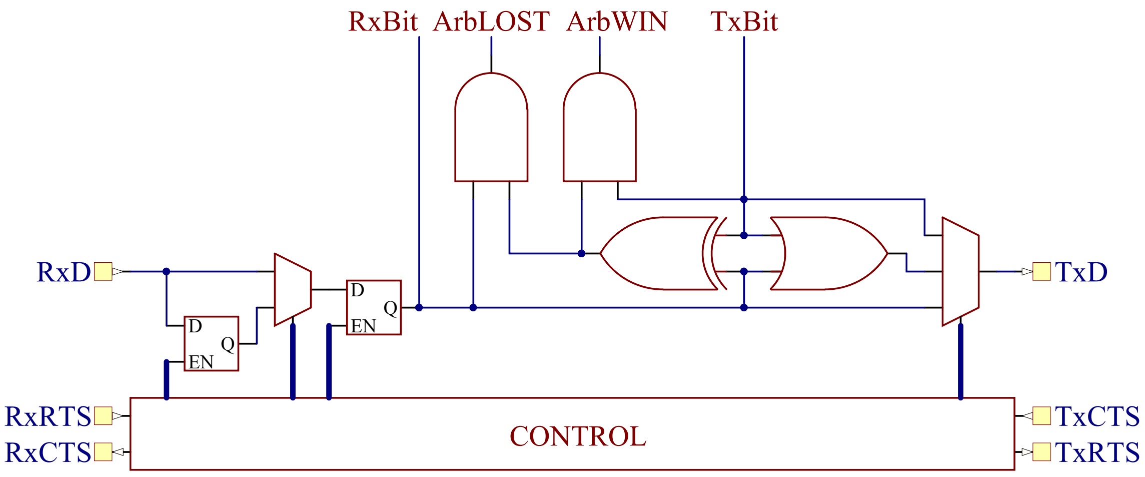


Figure 4.3: Serial interface. TxD signal from the serial interface can be either RxD + TxBit, RxD or TxBit, depending on the selected mode. Signals ArbLOST and ArbWIN are used for the arbitration process.

4.1.1 Forwarder Mode

The simplest mode for the interface is bit forwarding. Every interface instance buffers the received data bit and forwards it at the next clock cycle, if there is space available in the upstream interface. This is the case of the normal data forward flow.

Upstream interface can halt the transmission by asserting it's CTS signal to low, resulting the downstream interface to make use of it's shadow buffer and clear it's CTS signal at the next clock cycle.

In addition, the serial interface instance clears it's RxCTS signal if the local controlling master does not read the data as soon it becomes available, effectively avoiding the overflow of the buffered data.

All these three scenarios have been simulated in order to verify the proper behavior of the serial interface (Fig. 4.4). First of all, there is exactly one clock cycle delay in case of

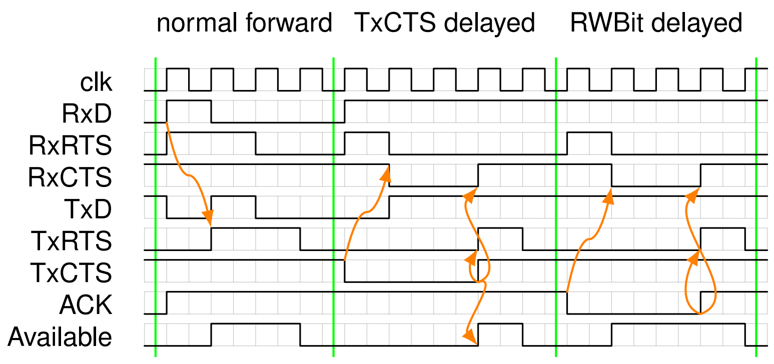


Figure 4.4: Serial interface forward mode simulation. Three scenarios are simulated: normal forward where upstream interface constantly has space for the new data and upper layer component is constantly accepting it, TxCTS delayed where upstream interface clears its CTS signal informing there is no space available and ACK delayed where the master controller component does not read the data causing the transmission delay.

normal forward flow – CTS signal from the upstream port (TxCTS for the UUT) and ACK from the master controller are constantly high, presented on the left side of the simulation timing diagram.

Middle portion simulates the scenario where the upstream port has cleared it's CTS towards the UUT, causing the transmission halt. Signals Available and CTS towards the downstream port are set high as soon as the data can be forwarded.

Right side of the diagram presents the results of the master controller unavailability simulation. Signals CTS towards the downstream port and RTS towards the upstream port are set high as soon as the master controller reads the data by asserting the ACK signal.

4.1.2 Replacer Mode

Replacer mode is similar to the forward mode in the implementation of flow control. The only difference is that the data bit received from the serial line is simply discarded and the local value received from the TxBit line is transmitted instead.

This mode is useful for reading the contents of the mini-column's block RAM and for the additional *capture* mode implemented by the data buffer component (Section 4.2.4).

The replacer mode was simulated by transmitting bits 0 and 1 to the interface RxD port while setting the local TxBit to 1 and 0, respectively (Fig. 4.5). Interface instance successfully replaced the data on the serial line with it's local data, proving the replacer mode is functioning properly.

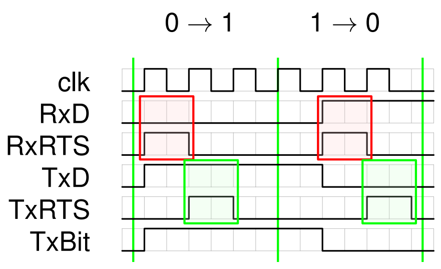


Figure 4.5: Serial interface replacer mode simulation. Red rectangles mark the data sent to the interface RxD port, 0 and 1. Received data is dropped and the local bit from the TxBit line is transferred instead, marked by green rectangles.

4.1.3 Arbiter Mode

Arbitration mode is useful for performing the competitive learning, e.g. picking the set of highest SP overlap counts. There should be no more than one clock cycle delay for a data bit arbitration, similarly to the forwarder and replacer modes. The idea behind the arbitration mode is that the high value is transmitted by the interface if either received bit or the bit set by the master controller is high. Low value should be transmitted otherwise.

This can be achieved by transmitting the value $\mathrm{TxD} = \mathrm{TxBit} + \mathrm{RxD}$ to the upstream port. As soon as there is a difference in between the local value and the received frame the mini-column has to switch either to the *arbitration lost* or *arbitration won* mode and continue forwarding the received frame or it's own data (Eq. (4.1)).

$$\mathrm{TxD} = \begin{cases} \mathrm{RxD} + \mathrm{TxBit}, & \text{if MODE} = \text{arbitration} \\ \mathrm{RxD}, & \text{if MODE} = \text{lost arbitration} \\ \mathrm{TxBit}, & \text{if MODE} = \text{won arbitration} \end{cases} \tag{4.1}$$

Serial interface assists the master controller in proper mode selection by constantly out-putting the signals ArbLOST and ArbWin (Fig. 4.3 and Eqs. (4.2) and (4.3)).

$$\mathrm{ArbLOST} = \mathrm{RxD}(\mathrm{RxD} \oplus \mathrm{TxBit}) \tag{4.2}$$

$$\mathrm{ArbWIN} = \mathrm{TxBit}(\mathrm{RxD} \oplus \mathrm{TxBit}) \tag{4.3}$$

Two scenarios where simulated to verify the proper behavior: interface winning and loosing the arbitration process (Fig. 4.6).

In order to make the interface instance to loose the arbitration process bit pattern 101 was transmitted to it's RxD port and 100 to it's TxBit signal line. Arbitration process carried on up to the first difference. As high signal was transmitted to the RxD and low signal to the TxBit, the interface asserted the ArbLost signal, correctly signaling that the arbitration process was lost.

Transmitting bit patter 100 to the RxD port and 101 to the TxBit caused the interface to win the arbitration process as the local value was higher than the value received from the line. ArbWin signal was correctly set to high during the arbitration of the third bit.

The bit pattern transmitted by the interface to it's TxD port was TxD $= 101 = 101 + 100$ in case of both tested scenarios.

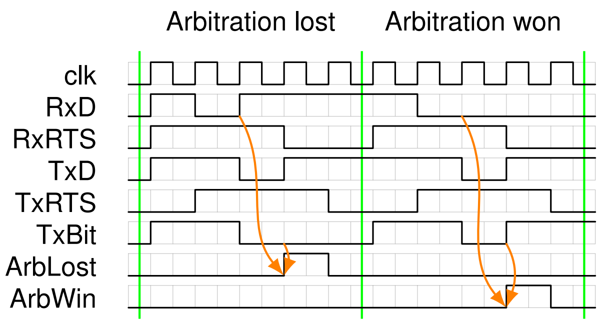


Figure 4.6: Serial interface arbitration mode simulation. Two scenarios are simulated. Firstly, "101" is transmitted on the RxD port while "100" is locally transmitted to the interface, causing it to loose the arbitration. Secondly, "100" is transmitted to the RxD port and "101" to the TxBit, causing the interface to win the arbitration. "101" is transmitted to the TxD port as the result of arbitration in both cases.

4.2 Data Buffer

Although the serial interface HW component provides fine control of the communication, bit level data manipulation is not always needed. Writing or reading the state of the HTM matrix or transmitting the calculated overlap counts require operating with data words, for example.

Data buffer extends the functionality of the serial interface by operating with data words instead of single bits. Data buffer can work in one of the following modes: *forwarder*-, *replacer*-, *arbiter*- or *capturer* mode. Upper level components do not have to monitor the availability of input data or possible communication hold requests from the upstream port as the control and status lines of the underlaying serial interface are controlled by the data buffer.

With of the data word of the buffer can be configured at design time. Current HTM realization uses the data with of 8 bits. Bit with equal to power of two is only supported by the current buffer implementation. This requirement simplifies the internal bit counter implementation, e.g. 8 bit data can be counted using the three bit counter and there is no need to reset the counter in between adjacent data frames, the counter simply overflows back to zero.

Processing of multiple data words is supported by integrated frame counter. This approach simplifies the upper level component design.

4.2.1 Forwarder Mode

Forwarder mode is intended for the simple data accumulation purposes. The mode is named as *forwarder* in order to emphasize that the accumulated data is actually forwarded during the accumulation process.

The controlling state machine of the forwarder mode process consists of two states: *IDLE* and *FWD* (Fig. 4.7). The *FWD* state is entered as soon as the buffer is set to the *forwarder* mode by assigning the corresponding control line. Terminate condition is matched as soon as the last bit of the last frame required has been accumulated and forwarded.

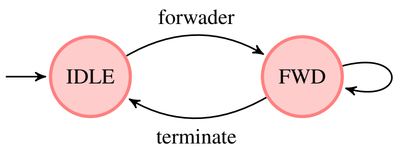


Figure 4.7: Data buffer forwarder mode FSM. The *FWD* state constantly reads the data from the underlying serial interface. As soon as the required amount of bytes have been received and forwarded the FSM returns to the *IDLE* state.

Forwarder mode was simulated by sending two 8 bit data frames to the buffer, 0xAA and 0x55 (Fig. 4.8). Ready signal was asserted after the successful reception of every data frame. The buffer asserted the dataStatus signal and returned to the *IDLE* state after all of the required frames where successfully received.

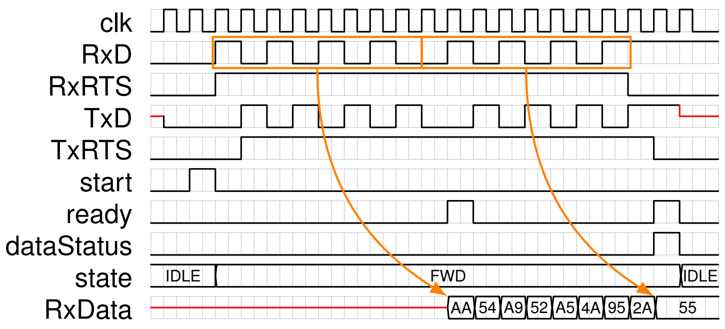


Figure 4.8: Data buffer forwarder mode simulation. The buffer acts like a data accumulator in the forwarder mode. Received data bits are forwarded to the TxD port with one clock cycle delay during the accumulation.

4.2.2 Replacer Mode

Replacer mode, as suggested by it's name, simply *replaces* the received data with the contents of the local TxData buffer. Replacer mode state machine consists of two states, *IDLE* and *RPL* (Fig. 4.9). The replacer mode is needed in order to read the state of the HTM. HTM's controller can read the contents of the mini-column's block RAM by sending dummy bytes to the serial interface. Every mini-column then replaces the appropriate bytes in the message with the contents of it's block RAM and the return message from the communication chain contains the required values.

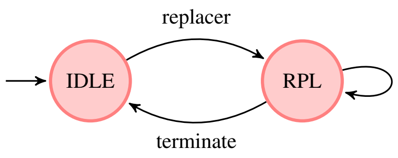


Figure 4.9: Data buffer replacer mode FSM. The *RPL* state drops and replaces the incoming data frames with it's TxData buffer contents. *IDLE* state is returned as soon as the required amount of frames has been processed.

The replacer mode was simulated by sending two 8 bit data frames to the buffer: 0x81 and 0xFF. The contents of the frames transmitted to the buffer is not really important as the data gets replaced. But for better observability of the ongoing replace process the frames transmitted to to the buffer have to be different compared to the buffer local data. As can be seen from the simulation results (Fig. 4.8) the buffer instance replaced the received frames. Data frames transmitted to the TxD port equal to the contents of the buffer's local TxData register.

