TECHNISCHE UNIVERSITÄT WIEN

Institut für Computertechnik
Institute of Computer Technology

A MASTER THESIS ON

Context-aware Monitoring for NEMS

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

Diplom-Ingenieur
(Equivalent to Master of Science)

in

Embedded Systems 066 504

by

Philipp Lehninger

01327039

Supervisor(s):
Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Proj.Ass. Ardavan Elahi, MSc

Vienna, Austria
February 2025

Abstract

Monitoring the correct behavior of an arbitrary system can be costly if it the monitoring system has to be taylored to the system under observation. To ease the configuration of such a device health monitoring system, Confidence-based Context-Aware Monitoring (CCAM) was developed. Although it can be considered a lightweight algorithm on modern embedded computing systems, it was not designed for extremely hardware-constrained use cases. To further extend the field of possible applications, the algorithm has been simplified and implemented in hardware on a novel NEMS technology. Simultaneously, this work is a case study for logic design on NEMS. Therefore, it has been investigated how to leverage the potential of that technology and a trade-off analysis for two implementations has been performed. The insights allow for improvement of further circuit designs and hint towards potential improvements of synthesis targeting NEMS.

Kurzfassung

Die Überwachung des korrekten Verhaltens eines beliebigen Systems kann kostspielig sein, wenn das Überwachungssystem an das zu überwachende System angepasst werden muss. Um die Konfiguration eines solchen Systems zur Überwachung des Gerätezustands zu vereinfachen, wurde Confidence-based Context-Aware Monitoring (CCAM) entwickelt. Obwohl es als ressourcen-schonender Algorithmus auf modernen Embedded Systems betrachtet werden kann, wurde er nicht für Anwendungsfälle mit extrem beschränkten Hardware-Anforderungen konzipiert. Um das Feld der möglichen Anwendungen weiter zu erweitern, wurde der Algorithmus vereinfacht und in Hardware auf einer neuartigen NEMS-Technologie implementiert. Gleichzeitig ist diese Arbeit eine Fallstudie für den Logikentwurf auf NEMS. Daher wurde untersucht, wie das Potenzial dieser Technologie genutzt werden kann und es wurde eine Trade-off-Analyse für zwei Implementierungen durchgeführt. Die Erkenntnisse ermöglichen die Verbesserung weiterer Schaltungsentwürfe und geben Hinweise auf mögliche Verbesserungen der Synthese für NEMS.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Copyright Statement

I, Philipp Lehninger, hereby declare that this thesis is my own original work and, to the best of my knowledge and belief, it does not:

- Breach copyright or other intellectual property rights of a third party.
- Contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
- Contain material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.
- Contain substantial portions of third party copyright material, including but not limited to charts, diagrams, graphs, photographs or maps, or in instances where it does, I have obtained permission to use such material and allow it to be made accessible worldwide via the Internet.

Signature: _____

Vienna, Austria, February 2025 Philipp Lehninger

Acknowledgment

During my journey through the world of CCAM and NEMS, I was happy to not only have friendly colleagues but also friends at the institute. I am not good at writing this kind of text, but there are a lot of people whom I would like to thank for their support, their valuable input, and for the good time I have spent with them.

First, I would like to start with the colleagues in my office. Throughout my time at ICT, I have been happy to share the office with Sofia, David, Kathi, Anoush, Dominik, Jan, and Jasper. I don't want to miss the decent sarcasm of Sofia during my work and Anoush has always been motivating and supportive and he puts a lot of effort into doing what needs to be done. He really works too much! Hearing Dominik trying to speak Dutch with Jasper and saying weighty sentences like 'Ik wil een boterham' has often been funny and I am sorry that Dominik and Jasper will leave the ICT soon.

Special thanks to Daniel Schnöll, Daniel Hauer, Maximilian Götzinger for their valuable insights from their past work on CCAM. My discussions with Daniel Schnöll have turned out to be extremely valuable.

Thanks to Axel for allowing me to be part of the ICT and contribute to this exciting project, as well as for his feedback that seems to be always on-point.

Thanks to all members of the i-EDGE team I have worked with so far, as all of them contributed to a comfortable and supportive meeting environment, especially to Elliott, Victor, and Simon. I know that you have been busy giving Anoush and me support and I have really appreciated your help. Last but not least, I would also like to thank my family and friends who are not part of ICT.

Project funded by

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Economic Affairs,
Education and Research EAER
**State Secretariat for Education,
Research and Innovation SERI**

This work was supported by the i-EDGE project, which has received funding from the European Union (grant number 101092018), the Swiss State Secretariat for Education, Research and Innovation (SERI) and UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee (grant numbers 10061130 and 10063023).

Contents

List of Tables

List of Figures

Acronyms

6-T Six-Terminal. 12

ADC Analog-to-Digital Converter. 12

ASIC Application Specific Integrated Circuit. 2

CAS Compare-And-Swap. 63

CCAM Confidence-based Context-Aware Monitoring. 1, 40

CMOS Complementary Metal-Oxide-Semiconductor. 2

DAB Discrete Average Block. 28, 73

DPM Datapath Module. 74

FIFO First In First Out. 65

FPGA Field Programmable Gate Array. 2

FPU Floating-Point Unit. 12

IIoT Industrial Internet of Things. 1

LED Largest Element Detector. 60

MCDP Matching Confidence Datapath. 74

MMU Memory Management Unit. 74

NEM Nano Electro-Mechanical. 2

PDK Process Development Kit. 43

RoSA Research on Self-Awareness. 10

RTL Register-Transfer Level. 12

SuO System under Observation. 1

TCAM Threshold-based Context-Aware Monitoring. 24

VCDP Validity Confidence Datapath. 74

Chapter 1

Introduction

As manual adjustments and maintenance of industrial systems are very expensive and time-consuming, it is desirable to reduce the need for manual (re-)configurations and device health checks.

Especially in the field of the Industrial Internet of Things (IIoT), the amount of small devices grows rapidly and so does the maintenance effort. Therefore, generalized methods that are able to automatically monitor the system state of a variety of systems are required. By detecting deviations from the normal system behavior, respectively, and untypical transitions between sensor measurements, the maintenance costs can be greatly reduced.

Because devices in the IIoT domain are usually resource-constrained, on many devices only lightweight monitoring algorithms can be used. As a possible solution, Confidence-based Context-Aware Monitoring (CCAM) has been proposed [10]. The monitoring algorithm treats the System under Observation (SuO) mostly as a black box as depicted in Figure 1.1. This allows for the application of CCAM on various SuOs, without prior knowledge of internal processes.

It is an algorithm that compares samples from system input and output variables with preceding

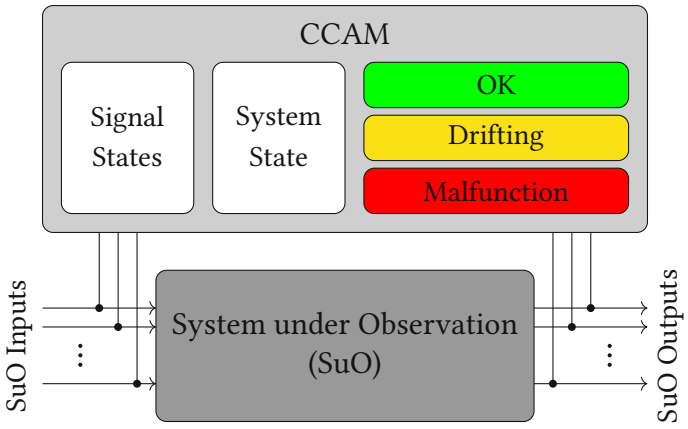


Figure 1.1: CCAM monitoring a black box [3]

samples and calculates a confidence value of how likely a new sample fits the preceding samples, based on fuzzy logic. Similar samples of the same system variable form a dataset. These clustered datasets are called *signal states*. If the distance between a new sample and the current dataset is too big, a change of the signal state is detected. Moreover, the relations between system inputs and outputs are treated as bijective functions. If the signal state of a system input changes, but the signal state of a corresponding output does not, the monitored system will be considered malfunctioning. More detailed descriptions can be found in section 2.1 and in chapter 3.

Although CCAM is considered to be a light-weight context-aware monitoring algorithm, for instance, compared to approaches with neural network classifiers [11], it has to be further optimized to fit extremely resource-constrained use cases, like integration in logic onto an Field Programmable Gate Array (FPGA) or an Application Specific Integrated Circuit (ASIC).

A very challenging and interesting use case would be the implementation of CCAM in a harsh environment, where even traditional computer architectures and Complementary Metal-Oxide-Semiconductor (CMOS) technology may not be suitable. Such harsh environments could be either in the domains of aerospace or industrial with high temperature or radiation doses.

The Nano Electro-Mechanical (NEM) relays [12] developed by the i-EDGE project are a novel technology that is designed for harsh environments. The logic gates built with it can still operate at high temperatures (above $225°C$ and up to $300°C$) and high radiation levels. Furthermore, there is no leakage current without switching activity, so battery-powered devices can continue operation much longer. Thus, it may have the potential to help to overcome limits that CMOS cannot. The NEM technology of the i-edge project is explained in the next section (1.1).

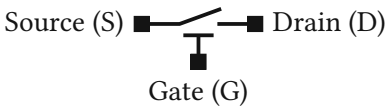
1.1 NEM Devices

This section explains the underlying target technology for the resource-constrained implementation use case. Although this work always refers to the NEM switches designed by the i-EDGE project when mentioning NEM switches, there exist different approaches and motivations for NEM switches. [13] [14] [15].

In the course of ZeroAMP, the predecessor of the i-EDGE project, three different switch designs have been created [16] [12]. They differ in their functionality as well as in the number of terminals. Their design and characteristics are explained in detail in the thesis of Elliott Worsey [2].

1.1.1 NEM 3-T Device

The Three-Terminal (3-T) Device is a small relay that connects the *source* terminal with the *drain* terminal when the voltage applied between the *source* and *gate* terminal exceeds a certain threshold. Figure 1.2 depicts the device symbol and the truth table describing the switch behavior. As the polarity of the gate voltage is irrelevant, the 3-T device can be handled in the same way as the n-MOS and p-MOS transistors. Therefore, most CMOS logic gate designs can also be implemented in NEMS without relevant changes in the circuits.



(a) Symbol

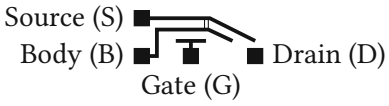
G	S	D
0	0	Z
0	1	1
1	0	0
1	1	Z

(b) Truth Table

Figure 1.2: NEM 3-Terminal Device (3-T)

1.1.2 NEM 4-T Device

The Four-Terminal (4-T) Device is designed like the 3-T device, however, the beam of the relay is divided into two separate parts that are mechanically connected but electrically insulated. This leads to the advantage that the voltage controlling the switch state is applied between the *gate* terminal and the new *body* terminal and independent from the potential of the *source* terminal. Figure 1.3 shows the symbol of a 4-T device and the corresponding truth table.



(a) Symbol

G	B	S	D
0	0	0	Z
0	0	1	Z
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	Z
1	1	1	Z

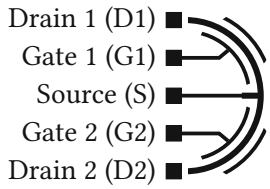
(b) Truth Table

Figure 1.3: NEM 4-Terminal Device (4-T)

1.1.3 NEM 7-T Device

The Seven-Terminal (7-T) Device is the most complicated of the three NEM switches designed to date. The device symbol and the truth tables are shown in Figure 1.4. Its circular beam starts in a neutral

position and can be moved to one of two *drain* terminals. Once it is connected to a *drain* terminal, it will its position until the conditions to move to the other *drain* terminal are met. As the device keeps its position and the there are two truth tables, one for setting the switch position and another one for the actual *drain* outputs. In addition to the two *gate* terminals, there are also two *auxiliary gates* but they are connected to the *gate* terminals to increase the attraction of the beam. Therefore, the 7-T device is actually treated as if it had only five terminals.



(a) Symbol

G1	S	G2	pos_{n+1}
0	0	0	pos_n
0	0	1	right
0	1	0	pos_n
0	1	1	left
1	0	0	left
1	0	1	pos_n
1	1	0	right
1	1	1	pos_n

(b) Device State Table

pos_n	S	D1	D2
neutral	0	Z	Z
neutral	1	Z	Z
left	0	0	Z
left	1	1	Z
right	0	Z	0
right	1	Z	1

(c) Output Truth Table

Figure 1.4: NEM 7-Terminal Device (7-T)

1.2 Research Questions

To bring context-aware system monitoring and cheaper maintenance also into domains with harsh environments on the one hand, but also to help demonstrate the applicability of the NEMS relay technology on the other hand, this thesis will evaluate how much the arithmetic and computational effort of the CCAM algorithm can be reduced and optimize it towards the benefits of the target technology.

The research questions can be reformulated as following questions:

- What are the tradeoffs of the CCAM algorithm regarding area, latency, and accuracy?
- How can the algorithm be optimized towards the NEMS relay technology by utilizing the 4-terminal and 7-terminal relays?

1.3 Methodology

The methodology for resolving the answers to the research questions consists of different steps.

1. Describe the rather unmodified algorithm in Verilog and simulate it
   (a) Identify area-expensive (arithmetic) operations
   (b) Derive Finite State Machine(s) from algorithm

(c) Identify design parameters

(d) Determine memory requirements

2. Simplify the algorithm and describe it in Verilog

   (a) Simplify the algorithm and its arithmetic

   (b) Compare the simplified algorithm with the original CCAM algorithm

   (c) Describe the stripped-down algorithm in Verilog and simulate its behavior

3. Explore the tradeoffs between different implementations when changing design parameters

   (a) Explore implementations of area-expensive operations

   (b) Consider serialization vs. parallelization

   (c) Consider resource sharing

   (d) Implement promising designs and compare them

4. Implement the simplified algorithm on NEM hardware

   (a) Complete the entire RTL-to-GDSII flow to prepare the designs for fabrication

Metrics for comparison between original CCAM and simplified CCAM are:

- Percentage of correct outputs over time series (Accuracy)
- NEM Switch Device Count
- Cycle Count (Time)

Power has not been included in the set of metrics as there is hardly any useful information available at the time of writing this thesis. In the later course of the project, more information on power dissipation will be measured and provided to the design tools. The device count of the NEM switches is correlated with the necessary chip area, however, the probability of the occurrence of faults that originate from manufacturing does not only depend on the chip area but also on the device count. This is because stuck-on and stuck-off faults of the NEM switches are rather caused by mechanical breakdown or unwanted mechanical connections, than by doping and impurity issues.

Chapter 2

Literature Review

As there is related work and literature from different topics that have been considered to be relevant for this thesis, this chapter focused on the review of literature about Context-Aware Monitoring and Fuzzy Logic for a general overview. Literature about possible implementations of sub-components is covered in the respective sections about the implementation of these sub-components.

2.1 Device Health Monitoring

2.1.1 Context-Aware Health monitoring (CAH)

The early version of CCAM has been named *Context-Aware Health monitoring (CAH)* and is introduced in [17] with an AC-motor case study. In [18] it was shown, that CAH is also applicable to a completely different application domain by monitoring a hydraulic system.

Striking differences to its successor CCAM are, firstly, that it uses threshold-based instead of confidence-based comparisons, and secondly, that signal state detection is only performed on stable signals. To assess if a sample belongs to a signal state, the sample value is compared to an average of previous sample values, whereas CCAM compares a sample with all entries of a sliding window history (stated in [19, p. 88]).

These algorithmic changes in CCAM resulted in "equally good or better results" than with CAH [10]. Moreover, CAH requires signal preprocessing (lowpass filtering) to remove noise but CCAM does not anymore.

2.1.2 CCAM for Smart Grids

Daniel Hauer et al. adapted CCAM for monitoring of Smart Grids [3]. In this context load day profiles have been investigated, so only one signal is monitored and thus only the signal state detection of

CCAM has been used but not the system state detection functionality. The high complexity of load profiles of Smart Grids hamper a proper detection of signal states and therefore, some adaptations have been necessary.

Instead of checking the similarity to other signal states only if a sample does not fit the current state, *Continuous State Reevaluation* compares a new sample to all already created signal states and selects the most fitting one.This change notably enhances the detection of signal states at the cost of additional computational effort (linear growth with the amount of created states).

Another deviation from CCAM is *State Mooring*. As load profiles can have quick as well as slow state transitions, slow transitions would be recognized as signal drift. The drift detection of CCAM is replaced by a similar concept, however, instead of raising a drift alarm, state mooring influences the state matching calculation. The mooring history which reminds of the Discrete Average Blocks (DABs) in the original CCAM adds the initial sample values of a signal state until it is full. If the mooring history of a signal state is full, samples are not only compared to the values inside the sample history but also to the mooring history's mean value. This adaption is reasonable if the monitoring system does not need to have a distinction between signal drift and state changes. However, in the majority of use cases in the domain of embedded systems a semantic difference between drift and signal transition will exist.

2.1.3 Mathematical Analysis of CCAM and Enhancements

In the thesis of Daniel Schnöll the Signal State Detector of CCAM is analyzed from a mostly mathematical point of view. [20]. With the gained findings, enhancements to improve the accuracy of the signal state change detection have been proposed.

As different naming conventions have been introduced for distances and confidences, denominations of both [10] and [20] are used in this section.

This thesis stays with the denominations defined in [10] throughout all other sections but the term *co-confidence* has been adopted from [20]. Figure 2.1 shows an example confidence function in green and its so-called co-confidence function in red. The idea behind it is that the co-confidence is the confidence in the opposite case. As an example, if $c_1(x)$ is the confidence that a property of a signal is "X", its co-confidence $c_2(x)$ is the confidence that this property is not "X". In CCAM these co-confidences are defined as the negated confidences s.t. $c_2(x) = \overline{c_1(x)} = 1 - c_1(x)$. Fuzzy algebra is explained in the section 2.2.

Daniel Schnöll identified mainly five problems ([20, p. 26]) in the original CCAM algorithm.

1. Normalization of the distance

The normalized distance $d_{i,j} = \Delta_{norm,i,j} = \left|\frac{v_i - H_{i,j}}{v_i}\right|$ uses a division for normalization.If sample

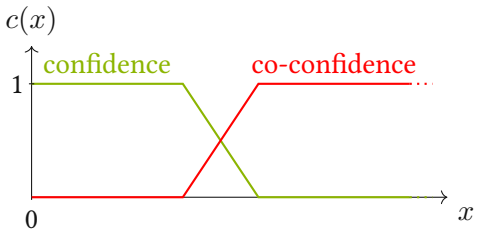


Figure 2.1: Confidence Function with its Co-Confidence Function

values can happen to be 0, this possibly leads to a division by 0 and even a close vicinity to 0 would result in an enormous distance. CCAM would be unable to keep any signal state.

2. Definition range of the sample value confidence

Descriptions and diagrams in [10] show fuzzy functions for the confidence mapping of distances that are defined for negative distances, too. This is misleading as the calculation of distances includes an absolute value function. Therefore, fuzzy functions only need to be defined over $\mathbb{R}_{0+}$.

3. Potential over-complication of fuzzy operations

Like the previous problem, this problem is not necessarily a problem of the algorithm itself but of its description.

4. Reduction to threshold-based algorithm

In CCAM any fuzzy confidence function a and its co-confidence function b are defined such that $a + b = 1$ always holds. It has been proven that, under this condition, the decision if a sample matches a signal state could be reduced to a threshold-based decision without changing the decision outcome.Nevertheless, confidences contain more information than boolean decision outcomes as they show how confident CCAM has been about its decisions.

5. Outlier creating states

In case the first sample that leads to the creation of a new signal state is an outlier, it may happen that subsequent samples are still close enough to this outlier such that the falsely created state will not be left or deleted. However, such false state changes should be avoided.

Among the proposed enhancements, there is a method for automatic configuration parameter detection. Selecting optimal configuration values for the kink points of all fuzzy functions in a use case still involves human involvement and automatic configuration would avoid misconfiguration and make the adoption of the algorithm easier.

Furthermore, integrated preprocessing was discussed. With the absence of the division in the distance calculation, the algorithm becomes more sensitive to drift. For history lengths $|H| > 10$ the addition of a *mean of differences* or *linear approximation* have been suggested. However, both options significantly increase the computational effort, including multiplications or even divisions.