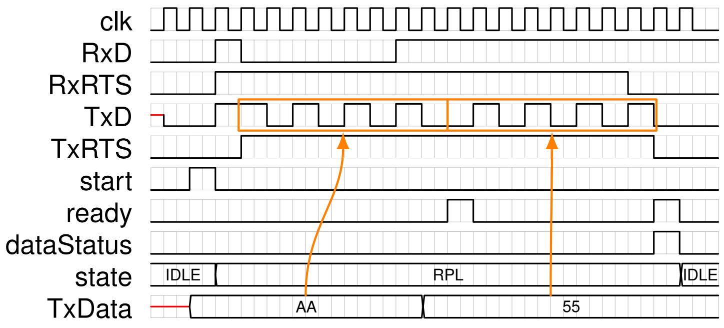


Figure 4.10: Data buffer replacer mode simulation. Received data is simply replaced by the local data in case of the replacer mode. Two bytes of data are sent to the buffer RxD port: 0x81 and 0xFF, which are replaced by the local data 0xAA and 0x55, respectively.

4.2.3 Arbiter Mode

Arbiter mode is used for the competitive learning, e.g. selecting the highest overlap values during the SP process. The goal for the arbitration process is to insert the local TxData value into the transmitted data packet consisting of several data frames. Data can be inserted only if the binary value of the currently transmitted frame is less the value present in TxData holding register. Additional requirement are that the local data can be inserted only once per data packet currently processed and the rest of the original transmitted packet has to be delayed by one frame and transmitted after successfully inserting the local value.

Arbiter FSM consists of five states: *IDLE*, *ARB*, *LOST*, *JWON* and *WON* (Fig. 4.11).

Active arbitration state *ARB* is entered as soon as the buffer is set to the arbitration mode by asserting the corresponding control lines. This state sets the underlying serial buffer to the arbitration mode and monitors the signals arbLost and arbWin. As soon as the bit in the transmitted frame is 1 and the corresponding TxData bit is 0 the buffer looses the arbitration and enters the *LOST* state. The rest on of the currently transmitting frame is simply forwarded during this state and *ARB* state is re-entered as the last bit of the current frame has been transmitted. Switching in between *LOST* and *ARB* states skips the beginning frames which have higher value compared to the TxData register.

In case the TxData value is higher than the transmitting frame the arbiter FSM enters the state *JWON*, which is the shorthand for arbitration Just WON. This mode sets the underlying serial interface to the replacer mode and replaces the currently transmitting frame with it's TxData register contents. Important to notice here is that the original frame is still accumulated into the buffer's RxData register.

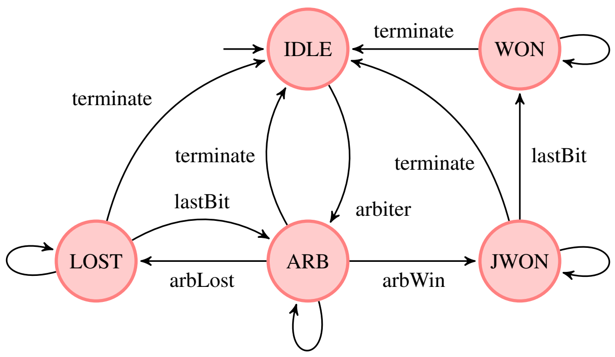


Figure 4.11: Data buffer arbiter mode FSM. Beginning frames with higher value than the buffer's local data are simply forwarded by switching between the *ARB* and *LOST* states. The state *JWON* is entered as soon as the value of the local data is higher compared to the frame currently transmitting. The original data packet is appended to the transmitted local data in the *WON* state, shifting the original data by one frame.

Arbitration *WON* state is entered as soon as the last bit of the TxData register has been transmitted. The *WON* state, as well as *JWON*, sets the underlying serial interface to the replacer mode. The important difference is that the previously accumulated data is transmitted in this state instead of the local TxData, effectively delaying the original data packets by one frame. The terminate condition is met as soon as the last bit of the last frame has been processed.

Proper behavior of the buffer's arbitration mode was simulated by transmitting the data packet consisting of three frames: 0xAF, 0x81 and 0x55 (Fig. 4.12). Buffer's local data, the TxData register, was loaded with the value 0xAA.

The first data frame is forwarded unaltered by the buffer as the transmitted frame 0xAF has higher value compared to the local data 0xAA. The state *LOST* is entered as soon as the arbLost signal is asserted by the underlying serial interface, resulting in the forwarding of the original data frame. The arbitration process is won during the transmission of the second frame and the local data 0xAA is transmitted instead of the original frame 0x81. The original second frame 0x81 is delayed and transmitted as the third frame.

The status of the performed arbitration process is communicated using the dataStatus signal. It is set to high if the local data was successfully inserted into the transmitting data packet.

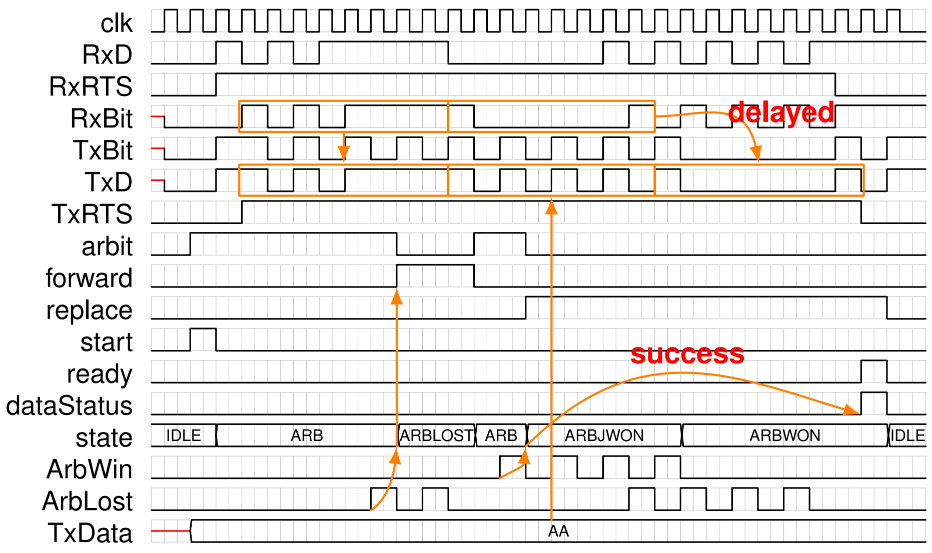


Figure 4.12: Data buffer arbitration mode simulation. Three bytes are transmitted to the data buffer: 0xAF, 0x81 and 0x55. The local data for the arbitration is loaded into TxData register: 0xAA. Data buffer wins the arbitration while transmitting the third bit of the second byte. Successful arbitration is reported by asserting the signal dataStatus to high after required amount of bytes have been processed. The original second byte is stored and transmitted instead of the third byte.

4.2.4 Capture Mode

Capture mode operation complements the arbitration mode in order to complete the competitive learning process, e.g. in case the mini-column successfully inserted it's overlap count into the data packet the value could have been overwritten by the following mini-columns in the communication chain.

In order to determine if the transmitted overlap count was not overwritten the last bit of the frame is reserved for the capture operation. The original overlap count is shifted to the left by one position and the last bit is set to zero. Capture mode operation then tries to *capture* a frame from the transmitted packet by setting it's last bit to high, if not set by previous a mini-column already.

Capture mode FSM consists of four states: *IDLE*, *CAPT*, *LOST* and *CFWD* (Fig. 4.13).

First, the received data frame is compared to the contents of the TxData register up to the second last bit in the *CAPT* state. The state *LOST* is entered as soon as the data does not match. The rest of the bits of the current frame are simply forwarded in the *LOST* state and the state *CAP* is re-entered at the beginning of the next frame.

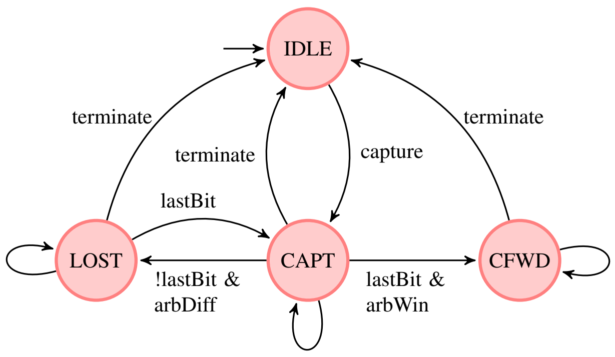


Figure 4.13: Data buffer capture mode FSM. Buffer compares the received frame to the TxData register in the *CAPT* state and enters the state *LOST* in case the data does not match. The state *CAPT* is re-entered if there are more frames to follow. *CFWD* state is entered as the result of the successful capture and the remaining frames of the packet are forwarded unaltered.

In case the transmitted data frame matches the local TxData register the buffer tries to set the last bit of the frame to high, if it is not set already. Underlaying serial interface is set to arbitration mode for this operation. In case the arbWin signal gets asserted, the frame was successfully captured and the FSM enters the state *CPFD*. The *CPFD* state is used to forward the remaining frames of the data packet.

The result of the capture operation is communicated by the signal dataStatus, it is set high in case of successful capture of a frame. Terminate condition is met as soon as the last bit of the last frame has been processed.

Proper behavior of the capture mode was simulated by transmission of the data packet consisting of three frames: 0xAA, 0x81 and 0x80 (Fig. 4.14). Local TxData register was loaded with value 0x81.

Capturing of the first data frame failed as the third bit of the transmitted frame and the local TxData register are different, as signaled by the underlying serial interface.

The second frame equals to the TxData register up to the second last bit and the frame capture was attempted. Capturing of the second frame did not succeed as the last bit of the transmitted frame was set already.

As the capture attempt of the second frame failed the FSM stays in the *CAPT* state and starts comparing the next frame to the TxData register value. As the frame matched the local data and the last bit of the transmitted frame 0x80 was not set, capturing of the third frame succeeded.

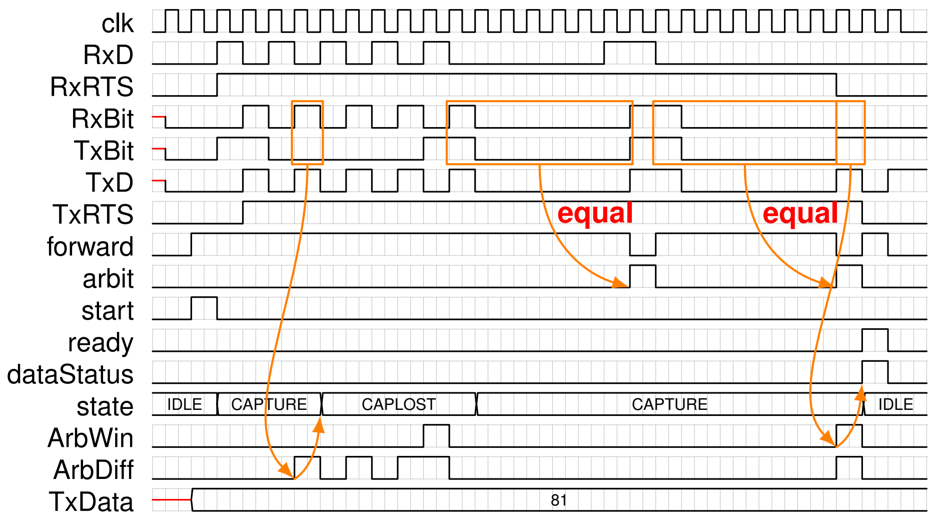


Figure 4.14: Data buffer capture mode simulation. Three bytes are transmitted to the buffer: 0xAA, 0x81 and 0x80. Local data for capture process is stored in the TxData register: 0x81. Capturing the first received frame fails as the received frame and local data are different. Capturing of the second received frame 0x81 is not successful as the last bit is set already. Capturing of the third frame succeeds. Successful capture is reported by asserting the signal dataStatus to high.

4.3 Spatial Pooler

Spatial pooler is the first stage of the HTM algorithm. It converts the input data into the SDR with the fixed dimension and sparsity which is then used as the input for the TP.

Operation of the SP consists of the four major phases: receiving the input data, calculating the overlap counts, determining the winning mini-columns and optional learning phase. In the implementation presented in this thesis one additional feature was added to the SP component: generation and transmission of the output SDR. This feature is needed in order to compare the SP hardware component output to the one of SP software realization or feed it to the TP software process.

The FSM of the spatial pooler consists of three distinguishable sub FSM-s: feedforward input processing (FF IN), learning (LEARN) and the output SDR generation (OUT SDR) (Fig. 4.15).

Determining the winning mini-columns is skipped from the SP component, as can be seen from the SP FSM partitioning. This design decision was made because the SP component uses the direct access to the serial interface instance (Section 4.1) and winning cells detection can benefit more from the features provided by the data buffer component (Section 4.2). Therefore the SP component does calculate the overlap count but outputs the calculated value to the mini-column component which has the buffer hardware included.

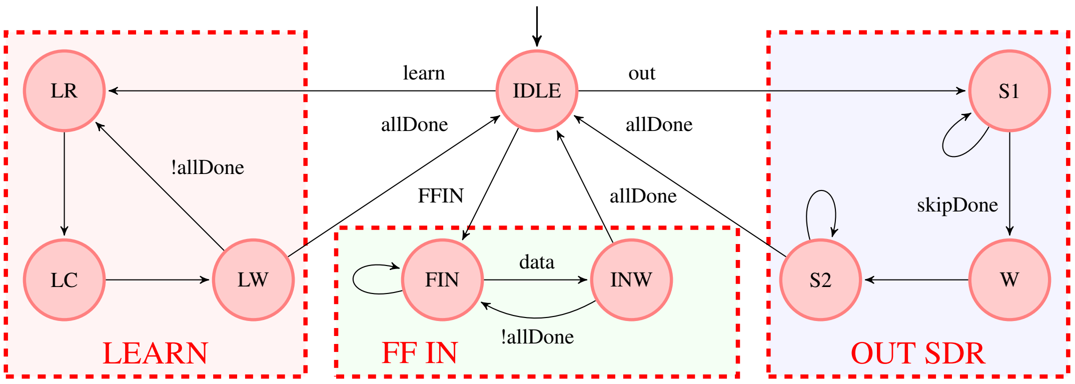


Figure 4.15: Spatial pooler FSM consist of three sub FSM-s: feedforward input processing (FF IN), learning (LEARN) and output SDR generation (OUT SDR). LEARN FSM increments or decrements the connection permanence values, OUT SDR generates and transfers the output SDR and FF IN receives the feedforward input.

4.3.1 Feedforward Input

The feedforward input sub FSM (FF IN) is responsible of receiving the input data. All the HTM mini-columns are connected to the approximately half of the input bits, therefore the SP component either has to store the connection indexes to the mini-columns internal block RAM or use some other method to keep track of the connections.

As the RAM is limited resource, the implementation presented in this thesis takes the approach of not storing the connection indexes. Considering the facts that every mini-column has to be connected to the approximately half of the random inputs and the SP feedforward inputs do not change in time, the information can be generated using the LFSR.

LFSR generates pseudo random bit patterns, depending on the feedback polynomial and the initial seed value. The correlation between the generated patterns can be calculated using different techniques [23] and is the same for every vector pair in case of the LFSR uses a primitive feedback polynomial.

Using primitive polynomials guarantees that the LFSR register includes all the possible values, i.e. generates the pseudo bit pattern of maximum length. The only restricted value for the LFSR is all zeros, which is never generated, unless the LFSR register is initialized with it.

LFSR with primitive polynomial $X^4 + X^3 + 1$ (Fig. 4.16) generates $2^n - 1 = 15$ bits long pattern and the LFSR register travels all the possible values in the range $[1\ldots15]$ (Table 4.1). The least significant bit, b0, is used as the output of the LFSR. As the LFSR register travels all the possible values the n bit long register can hold, except all zeros, the output bit of $n = 4$ bit long LFSR is guaranteed to have $b_{ones} = 2^n/2 = 8$ one bits and $b_{zeros} = b_{ones} - 1 = 7$ zero bits.

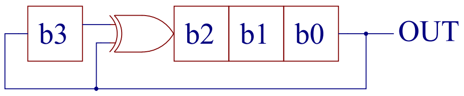


Figure 4.16: LFSR using primitive polynomial $X^4 + X^3 + 1$.

Table 4.1: LFSR register values. The LSB b0 is used as the output of the LFSR.

Seq.	b3	b2	b1	b0	Seq.	b3	b2	b1	b0	Seq.	b3	b2	b1	b0
1	0	0	0	1	**6**	1	0	1	0	**11**	1	0	1	1
2	1	1	0	0	**7**	0	1	0	1	**12**	1	0	0	1
3	0	1	1	0	**8**	1	1	1	0	**13**	1	0	0	0
4	0	0	1	1	**9**	0	1	1	1	**14**	0	1	0	0
5	1	1	0	1	**10**	1	1	1	1	**15**	0	0	1	0

The bit pattern generated by the LFSR with seed value 0001 equals to 100110101111000. Changing the seed value does not change this pattern but rotates it. The initial seed for every mini-column can be chosen by different criteria, favoring more connections in certain positions or maximizing the distribution of connections over the entire range of the HTM mini-columns. Or as there are many alternative feedback polynomials available for the same pattern length, especially for the longer bit patterns, different mini-columns could even be initialized using different LFSR feedback.

Possible connection matrix of the 15 mini-column HTM to the 15 bit input vector is presented in the Table 4.2. Cell value 1 indicates that there is possible connection between the input bit (table column) and the HTM mini-column (table row). The seed values for different HTM mini-columns are initialized in favor of maximizing the bit value transitions in every table column, distributing the possible connections. Every SP instance advances it's LFSR as the new feedforward input bit is transferred to it. Depending on the output of the LFSR the mini-column either reacts to it or forwards without notice. If the input bit belongs to the mini-column, it is saved as the *activity* bit into the mini-column's block RAM, as the highest bit of the corresponding permanence value memory field (Fig. 4.17).

The LFSR output vector L and input vector I of length m are defined as:

$$L = (l_1, l_2, l_3, \ldots, l_m) \tag{4.4}$$

Table 4.2: SP connections generated by LFSR-s with different seed values, 1 indicates the connection in between mini-column C_i and input bit I_j.

Col	Seed	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
C_1	**1**	1	0	0	1	1	0	1	0	1	1	1	1	0	0	0
C_2	**2**	0	1	0	0	1	1	0	1	0	1	1	1	1	0	0
C_3	**3**	1	1	0	1	0	1	1	1	1	0	0	0	1	0	0
C_4	**4**	0	0	1	0	0	1	1	0	1	0	1	1	1	1	0
C_5	**8**	0	0	0	1	0	0	1	1	0	1	0	1	1	1	1
C_6	**5**	1	0	1	1	1	1	0	0	0	1	0	0	1	1	0
C_7	**12**	0	0	1	1	0	1	0	1	1	1	1	0	0	0	1
C_8	**6**	0	1	1	0	1	0	1	1	1	1	0	0	0	1	0
C_9	**9**	1	0	0	0	1	0	0	1	1	0	1	0	1	1	1
C_{10}	**7**	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0
C_{11}	**15**	1	1	1	0	0	0	1	0	0	1	1	0	1	0	1
C_{12}	**10**	0	1	0	1	1	1	1	0	0	0	1	0	0	1	1
C_{13}	**11**	1	1	0	0	0	1	0	0	1	1	0	1	0	1	1
C_{14}	**13**	1	0	1	0	1	1	1	1	0	0	0	1	0	0	1
C_{15}	**14**	0	1	1	1	1	0	0	0	1	0	0	1	1	0	1

$$I = (i_1, i_2, i_3, \ldots, i_m) \tag{4.5}$$

where, $l_i, i_i \in \{0,1\}, \forall i \in \{1,2,\ldots,m\}$ and the activity vector is defined as:

$$A = (a_1, a_2, a_3, \ldots, a_n) \tag{4.6}$$

The activity vector $A \subset I$ is defined for every i where LFSR output is one:

$$A(j) = (i_i | l_i = 1) \tag{4.7}$$

The length of the activity vector is half of the the length of the input vector $n \approx m/2$, as defined by the properties of the LFSR output bit pattern L.

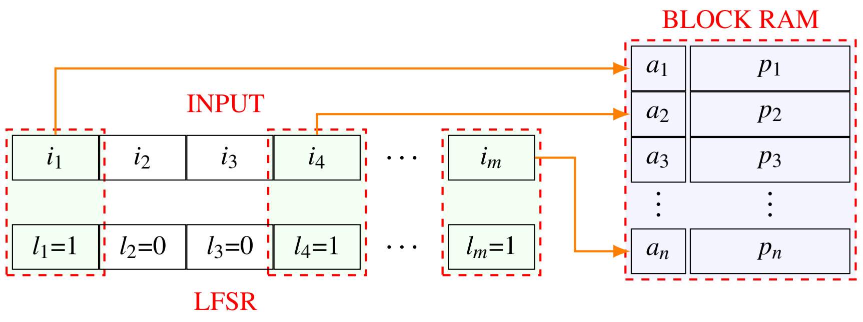


Figure 4.17: LFSR used as the input address decoder. If the output of the LFSR is 1, the corresponding input bit i_i is saved to the mini-columns block RAM as the highest bit of the permanence value p_j.

The feedforward input processing FSM FF IN transients between the two states, *FIN* and *INW*. First, the FSM waits for the new data in the state *FIN* and enters the sate *INW* as soon as the data becomes available. The LFSR output is checked in the state *INW* and if it's 1, the data bit is saved to the block RAM as the activity bit and and the RAM address gets incremented.

Calculating the mini-columns feedforward input overlap count can be combined with the reception of the input data. The overlap counter is activated in the stage *INW* if two highest bits of the current memory word are high, the activity bit and the highest bit of the permanence value, i.e. the input bit has value 1 and the permanence value is over the half of the full range of the register.

The SP feedforward input was simulated by transferring the bit vector 10001010 to it. The LFSR was configured according to Fig. 4.16 and initial seed set to 1. The output stream of the LFSR matches the Table 4.2 row 1, which is the correct result. Write enable signal *we* is enabled and the RAM address gets incremented for every LFSR output high cycle (Fig. 4.18). Overlap counter also works as expected, it gets incremented if the LFSR output and the input bit RxBit are high marking – the input is active and the permanence value is over the threshold.

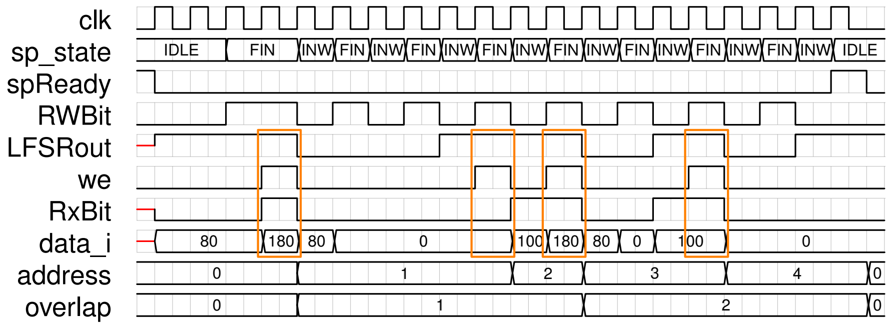


Figure 4.18: Spatial pooler feedforward input. If the LFSR output is high, the corresponding input bit is stored to the mini-column's memory. Overlap counter gets incremented if two highest bit of data_i register are set, the activity bit received from the interface and the highest bit of the corresponding permanence value.

4.3.2 Learning

Learning process of the SP strengthens the mini-column connections which are connected to the active input bits and weakens the others. Incrementing the permanence value of a connection strengthens it while decrementing it has the opposite effect. Prerequisite for the learning is that the mini-column has to belong to the set of winning mini-columns with the highest overlap counts.

Learning process FSM contains three states: *LR*, *LC* and *LW* (Fig. 4.15), forming the read (*LR*), modify (*LC*), write (*LW*) sequence. Learning does not include any communication and is the local operation for a mini-column.

Incrementing or decrementing of a permanence value can be carried out by a up/down counter hardware and possible over- or underflow has to be avoided. Over- or underflow can be checked by configuring the counter to be 1 bit wider than the permanence value field and using the highest bit as the inhibition for the write signal, i.e. if the highest bit of the counter output is set, the write enable signal gets inhibited in the *LW* state.

Learning process was simulated using four different scenarios: normal increment, increment with the overflow, normal decrement and decrement with the underflow (Fig. 4.19).

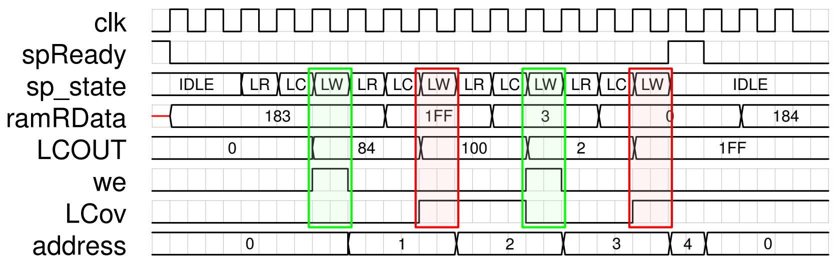


Figure 4.19: Spatial pooler learning. Permanence values get either incremented or decremented based on the activity bit a_i stored as the MSB of the memory field. Write enable *we* is inhibited in case the LCov signals the over- or underflow. These situations are marked by the red rectangles. Green rectangles mark the normal increment or decrement cycle.

SP was configured to have 4 feedforward inputs connected to it, resulting in four permanence values stored in to the block RAM. The RAM was initialized with data words 10000011, 11111111, 00000011 and 00000000 (Table 4.3). The highest bit of the memory fields hold the activity bit a_i and the lower 8 bits hold the permanence values p_i.

As per the activity bit a_i value the first two permanence values have to be incremented and the last two have to be decremented during the learning process. Processing of the the second and fourth field yield to over- and underflow respectively and remain unchanged, proving the detection mechanism is functional.

Table 4.3: SP learning. The permanence values p_i get either incremented or decremented, based on the activity bits a_i . If the operation yields to a overflow the value remains unchanged. Address 1 receives the overflow and address 3 the underflow.

address	a_i	p_i before	p_i after
0	1	10000011	10000100
1	1	11111111	11111111
2	0	00000011	00000010
3	0	00000000	00000000

4.3.3 Output SDR Generation

It has to be possible to acquire and store the SDR generated by the SP in order to analyze it or to use it as the input for the SW based TP.