2.1.4 RoSA Framework

The original CCAM algorithm has been implemented on top of the RoSA framework, hence RoSA is briefly introduced. The Research on Self-Awareness (RoSA) framework was built to make modeling and evaluation of self-awareness concepts easier [4]. It is organized into independent components called agents with a hierarchical structure (Figure 2.2). Correspondingly, in order to implement CCAM, for each monitored signal a signal state detector agent is instantiated that reports to a common system state detector agent on a higher hierarchy level. This results in a structure like in Figure 2.3.

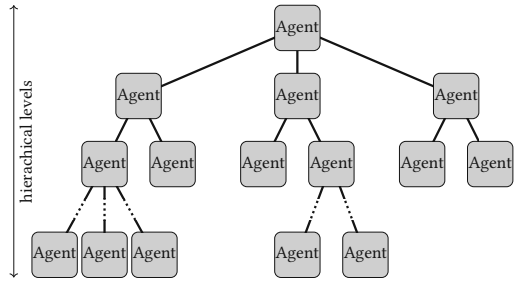


Figure 2.2: General hierarchical model

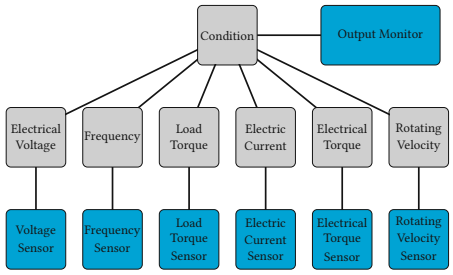


Figure 2.3: Architecture of CCAM for AC motor case-study

Figure 2.4: Agent-based structure in RoSA [4]

The C++ implementation of RoSA is open-source and includes also the implementation of CCAM. The source code is available at the following Git repository:

`https://phabricator.ict.tuwien.ac.at/source/SoC_Rosa_repo.git`

As there were two minor bugs in the code on the master branch at the time of writing this thesis, a separate branch named *ccam_fixes_2024* was set up in the repository, including the bug fixes and some changes for easier information extraction. When referring to the original CCAM in comparison, the code on this branch is spoken of.

2.2 Fuzzy Logic

Since CCAM involves fuzzy logic for calculating the confidences of its system assessment, literature on hardware implementations with fuzzy logic seems relevant. In general, fuzzy logic extends Boolean logic by extending the set of Boolean values $\mathbb{B} = 0, 1$ to the interval (0,1]. Instead of just having Boolean states of *true* or *false*, this allows the expression of uncertainties.

Watanabe et al. presented a VLSI Fuzzy Logic Controller [5] while also giving a good explanation how fuzzy logic is applied to a control engineering problem.

Typically, fuzzy logic controller consist out of three stages:

- Fuzzification
- Rule Evaluation
- Defuzzification

2.2.1 Fuzzification

Fuzzification applies fuzzy membership functions to "crisp" input data, like discrete sensor measurements, and returns confidences on the membership to multiple classes [5]. These confidences are real numbers and lie within the interval $(0, 1]$. A simple example for a temperature value as input would be a classification into "low", "medium" and "high" temperature, whereby the membership functions are allowed to overlap (See Figure 2.5). Membership functions are can have different shapes; mostly triangular, trapezoidal and gaussian shapes.

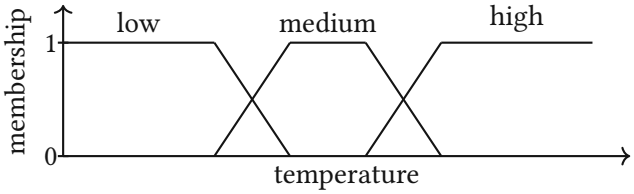


Figure 2.5: Example membership functions

2.2.2 Rule Evaluation

Rule Evaluation or *Inference* applies the membership confidences to a set of rules in the form

`IF A' is A1 [AND B' is B1] THEN C is C1'`. The operations *AND* and *OR* in these rules are fuzzy logic operands. The standard algebraic operations on fuzzy logic are related to their Boolean counterparts as Table 2.1 shows. Distributive and De Morgan's laws apply to fuzzy logic, too [21].

Table 2.1: Fuzzy Logic Algebra [1, p.124]

Operation	Definition
Negation	$T(\overline{P}) = 1 - T(P)$
Disjunction	$T(P \vee Q) = max(T(P), T(Q))$
Conjunction	$T(P \wedge Q) = min(T(P), T(Q))$
Implication	$T(P \rightarrow Q) = max(T(\overline{P}), T(Q))$

As multiple rules are applied, there will be multiple results for the THEN-parts of the rules(*actions*). The fuzzy result of C' can be acquired by computing $C' = max(C1, C2, C3, ...)$.

2.2.3 Defuzzification

Defuzzification transforms a fuzzy variable back to a real-numbered property.

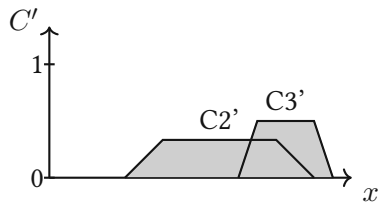


Figure 2.6: Example of a fuzzy result [5]

CCAM does not completely adhere to the concept of a fuzzy controller, as the confidences in CCAM are either used as confidence outputs without defuzzification or they are used to evaluate binary decisions. The defuzzification used for these binary decisions is done by a comparison of a confidence with its co-confidence.

```
if (confidence > co-confidence) then ... else ... ;
```

However, the outputs of a fuzzy system are usually not binary. Instead, defuzzification is a task that is more complicated than a comparison between two confidences. There are various defuzzification methods with *center of gravity (COG)*, shown in Equation 2.1, being probably the best known [22].

$$COG(A) = \frac{\Sigma_{x_{min}}^{x_{max}}(x * A(x))}{\Sigma_{x_{min}}^{x_{max}} A(x)} \tag{2.1}$$

The result is the center of gravity of the area under the membership functions (Figure 2.6).

2.3 Logic Design for NEMS

Although there are several publications on different NEM relay technologies, logic design for NEMS surpassing the scope of individual logic gates is still a field where little work has been done. In the course of the i-EDGE project, a NEM standard cell library [2], a switch box unit for FPGAs [23] and an Analog-to-Digital Converter (ADC) [24] have been presented, however, it does not include logic design and optimization on Register-Transfer Level (RTL) level. Besides that, research on some arithmetic blocks has been carried out but has not been published yet.

Liu et al. built some logic gates from Six-Terminal (6-T) switches [25] comparable to i-EDGE's 4-T switches that were used later in a case study designing an Floating-Point Unit (FPU) [26]. The underlying NEM synthesis tool that focuses on minimization of the circuit delay was presented in [27].

Optimization of logic synthesis for another 6-T NEMS technology has also been proposed in [28] and [29] leading to subsequent publications including a book on data structures and algorithms for synthesis of beyond CMOS technologies [30].

Chapter 3

Algorithmic Analysis

To simplify and implement the CCAM algorithm, an in-depth analysis of the source code is necessary. The analysis contains an extraction of precise decision diagrams and dataflow diagrams.

Decision diagrams can then be translated into finite state machines (FSMs), whereas dataflows allow for an overview of necessary datapaths. These datapaths have to be broken down into required arithmetic and logic operations and especially, computationally intensive operations in the datapath can be identified.

As for simplification, the goal is to find possible simplifications that do not alter the output behavior of the algorithm. So for this thesis, deviations in the behavior are only acceptable if absolutely necessary.

There are different versions of the CCAM code, but the official one is available in C++ on the repository of the RoSA Framework. For more details on RoSA and a link to the repository, see section 2.1.4. For the sake of easier modification and better exploration of the algorithm, CCAM has also been implemented in Python. In the repository SoC_Verilog_CCAM there are various variants of CCAM, starting with a direct translation of the algorithm in the folder *src/ccam/python/ccam_orginal*. Each simplification has been tested in Python and the quantization can also be emulated.

This chapter does not focus on design decisions and implementation of certain operations, but solely on the algorithm itself. Nevertheless, presented algorithmic simplifications will lead to design decisions that will occur in chapter 5.

In section 2.1.3 it has been stated that the relation $a + b = 1$ between confidences and their co-confidences has the consequence that the confidence-based decisions could be reduced to threshold-based decisions. This opens up two major simplification possibilities of different granularity.

Moreover, the limited memory makes it necessary to adapt the algorithm as dynamic memory allocation will not be possible in hardware. This also opens up the question on what to do if "allocation" of additional memory is not possible because it would exceed the capacity.

3.1 Testing

Adaptions of the CCAM algorithm have to be compared to a golden model to ensure correct behavior on the one hand and to compare relevant metrics of different versions of CCAM on the other. For this purpose, a set of Python scripts was written that read configuration files, start the execution of the algorithms, store the results in CSV files, and visualize the differences in diagrams. Since the original RoSA implementation is compiled into a C++ program but all other software CCAM versions are written in Python, the comparison script executes the CCAM versions via system calls.

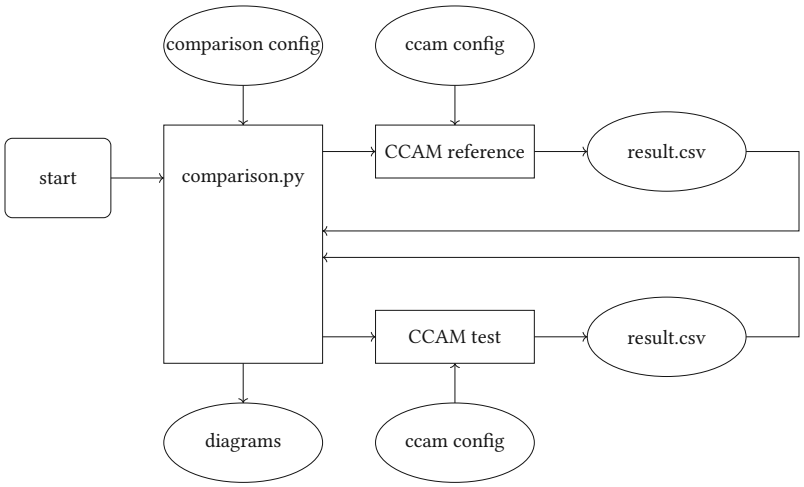


Figure 3.1: Setup for Comparison (SW to SW)

Figure 3.1 depicts the rough flow of the algorithm comparison. The script requires the path to a comparison configuration file in JSON as a command line argument. The command to start a comparison is therefore *python3 comparison.py -f <path-to-config>*. Inside the config file, the path to the executables and their intended working directory and the path to the CCAM config file are set. Listing 3.1 shows an example of such a configuration.

Listing 3.1: Comparison Configuration Example

```
{
    "level": "system",
    "reference": {
        "name": "RoSA",
        "cwd_path": "../../tests/20180529_NormalTwoTimesSameState",
        "command": "",
        "cmd_path": "./ccam_bugfixed",
        "config": "config_rosa.json",
        "type": "rosa"
    },
    "implementation": {
        "name": "Python",
        "cwd_path": "../../tests/20180529_NormalTwoTimesSameState",
        "command": "python3",
        "cmd_path": "../../src/ccam/python/ccam_original/run_py_ccam.py",
        "config": "config_py.json",
        "type": "python"
```

```
18            }
19        }
```

The *name* fields are to distinguish the time series in the diagrams and the *level* field can be *signal*, *system*, or *both*. It defines whether to run the Signal State Detector or the entire CCAM including the System State Detector. The RoSA implementation always runs the entire CCAM, but the Python versions can be configured to run only the Signal State Detector. The *both* level will output the results of the System State Detector but also the detailed results of each Signal State Detector.

The configuration of a Python CCAM version is an extended version of the config files used by RoSA. The additional fields are mainly needed for quantizing data, such as simulating a reduced bit width for samples or confidence values.

The comparison of the results of a behavioral simulation of a Verilog implementation with the golden model in Python is done by another script and will be mentioned later.

The data set for the algorithm testing consists of timeseries from a water pipe system that has already been used in [18] and [10]. Figure 3.2 shows the CCAM architecture for these case studies.

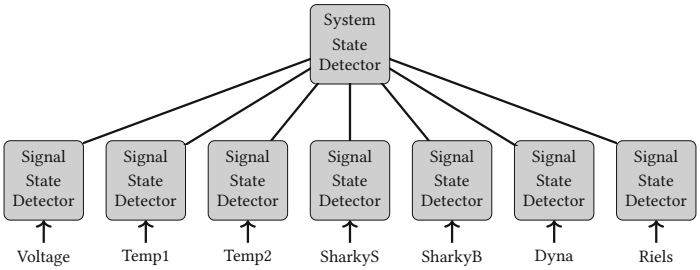


Figure 3.2: CCAM for water pipe case-study

The voltage of the water pump is an control input to the SuO and lies between 3 to 10 V during operation. The voltage samples in the data set have been normalized (0.3 - 1.0). All other monitored signals are sensor measurements and are treated as SuO outputs. The measurement unit of the temperatures is $°C$ and the unit of the remaining timeseries is l/s. The sampling frequency was 30.5 Hz in average. [18] The time series snippets have an average length of 20000 subsequent samples but CCAM was configured to apply a downsampling rate of 50, resulting in a sequence of 400 samples per snippet. A hardware implementation would therefore have to process new samples every 1639 milliseconds in this use case.

The accuracy metric used for CCAM has already been mentioned in section 1.3, but will now be explained in more detail. The correct operation of CCAM relies on the detection of Signal States in the Signal State Detector. The Signal State IDs are the essential input to the System State Detector and the confidence if the SuO is functional depends on the System State. Despite the appliance of fuzzy logic, creating, leaving and entering a state are binary decisions in the end. A divergence of the state detection may lead to different decisions afterwards, making the detection accuracy time-dependent. More than

the functional confidence or any other confidence in the system, the *State IDs* are the relevant results that have to be compared between different implementations. As the creation of a *Signal State* affects the *Signal State ID* of future states, a short visit in a state created by outliers in the samples could already lead to a divergence that detects the *Signal States* correctly but with different IDs assigned to them. Therefore a state mapping algorithm has been written to get a significant accuracy metric. Listing 3.2 shows the algorithm in Python.

Listing 3.2: State Mapping Algorithm

```
states1 = stateDf1["StateID"].unique()
states2 = stateDf2["StateID"].unique()

stateMapping1 = {}
stateMapping2 = {}

for s1 in states1:
    maxOverlap = 0

    for s2 in states2:
        o = 0
        for i in range(0, length):
            if (stateDf2.loc[i, "StateID"] == s2):
                if (stateDf1.loc[i, "StateID"] == s1):
                    o += 1

        if o > maxOverlap:
            maxOverlap = o
            stateMapping1[s1] = s2
    if (maxOverlap == 0):
        stateMapping1[s1] = 0
#---
for s2 in states2:
    maxOverlap = 0

    for s1 in states1:
        o = 0
        for i in range(0, length):
            if (stateDf2.loc[i, "StateID"] == s2):
                if (stateDf1.loc[i, "StateID"] == s1):
                    o += 1

        if o > maxOverlap:
            maxOverlap = o
            stateMapping2[s2] = s1
    if (maxOverlap == 0):
        stateMapping2[s2] = 0
#---
```

```
if (len(states1) > len(states2)):
    for s1 in states1:
        s2 = stateMapping1[s1]
        if (s2 in stateMapping2):
            if (stateMapping2[s2] != s1):
                stateMapping1[s1] = 0
else:
    for s2 in states2:
        s1 = stateMapping2[s2]
        if (s1 in stateMapping1):
            if (stateMapping1[s1] != s2):
                stateMapping2[s2] = 0
```

For every state detected by an CCAM implementation, the time-wise most overlapping state of the other implementation is selected as the corresponding state. If one state is the corresponding state of more than one state, the non-bidirectional mapping will be deleted, resulting to 1:1 mapping between the *State IDs*. When there is no corresponding state, the state will be mapped to ID 0. This happens when one detector detects more or less states than the other.

3.2 Data Flow

All calculations on sample values or confidences are part of the data flow in CCAM. Most of these calculations have a Boolean result in the end, and subsequent Boolean operations on them are somehow in a gray zone between data and control flow. The nomenclature used for confidence values in this section adheres to that used in [10].

Inside the Signal State Detector, the following three data flows can be found:

1. Matching Confidence Data Flow

This data flow includes all necessary calculations to decide if a sample matches a signal state.

2. Validity Confidence Data Flow

This data flow derives if a signal state is valid.

3. Stability Confidence Data Flow

This data flow is part of the drift detection.

The confidences of all Signal State Detectors are then passed to the System State Detector. The data flow of the System State Detector can be split into these parts:

1. Aggregation of Signal State Confidences

The matching confidence, the validity confidence, and the stability confidence as well as their co-confidences are aggregated over all observed signals.

2. Functioning Confidence Data Flow

The value of a counter that counts the subsequent time steps where the mapping of signal states indicates an inconsistency in the System State is translated into confidence if the system is working correctly or malfunctioning.

3. Overall Confidence Data Flow

All confidences about the decisions of CCAM are combined to an overall confidence.

Diagrams of all these data flows could already be interpreted as potential physical structures of the respective data paths in a hardware implementation and are thus part of chapter 5. All data flows will be presented and explained in detail and possible simplifications will be discussed below.

3.2.1 Matching Confidence Data Flow

A signal sample is added to a history of a signal state if it matches the older samples in the history. When a signal state does not exist or the sample does not fit into any available signal state, a new signal state is created.

The decision if a sample matches, depends on the matching confidence $c_{b,i}$ and the mismatching confidence $c_{n,i}$. So the inputs of this data flow are the new sample, the samples present in the sample history, the fill level of the sample history and its maximum fill level, and the configuration of the used fuzzy functions. The outputs are the matching confidence and its co-confidence, the mismatching confidence.

At first, the definitions from [10] are re-examined, together with additional explanation. Let the sample history H have a length $|H|$ s.t. it can store up to $|H|$ samples of the same signal from previous time steps that matched the current signal state. The current fill level of the sample history is s_a, the new sample value of signal i is $v_{i,new}$ and the samples in the history are $v_{h_{i,j}}$ with $j = 1..s_a$.

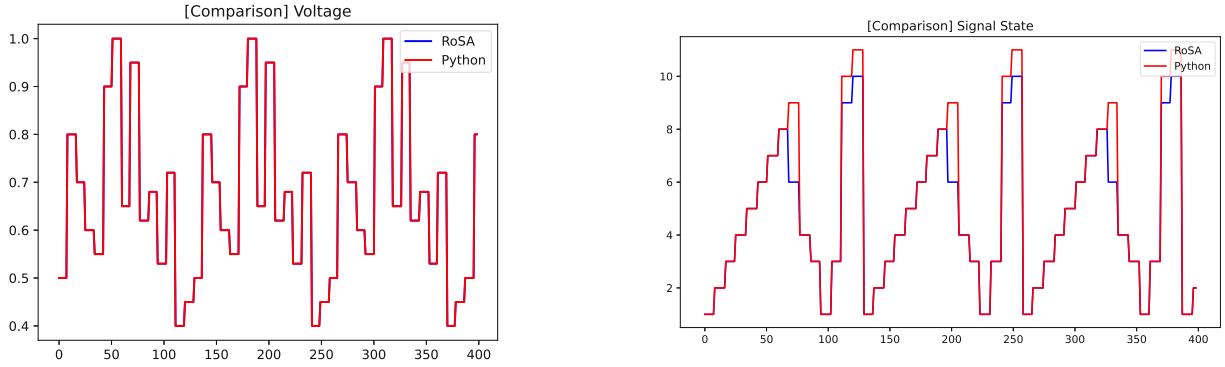
The distance metric that describes how well two samples match each other is defined as 3.1.

$$d_{i,j} = \left| \frac{v_{i,new} - v_{h_{i,j}}}{v_{i,new}} \right| \tag{3.1}$$

As already suggested and implemented by [20], the division in the distance metric could be removed. (This has been mentioned in section 2.1.3.) The argument was that values in the close vicinity of 0 result in a high distance that makes it impossible for the algorithm to keep a signal state. Besides that, this suggestion is highly welcomed as a division unit is also a complex hardware module that would cost a lot of chip area. Equation 3.2 shows the adapted distance metric.

$$d_{i,j} = \left| v_{i,new} - v_{h_{i,j}} \right| \tag{3.2}$$

This change from a relative distance in percent to an absolute distance affects the algorithm's results as Figure 3.3 shows on an example where the results from the RoSA framework are compared to a Python implementation with the divisionless distance metric. Also, the definition of the fuzzy function's kink points in percentages is more use-case agnostic than with absolute distances. Therefore, the configuration of the fuzzy functions have to be adapted, too.