In order to read the output SDR from the HTM the data packet consisting of $n = Col_{total}$

bits is transferred to the serial interface. The initial transmitted data is not important as every mini-column inserts it's state value $s_{sp\_i} \in \{0,1\}$ into the bit position $Col_{pos\_i}+1$ while forwarding the packet, where Col_{pos_i} is the mini-column's zero based address in the HTM matrix. The data packet outputted from the serial interface contains the output SDR of the SP process after passing all the mini-columns.

Output generation FSM, OUT SDR, consists of three states: *S1*, *W* and *S2*. In the first state, *S1*, the starting $n = Col_{pos\_i}$ bits are skipped. After skipping the required amount of bits the state *W* is entered, where the mini-column replaces the original data bit in the transferred message with it's state value s_{sp_i} and switches to the last state, *S2*. The last state just skips the remaining $n = Col_{total} - Col_{pos\_i} - 1$ data bits.

Simulating the output SDR generation was performed by first setting the mini-column to the active state and sending bit pattern filled with zeros to it's serial interface. The second half of the simulation set the mini-column to the inactive state and the bit pattern consisting of all ones was transferred to it.

Simulated mini-column was configured with the position index $Col_{pos\_i} = 1$. Simulation results proved that the column successfully replaced the bit $Col_{pos\_i}+1$ with it's state value (Fig. 4.20).

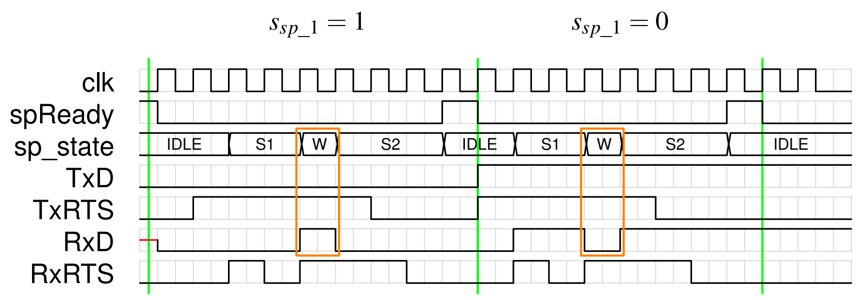


Figure 4.20: Spatial pooler output SDR generation. The bit position $Col_{pos\_i} = 1+1 = 2$ of the transferred pattern is replaced with the mini-columns state value s_{sp}.

4.4 Mini-Column Controller

Mini-column controller is the master controlling component of the HTM column's operations. It's main task is to decode the commands received over the serial interface and control the sub FSM-s. It is also responsible for multiplexing the shared resources in between different HW components (Figure 4.21).

Mini-column controller is connected to the adjacent controllers using the serial interface, forming the HTM matrix. The matrix can be scaled by adding or removing mini-column instances to or from the communication chain and can only be done on design time. The maximum amount of possible mini-columns depends on the actual FPGA used.

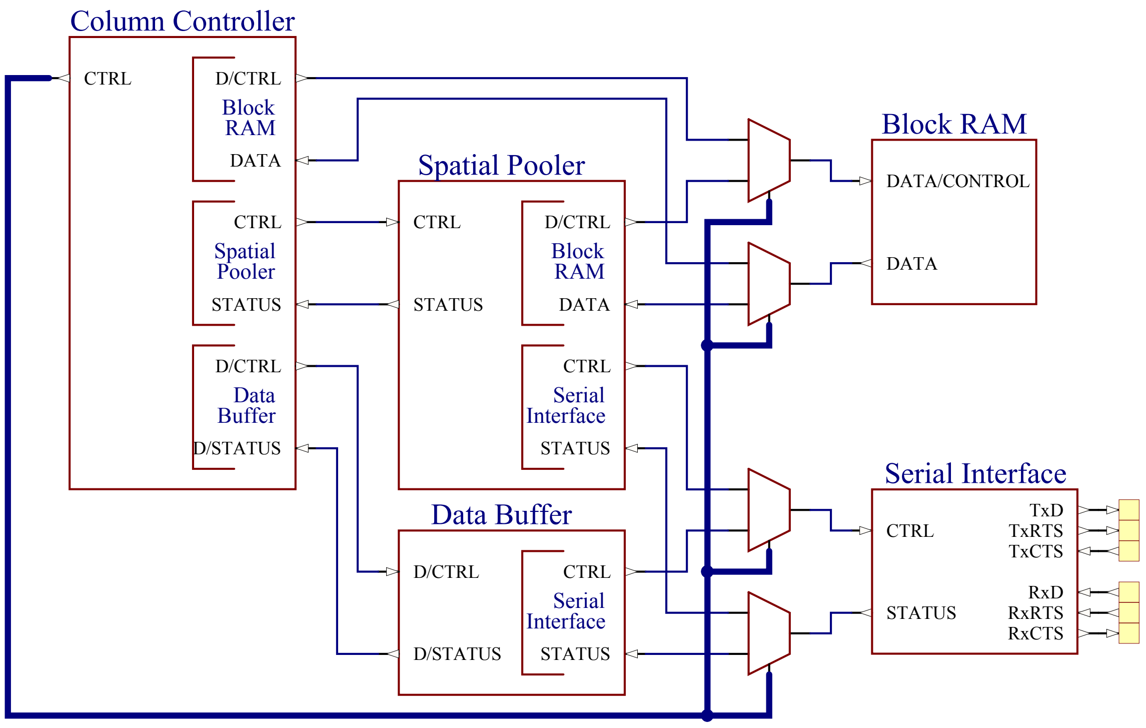


Figure 4.21: Mini-Column Controller's responsibility is to control the sub HW components and multiplex the shared resources. It is connected to the adjacent mini-columns using the serial interface, forming the HTM matrix.

4.4.1 Command Decoder

In order to control the operations of the mini-column a command protocol on top of the physical serial line is used. As the initial interest is to evaluate the feasibility of the HTM running on FPGA platform and there are no harsh environmental requirements the selected protocol structure is extremely simple, consisting of a command frame followed by the required amount of data frames. Error detection and correction schemes can be added if needed.

The bit with of the command and data bytes depends on the design time configuration of the data buffer component (Section 4.2) and has to be selected based on the minimal data type requirement for the mini-column proper operations. Selection has to be made very carefully, setting the base data type unnecessary wide yields to waste of resources, restricting it too much, on the other hand, might yield to the poor HTMs performance.

The data with is set to 8 bits in the current work, as considered to be adequate for all the required variables.

Command frame of the communication protocol is divided into the command group selector and the command selector. The command groups where introduced in order to simplify the decoder design. The command group is formed of the three starting bits of the command frame and selecting the proper group therefore needs a three bit decoder, while the rest of the bits within the group use one-hot encoding. The set of defined commands is presented in the Table 4.4.

Table 4.4: Mini-column commands. First three bits define the command group. The commands within the group are one-hot encoded in order to simplify the command decoder hardware.

Group	Command	Description
000	00001	SP feedforward input
	00010	SP overlap communication and winning cell selection
	00100	SP output SDR communication
	01000	SP perform learning
	10000	–
001	00001	Write SP permanence values to the RAM
	00010	Read SP permanence values from the RAM
	00100	–
	01000	–
	10000	–

FSM of the mini-column's controller consists of four states: *START*, *CWT*, *CDEC* and *ERR* (Figure 4.22).

START state sets the data buffer to the forward mode, frame count to one and connects the serial interface to it. This state is left at next clock cycle and state *CWT* is entered.

The *CWT* state is used for waiting the command to be transferred. Availability of the command is signalled by the data buffer component using it's ready signal. Next state, *CDEC*, is entered as soon as a new command has been received.

The *CDEC* state decodes the command and transfers the control to a required sub FSM, which is responsible of setting the multiplexers and initializing other necessary components for it's intended actions.

The *ERR* state was added for the debug purposes. The functionality of this state can be defined if the error detection and correction algorithms are needed.

Simulation of the command decoder was performed in conjunction with the actual command execution.

4.4.2 Feedforward Input And Output SDR Generation

Reception of the feedforward input and generation of the output SDR are performed by the SP sub component. The task for the mini-column's controller is to just connect the

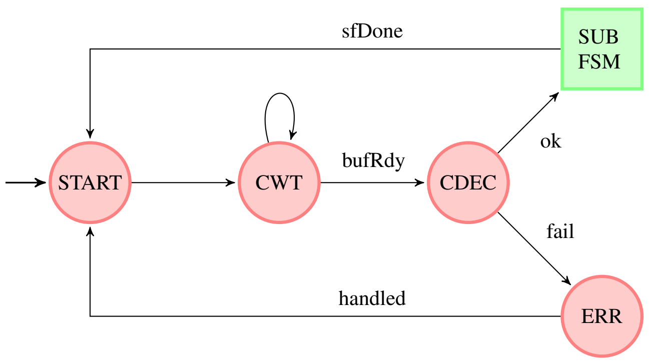


Figure 4.22: Mini-Column's Command Decoder uses the data buffer component for the command reception. *START* state sets up the data buffer for reception of 1 frame, followed by the *CWT* state. The *CWT* state just waits for the reception completion signal from the data buffer and proceeds to the *CDEC* state for command decoding and SUB FSM execution.

serial interface instance and the block RAM to the SP and request it to start the operation. Therefore the controllers feedforward input and output SDR generation sub FSM consists of only one single state: *SPBSY* (Fig. 4.23). This state just waits for the SP completion.

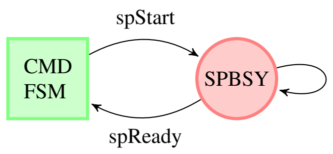


Figure 4.23: Processing of the feedforward input and outputting the SP SDR are the tasks of spatial pooler component, therefore the corresponding mini-Column's FSM consists of only one single state, *SPBSY* – waiting while the SP is busy.

The feedforward input processing was tested with four mini-column controllers, forming the 4 element HTM matrix. The length of the input pattern was initialized to 15 bits. All the mini-columns successfully accepted the feedforward input and returned to *CWT*, *command wait* state (Fig. 4.24). The total delay between completing the feedforward input processing for the first and last mini-columns was four clock cycles, equaling the number of elements in the HTM matrix.

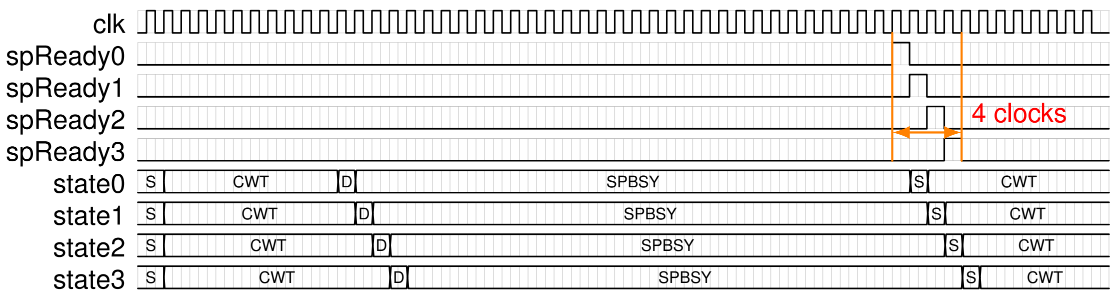


Figure 4.24: Feedforward input timing for HTM matrix consisting of four mini-columns. There is exactly one clock cycle delay for consecutive columns to complete the input processing, totaling the number of elements in the HTM matrix.

Generation of the output SDR was tested using the same size HTM matrix as for the feedforward input simulation. All the mini-column controllers wrote their state value into the transmitting data frame and returned to the *command wait* state (Fig. 4.25).

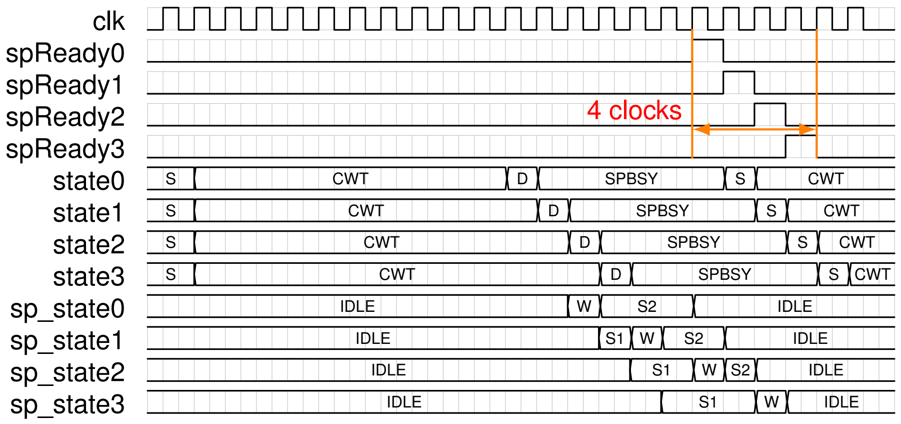


Figure 4.25: SP output SDR Generation. All the mini-column controllers successfully received and decoded the command. The SP FSMs wrote the state value into the frame bit position corresponding to their index in the communication chain. The total delay between the first and the last mini-column to complete the operation equals to the total number of mini-columns in the HTM matrix.

4.4.3 Mini-Column Activation

Activation of the mini-column is based on the Hebbian learning, only the columns with the highest overlap count should be activated.

Mini-column's activation FSM consists of two states, *SPOLP* and *SPWIN* (Fig. 4.26).

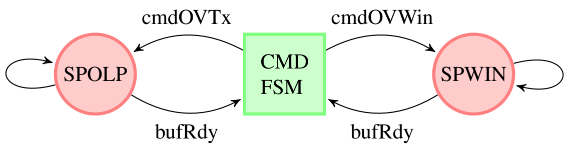


Figure 4.26: Mini-Column's activation FSM consists of two states, *SPOLP* for determining the set of highest overlap counts and *SPWIN* for activating the mini-columns. Both of these states make use of the features provided by the data buffer component.

In order to activate the required amount of mini-columns with the highest overlap counts the set of highest overlaps has to be determined first. This operation is performed in the *SPOLP* state, which consists of transmitting the data packet consisting of $n = Col_{total}$ frames initialized with zeros to the *HTM* serial interface. Every mini-column then tries to replace the original frame with it's overlap count by setting the data buffer to the *arbitration* mode. All the mini-columns shift their overlap count value to the left by one bit position and transmit zero as the last bit, which will be used as the capture bit in the next phase.

The data packet exiting the HTM matrix serial interface consists of the required amount of n highest overlap counts o appended with the initially zero capture bit c (Fig. 4.27).

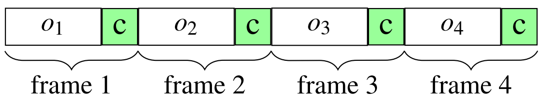


Figure 4.27: The data packet exiting the highest overlap counts selection consist of frames containing the overlap values with the special purpose appended capture bit.

The same packet is re-transmitted to the HTM matrix in order to determine the contributing mini-columns and activate them. This operation is performed in the *SPWIN* state which sets the data buffer to the *capture* mode. The data buffer interfaces use the last bit of a frame for the capture operation.

The mini-column's activation process uses the components which have been individually simulated. Therefore the activation process simulation was performed during the evaluation of the overall operations of the SP.

4.4.4 SP Permanence Values Read And Write

In order to initialize the HTM matrix it has to be possible to write the initial permanence values to the mini-columns block RAM. In addition to the possibility of writing it has to be possible to read the memory contents in order to store the operating state of the HTM for the later use.

Reading and writing of the permanence values is performed by transmitting data packet consisting of $n = Col_{total} * Col_{conn}$ data frames to the serial interface. The data packet is initialized with the permanence values for the write operation and zeros for the read operation.

Both, reading and writing operation start with the mini-columns skipping the first $n = Col_{pos\_i} * Col_{conn}$ data frames in the *SPRP1* and *SPWP1* states respectively. The frame skipping state is followed by the actual reading or writing of the Col_{conn} amount of frames in the *SPRP* or *SPRP* state. Permanence value reading and writing is completed by skipping the rest of the packet in the *SPRP2* or *SPWP2* state (Figs. 4.28 and 4.29).

4.5 HTM Controller

In order to control the behavior of the HTM and feed it with the input data the HTM hardware component has to be interfaced to the processor peripheral bus.

The HTM controller requires three FIFOs to be connected to it externally, one for transmitting and the other one for receiving the data to/from it. The third FIFO is used internally by the HTM controller in order to circulate the data packet if necessary (Figure 4.30).

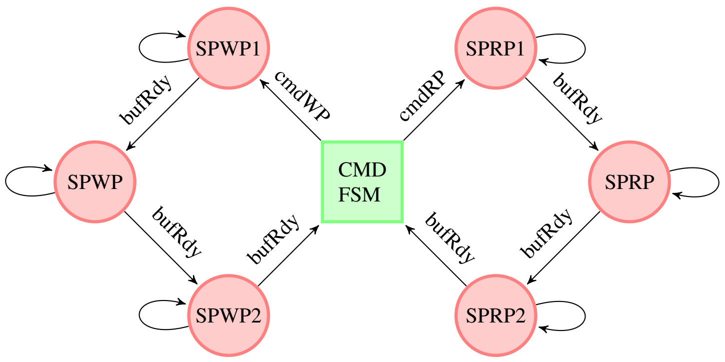


Figure 4.28: Mini-Column's SP Permanence Value R/W FSM. States *SPRP1* and *SPWP1* are used for skipping the beginning frames of the packet. Actual memory reading or writing is performed in the states *SPRP* or *PRWP*. The rest of the packet is skipped in the state *SPRP2* or SPWP2.

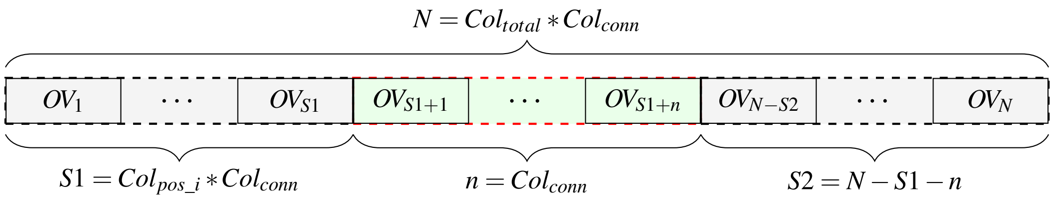


Figure 4.29: Mini-Column overlap value R/W data packet consists of all the overlap values for all the mini-columns concatenated into a single packet. Every mini-column reads or writes the values which belong to it, marked as green, and skip the rest. Skipped frames are colored gray.

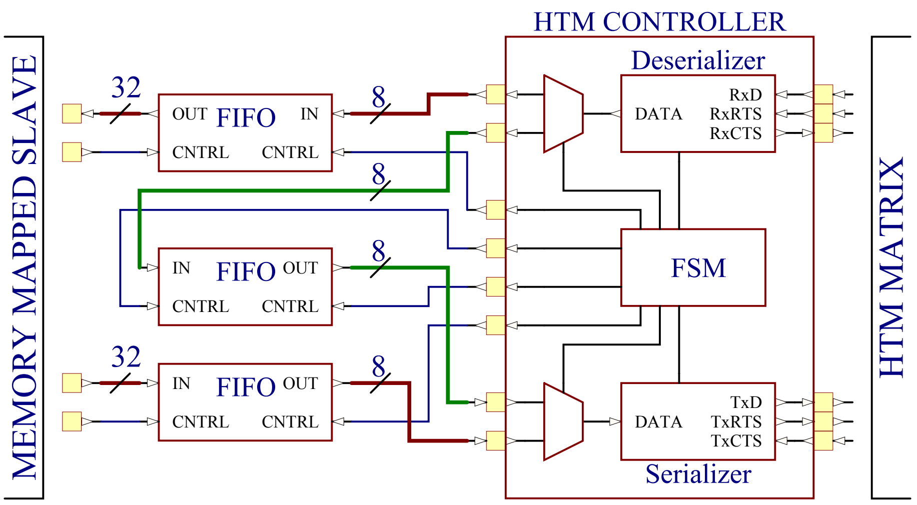


Figure 4.30: HTM Controller requires three external FIFOs, two for data transfer between the host processor and one additional FIFO for circulating the necessary data packets in the HTM matrix. Data buses for the host connection are drawn red while the internal circulating bus is green.

The external FIFOs provided to the HTM controller have to be of type First Word Fall Through (FWFT). FWFT FIFO simplifies the design of the HTM controller as read and write signals transfer to the read and write *acknowledge* signals instead, therefore no explicit read phase is necessary.

The capacity of the internal FIFO, used for data circulation, has to be chosen high enough not to generate the dead lock. The minimum capacity F_{min} of the FIFO is set by the maximum payload length P_{max} needed to be circulated. The entire packet has to fit into the serial buffers of the HTM mini-columns, each capable of storing 2 data bits, de-serializer and the circulation FIFO itself (Eq. (4.8)).

$$F_{min} \geq P_{max} - \left\lfloor \frac{2 * Col_{total}}{8} \right\rfloor - 1 \tag{4.8}$$

The data packet sent to the HTM controller should start with four control bytes, a 32 bit header, containing the information about the following payload data (Fig. 4.31). The control header includes alignment field *ALIGN* in order to allow the HTM matrix easy interfacing to the 32 bit host bus systems. Alignment field can be used if the total amount of bytes in the payload data is not aligned in the 4 byte boundaries – HTM controller automatically skips the required amount of bytes and does not transfer those to the HTM matrix serial bus. The same holds for the output packet generated by the HTM matrix, additional bytes are used to meet the 32 bit boundaries, if necessary. Description of the control header bits is presented in the Table 4.5.

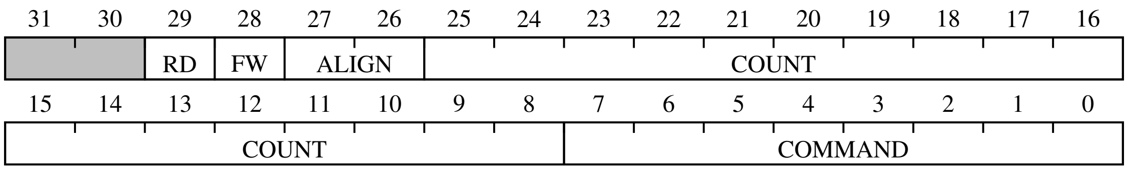


Figure 4.31: HTM controller packet header. It consists of the total count of the following payload data, optional alignment field, control bits describing the needed actions and the actual command byte.

4.6 HTM IP

In order to test the HTM SP in the real hardware two synthesizers from two bigger FPGA manufacturers have been used: Quartus from the Altera and Vivado from the Xilinx.

4.6.1 Register Map

Both of the IPs, Avalon IP for the Altera devices and Advanced eXtensible Interface 4 (AXI4) IP for the Xilinx SoC implement the same register map in order to enable the

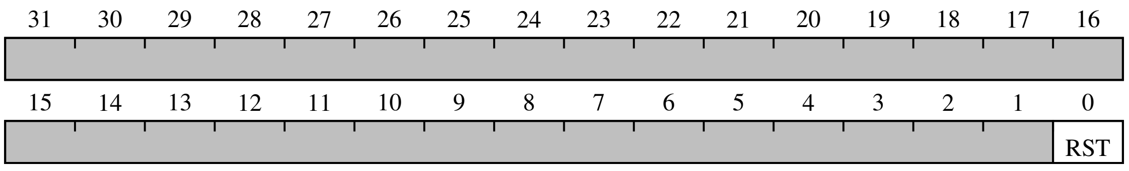
Table 4.5: HTM Controller Packet Header.

Bit	Description
31–30	Reserved. Software should write 0 to this field.
29	Read control. If this bit is set the response of the HTM matrix to the payload data is transmitted to the host RX FIFO.
28	Forward control. If this bit is set the payload data is first transmitted to the HTM matrix and the output packet is re-transmitted, effectively circulating the payload data.
27–26	Alignment control. Maximum three alignment bytes can be set. HTM controller automatically skips the alignment bytes from the received payload data and appends the same amount of bytes to the response packet transmitted to the host controller (if bit 29 is set).
25–8	Number of payload bytes.
7–0	Command byte. This byte is the actual HTM matrix command (Table 4.4).

usage of the same software drivers with minimal Board Support Package (BSP) related changes.

Register map for controlling the HTM and for data transfer is documented in Tables 4.6 to 4.13. Both of the IPs implement the 32 bit wide bus interface and make use of the alignment bytes for a transferred data packet.

Table 4.6: HTM IP Control Register 0x00



Bit	RW	Description
31–1	R	Reserved. Software should write 0 to this field.
0	RW	Reset the entire HTM matrix. This bit does not reset the previously written permanence values

Table 4.7: HTM IP Total Mini-Columns Register 0x01

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							Total Columns								
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
							Total Columns								

Bit	**RW**	**Description**
31–0	R	Number of mini-columns in the HTM matrix.

Table 4.8: HTM IP total Number of Feedforward Inputs Register 0x02

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							Feedforward input length								
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
							Feedforward input length								

Bit	**RW**	**Description**
31–0	R	Total number of bits in the Feedforward input.

Table 4.9: HTM IP Feedforward Input Connections Register 0x03

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							SP FF Connections								
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
							SP FF Connections								

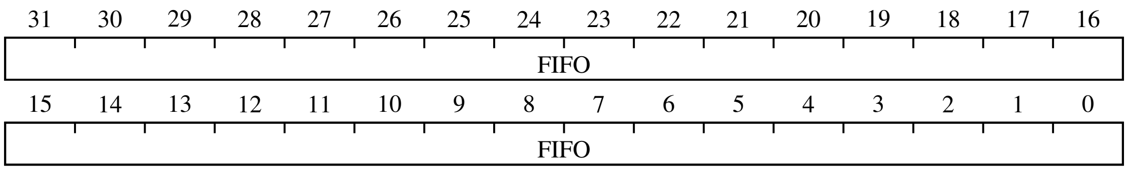
Bit	**RW**	**Description**
31–0	R	Total number of feedforward connections per mini-column.

Table 4.10: HTM IP SP Output Active mini-Columns Count Register 0x04

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							SP Output Count								
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
							SP Output Count								

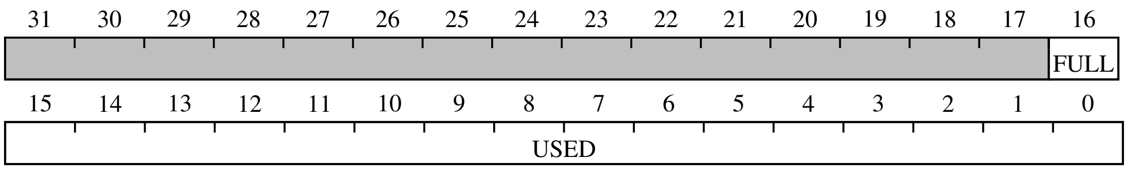
Bit	**RW**	**Description**
31–0	R	Total number of activated mini-columns in the SP process.