(a) Voltage Samples

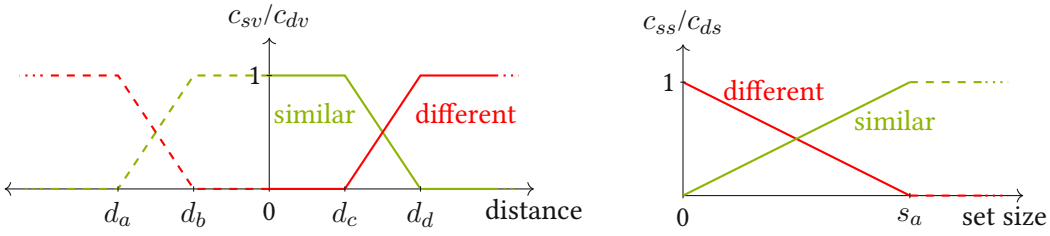
(b) Signal States

Figure 3.3: Signal States in original CCAM and with the adapted Distance Metric

As it does not make sense to keep CCAM with the relative distance as the golden model for all subsequent comparisons, the version using the absolute distance has been selected as the golden model from here on.

The fuzzy functions used in this data flow are depicted in Figure 3.4. While $c_{sv}(d)$ (Equation 3.3 [10]) maps the distance between two sample values to a confidence if these values are "similar", $c_{dv}(d)$ (Equation 3.4 [10]) is the fuzzy function for the confidence if the values are "different".

These confidence functions have also been originally defined for negative distances, but since the distance is defined as an absolute value, the function only needs to be considered for positive values. This is why the lines in Figure 3.4a are dashed for the negative x-axis. The definitions of $c_{sv}(d)$ and $c_{dv}(d)$ have been simplified to model the fuzzy function only for positive distances.



(a) c_{sv} and c_{dv}

(b) c_{ss} and c_{ds}

Figure 3.4: Confidence functions for matching data flow

$$c_{sv,i,j} = \begin{cases} 1, & \text{if } d_{i,j} \leq d_c \\ \frac{d_d - d_{i,j}}{d_d - d_c}, & \text{if } d_c < d_{i,j} < d_d \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

$$c_{dv,i,j} = \begin{cases} 0, & \text{if } d_{i,j} \leq d_c \\ \frac{d_{i,j} - d_c}{d_d - d_c}, & \text{if } d_c < d_{i,j} < d_d \\ 1, & \text{otherwise} \end{cases} \tag{3.4}$$

The fuzzy function $c_{ss}(k)$ (Equation 3.5 [10]) maps the number k of considered samples from the history to a confidence that the samples in the history seem to be in the "same set" as the new sample. More precisely, the algorithm tries to find the biggest set of distances that are all "similar" enough and therefore, the distances of the new sample $v_{i,new}$ and all history values $v_{h_{i,j}}$ have to be sorted in ascending order. The initial set size k starts with 1 but increases until the history fill level s_a. The bigger the set, the higher the confidence that the new sample matches the samples in the history and thus matches the signal state. The function $c_{ds}(k)$ (Equation 3.6 [10]) is the respective co-confidence function and yields the confidence that $v_{i,new}$ does not belong to the same set of distances ("different set").

$$c_{ss,i,k} = \begin{cases} 1, & \text{if } k \geq s_a \\ \frac{k}{s_a}, & \text{if } 0 \leq k < s_a \end{cases} \tag{3.5}$$

$$c_{ds,i,k} = \begin{cases} 0, & \text{if } k \geq s_a \\ \frac{s_a - k}{s_a}, & \text{if } 0 \leq k < s_a \end{cases} \tag{3.6}$$

After mapping the ascendingly sorted list of distances to the lists of confidences, $c_{sv,i,j}, j = 1..s_a$ will be sorted in descending order and $c_{dv,i,j}, j = 1..s_a$ will be still sorted in ascending order because the fuzzy function $c_{sv}(d)$ is falling monotonically while $c_{dv}(d)$ is monotonically rising. The conditions 3.7 and 3.9 are thus satisfied.

Equations 3.8 and 3.10 [10] show the definitions of $c_{b,i,k}$ and $c_{n,i,k}$ which are the confidences that

the new sample matches the subset or not.

$$c_{sv,i,j} \geq c_{sv,i,k} \quad \forall j \leq k \tag{3.7}$$

$$c_{b,i,k} = (\wedge_{j=1}^{k} c_{sv,i,j}) \wedge c_{ss,i,k} \tag{3.8}$$

$$c_{dv,i,j} \leq c_{dv,i,k} \quad \forall j \leq k \tag{3.9}$$

$$c_{n,i,k} = (\vee_{j=1}^{k} c_{dv,i,j}) \vee c_{ds,i,k} \tag{3.10}$$

Under the circumstances in which CCAM has been presented in various use cases, the algorithm can be simplified to be threshold-based rather than confidence-based. It can be argued that this reduction might not be possible to that extent when changing the definition of fuzzy functions used in the algorithm, however, in the way CCAM uses fuzzy logic, the following conditions that allow for a threshold-based implementation are all true:

- The co-confidence $\overline{c}(x_i) = 1 - c(x_i)$
- Operations on co-confidences are complementary
- Decisions in CCAM depend on comparisons like $c_z > \overline{c_z}$

As the confidence functions yield co-confidences that are the exact negated corresponding confidence, like $c_{dv} = \overline{c_{sv}}$, the resource-intensive fuzzification of co-confidences is redundant and can be replaced by an inversion of the confidence. Inversion of confidences can easily be implemented by bit-wise inversion (one's complement) with an unsigned numeric representation of confidences.

Equations 8+11, as well as 9+13, in the paper introducing CCAM [10], lead to 3.11 and 3.12. When comparing both equations, it becomes apparent that fuzzyAND and fuzzyOR operations are interchanged for the co-confidence $c_{n,i}$. This was referred to when calling the operations on co-confidences *complementary*.

$$c_{b,i} = \vee_{k=1}^{n}((\wedge_{j=1}^{k} c_{sv,i,j}) \wedge c_{ss,i,k}) \tag{3.11}$$

$$c_{n,i} = \wedge_{k=1}^{n}((\vee_{j=1}^{k} c_{dv,i,j}) \vee c_{ds,i,k}) \tag{3.12}$$

Adding the fact that DeMorgan's laws are applicable, inserting $c_{dv} = \overline{c_{sv}}$ and $c_{ds} = \overline{c_{ss}}$ yields $c_{n,i} = \overline{c_{b,i}}$ within a few steps. It will be shown that this does not only apply to the calculation of the confidence if a sample matches the state but also to all other occurrences of fuzzy algebra within the algorithm. Calculations with fuzzy algebra on co-confidences can thus be omitted completely in CCAM.

This has further consequences, as there is no typical *defuzzification* in CCAM but simple comparisons

if a confidence value is greater than its co-confidence value. The decision if a sample of a variable i is matching, depends on the outcome of $c_{b,i} > c_{n,i}$ and so it depends on a fixed confidence threshold:

$$c_{b,i} > c_{n,i} \Rightarrow c_{b,i} > 1 - c_{b,i} \Rightarrow c_{b,i} > 0.5 \tag{3.13}$$

All confidence functions in CCAM are defined as monotonically rising or falling. Therefore, a distinct mapping of the threshold for a confidence to a threshold for the original fuzzified metric is possible. Even c_{sv} and c_{dv}, although depicted as defined over negative distance values in Figure 3.4, are monotonic because the distance metric is an absolute value according to Equation 3.2. Therefore, a comparison against only one threshold is sufficient as a replacement for each fuzzification (mapping to confidence by applying a confidence function). The threshold for the distance metric, for example, is $d_t = c_{sv}^{-1}(0.5)$. The fuzzy algebra in CCAM, which involves partial sorting (minimum and maximum operations), ultimately reduces to much simpler Boolean algebra.

Although it might not be obvious in [10], the computation of $c_{b,i}$ requires ascending sorting of the distances $d_{i,j}$. With the reduction to Boolean logic, there is no need to calculate an exact confidence; it is just a decision flag. The confidence functions $c_{ss}(k, s_a)$ and $c_{ds}(k, s_a)$ (see Figure 3.4) also have their confidence threshold at 0.5. This translates to a threshold for k as $\frac{s_a}{2}$ because $c_{ss}(k, s_a) = \frac{k}{s_a}$. So more than half of the entries in the sample history have to be considered in order to get a $c_{ss,i,k} > 0.5$. Actually, the first $k_t = \lfloor \frac{s_a}{2} \rfloor$ values of $c_{sv,i,j}$ ($k = 1..k_t$) will not be considered for the result. As the distances are sorted ascendingly, these are the lowest distances and, respectively, the highest $c_{sv,i,j}$.

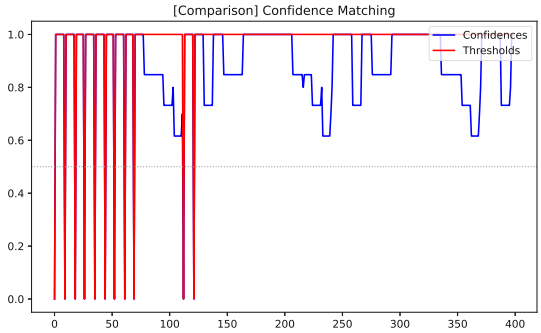
The condition that $c_{b,i} > 0.5$ can also be described the following way:

$$\forall k \in 1..s_a : \exists c_{b,i,k} = (\wedge_{j=1}^{k} c_{sv,i,j}) \wedge c_{ss,i,k} > 0.5$$
$$\forall k \in \lfloor \frac{s_a}{2} \rfloor + 1..s_a : \exists c_{b,i,k} = (\wedge_{j=1}^{k} c_{sv,i,j}) > 0.5$$
$$\forall k \in 2..s_a : c_{sv,i,k} > 0.5 \Rightarrow c_{sv,i,k-1} > 0.5$$
$$\forall k \in \lfloor \frac{s_a}{2} \rfloor + 1..s_a : \exists c_{b,i,k} > 0.5 \Rightarrow \forall j \in 1..k : c_{sv,i,j} > 0.5$$

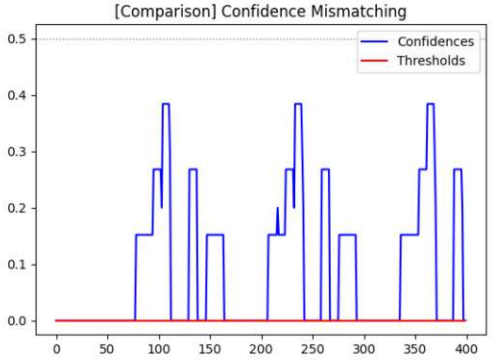
Considering the facts that $c_{sv,i,k}$ are sorted descendingly and that $k_t = \lfloor \frac{s_a}{2} \rfloor$, more than half of all $c_{sv,i,k}$ that are computed with the entries from the sample history have to match to fulfill the condition on $c_{b,i}$. Therefore, it is possible to eliminate the sorting and count the number of matching $c_{sv,i,k}$ instead. If more than k_t $c_{sv,i,k}$ match, the new sample will match the signal state.

Figure 3.5 shows an example run of the Signal State Detector on the voltage samples of the "15states4times" timeseries snippet on both the divisionless but not simplified version and the threshold-based implementation (called Threshold-based Context-Aware Monitoring (TCAM)). It can be seen

that if the confidence moves beyond 0.5, the binary confidence of TCAM also changes. The matching confidences and the mismatching confidence are the exact opposite of each other (complementing confidences) when they are computed but there is an exception where both confidences are zero. This happens when a new Signal State has been created. As there are no previous samples in that new state, the matching confidence cannot be calculated and is assigned to 0 by the control logic of the algorithm. The plots of the matching confidences usually do not show confidences below 0.5 because the plot always shows the last result of a detector execution with a new sample. When the Signal State is not matching anymore, a different but matching state will be entered (confidence above 0.5) or a new one will be created (confidence is exactly 0).



(a) Matching Confidence $c_{b,i}$



(b) Mismatching Confidence $c_{n,i}$

Figure 3.5: Confidence-based and threshold-based Matching Confidence

3.2.2 Validity Confidence Data Flow

The lowest of all $c_{sv,i,j}$, $c_{sl,i,new}$, is inserted into a Confidence History which stores the lowest sample value confidences from the n last detector executions. The Confidence History stores the same amount of elements as the Sample History. Instead of calculating the distances and performing the fuzzification again, this confidence can be acquired during the *matching confidence* calculation before. Analogously, the highest *mismatching confidence* $c_{dh,i,new}$ is inserted into a second Confidence History. After these new confidence values have been inserted into the histories, the minimum $c_{sl,i,j}$ and the maximum $c_{dh,i,j}$ are identified by iterating over these histories. A counter s_r that counts how many samples have been inserted into the Sample History since the last Signal State reentrance is fuzzified and combined with $(\wedge_{j=1}^{n} c_{sl,i,j})$, respective $(\vee_{j=1}^{n} c_{dh,i,j})$. All relevant equations are written below (3.14 - 3.17 [10]).

$$c_{sl,i,new} = (\wedge_{j=1}^{n} c_{sv,i,j}) \tag{3.14}$$

$$c_{dh,i,new} = (\vee_{j=1}^{n} c_{dv,i,j}) \tag{3.15}$$

$$c_{val,i} = (\wedge_{j=1}^{n} c_{sl,i,j}) \wedge c_{ss,i,s_r} \tag{3.16}$$

$$c_{inv,i} = (\vee_{j=1}^{n} c_{dh,i,j}) \vee c_{ds,i,s_r} \tag{3.17}$$

The confidence that a state is valid is called $c_{val,i}$ and if it is greater than the confidence that a state is invalid $c_{inv,i}$, the Signal State is considered *valid*. More precisely, the assertion of the valid flag of an execution t can be described as $valid_{i,t} = valid_{i,t-1} \vee (c_{val,i} > c_{inv,i})$.

As in the section about the *matching confidence* data flow, the relations $c_{dv}(d) = 1 - c_{sv}(d)$ and $c_{ds}(k) = 1 - c_{ss}(k)$ are assumed. Therefore, it can be quickly proven that $c_{sl,i,new} = \overline{c_{dh,i,new}} = (\wedge_{j=1}^{n} \overline{c_{dv,i,j}}) = (\wedge_{j=1}^{n} c_{sv,i,j})$ by applying DeMorgan's rules. Eventually, this is also true for $c_{val,i} = \overline{c_{inv,i}} = (\wedge_{j=1}^{n} \overline{c_{dh,i,j}}) \wedge \overline{c_{ds,i,s_r}} = (\wedge_{j=1}^{n} c_{sl,i,j}) \wedge c_{ss,i,s_r}$.

This shows that the co-confidence calculations are also redundant in the Validity Confidence Data Flow. Furthermore, it can be adapted to a threshold-based datapath, too. The Boolean variable $c_{sl,i,new,binary} = \wedge_{j=1}^{n}(d_{i,j} > d_{i,t})$ is 0 if one of all distances is less than or equal to the threshold $d_{i,t}$. The Confidence History would store up to n of these values in a FIFO fashion and as long as there is a '0' in the FIFO, the validity confidence bit is also '0'. This behavior could also be performed with a shift register or a binary counter instead of iterating over that history. The code in Listing 3.3 sets a counter value if a '0' should be inserted and decreases the counter every detector execution that inserts a '1'. The variable *vac* corresponds to $c_{ss,i,s_r,binary} = s_r > \frac{n}{2}$.

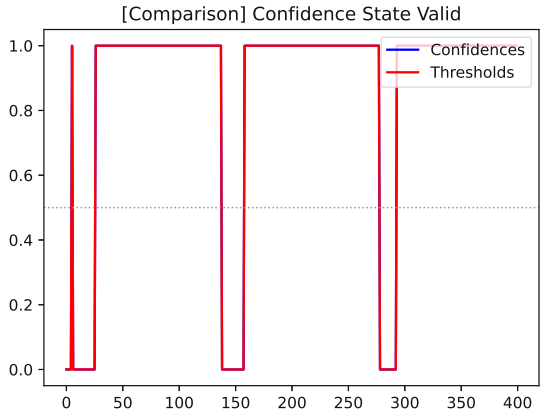
Listing 3.3: Simplified Validity Confidence Code

```
vac = 1 if (self.numberOfInsertedSamplesAfterReentrance > (maxHistorySize//2)) else 0
validityConfidence = 1 if (self.invalidityCounter == 0 and vac == 1 and lowestConfidence
    == 1) else 0

if lowestConfidence == 0:
    self.invalidityCounter = maxHistorySize - 1
elif self.invalidityCounter != 0:
    self.invalidityCounter = self.invalidityCounter - 1

return validityConfidence
```

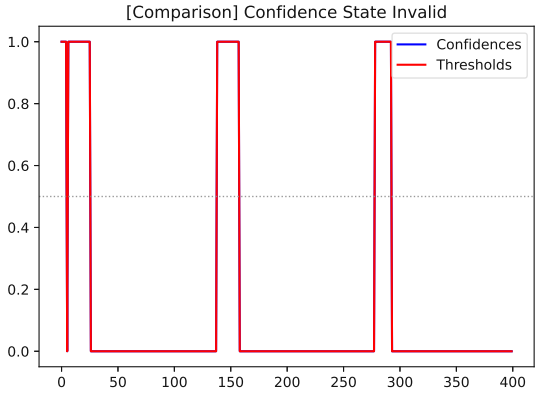
Still, the counter value has to be stored for each Signal State, but the memory requirements have been reduced to a counter value with a bit width of $\lceil log_2(n) \rceil$ instead of n memory bits or even more when using confidences ($n * C$ bits, when C is the confidence bit width.) The runtime for this calculation also

reduced from $O(n)$ to $O(1)$. However, the validity confidence $c_{val,i,binary}$ is mainly necessary for the decision if a state is valid and a valid state cannot become invalid. Thus, loading the *invalidity counter* of a re-entered Signal State from memory does not have an influence on the validity of a state anymore. As a result, the counter could just be reset when a state reentrance occurs which saves $\lceil log_2(n) \rceil * |States|$ bits of memory. The only diverging effect of this change is that the flag *validAfterReentrance* could be set at most $\lfloor \frac{n}{2} \rfloor$ executions earlier.

Figures 3.6 and 3.7 show the comparison of the validity confidence in CCAM and TCAM on different timeseries snippets.



(a) Validity Confidence $c_{val,i}$



(b) Invalidity Confidence $c_{inv,i}$

Figure 3.6: Confidence-based and threshold-based Validity Confidence ("severalChanges" snippet)

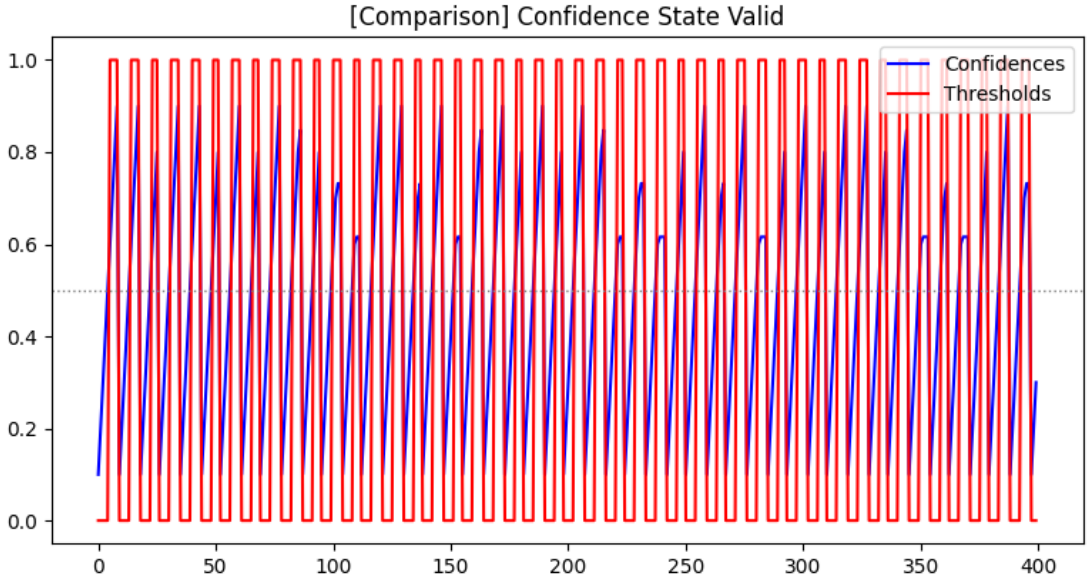


Figure 3.7: Validity Confidence $c_{val,i}$ ("15states4times" snippet)

3.2.3 Stability Confidence Data Flow

To obtain a stability confidence, the new sample is inserted into a Discrete Average Block (DAB). The DAB is a history with a fixed length that does not accept values once it is full. Instead, a new DAB will be filled. As only the first and the most recent DAB are relevant for the algorithm, it is sufficient to provide only memory for two DABs. When the latest DAB is full, the mean is calculated, as well as the mean of the first DAB. If the size of a DAB is defined as a power of two, the division necessary for the mean can be implemented as a shift. Then the distance metric (Equation 3.18 [10]) is derived from those two mean values, similar to the Matching Confidence data flow, however, the fuzzy function for the stability might be a bit different. To again avoid divisions, the distance metric has been changed to Equation 3.19.

$$d_{dft} = \left| \frac{v_{avg,1} - v_{avg,2}}{v_{avg,1}} \right| \tag{3.18}$$

$$d_{dft} = |v_{avg,1} - v_{avg,2}| \tag{3.19}$$

The fuzzy function $c_{stb,i}$ (Equation 3.20 [10]) yields the confidence that the observed signal is stable, whereas and $c_{dft,i}$ (Equation 3.21) is the co-confidence that expresses if the signal is drifting.

$$c_{stb,i} = \begin{cases} 1, & \text{if } d_{dft,i} \leq d_{c,dft} \\ \frac{d_{d,dft} - d_{dft,i}}{d_{d,dft} - d_{c,dft}}, & \text{if } d_{c,dft} < d_{dft,i} < d_{d,dft} \\ 0, & \text{otherwise} \end{cases} \tag{3.20}$$

$$c_{dft,i} = \begin{cases} 0, & \text{if } d_{dft,i} \leq d_{c,dft} \\ \frac{d_{dft,i} - d_{c,dft}}{d_{d,dft} - d_{c,dft}}, & \text{if } d_{c,dft} < d_{dft,i} < d_{d,dft} \\ 1, & \text{otherwise} \end{cases} \tag{3.21}$$

Also the flag $isStable = c_{stb,i} > c_{dft,i} = c_{stb,i} > 0.5$ can be assigned by comparing the distance between the means with a drift threshold.

As the removal of the division from the distance metric has the disadvantage to make the algorithm more prone to noise, especially the results of the drift detection became worse. Some systems experience signal drift at their inputs, whereas others do not require a drift detection at all. Therefore, the drift detection was assessed as an optional feature of CCAM. An implementation of a reliable drift detection on NEMS has been defined as future work.

3.2.4 Aggregation of Confidences

The System State Detector receives also the matching confidence, the validity confidence and the stability confidence from all Signal State Detectors and their co-confidences as well and uses them to aggregate them to respective confidences for the System State Detector (Equations 3.22-3.27 [10]). The number of observed SuO signals is m.

$$c_b = \wedge_{i=1}^{m}(c_{b,i}) \tag{3.22}$$

$$c_n = \vee_{i=1}^{m}(c_{n,i}) \tag{3.23}$$

$$c_{val} = \wedge_{i=1}^{m}(c_{val,i}) \tag{3.24}$$

$$c_{inv} = \vee_{i=1}^{m}(c_{inv,i}) \tag{3.25}$$

$$c_{stb} = \wedge_{i=1}^{m}(c_{stb,i}) \tag{3.26}$$

$$c_{dft} = \vee_{i=1}^{m}(c_{dft,i}) \tag{3.27}$$

Thus, the System State Detector needs to calculate minimum and maximum operations. If the Signal State Detectors are threshold-based and provide only 1 bit variables, they become Boolean operations. Even when the Signal State Detectors are still confidence-based, they can forward the decision flags instead of the confidence values, so that the System State Detector operates on binary "confidences". However, these aggregated confidences are only needed for the computation of the *overall confidence*.

3.2.5 Functioning Confidence Data Flow

The data flow of the functioning confidence c_{ok} is mostly part of the control flow than a separate data flow. A System State can be defined as a mapping of Signal States of all observed SuO signals (Equation 3.28). To check against the assumed bijective function property of the SuO, the mapping is separated in a SuO input and a SuO output group (Equations 3.29 - 3.32).

$$sysstate_t = (id_{sigstate,1,t}, ..., id_{sigstate,m,t}) \tag{3.28}$$

$$sysstate_{inputs,t} = (id_{sigstate,1,t}, ..., id_{sigstate,m_{inputs},t}) \tag{3.29}$$

$$sysstate_{outputs,t} = (id_{sigstate,m_{inputs}+1,t}, ..., id_{sigstate,m,t}) \tag{3.30}$$

$$match_{inputs} = (sysstate_{inputs,t} == sysstate_{inputs,t-1}) \tag{3.31}$$

$$match_{outputs} = (sysstate_{outputs,t} == sysstate_{outputs,t-1}) \tag{3.32}$$

In case of a mismatch of the System State, all stored System States are compared against the tupel of current Signal State IDs and the matching System State will be entered. If there is no matching System State but a *potential* System State, the potential System State will be entered. A potential System State matches either with all Signal State IDs from the input or output group. Only when there exists no potential state, a new System State will be created. A counter increments for every subsequent execution of the System State Detector in which a *potential* state is active. If a matching state or a newly created state is entered, this *disparity counter (dc)* is reset to zero. The resulting counter value is fuzzified and provides the functioning confidence c_{ok} (Equation 3.33 [10]) and its co-confidence, the broken confidence c_{brk} (Equation 3.34 [10]). This means that the disparity counter contains the same information as the functioning confidence. The decision $(c_{ok} > c_{brk}) = (c_{ok} > 0.5)$ can be further simplified to $(dc_t > \frac{dc_a}{2})$.

$$c_{ok} = \begin{cases} 0, & \text{if } dc_t \geq dc_a \\ \frac{dc_a - dc_t}{dc_a}, & \text{if } 0 \leq dc_t < dc_a \end{cases} \tag{3.33}$$

$$c_{brk} = \begin{cases} 1, & \text{if } dc_t \geq dc_a \\ \frac{dc_t}{dc_a}, & \text{if } 0 \leq dc_t < dc_a \end{cases} \tag{3.34}$$

3.2.6 Overall Confidence Data Flow

The overall confidence provides an estimation how confident CCAM is about its estimations overall. Therefore, it combines the aggregated matching confidence c_b, validity confidence c_val, stability confidence c_{stb} and the functioning confidence c_{ok} as well as some of their co-confidences.

Until the two DABs of a Signal State have not been filled, there is no meaningful stability and drifting confidence available and the flag *stability_known* states that. This and other flags will choose the *State Condition* of the System State:

- UNKNOWN: At least one signal stability is unknown
- STABLE: All signals are stable ($\forall i : c_{stb,i} > c_{dft,i}$)
- DRIFTING: At least one signal is not stable ($\exists i : c_{stb,i} \leq c_{dft,i}$)
- MALFUNCTIONING: ($c_{ok} \leq c_{brk}$)

Depending on the State Condition, the overall confidence will be calculated differently as shown in Equation 3.35 [10].

$$c_{all} = \begin{cases} 0, & \text{if } UNKNOWN \\ (c_b \vee c_n) \wedge c_{ok} \wedge c_{stb} \wedge c_{val}, & \text{if } STABLE \\ c_b \wedge c_{dft} \wedge c_{val}, & \text{if } DRIFTING \\ c_n \wedge c_{brk} \wedge c_{val}, & \text{if } MALFUNCTIONING \end{cases} \tag{3.35}$$

3.2.7 Numeric representations

To avoid computationally costly floating-point arithmetics in hardware, integer or fixed-point representations are desired. Regarding the numeric representation of sample values, it has to be differentiated between a representation of the voltage the ADC measured at its input or a representation of the metric that is measured by the sensor (e.g. temperature in $°C$). As the samples are most likely directly from ADCs without any conversion, they are assumed to be unsigned integers. However, the dataset of the hydraulic system [18] contains samples in signed floating point format and each metric has a different range. For the conversion to an unsigned binary representation, the following formulas are used (3.36-3.38):

$$v_{clamped} = min(max(v_{real}, a), b) \tag{3.36}$$

$$\Delta = \frac{b-a}{2^n - 1} \tag{3.37}$$

$$v_{binary}(v_{real}, a, b, n) = \lfloor \frac{v_{clamped} - a}{\Delta} \rceil \tag{3.38}$$

$$Q(v_{real}, a, b, n) = v_{binary} * \Delta + a \tag{3.39}$$

The parameters are the lowest representable value a, the highest representable value b and the targeted bit width n. At first, the real number v_{real} is clamped to ensure that it lies within the representable range. Then the quantization step size Δ and the converted binary representation v_{binary} are calculated.

To get the quantized real numbered representation of the binary value, formula 3.39 can be utilized.

This approach can also be applied on confidence values. The defined range of confidence values is $D = \{x \in \mathbb{R} | 0 \leq x \leq 1\}$, so this range is mapped to an unsigned integer or fixed-point number with a certain bit width C. Figure 3.8 shows an example of the mapping to binary confidences. Note that the values of the binary confidences are given in decimal representation but are actually bit vectors with bit width C. The binary confidence values and their assigned real numbered range are listed in the table in figure 3.9b.

The presented binary representation minimizes quantization errors and keeps the decision boundary

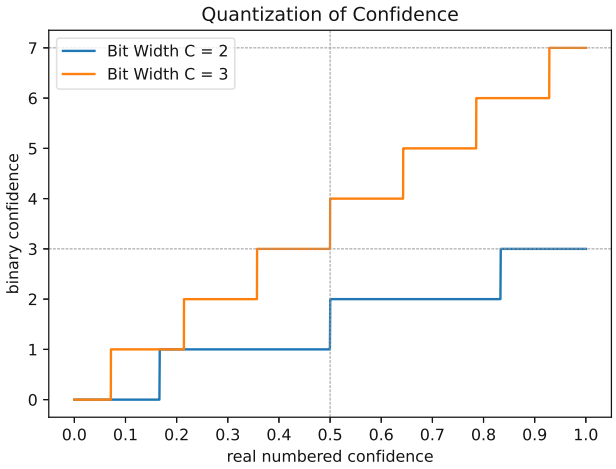


Figure 3.8: Quantization Steps of Confidence

for confidences around 0.5 precise, however, it has to be mentioned that it is **not** a fixed-point format but a mapping to integers. An appropriate fixed-point representation would be a $UQ0.n$ format. The Q notation $Qm.n$ means that the fixed-point number has m integer bits and n fractional bits. The U stands for *unsigned*. The table in figure 3.9a shows the usage of the UQ0.2 format for confidences with bit width 2.

2^{-1}	2^{-2}	Decimal	Range
0	0	0.0	0.0 - 0.249
0	0	0.25	0.25 - 0.499
1	0	0.5	0.5 - 0.749
1	1	0.75	0.75 - 1.0

(a) Fixed-point format UQ0.2

Binary	Decimal	Range
00	0.0	0.0 - 0.166
00	0.333	0.166 - 0.499
10	0.666	0.5 - 0.833
11	1.0	0.833 - 1.0

(b) Using quantization formulas 3.36-3.39

Figure 3.9: Binary confidence representations with bit width 2

3.3 Control Flow

The version of the code at the time of [10] might have had no clear separation between Signal State Detector and System State Detector, the available code of the implementation inside RoSA, however, had. Instead of using the flow graph shown in [10], the control graphs in this chapter have been derived from the code of the CCAM implementation in Python.

3.3.1 Signal State Detector Control Flow

The control flow of the Signal State Detector is depicted in Figure 3.10. Compared to the RoSA implementation, it does not have global variables whose modification could be confusing, but just local

variables and class properties. By several tests, it has been checked that the unmodified Python algorithm and the RoSA implementation provide the same outputs for the same input time series.Under these circumstances, the Python code has been taken as a template for the analysis.

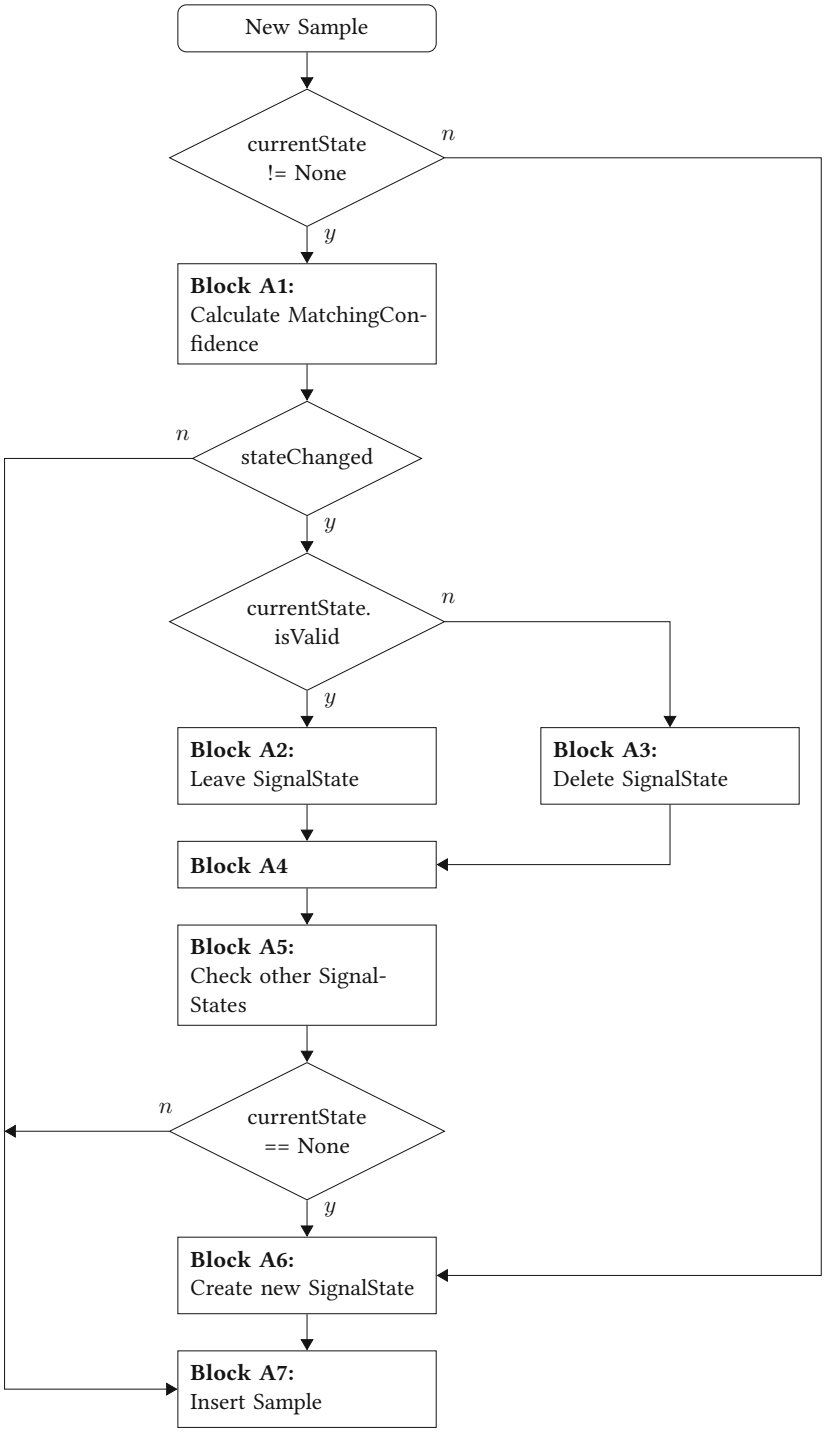


Figure 3.10: Control Flow of Signal State Detector

The descriptions of the code blocks in Figure 3.10 give a grasp of what the algorithm does in these blocks. In Block A7, however, the sample is not only inserted into the history of the active Signal State

but also calculates the Validity Confidence to assess if the Signal State becomes *valid* and the Stability Confidence to assess if the observed signal is drifting.

In order to derive a Finite State Machine(FSM) from the given control flow diagram, code blocks could be translated to possible FSM states whereas decisions form transitions between states. A notable difference is that an FSM for hardware should not remain in an accepting state during normal operation and that time constraints have to be considered. The basis design principle is as follows. The FSM starts in a state that waits for a new sample as input and changes to a different state when a sample arrives. When all necessary computations have been performed, the hardware is ready again and enters the initial state again.

When identifying the necessary states of an FSM, there has to be an arguable reason why states cannot be combined into a bigger one and why a decision requires the transition to another state. Thus, data dependencies within the algorithm have to be resolved and described as time constraints in discrete time steps. Changing from one state to another takes one discrete time step Δ, so a state transition helps to meet some of these constraints. If the time steps between two actions have to be longer than 1 Δ, such a constraint has to be included in the transition condition. The following data dependencies have to be considered:

- Read-After-Write (RAW)
- Write-After-Read (WAR)
- Write-After-Write (WAW)

While in ordinary sequential logic with flip-flops such dependencies usually need just 1 Δ between the accesses of the particular variable, memory access and function calls may need more time steps before finishing. Depending on the interface of the memory, the memory access has an effect on the design of the FSM.

State machines in CCAM have been designed as Mealy FSMs with additional stateful variables to store data. Figure 3.11 shows the FSM of the Signal State Detector. It has exactly 8 states and thus utilizes the bit width of the FSM state register efficiently. To get a clear arrangement, the FSM states and input variables have been replaced by shorter abbreviations. Table 3.1 and 3.2 show the mapping of these abbreviations to their more meaningful names.

3.3.2 Signal State Flag Simplification

After the validity and invalidity confidences have been calculated, the code listed in Listing 3.4 from the original CCAM implementation sets the state flags that are needed for further decisions of the algorithm.

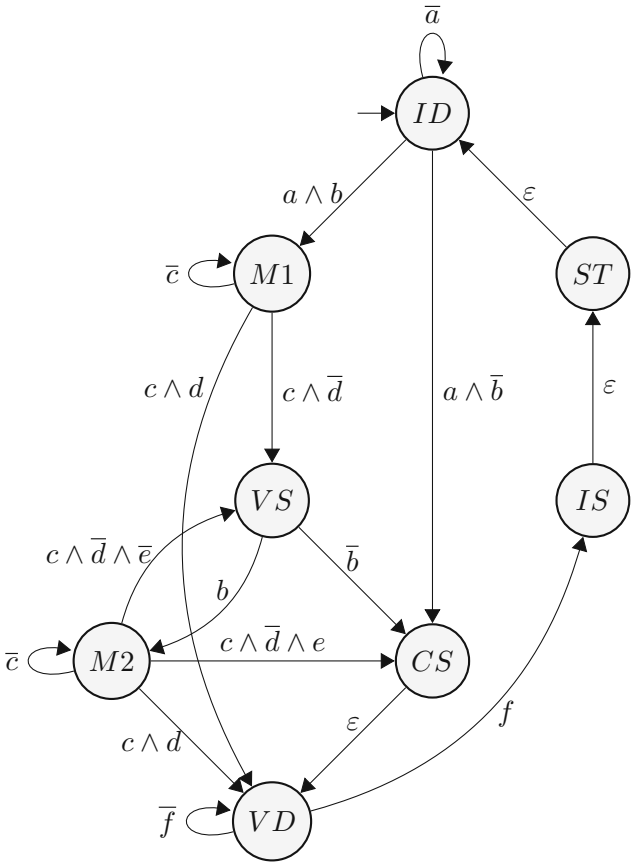


Figure 3.11: State Diagram of Signal State Detector FSM

Table 3.1: FSM Variable Translation

a	sample_valid
b	slot_used
c	md_valid
d	match
e	last_addr
f	vd_valid

Table 3.2: FSM State Abbreviations

ID	Idle
$M1$	MD1
VS	Visit State
$M2$	MD2
CS	Create State
VD	VSD
IS	Insert Sample
ST	Store

Listing 3.4: Signal State Flag Assertions (Original)

```
if (self.currentState.isValid):
        self.currentState.justGotValid = False

    if (self.currentState.validityConfidence > self.currentState.invalidityConfidence):

        if (not self.currentState.isValid):
            self.currentState.justGotValid = True

        self.currentState.isValid = True
        self.currentState.isValidAfterReentrance = True
```

This code can be reformulated so that it does not contain if-statements as shown in Listing 3.5.