Table 4.11: HTM IP FIFO Register 0x10



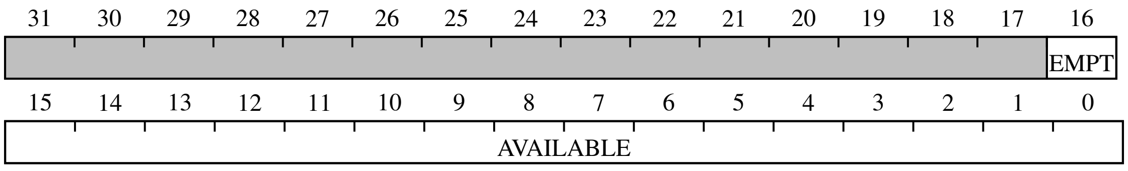
Bit	RW	Description
31–0	W R	Write to the TX FIFO. Read from the RX FIFO.

Table 4.12: HTM IP TX FIFO Status Register 0x11



Bit	RW	Description
31–15	R	Reserved. Software should write 0 to this field.
16	R	TX FIFO full flag.
15–0	R	TX FIFO used count. Number of 32 bit words in the TX FIFO.

Table 4.13: HTM IP RX FIFO Status Register 0x12



Bit	RW	Description
31–15	R	Reserved. Software should write 0 to this field.
16	R	RX FIFO empty flag.
15–0	R	RX FIFO available count. Number of 32 bit words available in the RX FIFO.

4.6.2 Avalon Bus HTM IP

Avalon bus HTM IP was synthesized targeting the Altera's MAX10M50 FPGA. The target device provides:

- 50K logic elements
- 1638 Kbit of on chip memory, configurable to 9 bit width memory blocks suitable for the chosen SP permanence value fields.

Synthesized HTM matrix consisted of 128 mini-columns, each accepting 128 bit feedforward inputs and connecting to 64 input bits.

The total number of logic elements occupied by the selected matrix was 42768 logic elements. Distribution of the logic usage per HTM SP elements is provided in Table 4.14.

Table 4.14: Logic usage per HTM Avalon IP mini-column

HTM Building Block	Logic Elements Usage
Column Controller	93
SP Controller	121
LFSR	9
Bit Collector	96
Serial Interface	12
Total	331

4.6.3 AXI4 Bus HTM IP

AXI4 IP was synthesized targeting the Xilinx ZYNQ-7020 SoC.

The synthesized HTM consisted of 128 mini-columns, each accepting 128 bit feedforward inputs and connecting to 64 input bits.

The total number of logic occupied by the matrix was: 32710 LUTs and 22324 Flip-Flops. The logic usage per HTM SP elements is provided in Table 4.15.

Table 4.15: Logic usage per HTM AXI4 IP mini-column

HTM Building Block	LUTs	Flip-Flops
Column Controller	21	30
SP Controller	85	64
LFSR	3	8
Bit Collector	134	67
Serial Interface	9	4
Total	252	173

5 PERFORMANCE ESTIMATION

The performance of the SP process can be estimated using the formulas derived from it's intended behavior.

5.1 Feedforward Input

Communicating the input SDR to the HTM takes 2 clock cycles per input bit (Section 4.3.1). Therefore it takes $2Col_{total}$ clock cycles for the first bit to travel all the mini-columns. Additional $2m$ clock cycles are required for the last bit of the input pattern to exit the last mini-column, where m stands for the length of the input pattern. Total clock cycles for the input processing can be calculated using Eq. (5.1).

$$CLK_{in} = 2(Col_{total} + m) \tag{5.1}$$

5.2 Mini-columns Activation

The next step after applying the input is the overlap calculation and mini-columns activation (Section 4.4.3). The total cycle count for this step depends on the required output sparsity which sets the amount of overlap frames to be transmitted and consists of the three phases:

1. Col_{total} cycles are required for the first bit of the highest overlap count to exit the HTM.
2. It takes $L_{ov}C_{win}$ cycles for the last bit of all the overlap frames to re-enter the HTM, where L_{ov} stands for the length of a single overlap frame and C_{win} is the total number of required overlaps, i.e. the required density of the output SDR.
3. Additional Col_{total} cycles are required for the last bit to exit the HTM.

The total cycle count for the mini-columns activation can be calculated using Eq. (5.2):

$$CLK_{ov} = 2Col_{total} + L_{ov}C_{win} \tag{5.2}$$

5.3 Learning

The SP learning phase consists of the n read-modify-write cycles (Section 4.3.2), where n stands for the total amount of mini-column's feedforward connections. Therefore the total amount of required clock cycles for the learning process equals the number of permanence values multiplied by three (Eq. (5.3)).

$$CLK_{learn} = 3n \tag{5.3}$$

5.4 Output SDR Generation

Generation of the output SDR of the SP depends of the size of the HTM matrix. It does not require any wait cycles, all the mini-columns modify the bit position corresponding to their location while forwarding the message (Section 4.3.3). Therefore the total output generation cycles equal to Col_{total} cycles for the first bit to exit the HTM and additional Col_{total} cycles for the last bit (Eq. (5.4)).

$$CLK_{out} = 2Col_{total} \tag{5.4}$$

5.5 Total SP Processing

The total cycles needed for the SP process equals to the sum of the processing steps (Eqs. (5.1) to (5.4)). The execution time t can be calculated by dividing the total cycles by the selected clock frequency f of the hardware (Eq. (5.5)).

$$t = \frac{CLK_{in} + CLK_{ov} + CLK_{learn} + CLK_{out}}{f} \tag{5.5}$$

6 EVALUATION

Evaluation of the SP hardware realization was performed by applying the same set of the input data to the hardware and software implementations. The input data was applied for 10000 times in order to monitor the learning capabilities. The correctness of the functionality and the performance of the SP where both verified.

6.1 Supporting Software Description

In order to verify the functionality of the SP process it had to be supplied with the input data and the output SDRs had to be compared against the known good results. This verification process required some support software development.

Software development was divided into two main steps, development of the SP software process and development of the hardware driver.

6.1.1 Spatial Pooler SW Implementation

The pure software SP process was implemented using the C++ programming language. In addition to the implementation of the SP itself, input data encoder had to be developed.

The PC software was developed to have the same configuration options as the implementation running in the hardware and all the classes where developed as templates, which can be instantiated using the variable data type. The data type has to match the selected data with of the hardware implementation.

6.1.2 Example Usage

This section provides an example use-case of the SP software.

In order to run the software the SP has to be initialized first. Initialization involves defining the number of mini-columns, length of the input, output sparsity and other parameters (Listing 6.1).

```
//SP parameters
/**< total number of mini−columns in HTM */
unsigned columns= 128;
/**< overlap count (sparsity) */
unsigned overlap= 4;
/**< length of the input SDR */
```

```
unsigned inputLen= 128;
/**< number of bits connected (each cell connects to 'connectCount'
   number of inputs) */
unsigned connectCount= 64;
/**< threshold value for the permanences */
unsigned TH= 128;
/**< threshold delta for initial randomization procedure */
unsigned THDelta= 5;
/**< specify if the mini-columns perform the learning operation */
bool learning= true;
/**< specify if the boost operation is enabled */
bool boosting= false;

//instantiate the SP
HTM::POOLER::SpatialPooler<uint8_t> sp(columns, inputLen, inputLen,
   overlap, true, TH);
//and randomize the permanence values
sp.randomize(connectCount, TH-THDelta, TH+THDelta);
```

Listing 6.1: SP Initialization

The next step after successfully setting up the SP is to prepare some input data to apply to it. Data encoder is used for that purpose. For easy access the input data is inserted into the C++ vector (Listing 6.2).

```
//encoder parameters
/**< minimum value for the SDR encoder */
unsigned encMin= 0;
/**< maximum value for thr SDR encoder */
unsigned encMax= 100;
/**< number of bits in the SDR encoder (bin size) */
unsigned encBin= 4;
/**< length of the input SDR */
unsigned inputLen= 128;

//instantiate the encoder
HTM::ENCODER::ScalarEncoder<uint8_t> encoder(encMin,encMax,inputLen,
   encBin);
//encode the input data
vector<HTM::MATRIX::bitMatrix> SDR;
for (int data= 0; data < 100; data+= 10)
  SDR.push_back(encoder.encode(data));
```

Listing 6.2: Input Data Encoder Initialization

Testing of the SP is performed by applying the input data to it. In order to better check the learning capabilities of the SP the input data can be applied in the loop (Listing 6.3).

```
//apply the input data
for (unsigned i= 0; i<10000; i++)
  for (auto const& s:SDR)
    sp.input(s, learning, boosting);
//and print the state of the mini-columns
cout << "Mini-column connections: " << endl << sp.printConnected();
```

Listing 6.3: SP Testing

6.1.3 Spatial Pooler Driver

The SP driver software is intended for controlling and testing the SP hardware implementation.

The basic operations of the driver are the same as for the software version, excepts that the HTM configuration has to be done during the HDL synthesis. However, the configuration parameters can be read and verified using the control registers (Section 4.6.1).

As there is no configuration involved in the runtime, the usage of the SP hardware implementation is limited to just applying the input data to it. Therefore the example use-case consisting of a single loop is not presented here.

The only housekeeping task the driver has to do is to monitor the level of the TX FIFO in order not to overflow it.

6.2 Functional Evaluation

The input data for the evaluation was encoded using the scalar encoder with the minimum and maximum values 0 and 100, respectively (Listing 6.2). The bin size for the encoder output was set to 4 and the total length of the encoding to 128, i.e. every encoded output has 4 out of the 128 bits set.

The input data set used for the evaluation consisted of values $0, 10, \ldots, 90$ (Fig. 6.1). The data set was applied for 10000 times, giving the SP enough inputs to perform the learning and train the feedforward connections. The final SP connections where identical for the software and hardware processes, proving the HTM SP hardware model functions according to the design requirements.

The graphical representation of the evaluation output is presented in the Fig. 6.2. The presented matrix visualizes the state of the mini-columns feedforward connections. If the permanence value of a connections is over the threshold value, i.e. the connection has been established, the corresponding position is marked as red rectangle in the output figure. Green rectangles indicate the inactive connections.

Every row presents the results for a single mini-column. The mini-columns which where not contributing to the generation of the output SDRs, i.e. the ones which constantly lost the overlap count comparison where skipped from the figure.

Every input presented to the HTM has 4 mini-columns matching it's SDR encoding, i.e. every input has 4 mini-columns which have trained their feedforward connections to match it. The amount of connected mini-columns matches the initialization of the SP – the SP was initialized to activate 4 mini-columns for every input pattern (Listing 6.1).

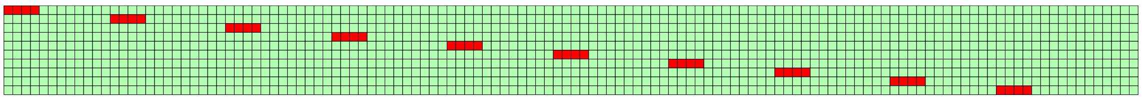


Figure 6.1: Output of the encoder has 4 out of the total 128 bits set, as per encoder initialization. Minimum and maximum values for the encoder where set to 0 and 100, respectively. Encodings of the inputs 0, 10,. . . ,90 are shown as the the matrix rows where the red boxes indicate the *ON* bits.

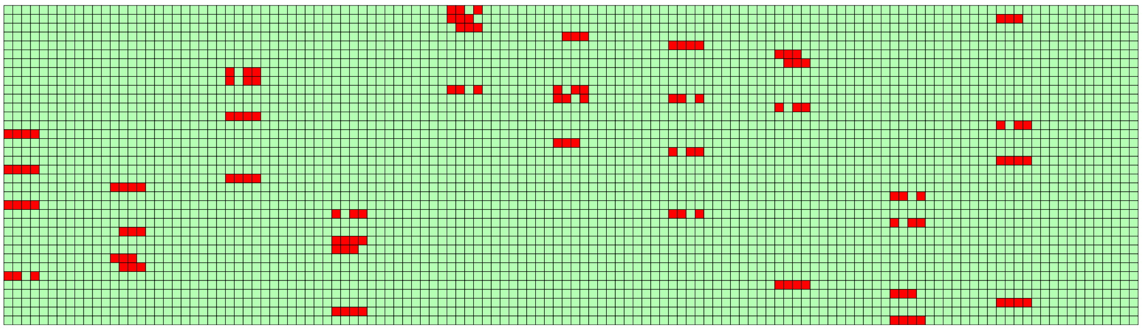


Figure 6.2: The state of the SP feedforward connections after applying the set of inputs (Fig. 6.2) for 10000 times. Every row of the matrix shows the connections for a mini-column that has been participating to the learning process, i.e. the mini-columns which where not activated due to the results of the overlap count calculations are skipped from the figure. Every possible input value has matching 4 mini-columns connected to it.

6.3 Performance Evaluation

The performance of the SP was evaluated using the hardware timer providing the operating system tick. The granularity of the system tick was set to 1ms.

Performance evaluation consisted of registering the initial system tick count, applying the set of input patters for 10000 times and acquiring the system timer value after the execution of all the inputs. The difference of these two values was divided by the total amount of the input SDRs applied in order to get the execution time per unit.

The same process was conducted for the SW based algorithm in order to get some data for the comparison.

As can be seen from the processing times, 128 mini-column HTM is capable to outperform the conventional PC with quite a bit of a margin (Table 6.1). Processing times on the main HW target, ZYNC-7020 SoC and for the smaller MAX10M50 FPGA where the same as the driving clock was the same in both cased: 100MHz. The clock frequency can be reduced for the low power designs to meet the actual required cycle times.

Table 6.1: 128 mini-column HTM SP processing time for a 128 bit input SDR

SP	i7 PC	ZYNQ	MAX10
t(ms)	0.11	0.02	0.02

The theoretical SP processing time calculated using the Eq. (5.5) for the 128 mini-column

HTM with 128 bit feedforward input length equals to $t = 0.012$ms, which is shorter than the results acquired from the experiment. This phenomena can be explained by the performance of the processor. The clock of the processor was also set to 100MHz and it was not enough to feed the data FIFO fast enough.

The processor performance penalty was also proven by the fact that the FIFO level was zero after applying all the input SDRs. Therefore it would be beneficial to use the Direct Memory Access (DMA) controller for the data transfers in order to achieve the optimum performance both in performance and power wise.

7 CONCLUSIONS

Based on the execution results of the HTM SP on the SoC and the theoretical performance estimations it can be concluded that the leading ideas of this theses proved to be feasible.

Implementation of the LFSR utilizes small amount of the hardware resources compared to the physical memory it helps to conserve. The total amount of the memory and logic used by the implementation is definitely one of the key factors in narrowing down the possible target platforms. Less resources required allow smaller devices to be used and less power consumption.

The serial communication scheme proposed and described in this theses proved to be a feasible approach too. There are several reasons for this. First of all, the address decoder LFSRs need to be advanced for every additional input bit, therefore there would be no benefit in transferring data using the faster parallel buses. Another consideration is the access of the mini-column's block RAM – inferring single port memories requires the access to be divided in the time domain, supporting the *serial* nature of the implementation.

And most importantly, the ultimate performance was not the primary target of this work. The key topic addressed by this thesis is more related to the balance in between the performance and the usage of the resources, i.e. the implementation has to show up improved execution time over the pure software solution using general purpose off-the-shelf and power efficient SoC hardware.

As the conclusion, this thesis provides another approach for the implementation of the HTM SP algorithm targeting the SoC devices with the FPGA resources available. Although the entire HTM algorithm has not been fully implemented the proposals and implementation details provided in this thesis can form a good platform for the following researches in the future.

7.1 Future Works

The future works for the the implementation of the entire HTM functionality:

- Evaluate the possibility to use the LFSR with configurable feedback and length for the TP connection segments addressing. TP can possibly add new segments in order to learn different sequences, i.e. the connection addresses have to have some room for runtime adjustments.

- The current implementation of the SP does not support the inhibition radius smaller than the entire HTM matrix. Implementation of the local inhibition might benefit from the Network On Chip (NOC) architectures with light-weight routers.
- Implement the boosting functionality for the mini-columns. The selected boosting algorithm should correlate to the implementation of Numenta's algorithm but it might benefit from some simplifications for constraining the usage of the FPGA logic.

In addition to the functional additions the proposed architecture could benefit from some additional optimization. Every saving in the resources scales up quite fast – the HTM matrix consists of large number of mini-columns:

- Although the best effort was used in generation of the HDL code, the synthesizer output has to be closely analyzed.
- It might be possible to share and schedule more resources in time domain.
- Current implementation of the mini-columns serial interface does not support generation of the output data without the input. Some FSM states and related logic can be reduced if the mini-column could independently start or continue the communication.

Bibliography

[1] D. Graupe, *Deep Learning Neural Networks. Design and Case Studies.* World Scientific Publishing Co. Pte. Ltd., 2016.

[2] A. Dijksterhuis and H. Aarts, "Goals, Attention, and (Un)Consciousness ," *Annual Review of Psychology*, vol. 61, pp. 467–490, 2010.

[3] Numenta, "Hierarchical Temporal Memory including HTM Cortical Learning Algorithms. White paper VERSION 0.2.1," http://numenta.org/resources/HTM_CorticalLearningAlgorithms.pdf, September 2011.

[4] J. Hawkins and S. Blakeslee, *On Intelligence.* Times Books, 2004.

[5] D. George, "How the Brain Might Work: A Hierarchical and Temporal Model for Learning and Recognition," Ph.D. dissertation, Stanford, CA, USA, 2008, aAI3313576.

[6] A. Lavin and S. Ahmad, "Evaluating Real-time Anomaly Detection Algorithms - the Numenta Anomaly Benchmark," in *14th International Conference on Machine Learning and Applications (IEEE ICMLA)*, 2015.

[7] Numenta, "Applications of Hierarchical Temporal Memory (HTM)," 2014. [Online]. Available: http://numenta.com/papers-videos-and-more/resources/applications-of-hierarchical-temporal-memory/

[8] ——, "Numenta Platform for Intelligent Computing," 2016. [Online]. Available: http://numenta.org/

[9] J. Hawkins and S. Ahmad, "Why Neurons Have Thousands of Synapses, a Theory of Sequence Memory in Neocortex," *Frontiers in Neural Circuits*, vol. 10, no. 23, 2015.

[10] S. Ahmad and J. Hafkins, "How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites," 2016. [Online]. Available: arXiv:1601.00720 [q-bio.NC]

[11] S. Purdy, "Encoding Data for HTM Systems," 2016. [Online]. Available: arXiv:1602.05925 [cs.NE]

[12] S. Ahmad and J. Hawkins, "Properties of Sparse Distributed Representations and their Application to Hierarchical Temporal Memory," 2015. [Online]. Available: arXiv:1503.07469 [q-bio.NC]

[13] Y. Cui, S. Ahmad, and J. Hawkins, “The HTM Spatial Pooler: a neocortical algorithm for online sparse distributed coding,” *bioRxiv*, 2016. [Online]. Available: http://biorxiv.org/content/early/2016/11/02/085035

[14] J. Mnatzaganian, E. Fokoué, and D. Kudithipudi, “A Mathematical Formalization of Hierarchical Temporal Memory’s Spatial Pooler,” 2016. [Online]. Available: arXiv:1601.06116v3 [stat.ML]

[15] Numenta, “Overview of the Temporal Pooler,” 2015. [Online]. Available: https://github.com/numenta/htmresearch/wiki/Overview-of-the-Temporal-Pooler

[16] X. ZHOU and Y. LUO, “Implementation of Hierarchical Temporal Memory on a Many-core Architecture,” Master’s thesis, Halmstad University, School of Information Science, Computer and Electrical Engineering (IDE), 2013.

[17] L. Streat, D. Kudithipudi, and K. Gomez, “Non-volatile Hierarchical Temporal Memory: Hardware for Spatial Pooling,” 2016. [Online]. Available: http://arxiv.org/abs/1611.02792

[18] M. Deshpande, “FPGA Implementation and Acceleration of Building blocks for Biologically Inspired Computational Models,” Master’s thesis, Portland State University, 2011, paper 160. [Online]. Available: http://pdxscholar.library.pdx.edu/open_access_etds/160

[19] A. M. Zyarah, “Design and Analysis of a Reconfigurable Hierarchical Temporal Memory Architecture,” thesis, Rochester Institute of Technology, 6 2015. [Online]. Available: http://scholarworks.rit.edu/theses/8703

[20] P. Vyas and M. Zaveri, “Verilog implementation of a node of hierarchical temporal memory,” *Asian Journal of Computer Science & Information Technology*, no. 3.7, pp. 103–108, 2013. [Online]. Available: http://www.innovativejournal.in/index.php/ajcsit/article/viewFile/370/355

[21] Xilinx, “All Programmable SoC with Hardware and Software Programmability.” [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[22] Altera, “MAX Series FPGAs.” [Online]. Available: https://www.altera.com/products/fpga/max-series.html

[23] B. Zhang and S. N. Srihari, “Properties of binary vector dissimilarity measures,” in *Proc. JCIS Int’l Conf. Computer Vision, Pattern Recognition, and Image Processing*, vol. 1, 2003.

Appendices

A DDECS 2017

Student paper "Hierarchical Temporal Memory implementation on FPGA using LFSR based spatial pooler address space generator" published and presented in DDECS2017 conference, Apr. 2017 Dresden, Germany.

Hierarchical Temporal Memory implementation on FPGA using LFSR based spatial pooler address space generator

Madis Kerner[1] and Kalle Tammemäe[2]

Abstract— **Hierarchical temporal memory (HTM) is the model of the neocortex functionality, developed by Numenta, Inc. The level of implementation does cover only the subset of actual neocortex layers functionality, but, however, is sufficient to be useful in different domain areas e.g. for a novelty or anomaly detection. Numenta provides their implementation of the HTM for commercial or research purposes as a software solution. The purpose of this work is to investigate the feasibility of implementing the HTM algorithm partly or entirely on FPGA, providing the suitable building block for the resource limited cyber physical systems. The uniqueness of the provided solution is based on resource efficient Linear Feedback Shift Registers (LFSR) as connection address generators, as well as using a simple serial interface for inter-column communication.**

I. Introduction

There have been different Artificial Neural Networks (ANN) developed over time [1]. The HTM cortical learning algorithm, developed by Numenta, Inc. [2] is one of the newest among them. HTM, as opposed to other types of ANNs, tries to mimic the functionality of the mammalian neocortex layered structure and operation as closely as possible. Every operation or execution step of the HTM algorithm is paired, partially or in whole, with its mammalians brain counterpart. Its ability to classify the input patterns and make predictions based on the input pattern sequences has been successfully used in different applications [3] and has been proven to be successful in detecting anomalies in real-world time-series data [4].

HTM is comprised of up to thousands of interconnected neural columns, with tens of cells each. All the columns are connected to a subset of input bits and all the cells are connected to a subset of another cells. The operation of the HTM is divided into two main tasks, a spatial pooler (SP) and a temporal pooler (TP). The SP, operating with columns, input data and connections between them, classifies and learns the input patterns. Columns, which are connected to more active input bits, switch to the active state and TP process follows. The temporal pooler operates on cells and their interconnections. TP's task is to recognize and remember the sequences of the input data. If a cell with high amount of active connections to the set of active cells belongs to the active column, it's activation was predicted, i.e. the sequence has been learned in the past.

Establishing and strengthening the connections between input data and HTM columns during spatial pooling or between different HTM cells during the TP process is based on competitive Hebbian learning. The connections between often firing entities get stronger while the rest will be suppressed. Every input connection in the HTM implementation has a characteristic permanence value assigned to it. In case of the value being over the specified threshold, the connection is in active, enabled state – active connections in between input bit vector and HTM columns allow active input bits to contribute to the column activation during the SP phase. The total amount of active input bits connected to each of the columns is defined as an overlap count. The columns having a higher overlap count become active and can suppress the activity of their neighbors. The maximum distance between a active column and columns affected by its activity is defined as an inhibition radius.

The data format HTM operates on is Sparse Distributed Representation (SDR) [5]. Sparsity means that only few bits of the entire data set are active concurrently and distributed means that the semantic meaning of the data is distributed across the bits, allowing sub-sampling of the data and providing fault tolerance. Few missing or additional bits will not change the SDR's ability to correctly represent the encoded input value.

HTM functionality is described thoroughly in [2], [6].

II. Background

Numenta's implementation of the HTM algorithm is freely available for investigation and further development [7]. The algorithm itself is not computationally very intensive; the majority of the computations are based on simple integer type count-and-compare operations in contrast to classical neural networks, where learning requires evaluation of non-linear equations. But the column-based architecture of the neocortex calls for the parallel execution. Evaluating hundreds of connections requires executing many software loops, which can severely degrade the overall throughput of the algorithm on small resource limited devices. One possible solution would be to make use of multi-core architectures, which has been shown to be good solution for increased throughput [8]. But constructing hundred or more cells HTM would quickly exploit the available cores. Another approach would be to implement a custom hardware component with single cell functionality and use it as the building block for the HTM matrix construction [9], [10], [11]. A solution based on multi-core architectures can be useful when the power consumption of the system is not limited, but smaller cyber physical devices often run on batteries and do not

[1]Master student at Dept. of Computer Systems, Tallinn University of Technology, Estonia

[2]Associate Professor at Dept. of Computer Systems, Tallinn University of Technology, Estonia

host several CPUs. Therefore the task oriented FPGA based solution could be beneficial.

III. FPGA as a platform for HTM implementation

Due to the parallel and concurrent nature of the HTM execution algorithm the FPGA based solution can yield to increased throughput. However, there are few considerations. First of all, the huge amount of possible connections. This calls for the huge crossbar switch but having all the possible connections hardwired severely limits the scalability and does not support the expected flexibility of a neural network. In order to overcome this limitation, serial communication schemes can be used, i.e. neighboring columns connect via the serial line interface and the message passes from one column to another. Another consideration is the memory footprint of the column implementation.

A. *Target platform*

Some of the sequential tasks related to the HTM execution, like initialization, saving the state for later analyzes, loading the state or performing another kind of maintenance are more suitable for the conventional CPU resource. In addition, having a CPU resource available allows partitioning the computations between HW and SW. Therefore a suitable HW platform for the HTM realization can be a system on chip (SoC), with FPGA resources closely coupled to the CPU-based processing system. The SoC chosen as the target platform for this work is a Xilinx ZYNQ-7020. The chosen SoC provides:

- Dual-Core ARM Cortex-A9 processing system.
- 140 36Kbit dual port block RAMs. Each memory block can be split into two single port RAMs.
- 85K logic cells.

A high level block diagram of the HTM algorithm implemented on a ZYNQ based SoC is shown in Fig. 1.