Listing 3.5: Signal State Flag Assertions (Simplified)

```
newIsValid = self.currentState.validityConfidence > self.currentState.invalidityConfidence
self.currentState.justGotValid = (not self.currentState.isValid) and (newIsValid or self.
    currentState.justGotValid)
self.currentState.isValid = self.currentState.isValid or newIsValid
self.currentState.isValidAfterReentrance = self.currentState.isValidAfterReentrance or
    newIsValid
```

The values of these flags at the beginning of the shown code block have been assigned in the previous execution of the Signal State Detector. As these flags are properties of the currently active Signal State, they need to be stored across state changes and should be stored in a memory for state properties (called *Signal State Memory* in hardware). However, some assumptions can be made about the initial flag values.

According to the code listing above, a Signal State is considered *valid* if it has been valid before or when the Validity Confidence is high enough. When a Signal State is exited, it is deleted in case it is not valid. Therefore, we can assume that every old state that is re-entered has already been valid before. Instead of storing the *valid* flag beyond state changes, it is sufficient to store it only for the currently active Signal State.

Similarly, the flag *validAfterReentrance* is set to *false* when a Signal State is exited and set if the Validity Confidence is high enough. So this flag also does not need to be stored for inactive states as it is known to be *false* for them. The flag *justGotValid* should detect the execution step when a state becomes valid as the name suggests. As it is *false* for all Signal States that already have been valid before, it will be so for all re-entered states, too. Summing up, all of these three flags does not require to be stored per Signal State, reducing the amount of required memory.

3.3.3 System State Detector Control Flow

The control flow of the System State Detector in Figure 3.12 is very similar to the control flow of the Signal State Detector (Figure 3.10). Instead of executing the algorithm when a new signal sample arrives, the System State Detector is evoked when new Signal State information is available for all monitored signals (from all Signal State Detectors).

In Block B1, all saved Signal State IDs are compared with the updated Signal State IDs. The result is a relation enum with four possible values:

- MATCH: all Signal State IDs are matching
- ONLYA: only all Signal State IDs in group A are matching
- ONLYB: only all Signal State IDs in group A are matching

- MISMATCH: mismatches in both groups

Signal groups A and B are just more general group names for what would be defined as "Inputs" and "Outputs" signal groups in a typical CCAM use case.

If the comparison of a System State with the updated Signal State IDs yields an ONLYA or ONLYB relation, the System State will be classified as a "potential" state. The algorithm will enter a potential System State if and only if the current state is not matching, there is no matching state in the list of System States and there is at least one potential state.

The state diagramm of the derived FSM is provided in Figure 3.13. As the algorithm also looks for *potential System States*, the transitions $T2 \rightarrow PP$ and $VS \rightarrow PP$ are added compared to the Signal State Detector FSM. Furthermore, the check if a System State is matching is much easier and can be done in one cycle, removing the loop transistions of the states $T1$ and $T2$. See Table 3.3 and 3.4 for the meaning of the abbreviated states and variables.

Table 3.3: FSM Variable Translation

a	sig_state_ids_valid
b	slot_used
c	group1_matching
d	group2_matching
e	last_addr
f	potential_state_found

Table 3.4: FSM State Abbreviations

ID	Idle
$T1$	Test IDs 1
VS	Visit State
$T2$	Test IDs 2
CS	Create State
IS	Insert SignalStateInfo
PP	Postprocessing
OP	Output

3.4 Memory Management

As the usage of object-oriented data structures in the original C++ or Python code obfuscates the time window in which data has to be held in registers or memories, the first step in identifying necessary memory would be to reduce the scope of variables in the code as much as possible.

A rough heuristic would be that class properties are likely to require memory as the content of these variables depends on the particular instance of that class, whereas local variables in sequential functions often resolve to registers or even wires. This has led to the simplifications that have already been discussed in the previous sections.

The second set of tasks that deal with the memory topic is about designing circuitry for accessing the memory. Signal state or system state-related data will require a memory that is accessed with an address that is linked to that corresponding state. Most memory data is signal state-dependent and there is no requirement to have a common shared memory between different instances of the signal state

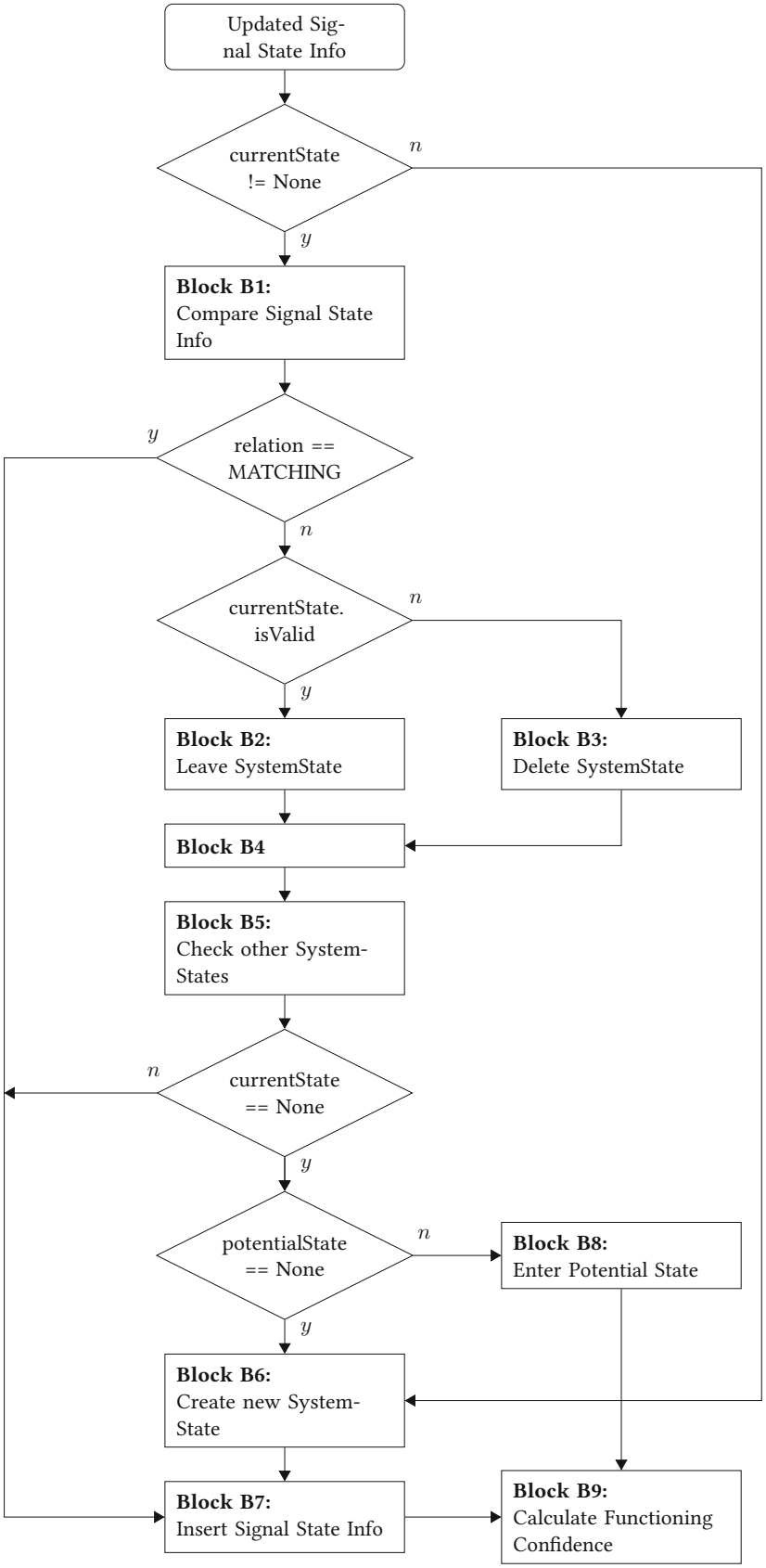


Figure 3.12: Control Flow of System State Detector

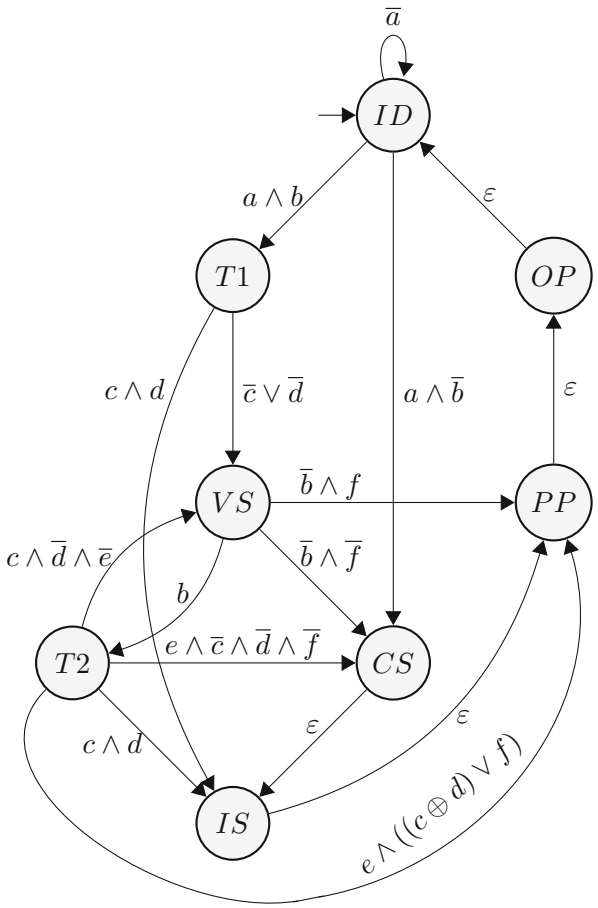


Figure 3.13: State Diagram of System State Detector FSM

detector or the system state detector, so the easiest addressing would be counting up the needed state in a simple enumeration from 0 to the maximum possible state ID. CCAM also deletes Signal and System States, therefore the address range may be smaller than the range of possible state IDs.

In the original C++ and the equivalent Python code, some data structures include dynamic lists.

Relevant lists are the list of samples in the sample history, the list of confidences in the confidence history, and the discrete average block (DAB) for the drift detection as well as the lists of signal and system states.

On the hardware side, lists with dynamic lengths would require dynamic memory allocation in combination with a big shared memory. This would lead to a too-high complexity for a minimal implementation of the algorithm. The simpler approach would be defining maximum list lengths with reasonable bounds.

The histories and the DAB have defined a fixed length anyway, but the list of detected states is supposed to grow over time. Depending on the use case, the amount of possible states could be estimated.

If the memory is too small to fit the amount of detected states, CCAM cannot operate reliably

anymore. A clever replacement policy could increase the possibility that CCAM still works reasonably with too many detected states, however, this operation mode should be avoided as proper operation can not be guaranteed anymore. In general, it is suggested to define the maximum list length of the CCAM Signal or System States list according to the expected number of states during a long-term operation.

3.4.1 State Slots

Figure 3.14 shows a visualization of the structure of a double-linked list like it is probably used in the high-level code. In each list element there is not only the content stored but also references (pointers) to the previous and next elements in the list. These references are essentially already real memory addresses when there is no paging or sophisticated memory management as in modern computers. However, the start and end of the list should be distinguishable by storing a NULL pointer in the previous or next address fields. Therefore, the address 0 or any other address that is used to work as a NULL pointer could not be in the range of the addressable memory, so this requires a bigger address range than the addressable range for the actual memory.

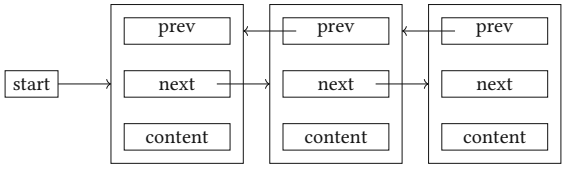


Figure 3.14: Double-Linked List

List iteration in the CCAM algorithm is always one-way, which could also be implemented with a single-linked list without a reference to the previous list element if no elements are deleted. A deletion operation would require the modification of the *next* reference of the previous list element and either there is a *prev* pointer as in the double-linked list, or the algorithm has to remember the address of the previous element somehow. Indeed, in CCAM the current Signal or System State could be deleted from the memory and thus from the list of detected states.

A hardware realization would need additional memory space to fit all the references and the actual memory allocation that selects addresses for empty space in the memory had still to be added, too. It becomes apparent, that this approach is not the right choice for an area-minimized ASIC for a small algorithm.

Therefore, a simpler approach has been chosen. Lists have a defined maximum length and iteration is implemented by counting up the memory address. More precisely, a **read iteration** over the list starts at address 0 and counts over the whole address space of the list. There is a flag for each position, if it is empty or not (*slot_used*). For **appending** a new element to the list, the list positions are iterated from the

beginning until an empty position has been found. Then the data is stored and the flag is set. **Deletion** only resets the flag. The addition of a new list element is depicted in Figure 3.15 for a maximum list length of 8. Empty spaces between list elements may come from previously deleted elements and this is also the reason to iterate over the whole address range during a read iteration as the first encountered empty position might not guarantee that all elements have been read.

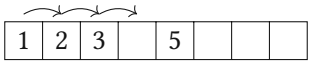


Figure 3.15: Allocating space for a new element

CCAM creates a new state after it has iterated over all detected states in its state list and no state has been fitting. So insertion into the list can only occur after a read iteration and thus, the FSM for the memory interaction can already select an empty slot during the read iteration and if a new state has to be created, the address for the insertion is already known.

3.4.2 State ID

Each time a Signal or System State becomes *valid* for the first time (the flag *justGotValid* is true), the State ID is incremented. Therefore, for each created state that became valid, there is a State ID in the system. As only states are deleted that did not become valid, there are no empty state slots between used ones in the memory. The result is that the state address of a newly created state (starts with 0) is always the State ID minus one ($addr = id - 1$). Under this circumstance it is unnecessary to save the State ID of a state in a memory because the address of the state slot can be treated as the ID. Moreover, the hardware implementation of CCAM should start the State IDs at 0 anyway in order to use the counter and memory bit width more efficiently. The calculation of the flag *justGotValid* also becomes redundant.

3.5 Summary

In this chapter, the CCAM algorithm has been described in detail. After a brief introduction, the test environment is presented before the examination of the algorithm starts. At first, the data flow is presented using the same nomenclature as in [10] and possible simplifications are suggested. Besides the omission of co-confidence calculations, these include the reduction of all data flows to a threshold-based version. In a next step, the control flow is described, showing control flow diagrams directly derived from the code and state diagrams for hardware implementations. Furthermore, it is shown that various state flags don't need to be stored per Signal State. Last but not least, the memory management for CCAM is covered. As CCAM changes between detected states, state-related data has to be stored accordingly

to prevent data loss. Due to the simplistic memory management, the state IDs are equal to the state addresses and thus, the state IDs do not have to be stored per Signal or System State.

Chapter 4

Exploration of Basic Components

In this chapter, different implementations of relevant building blocks for a CCAM design are compared by their NEM device count and the estimated (worst case) clock cycle count. Although 3-T, 4-T and 7-T switches have different dimensions, the device count of all switch types has been combined for the sake of simplicity. The device count for a logic circuit correlates with the total area.

4.1 Prerequisites

All Verilog modules are synthesized with the synthesis tool Genus (Genus(TM) Synthesis Solution, Version 21.10-p002_1) which is part of the Cadence toolset. The logic gates used for the synthesis were taken from the *liberty file* which is provided in the i-EDGE Process Development Kit (PDK). The PDK has been created by the project members at the University of Bristol and includes schematics, layouts and simulation files for NEM switches, standard cells, basic memory cells and some arithmetic cells. A detailed description of the PDK and thus also a list of the contents of the standard cell library is provided in [2]. The relevant parts are described in the following subsections.

4.1.1 Logic Gates and Flip-Flops

In order to ensure comparable synthesis results, all designs have been synthesized with the same liberty file. As the list of the NEM standard cell library in [2] is not identical to the list of gates in the liberty file, all gates defined in the liberty file are listed in Table 4.1. The 4-T implementations of the MUX, XOR and XNOR gates and the device count of all gates have been added to the table as additional information. Also, note that the naming convention of these cells has changed in the PDK. The 4-T implementations of the XOR and the XNOR gate consist of only two 4-T switches, however, they need an additional inverted input (Figure 4.1) and the synthesis tool does not understand the requirement of complementary inputs for a logic gate, despite the existence of a *pin_opposite* attribute in the Liberty Reference Manual [31].

Therefore, an inverter is included in these gates to provide these inverted inputs, which increases their device count to 4.

(a) XOR gate (b) XNOR gate

Figure 4.1: Gates with a complementary binary input

Table 4.1: NEM standard cells in the liberty file [2]

Cell	Device Count	Height in μm	Width in μm
inv_3T	2	80	14.5
nand_3T	4	80	34.4
and_3T	6	80	48.9
nor_3T	4	80	34.9
or_3T	6	80	49.4
xor_3T	12	80	97.8
xnor_3T	12	80	97.8
xor_4T	4	80	42.7
xnor_4T	4	80	42.7
mux_3T	12	80	100
mux_4T	2	80	28.2
D_FF	26	80	250.7
D_FF_rst	38	80	353.3

The difference between the D-flip-flops (D-FFs) D_FF and D_FF_rst is the asynchronous reset of the latter. The D_FF design does not have a reset input but can be turned into a D-FF with a synchronous reset (SR) by adding some gates or a multiplexer in front of the D input. An additional enable input (EN) can be implemented by another multiplexer. Figure 4.2 shows a D_FF design that has been extended with both, an SR and EN input.

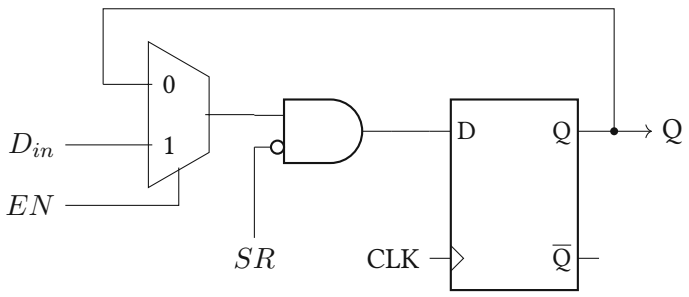


Figure 4.2: D flip-flop extended with EN and SR inputs

4.1.2 Memory

Memory blocks cannot be synthesized with the normal synthesis flow and have to be layouted by hand as long as no memory creation tool flow is set up. With the memory cells in the PDK (Table 4.2) unclocked registers, LUTs and RAMs can be constructed. All NEM memory cells are non-volatile because they are built from 7-T devices.

Table 4.2: NEM memory components [2]

Cell	Device Count	Height in μm	Width in μm
Wordline Driver (WLD)	6	80	49.9
RAM-U cell	5	66.9	43.9
LUT-U cell (min)	3	60	33.8
LUT cell	5	80	55.2

The LUT cell with only three devices has the shortcoming that the data input needs to be driven with a logical '1' in read mode. Therefore, it has been extended by two additional switches to automatically fulfill this requirement when *write* is set to '0'. Besides a LUT, the LUT cell also allows us to build an unclocked register as shown in Figure 4.3. Equation 4.1 gives the device count of such a register.

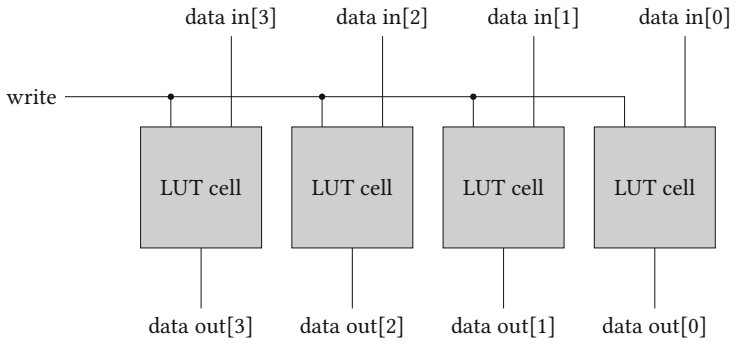


Figure 4.3: Unclocked NEM register [2]

$$D_{Reg}(n) = n * D_{LUT\_Cell} \tag{4.1}$$

The disadvantage of this register is that the data output during a write operation is not defined and a floating voltage. Therefore, it can only be used when write and read operations never occur in the same clock cycle.

Figure 4.4 shows the structure of a LUT. If the output of the LUT should have more than one bit, the LUT structure could be duplicated and placed in parallel. The usage of this LUT design for long output vectors, however, is not recommended as the parallel data input is impractical for the configuration of such a LUT with a large amount of data.

Formula 4.2 calculates the device count of a LUT with an address of bit width k, $n = 2^k$ possible

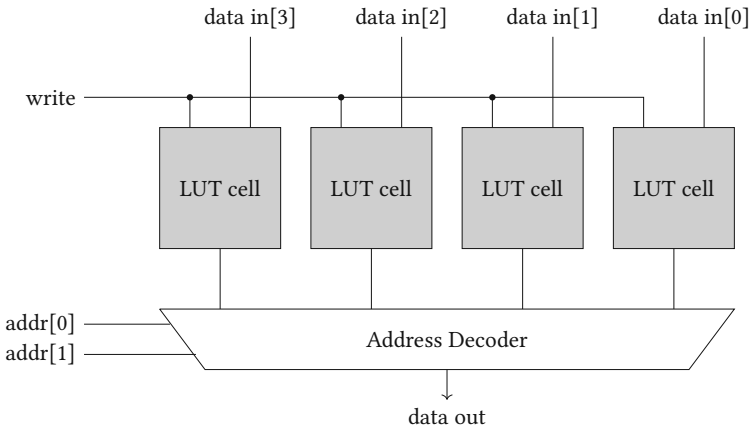


Figure 4.4: NEM LUT structure [2]

input combinations, and a 1-bit output.

$$D_{LUT} = 2^k * D_{LUT\_Cell} + D_{Addr\_Decoder(k)} \tag{4.2}$$

Figure 4.5 depicts an example of a NEM RAM with a size of 4 by 4 cells. Data is written and read in serial (element-wise but not bit-serial). The word line driver (WLD) converts the select signal of the row and the read/write signal to the word lines that are needed by the RAM cells.

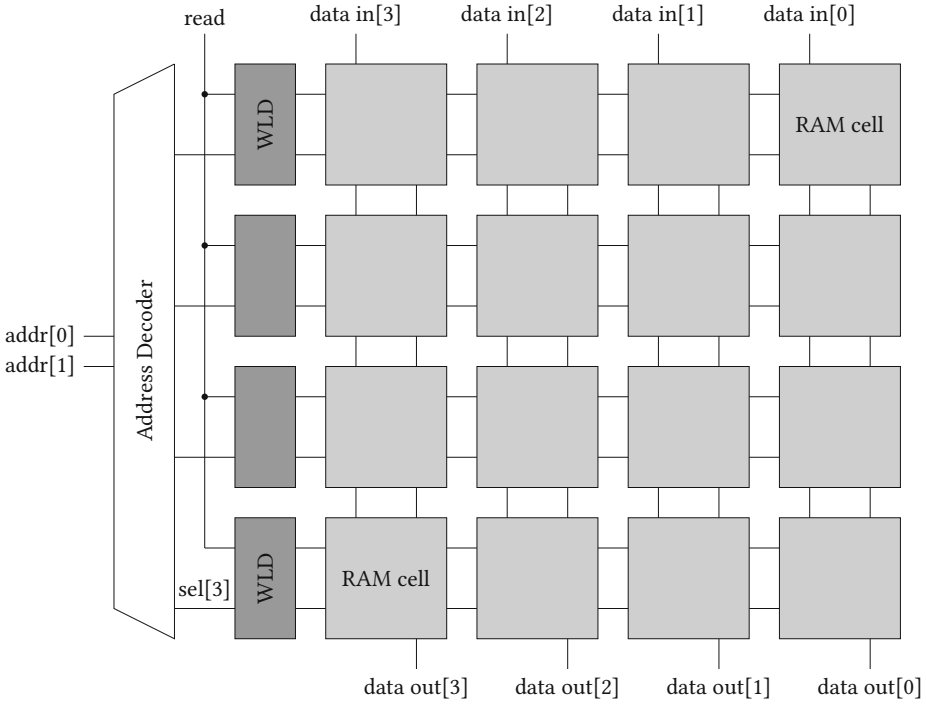


Figure 4.5: NEM RAM structure [2]

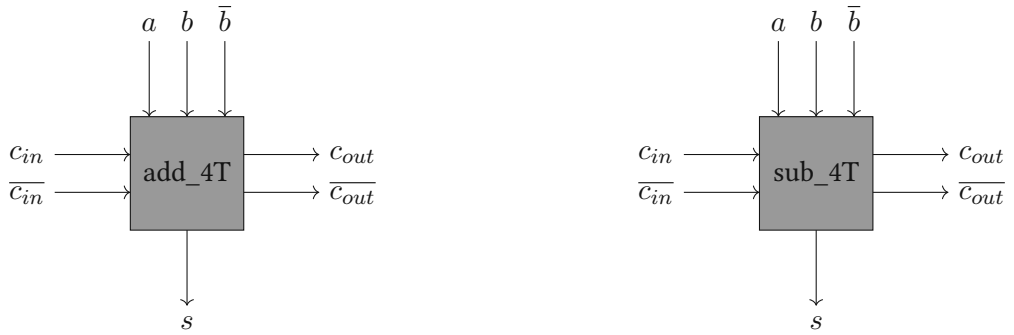
The device count of a RAM with m rows and n bits per row can be calculated as shown in Formula 4.3. A table for the device count of a binary decoder for different bit widths is derived from synthesis

results and will be given later.

$$D_{RAM}(m,n) = m * n * D_{RAM\_Cell} + m * D_{WLD} + D_{Binary\_Decoder(m)} \tag{4.3}$$

4.1.3 Arithmetic Blocks

Optimized designs for a ripple carry adder and a subtractor have already been available in the PDK but they also require additional inverted inputs and could thus not be added to the liberty file. Figure 4.6 shows the adder and subtractor designs as black boxes. Both blocks have a NEM device count of 6.



(a) Full adder (b) Full subtractor

Figure 4.6: Symbols for 4-T optimized adder and subtractor

The *borrow in* and *borrow out* signals of the subtractor ports are also called *carry in*(c_{in}) and *carry out*(c_{out}) as in the adder in order to better distinguish between them and the input b.

In the course of the i-EDGE project, also a multiplier and an optimized comparator have been designed, however, they have not been part of the PDK and have not been described in a publication at the time of writing this thesis. The comparator compares two binary numbers and outputs a *lesser*(L), a *greater*(G), and an *equal*(E) signal. For the exploration in this chapter, mainly the synthesis results have been considered instead of adding a lot of optimized sub-circuits that would have to be designed and integrated by hand.

4.2 Optimization towards 4-T

Optimizing CCAM towards NEMS technology does not only mean to deal with the area and device count restrictions but also to exploit the special properties of this technology as much as reasonably possible. While most logic gates built with 3-T switches have the same device count and circuits as a CMOS implementation, 4-T and 7-T devices pave the way for logic gate designs that provide the same behaviour but through a different concept that allow for more efficient or robust cells. As the 7-T switch keeps its position without further appliance of voltages, it is used for stateful components such as

latches, flip-flops and memory cells. The 4-T device can help decrease the device count in combinational circuits, mainly by providing the efficient 4-T MUX and demultiplexer, presented in [12], as well as the 4-T XOR and XNOR gates and the previously presented full adder and subtractor cells, add_4T and sub_4T, which are all part of the i-EDGE PDK.

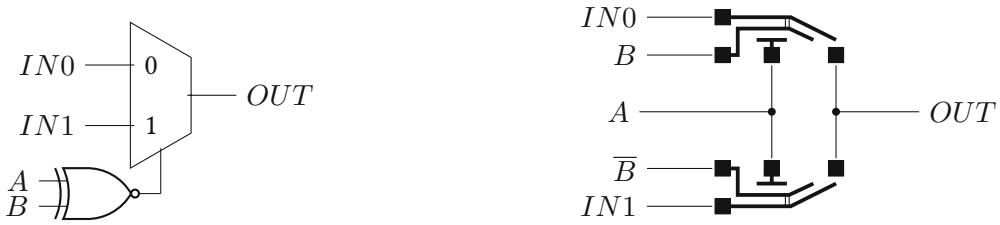
In the following sections, the synthesis results will show that Genus will almost not make advantage of these cells, even though there is often optimization potential. This can also be seen by looking at the MUX-based AND and OR gates in Figure 4.7. A 4-T multiplexer is much smaller than the CMOS-like 3-T AND and OR gates and by swapping the MUX inputs, even inverters can be saved.

(a) $Y = A \vee B$ (b) $Y = A \wedge B$

(c) $Y = A \vee \overline{B}$ (d) $Y = A \wedge \overline{B}$

Figure 4.7: Trivial logic functions with 2:1 multiplexers

A generalization of the 4-T MUX circuit is the MUX-XOR structure in Figure 4.8. Its potential for logic optimization has been mentioned in [29], where it represents a 6-T relay from a different NEMS technology. The inverted input $\overline{B}$ can be provided by an inverter. If B is assigned to a fixed value, the structure is a 4-T multiplexer and the inverted input is not needed.



(a) Equivalent Circuit (b) Circuit

Figure 4.8: MUX-XOR Structure

As a measure to obtain better synthesis results, the liberty file has been extended by the 4-T AND and OR gates with and without one inverted input, and by the MUX-XOR gate (including an inverter). The list of additional cells can be seen in Table 4.3. For even better results, the synthesis tool would

have to be adapted or replaced.

Table 4.3: Extension of the Liberty File

Cell	Device Count	Height in μm	Width in μm
and_4T	2	80	28.2
and_not_4T	2	80	28.2
or_4T	2	80	28.2
or_not_4T	2	80	28.2
mux_xor_4T	4	80	42.7

4.3 Binary Encoder and Decoder

Encoders and decoders are necessary for addressing memory and binary/unary conversions used in some sorting approaches. Therefore, their synthesis results help to calculate the device count of some sub-designs in CCAM. The results are plotted in figures 4.9 and 4.10.

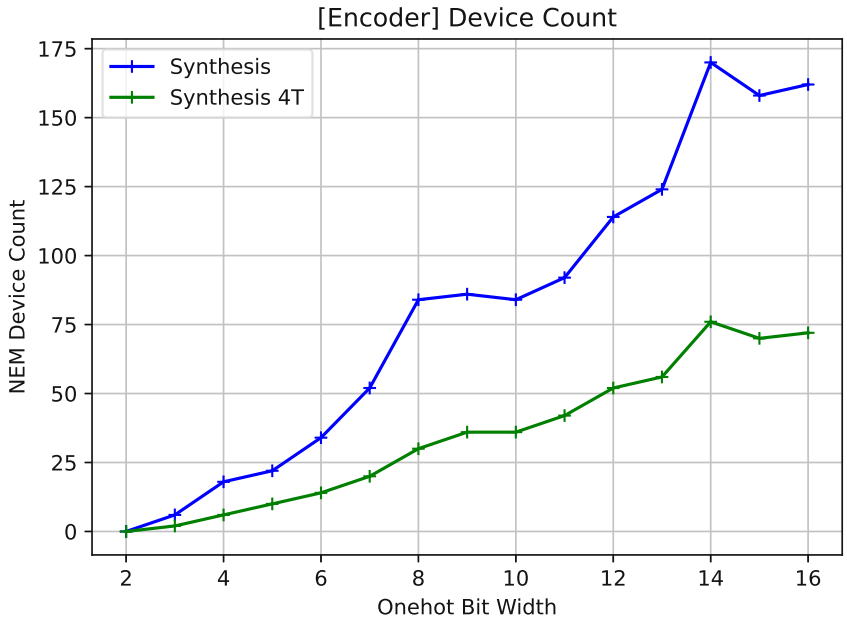


Figure 4.9: Device Count of the Onehot-to-Binary Encoder

4.4 Comparator

All the comparisons in CCAM of bit vectors with a potentially higher bit width are comparisons on sample or confidence values and can be formulated as either *less than*(LT) or *greater than*(GT) comparisons with no need for additional outputs. Therefore, no in-depth exploration of comparators is made but an estimation of the NEM device count of an LT block.

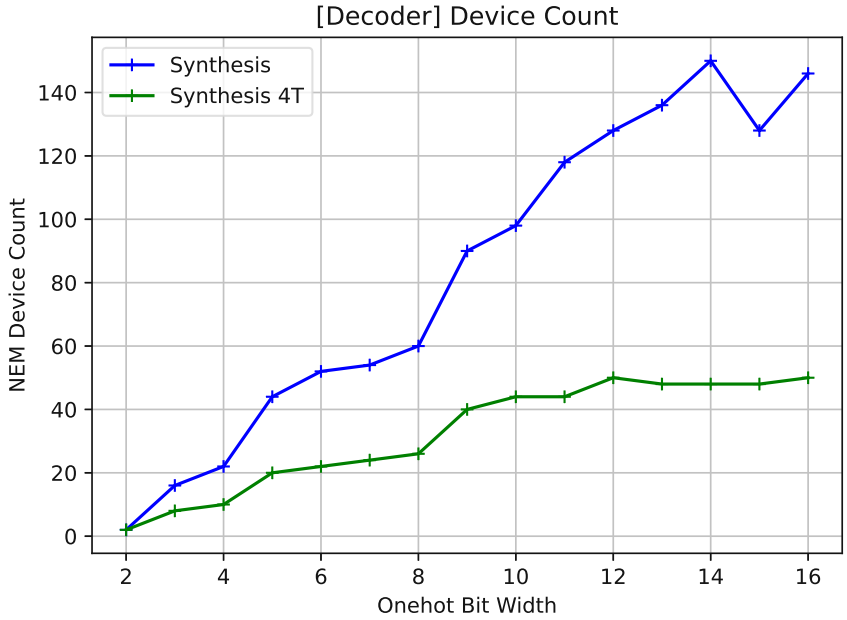


Figure 4.10: Device Count of the Binary-to-Onehot Decoder

The probably least device-intensive LT block implementation in NEMS could be the structure depicted in figure 4.11 as it would make use of the cheap MUX and XNOR gates. The logic function $L_n = \overline{A_n} \wedge B_n$ compares one digit of two binary numbers and results in '1' if A_n is less than B_n. The circuit checks the most significant bits(MSBs) at first and in case they are equal, the next digits are compared. The shown circuit has actually still optimization potential. The function $L_n = \overline{A_n} \wedge B_n$ can be implemented with a MUX. Furthermore, the MUXes that have an XOR or an XNOR gate at their *select* input could be combined with the X(N)ORs to MUX-XOR gates. Such a MUX-XOR gate consists only of four devices in i-EDGE's NEM technology.

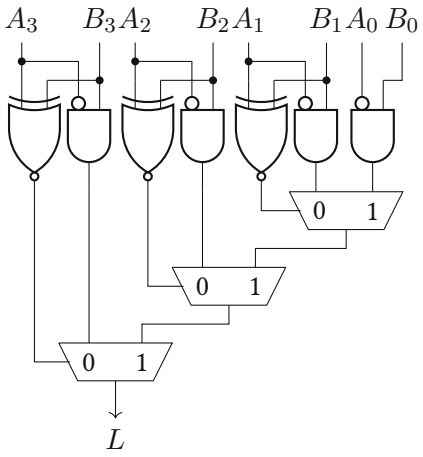


Figure 4.11: Less-Than comparison

However, this circuit results in a long critical path for comparisons of large bit vectors. A long chain

of multiplexers can also have problems with voltage degradation and require the insertion of buffer cells. The circuit is thus seen as a limit of the lowest device count achievable and compared with synthesis results. The equation for the NEM device count of the shown LT block with bit width n is written below in Equation 4.4.

$$D_{LT_{min}}(n) = n * D_{MUX} + (n-1) * D_{MUXXOR} = 2n + (n-1) * 4 = 6n - 4 \tag{4.4}$$

In figure 4.12 the device count from synthesis results and the estimation of the introduced LT structure are depicted. The blue line shows the synthesis results from the basic liberty file, the green line shows synthesis results where the AND and OR gates are built from 4T multiplexers.

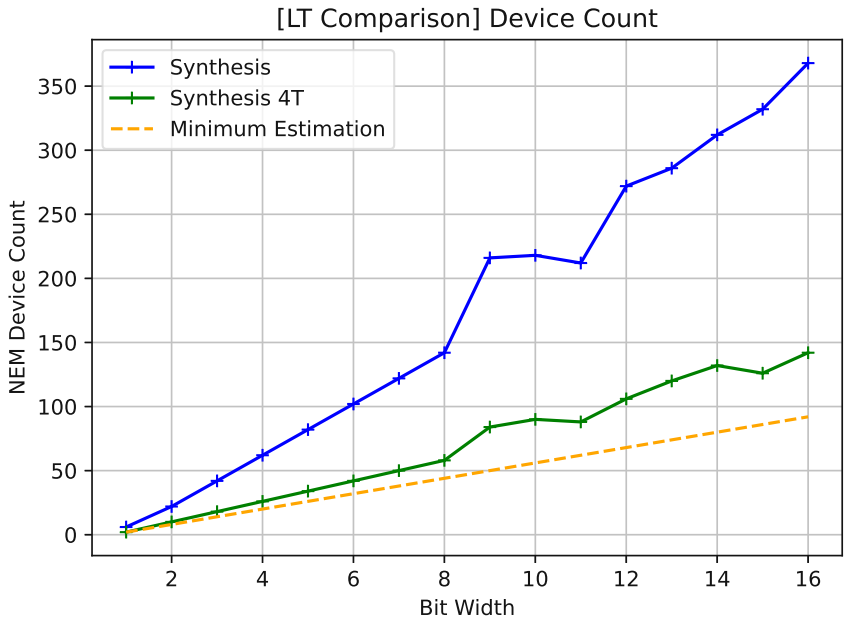
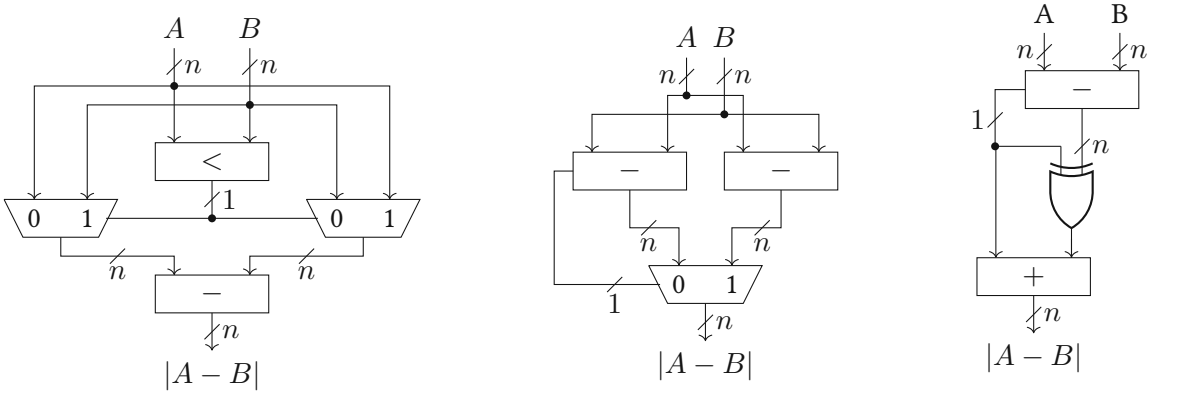


Figure 4.12: Device Count of the LT Block

4.5 Absolute Subtractor

Adders and subtractors are basic arithmetic operations that are present in almost every digital system. Most adders and subtractors in the design are incrementers or decrementers for counters, which means that these components always increment or decrement by 1. Such an incrementer can also be implemented by a chain of half adders. As the insertion of the arithmetic blocks from the PDK by hand introduces a lot of work overhead apart from the usual synthesis flow, and because this scale of micro-optimization is unnecessary for a rough tradeoff analysis, only the big absolute subtractor block is built by hand, whereas all other subtractors and counters are synthesized from Verilog code.