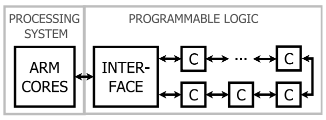


Fig. 1: HTM implementation based on Xilinx ZYNQ SoC. All the columns are connected in series. The processing system can provide computational support and can be used to analyze the state of the HTM.

The total number of available memory blocks in the SoC is a limited resource. Therefore all data types have to be minimal, but still capable of reflecting the needed functionality. E.g. the underlying data type for the permanence values of a connections has to be carefully chosen. The entire possible memory field has to be utilized, available memory blocks in HW can only be configured using the allowed set defined in the product's data sheet. Legal configurations for Xilinx ZYNQ include 2K of 9-bit words, for example, leaving one extra bit while using 8-bit permanence values. This extra bit gets used for saving the input activity during the SP execution (Fig. 2). Configuring the ZYNQ's 140 36Kbit dual port memory blocks as two single port blocks yields to 280 18Kbit memory units. Using a 9-bit memory word allows allocation of one 2K*9bit memory block to each column and build the HTM matrix consisting of $>$200 columns.

IV. Spatial pooler implementation

Spatial pooling is the first stage of the HTM operation. It consists of four main sub tasks: receiving the input data, calculating the overlap count for each of the columns, detecting the set of winning columns with the highest overlap and optional learning task. Every column is connected to approximately half of the input data bits on a semi-random basis. E.g. input data of 128 bits would require storing of 64 connection addresses in the memory of every column. But as the memory is a limited HW resource, it has to be conserved when possible and feasible. Therefore efficient method for re-generating the information about existing connections would be preferable. One suitable approach is to use a LFSR for this. LFSR can be efficiently implemented in hardware and it is capable of generating semi-random bit patterns for indicating connections between columns and upcoming input bits. Using the LFSR as a connection address decoder eliminates the need to store the connection data into the columns memory. In case of 64 possible connections every connection address has to be coded using 6 bits, 384 bits in total per column. When using LFSR, 9.6KB of memory can be freed up for HTM consisting of 200 columns.

A. *LFSR as a connection generator*

LFSRs can generate pseudo random bit patterns of different length, depending on the feedback polynomial and the length of the actual LFSR register [12]. As column connections have to be randomized, column LFSRs have to be initialized using a different seed value. Using a seed value based on the cell index effectively randomizes the connections. But it is important to notice that zero is the only value that can't be used as the initial seed, because LFSR would stay at constant zero state otherwise. Another important aspect while choosing the suitable LFSR length and polynomial is that the selected feedback polynomial has to be primitive. LFSR-s using the primitive polynomial generate the maximum length pseudo random sequence. Using primitive polynomial $P(x) = x^7+x^6+1$ generates 127 bit pseudo-random bit-stream, i.e. 127 unique internal LFSR states, which is the maximum number the 7-bit memory field can hold. Therefore every bit is guaranteed to be high for half of the possible states, effectively generating the pseudo-random bit stream with half of the bits set to high. This behavior can efficiently be used for randomizing half of the possible input bit connections to a column.

B. *Sending the input data*

Input data is sent to the SP as the bit vector. Every column advances it's LFSR while receiving the data, determining if the cell is connected to the bit just received or not. If the output of the LFSR is high, the cell is connected to the corresponding bit and vice versa if the LFSR output is zero.

The bit is saved as the most significant bit (MSB) into the column's corresponding SP permanence memory field and the memory address gets incremented in case the connection exists (Fig. 2).

Total amount of clock cycles for input data transfer consists of L_{in} cycles for transferring all input bits to the first column plus C cycles, the clock count for the last input bit to proceed up to the last column. Furthermore, processing every input bit takes two clock cycles per column (Eq. (1)).

$$CLK_{in} = (L_{in} + C) * 2 \tag{1}$$

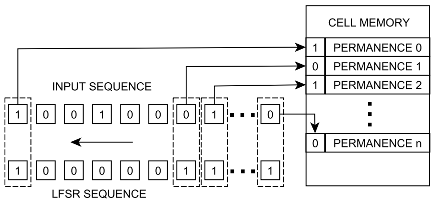


Fig. 2: Using LFSR as the connection address decoder. LFSR is advanced synchronously with receiving bits from the serial interface. Input bit arrival and LFSR advance direction are indicated by an arrow. In case the LFSR output is one, the corresponding input is stored to the column's memory as the highest bit of the permanence value. Legal memory configuration of Xilinx ZYNQ involves 9-bit memory word, leaving one bit for saving the input activity while having 8-bit permanence values.

C. Overlap counting

During the HTM's spatial pooling process every cell has to calculate its overlap count. Overlap counting is a simple comparison – if the permanence value is above the specified threshold and the corresponding input bit is active, the overlap count gets incremented. Threshold comparison can be done effectively by setting the value to exactly half of the maximum unsigned number the memory field can hold. If the permanence value is stored as an eight bit number, the maximum unsigned value it can accommodate is hex 0xFF and half range is 0x7F. Anything above the half range would set the MSB to high. Therefore the threshold comparison can be carried out by simply checking the MSB – connection is enabled if the MSB is set. Overlap counting can be combined with the receiving of the input bit pattern; while saving the input activity bit combined with it's corresponding permanence (Fig. 2), the overlap counter increments if the two highest bits are both 1 (the activity bit itself and the MSB of the permanence value).

D. Determining winning columns

After transferring the input bit vector to the columns and performing the overlap calculation, the pre-specified amount of columns with the highest overlap count have to be activated. Total amount corresponds to the required sparsity of the output SDR of the SP process. A 200 column HTM with a SP sparsity set to 2% should activate 4 columns only.

Determining if the columns overlap count is higher compared to ones of the columns within its inhibition radius can be achieved by sequentially forwarding the required amount of overlap count frames over the serial channel. Every column receiving the frames transmits the received bits OR-ed with the corresponding bit of the calculated overlap count up to the first difference, where a column either looses or wins the arbitration process. In case a column lost the arbitration the rest of the frame is forwarded unaltered and the same arbitration schema is repeated for the next frame (if any is left). If a column wins the arbitration, meaning its overlap count is higher than the frame currently transmitting, the column continues the frame with its own overlap value. The received frame gets stored and transmitted next, if any frames left, i.e. the column enters to the store-and-forward transfer mode.

Winning the arbitration process and transmitting its overlap count is not enough in order to decide whether the column belongs to the top winning set or not. There can be enough columns with higher overlap count on the communication path. Therefore one extra zero bit is inserted to the line together with overlap value, which will be used as a capture bit. The top overlap counts exiting the the last column will be transferred again. During this phase all the columns try to capture a frame matching their overlap count by setting the last bit, if not set by a preceding column already.

The total amount of needed clock cycles consists of three parts. It takes C cycles for the first bit of first overlap count frame to exit the last HTM column, $L_{ov} * C_{win}$ cycles for the last bit of the last overlap frame to re-enter the first HTM column and additional C clock cycles for this last bit to exit the last column (Eq. (2)). C is the the number of columns, C_{win} the required amount of active columns and L_{ov} stands is the length of the overlap frame together with the capture bit.

$$CLK_{ov} = C + L_{ov} * C_{win} + C \tag{2}$$

E. SP learning

After successfully determining the active columns, the set can perform an optional learning operation. The learning column increments the permanence values of the active input bit connections, while suppressing the others. As the input activity bit is stored together with the corresponding permanence (Fig. 2), the learning process consists of sequentially reading the fields from the memory, incrementing or decrementing the value depending on the MSB and writing the value back while incrementing the memory address. Possible over- or underflow has to be detected and corrected. Altogether it takes three clock cycles for processing each permanence value and therefore the total clock cycles needed equals to the input connections count C_{conn} multiplied by 3 (Eq. (3)).

$$CLK_{learn} = C_{conn} * 3 \tag{3}$$

V. Serial communication

Serial communication in between adjacent columns has a negative impact to the maximum network update rate, but greatly reduces the connections count in the HW (Fig. 3). Every interface instance is capable of receiving and buffering one bit of information on every clock cycle. The transmitted bit (TX) can be either the received bit (RX) for simple forwarding, the column local TX + RX for arbitration or the column local TX for data replace mode. The received data is stored into the shadow buffer and the CTS signal towards the downstream port is cleared in case the upstream interface turns to be busy.

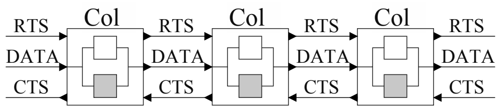


Fig. 3: Connections between adjacent columns. Every column receives the serial data from the downstream port and forwards it to the upstream port. Every cell can signal to its data source to wait in order to avoid possible buffer overflows and implements a shadow buffer for temporary data storage in case the upstream interface turns to be busy.

VI. Experimental results

The current level of the HTM algorithm implementation is limited to a SW implementation of the numeric encoder for encoding the input SDR-s, a SW implementation of the spatial pooler and VHDL implementation of the spatial pooler. The HDL description of the SP has not been synthesized to the target ZYNQ platform yet, but is being tested and verified in the simulator. Therefore the comparison can be carried out for the SP process, comparing the SW execution time to the simulated clock cycles of the HDL version. The total amount of clock cycles for the SP process can be calculated by adding up the cycle counts from Eqs. (1) to (3).

Reading the output SDR from the HTM matrix consumes additional $CLK_{out} = 2 * C$ clock cycles. C cycles are required for the first bit to proceed from the first to the last column and additional C clock cycles for the last bit to exit the last column. The total time consumed by the SP process and reading the output SDR can be calculated by adding up the clock cycles from those steps and dividing the result by the clock frequency (Eq. (4)).

$$t = \frac{CLK_{sp} + CLK_{out}}{f} \tag{4}$$

Using Eq. (4) we can calculate the total SP processing time for the HTM matrix consisting of 200 columns, each connected to 67 of possible 127 input bits for the FPGA clocking frequency $f = 100$MHz: $t \approx 0.017$ms. Same result was acquired using the ModelSIM HDL simulator. In order to compare this result against the pure software SP realization the same size HTM was used for time measurements on different platforms, Intel i7 CPU running at 2.7GHz and ARM processing unit of ZYNQ running at 650MHz (Table I). Total amount of physical memory used by one column is $M = N * 9$, where M stands for number of memory bits used, N number of input connections for a column and 9 is the width of memory word. With an input vector length of 127 bits and having 67 bits connected to every cell consumes 67 9-bit memory words out of 2K assigned for each of the columns, leaving the rest for the upcoming TP realization.

TABLE I: SP execution time.

SP	i7	ARM on ZYNQ	FPGA
t(ms)	0.266	1.875	0.017

VII. Conclusion and Future work

Based on the SP simulation results it is possible to expect up to 100-fold acceleration of an HTM being implemented on a FPGA compared to the pure SW implementation running on ZYNQ ARM processing unit, although the implementation of the TP process has not yet been started. Using the LFSR as SP address decoder has turned out to be an efficient approach in order to conserve the amount of physical memory used by the SP process, leaving more FPGA resources for the TP related data-structures. In addition, the latency of the serial connection in between columns proved not to become a performance bottleneck.

In the future, ongoing work will include the implementation of the entire functionality of the HTM on the FPGA: SP HDL synthesis, TP implementation as a software process, TP HDL simulation and synthesis, and testing with real-time streaming data.

References

[1] D. Graupe, *Deep Learning Neural Networks. Design and Case Studies.* World Scientific Publishing Co. Pte. Ltd., 2016.

[2] Numenta, "Hierarchical Temporal Memory including HTM Cortical Learning Algorithms. White paper VERSION 0.2.1," http://numenta.org/resources/HTM_CorticalLearningAlgorithms.pdf, September 2011.

[3] ——, "Applications of Hierarchical Temporal Memory (HTM)," 2014. [Online]. Available: http://numenta.com/papers-videos-and-more/resources/applications-of-hierarchical-temporal-memory/

[4] A. Lavin and S. Ahmad, "Evaluating Real-time Anomaly Detection Algorithms - the Numenta Anomaly Benchmark, year=2015, doi=10.1109/ICMLA.2015.141,," in *14th International Conference on Machine Learning and Applications (IEEE ICMLA).*

[5] S. Ahmad and J. Hafkins, "How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites," 2016. [Online]. Available: arXiv:1601.00720 [q-bio.NC]

[6] J. Hawkins and S. Ahmad, "Why neurons have thousands of synapses, a theory of sequence memory in neocortex," *Frontiers in Neural Circuits*, vol. 10, no. 23, 2015.

[7] Numenta, "Numenta platform for intelligent computing," 2016. [Online]. Available: http://numenta.org/

[8] X. ZHOU and Y. LUO, "Implementation of hierarchical temporal memory on a many-core architecture," Master's thesis, Halmstad University, School of Information Science, Computer and Electrical Engineering (IDE), 2013.

[9] M. Deshpande, "FPGA Implementation and Acceleration of Building blocks for Biologically Inspired Computational Models," Master's thesis, Portland State University.

[10] A. M. Zyarah, "Design and analysis of a reconfigurable hierarchical temporal memory architecture," thesis, Rochester Institute of Technology.

[11] P. Vyas and M. Zaveri, "Verilog implementation of a node of hierarchical temporal memory," *Asian Journal of Computer Science & Information Technology*, no. 3.7, pp. 103–108, 2013.

[12] Wikipedia, "Linear-feedback shift register," 2016. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Linear_feedback_shift_ register&oldid=754433904