An absolute subtractor always yields the positive difference of two numbers. The absolute subtractors in this section expect unsigned integers as inputs. Hamzaoglu et al. presented three ways to build an accurate unsigned absolute subtractor, while they proposed an approximate implementation [6]. These three circuits are shown in Figures 4.13a, 4.13b and 4.13c.



(a) Input multiplexing ($ASUB_1$)

(b) Output multiplexing ($ASUB_2$)

(c) Two's Complement ($ASUB_3$)

Figure 4.13: Accurate unsigned absolute subtractor circuits [6]

Using the adder and subtractor blocks from the PDK and the device count estimation of the LT block from the previous section, the device count estimations for a bit width n are as follows (Equations 4.5,4.6,4.7):

$$D_{ASUB_1}(n) = D_{LT_{min}}(n) + 2n * D_{MUX} + D_{SUB}(n) = D_{LT_{min}}(n) + 2n * D_{MUX} + n * D_{SUB} + n * D_{INV} = (6n - 4) + 4n + 6n + 2n = 18n - 4 \quad (4.5)$$

$$D_{ASUB_2}(n) = 2 * D_{SUB}(n) + n * D_{MUX} = 2n * D_{SUB} + n * D_{INV} + n * D_{MUX} = 12n + 2n + 2n = 16n \quad (4.6)$$

$$D_{ASUB_3}(n) = D_{SUB}(n) + n * D_{XOR} + D_{ADD}(n) = D_{SUB}(n) + n * D_{XOR} + (n-1) * D_{ADD} + D_{XOR_{min}} = 6n + 2n + 2n + 6n + 2 - 6 = 16n - 4 \quad (4.7)$$

Even with the optimistic estimation for the LT block, the first option has the highest device count. As the third approach has the lowest device count, it has been selected. Figure 4.14 shows the implementation in more detail. Because $c_{out,add}$ is not needed and the b input is 0, the last add_4T block on the lower

right side can be replaced with an XOR gate.

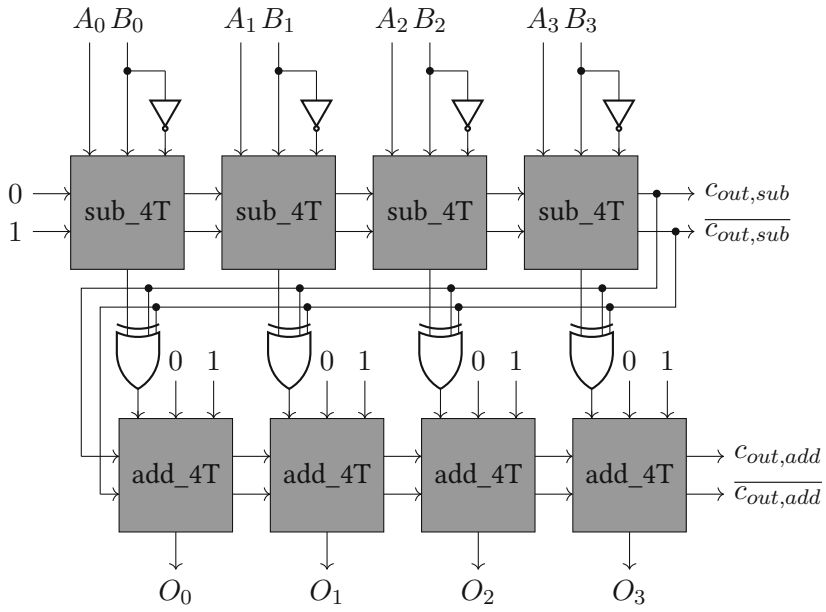


Figure 4.14: Circuit of 4-T optimized absolute subtractor

A comparison of synthesis results and the three presented physical implementations can be seen in figure 4.15.

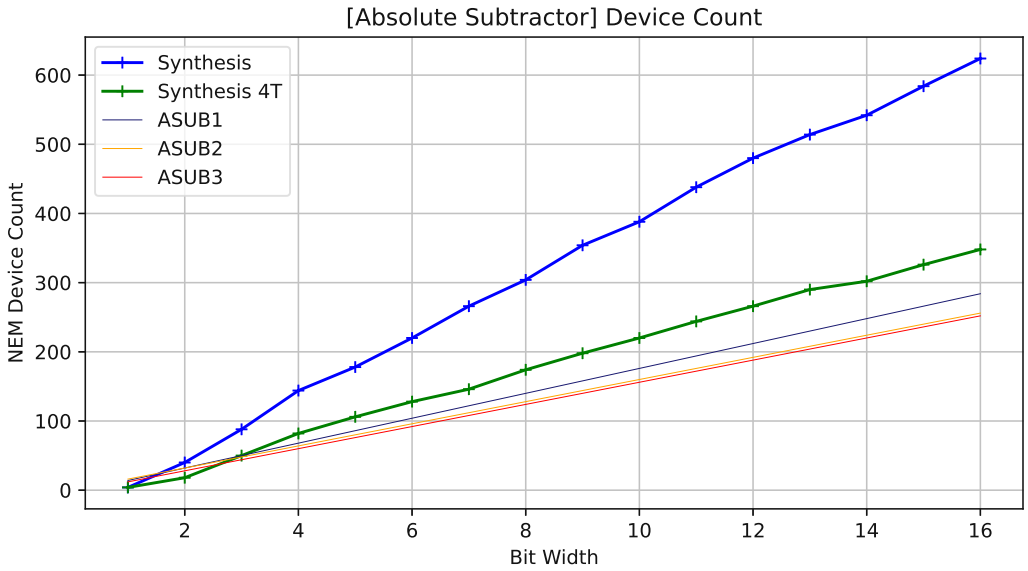
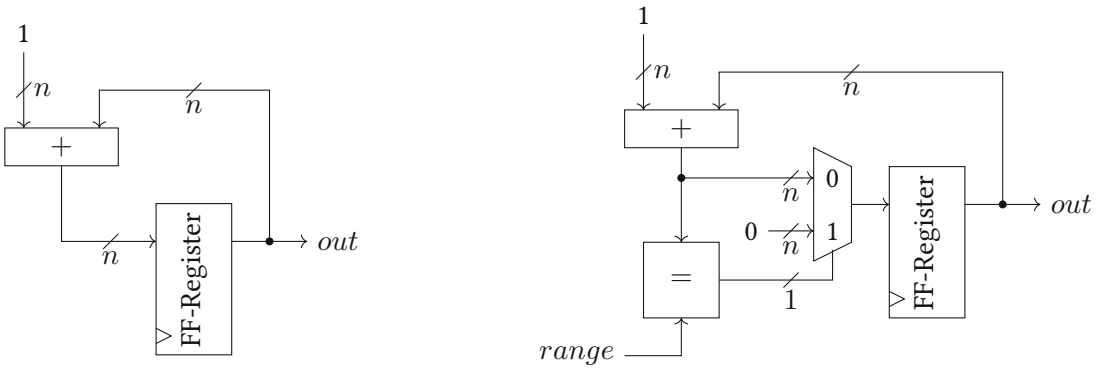


Figure 4.15: Device Count of the Absolute Subtractor

4.6 Binary Counter

There is a variety of requirements for counters, counters can be up-counting or down-counting, can have an asynchronous or synchronous reset, and can require an enable input or even a set input to set the counter to a desired value. Different counters have been used for various purposes in the realization of CCAM. If the output is not required to be binary but one-hot encoded, a shift register might be an option, however, flip-flops are expensive gates in NEMS. The bit width of a one-hot output is growing exponentially (2^n) compared to the bit width of a binary counter output and so is the number of required flip-flops. A one-hot output can be transformed into a binary value via an encoder and vice versa via a decoder.

For the following comparison an up-counting counter with a defined range, binary output, asynchronous reset but without additional inputs has been considered. The binary implementation with an incrementer (Figure 4.16) needs a comparison for counter ranges that are not of a power of 2, but the one-hot shift register with a binary encoder (Figure 4.17) does not.



(a) Overflow at a power of 2

(b) Overflow at arbitrary number

Figure 4.16: Circuit of Binary Counter

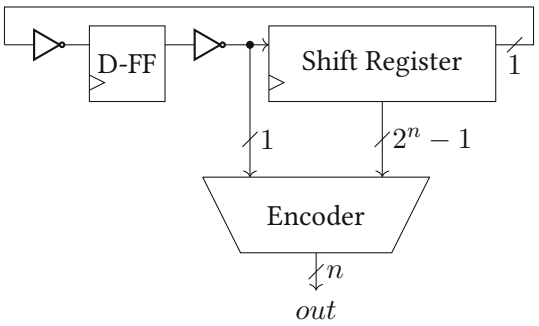


Figure 4.17: One-hot shift register with binary encoder

Apart from the synthesis results of the binary counter, also an implementation with the lowest device count has been looked for. The incrementer used for the counter in Figure 4.16 could be a chain of

the add_4T block, as was done with the absolute subtractor in Figure 4.14. However, the add_4T block is a full adder and has a device count of 6 while it still requires complementary inputs. To exploit the availability of the inverted D-FF outputs $\overline{Q}$, the structure shown in Figure 4.18 is suggested for counters. The AND gates are implemented with MUXes and thus the device count can be calculated as is written in Equation 4.8.

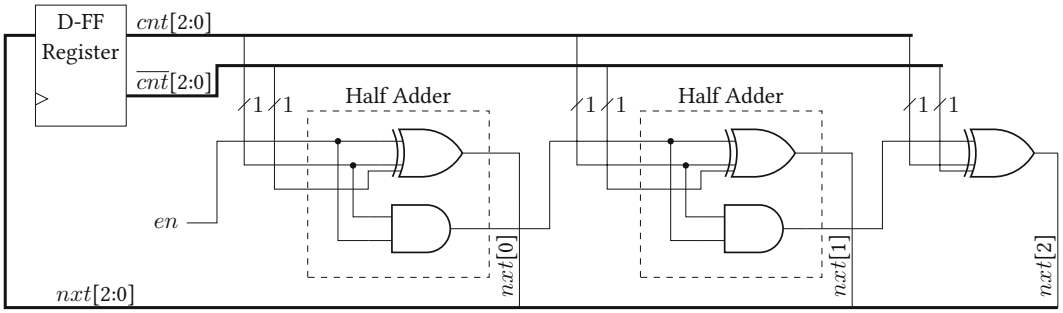


Figure 4.18: Optimized Incrementer of Binary Counter

$$D_{CNTR}(n) = n * D_{FF,rst} + n * D_{XOR} + (n-1) * D_{MUX} = 38n + 2n + 2 * (n-1) = 42n - 2 \quad (4.8)$$

A formula of the device count of a counter that can count until an arbitrary number $k-1$ that fits the counter bit width ($k < 2^n$), like in Figure 4.16b, is given in Equation 4.9. It is assumed that the *range* input comes from a FF-register and therefore is also present in inverted form.

$$D_{CNTR,DYN}(k) = D_{CNTR}(n) + n * D_{XOR} + n * D_{MUX};\ n = \lceil log_2(k) \rceil$$
$$D_{CNTR,DYN}(k) = 42n - 2 + 2n + 2n = 46n - 2 \quad (4.9)$$

Figure 4.19 compares the device count of the synthesized binary counter and the optimized circuit (Figure 4.18). The synthesis works quite well, even without the extended liberty file, as it can already pick the xor_4T cells. Figure 4.20 shows the synthesis results of the binary counter and the one constructed with the onehot shift-register. The dashed line marks the device count of the circuit of Figure 4.16b. Unlike the synthesized counters, it can be configured to different counter ranges during runtime.

To estimate a counter with a synchronous reset or the possibility to assign the counter value, multiplexers can be added to the device count formula as shown in Equations 4.10 and 4.11. These estimations include the asynchronous reset of the D-FFs. If the counter should only have a synchronous

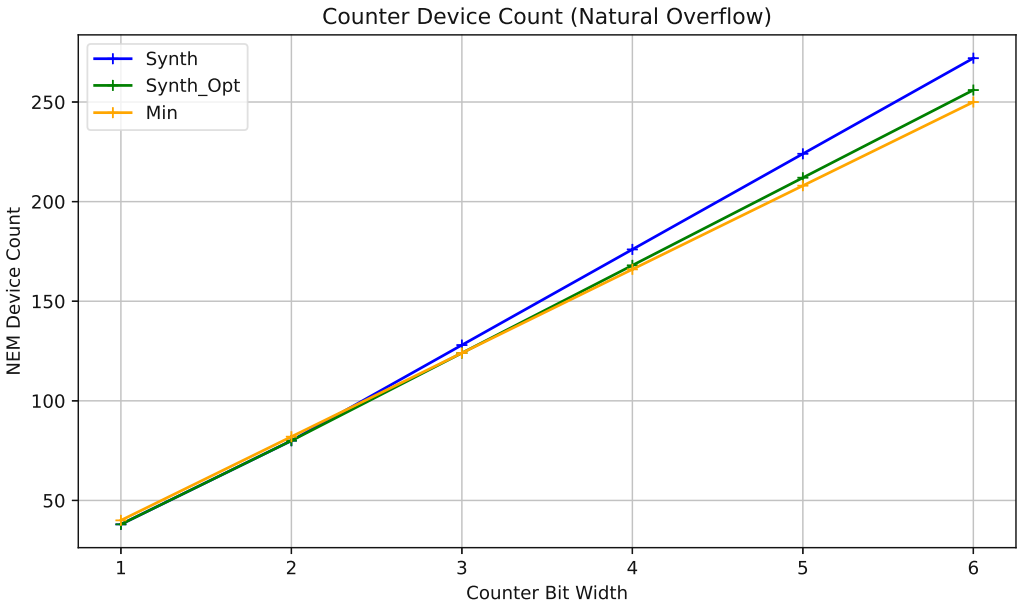


Figure 4.19: Device Count of Counter with Natural Overflow

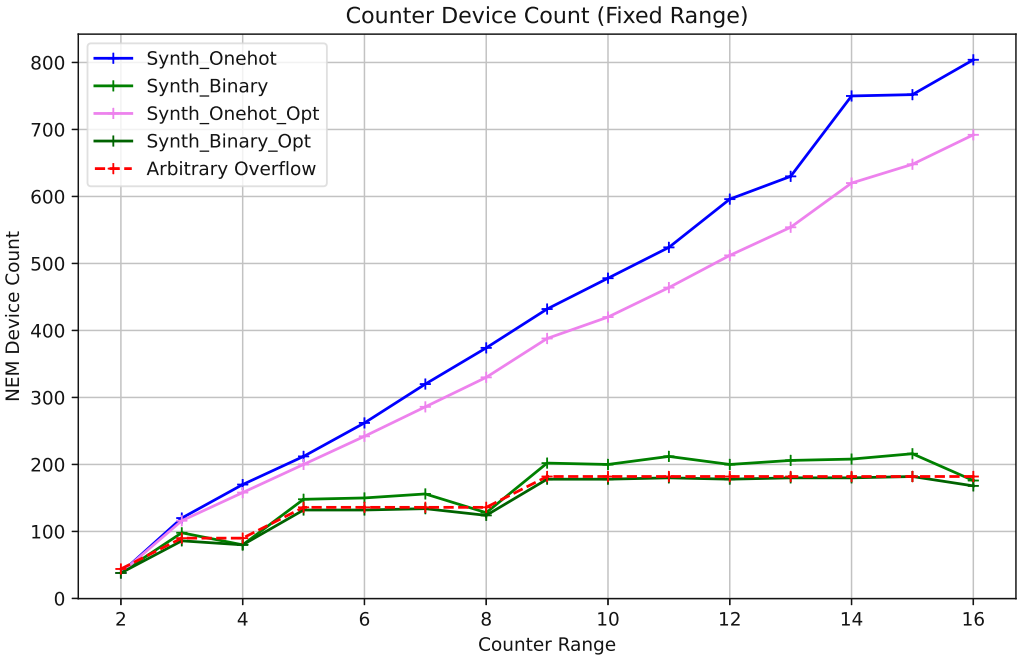


Figure 4.20: Device Count of Counter over Defined Range

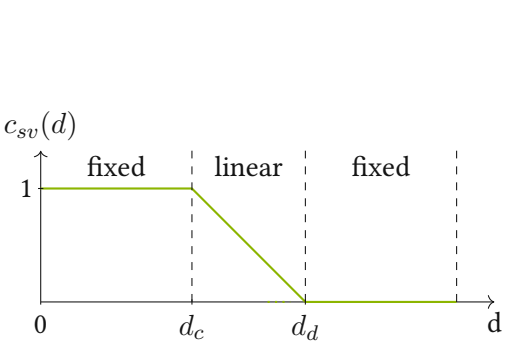
reset, the device count can be reduced by $12n$.

$$D_{CNTR,SET}(n) = D_{CNTR}(n) + n * D_{MUX} = 42n - 2 + 2n = 44n - 2 \tag{4.10}$$

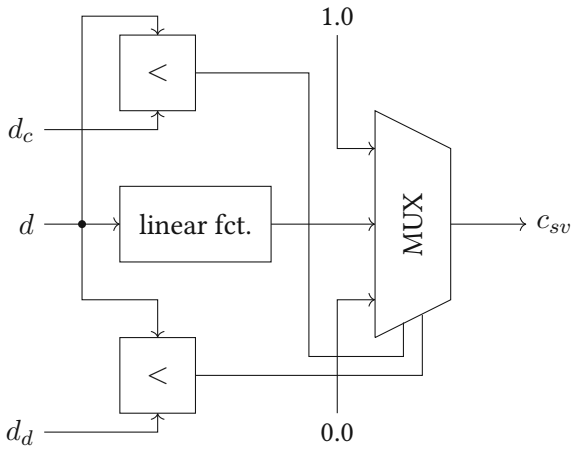
$$D_{CNTR,DYN,SET}(k) = D_{CNTR,DYN,SET}(k) + n * D_{MUX} = 48n - 2 \tag{4.11}$$

4.7 Fuzzification

As the shape of all fuzzy functions in CCAM is semi-trapezoidal, meaning that the shape looks like a halved trapezoid and that these functions are monotonically rising or falling, the function can be separated into one linear range and two static ranges. Figure 4.21 shows the sketched implementation of the mapping to a semi-trapezoidal fuzzy function.



(a) Function



(b) Sketched hardware implementation

Figure 4.21: Semi-trapezoidal function $c_{sv}(d)$

The two comparisons check if the distance lies in the linear or one of the static ranges and the multiplexing logic propagates the according confidence values. The case $d < d_c \wedge d > d_d$ cannot occur if the fuzzy function is configured correctly and can therefore be seen as a *don't care*. The linear function can be implemented in different ways. Let the constant Z be $Z = d_d - d_c$ then the linear function segment can be described as:

- $(d - d_c)/Z$
- $(d - d_c) * \frac{1}{Z}$
- $LUT(d - d_c)$

The values of slope are either stored in a LUT or a multiplication with a fixed factor has to be implemented. A division like in the original software code would only be an option if there was already an idle division unit in the design but no available multiplier.

The downside of the LUT approach is that the linear range of the fuzzy function should be configurable, and thus the size of the LUT needs to be sufficiently large to fit a very flat slope. If the LUT covers the entire range of possible input values, the comparators and the subtraction become redundant and the fuzzy function could be redefined freely, however, this is a very inefficient usage of memory for fuzzy functions with steep slopes. The advantage of a LUT is that the quantization error does not add up as it does with a multiplication.

Which implementation is the better option, depends on the design parameters and the fuzzy functions. Parameters are the bit width of the input value W, the bit width of the resulting confidence value C and the range R of the slope of the fuzzy function in sample quantization steps. As there is a variety of possible multiplier implementations and they have a high device count, it was intended to implement CCAM without a multiplier. Because of this, the higher accuracy and the easier implementation, the LUT approach has been chosen without a detailed analysis.

The formula of the required memory bits for the fuzzy function c_{sv} is shown in Equation 4.12. For clarification, it should be mentioned that for the fuzzification LUTs, the NEM RAM structure is used and not the NEM LUTs.

$$M_{FUZZ1} = R * C;\ 0 \leq R \leq 2^W \tag{4.12}$$

The resulting device count estimation is (Equation 4.13):

$$\begin{aligned} D_{FUZZ1} = D_{RAM}(R,C) + D_{SUB} + 2D_{LT} + 2C * D_{MUX} + 2 * D_{REG}(C) = \\ D_{RAM}(R,C) + 6C + C * D_{INV} + 2 * (6C - 4) + 4C + 10C = \\ D_{RAM}(R,C) + 34C - 8 = RC * D_{RAM\_Cell} + R * D_{WLD} + D_{Decoder}(R) + 34C - 8 \end{aligned} \tag{4.13}$$

The two 7-T registers in the equation are used to hold the configuration values d_c and d_d. The device count for a full LUT implementation would be $D_{RAM}(2^W, C)$.

For the fuzzification of sample numbers, the entire fuzzy function is a slope. Since c_{ss} and c_{ds} depend on the index of the sample from the history currently compared to and the fill level of the history, the fuzzy function changes according to the fill level $s_a \leq H$. The maximum history length H thus defines the number of memory bits of the LUT of this fuzzification as in Equation 4.14. The device count estimated in Equation 4.15.

$$M_{FUZZ2} = \frac{H^2 - H}{2} + H = \frac{H^2 + H}{2} \tag{4.14}$$

$$D_{FUZZ2} = \frac{H^2+H}{2} * (C * D_{RAM\_Cell} + D_{WLD} + D_{AND}) + 2 * D_{Decoder}(H) = \frac{H^2+H}{2} * (5C+8) + 2 * D_{Decoder}(H) \quad (4.15)$$

4.8 Sorting

Since the CCAM algorithm requires sorting of sample or confidence values, selecting a resource-efficient sorting algorithm is crucial. As unsorted data will probably arrive serially, considering a form of insertion sort might be worthwhile. The exact requirements for the sorting block depend on the design decisions in chapter 5. The chosen sorter implementations need to be able to work on a list with dynamic length which disqualifies some of the common hardware sorting techniques.

Most sorting circuits in hardware are built to maximize the achievable performance and therefore exploit parallelism. However, there already exist some papers that focus on the area usage of different sorting hardware and compare them. Abdelrasoul et al. [32] and Alif et al. [33]) implemented common software sorting algorithms (bubble sort, selection sort, insertion sort and merge sort). However, especially approaches that are very different from the software sorting algorithms were not included.

A recent and extensive survey is from Jalilvand et al. [34]. It organizes hardware sorting solutions into four categories.

- Comparison Based
- Comparison Free
- Partial Sorting
- In-Memory

In-memory sorting techniques are less relevant to this thesis because most of them use memristor technology or they assume the availability of modern memory access e.g. including paging. Moreover, the values in the histories have to be sorted in order of occurrence to keep the FIFO property and only after the distance calculation the distances can be sorted, or the confidences can be sorted after fuzzification.

Among the comparison-based sorters, there are also the widely used Bitonic Sorter [35] and hardware implementations of typical software sorting algorithms. Bubble sort in hardware can be built with a single comparison unit, but then it requires some kind of addressing logic for comparing and swapping the right registers and needs $o(n^2)$ cycles to sort. In [36], a parallelized version of bubble sort is described, but then a lot of comparison units are used like in a Bitonic Sorter.

Unary processing [37] can also allow for very simple swapping, because minimum and maximum functions can be built with AND and OR gates, but the bit width of unary signals is growing exponentially with the bit width of the corresponding binary bit width. Furthermore, binary-to-unary conversion logic is necessary.

Some comparison-free and partial sorting approaches use a *largest element detector* or something similar. They iteratively select the largest element from the remaining list and build a new sorted list. Campobello et al. proposed an FPGA implementation that gets the maximum value of a list by passing the elements bit-serially to a *multi-input maximum circuit* [38]. Jalilvand et al. also introduced a comparison-free sorter that identifies the largest element by operations on bit-serial unary elements [39].

Because of the requirement of sorting on dynamic lists and low device count, two especially relevant sorting techniques have been implemented and examined. These are described below in more detail.

4.8.1 Parallel Shift Sort

Parallel shift sort (PSS) is a very efficient hardware algorithm that is a kind of insertion sort therefore takes n cycles for sorting n values and consists of identical cells with low complexity [7]. As the data is expected to arrive most probably element-wise, sorting would take $O(n)$ cycles anyway. The downside is that it also uses n LT comparison units, which will be the biggest building blocks of that algorithm. Figure 4.22 shows the sorting engine of this sorter. The sorted elements can either be read in parallel or shifted out of the shift register. To know how many elements are actual values (due to the dynamic length), either a small shift register with *valid* bits or counters have to be added. Furthermore, the registers in the sorting engine have to be initialized correctly before sorting (each register should be set to the highest possible value e.g. 0xFF when the bit width is 8 bit).

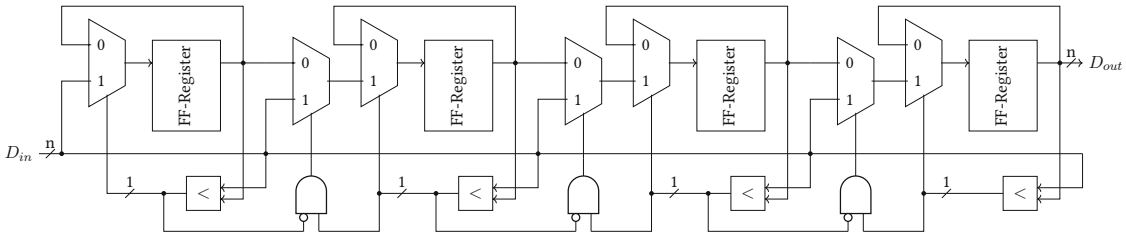


Figure 4.22: Parallel Shift Sort Engine [7]

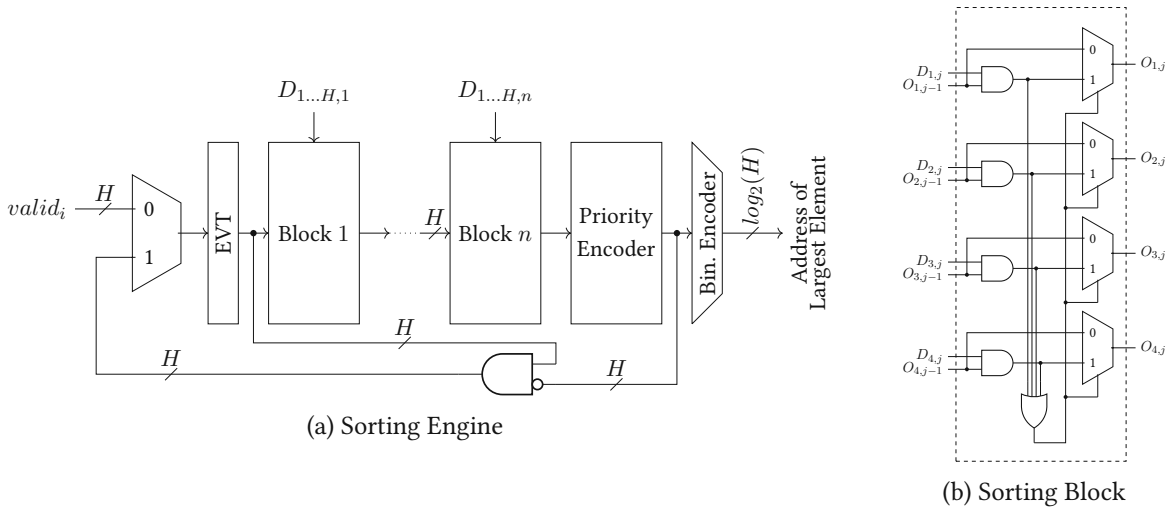
4.8.2 Comparison-free Sorter

A non-bit-serial comparison-free implementation has been presented by Ray et al. [8]. For sake of simplicity, this circuit will be called Largest Element Detector (LED) Sorter. Figure 4.23 shows the sorting engine of this approach.It determines the element with the highest set significant bit through a simple

network and uses an encoder to derive the address of the element. Sorting the entire list takes n cycles. The EVT is a vector table that stores which element has still to be sorted. This goes well with sorting lists with dynamic lists as these flag act as a *valid* bit for each element. The sorting network will then identify the largest element and the priority encoder ensures that a one-hot bit vector remains as some elements may have the same value. The one-hot vector is then encoded into the address of the memory that holds the unsorted elements. Originally, the approach presented in the paper requires two memories for sorting, the memory containing the unsorted elements and a memory that stores the sorted elements afterwards. Of course, the address of the sorted memory would have to be provided by an incrementing counter.

However, it is not necessary to store all sorted elements into a memory. Either the circuit after the sorter accepts the sorted elements sequentially or it needs to access them in parallel. The presented NEM RAM structure that is available in the project does not allow for parallel read-out of its rows and thus a shift-register could be used to gather the sorted elements.

To change the sorting order from descending to ascending, the sorting block can be changed so that the data bit inputs $D_{i,j}$ are inverted. By inserting XOR gates, the sorting order could be configured as needed.



(a) Sorting Engine

(b) Sorting Block

Figure 4.23: LED Sorting logic presented in [8]

4.8.3 Comparison

For a fair comparison of both sorting units that have been presented, the interface has to be the same. While the PSS works the best with shifting the data in serially and providing the sorted data in parallel, it is the other way around for the "largest element detector" circuit. The unsorted data and the EVT could be assigned in parallel but the sorter only yields one list element at a cycle. To parallelize the

incoming data, a shift register can be used. Serialization requires a counter and addressing logic. If the sorted data is shifted out serially, the subsequent datapath will probably require a list index provided by a counter.

The design decisions of the datapath using the sorter define the necessary interface and the sorting order. For a first comparison, both sorter designs have been configured in the code to have serial (sample-wise) input and serial output and have been synthesized.

The synthesis results with the extended liberty file are shown in Figure 4.24 and 4.25.

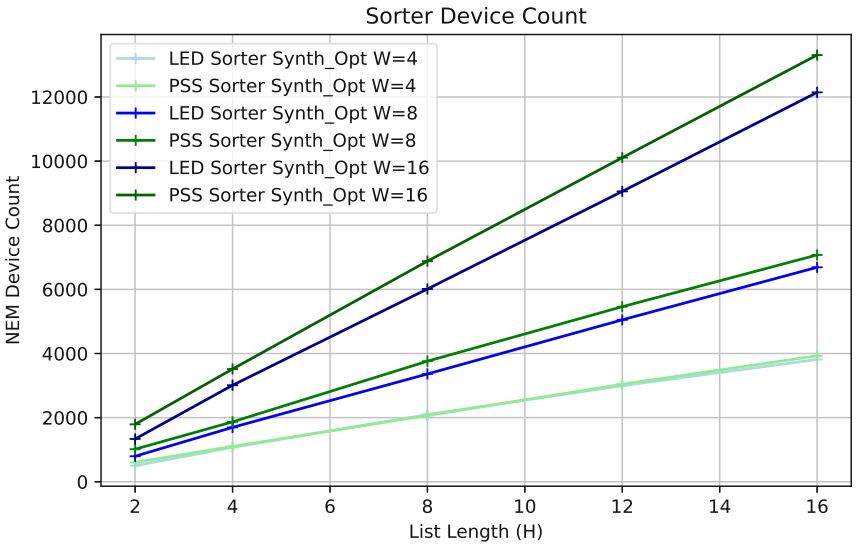


Figure 4.24: Device Count of LED Sorter and PSS over List Length

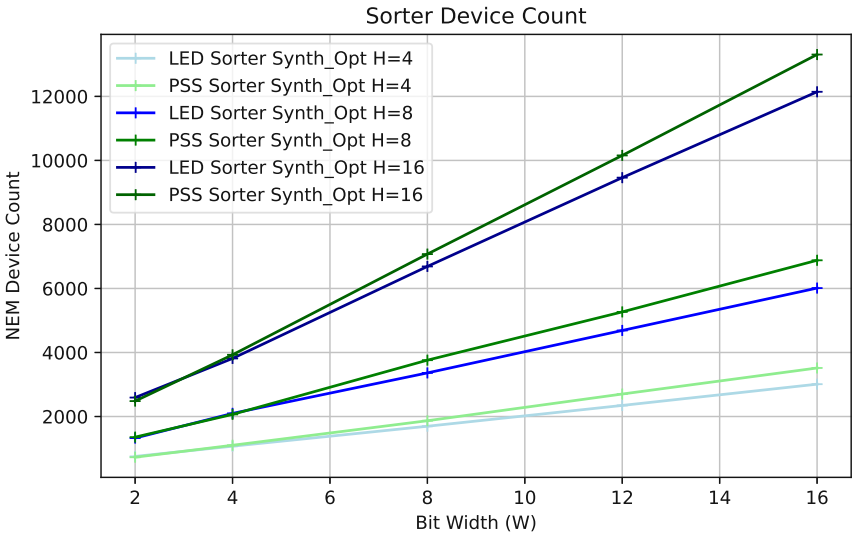


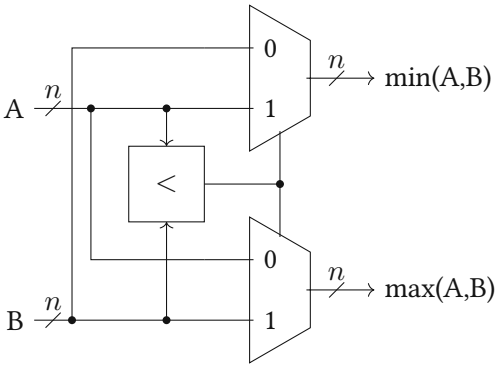
Figure 4.25: Device Count of LED Sorter and PSS over Bit Width

The device count of both sorters scales linearly with the bit width and the maximum list length.

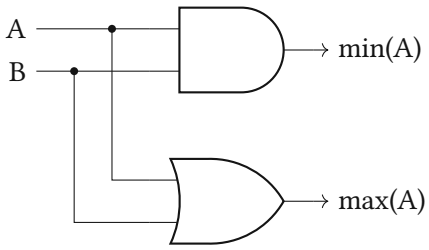
Since the execution takes H cycles for both of them, the LED Sorter outperforms the PSS. Another advantage is that the LED Sorter could be used for minimum and maximum operations as covered in the next section. Sorters that operate bit-serially per cycle have not been considered as sample-serial data would first have to be parallelized in $n \leq H$ cycles before taking W cycles to consider each bit, resulting in an execution time of $O(H * W)$.

4.9 Min/Max Operations

For a *min* or *max* operation on a list with exactly two elements, a so-called Compare-And-Swap (CAS) unit is sufficient. Figure 4.26 shows the hardware circuit for a CAS unit.



(a) Binary CAS Block [34]



(b) Unary CAS Block [37]

Figure 4.26: Compare-And-Swap (CAS) Block

The device count for a Min2 operation in unary is written in Equation 4.16. If the operation should be applied on binary values, binary encoders and a binary encoder are necessary as well.

A glimpse of the device count of the LT block combined with MUXes (Equations 4.17 and 4.18) shows that the minimum and maximum operations are relatively cheap for two operands.

$$D_{Min2,Unary} = 2 * D_{Decoder}(n) + 2^n * D_{AND} + D_{Encoder} \tag{4.16}$$

$$D_{Min2,Binary} = D_{LT}(n) + n * D_{MUX} = 6n - 4 + 2n = 8n - 4 \tag{4.17}$$

$$D_{Max2,Binary} = D_{LT}(n) + n * D_{MUX} = 8n - 4 \tag{4.18}$$

If the list can have more than two elements, the following naive algorithm can be implemented in hardware (Listing 4.1):

Listing 4.1: Min/Max algorithm

```
min = list[0]
max = list[0]
for element in list[1:]:
    if element < min:
        min = element
    if element > max:
        max = element
```

The synthesized algorithm would also use a CAS unit and store the result in a register but iterate over the list in k cycles for a list with k elements. If the list is available in parallel, a counter can be used to iterate over the list serially (one element per cycle). The estimated device count of this sequential approach is written in Equations 4.19 and 4.20.

$$D_{Min_k}(k,n) = (D_{Min2}(n) + n * D_{FF} + n * D_{MUX}) = 36n - 4 \tag{4.19}$$

$$D_{Max_k}(k,n) = (D_{Max2}(n) + n * D_{FF} + n * D_{MUX}) = 36n - 4 \tag{4.20}$$

A parallelized version could build a tree of reduced CAS blocks (only minimum or maximum is needed) which would lead to a device count as shown in Equations 4.21 and 4.22. The additional multiplexers ensure that only valid input values are considered since the circuit has cover K input vectors if $1 \leq k \leq K$. If one of the vectors is not valid, the select signal of the MUXes inside the CAS block is manipulated.

$$D_{Min_K}(k,n) = (K-1) * (D_{Min2}(n) + 2 * D_{MUX}) = (K-1) * 8n \tag{4.21}$$

$$D_{Max_K}(k,n) = (K-1) * (D_{Min2}(n) + 2 * D_{MUX}) = (K-1) * 8n \tag{4.22}$$

Finding the maximum or minimum values in a list, as needed for the $Min_k()$ and *max()* operations on confidence values, are special cases of sorting where the list does not have to be sorted completely. Since CCAM also needs a full sorter anyway, the use of such a more complicated full sorter may make sense for min/max operations as well when the sorter is idle (resource sharing). Especially, the comparison-free sorter with the largest element detector is suitable to find the largest element in 1 cycle ($O(1)$) without the need to sort all elements in the list. To get the minimum element, the sorting engine should yield the smallest element instead of the largest element.

4.10 History

In CCAM, histories are memories that store records of sample values or confidences per signal state over time. The maximum amount of values that can be stored for a certain signal state is called *history length* (H). If a new value is inserted into a full history, the oldest value will be replaced in a First In First Out (FIFO) manner. They are not typical FIFO memories, however, as all their values have to be readable without deleting them by read-out. In addition, the fill level needs to be accessible from outside of the history block.

While parallel write access to the history is not needed as only one sample comes at a time, both parallel or serial read-out of the stored sample values could be desired depending on the design of the logic reading from the history. These requirements lead to different architectures for the histories. Therefore, a history with serial read-out, as well as a history with parallel read-out, are presented and their device count is estimated. While it is kind of a comparison of apples to oranges, the results can be used to explore the trade-offs of designs that process the data in parallel or serially. The following explanations will target histories for samples, but they are also applicable to a confidence history. The confidence histories of the validity confidence datapath can use the same addresses as the sample history but the confidence history will insert the new value earlier than the sample history, which requires additional logic.

4.10.1 History with Serial Read-Out

When serial write access and serial read access are desired, the Random Access Memory(RAM) with a row width of the sample or confidence bit width is favorable. Figure 4.27 shows the rough implementation of the history. The pointer address (*ptr_addr*) stores the index on which the next history element will be inserted. In fact, this index is produced by a counter that increments after insertion. The counter overflow leads to a round-robin, respectively, a FIFO replacement strategy in case the history length is a power of 2. For other history lengths, the counter has to be compared against the history length to reset the counter when needed.

The iteration address (*iter_addr*) is used for iteration over the history as a sequence of read accesses and has to be provided by the datapath that reads the history. Additionally, the fill level has to be tracked as stated before. This can be done with a third counter without overflow. The fill level can range from zero elements to the history length H, which are $H + 1$ values in total. Instead of using a counter over $H + 1$ values, it is also possible to have a *hist_full* flag to remember when a history is filled completely and to count the maximum address for iteration (*max_iter_addr*) from zero to $H - 1$.

As the pointer address and the fill level of the history for a certain signal state have to be stored

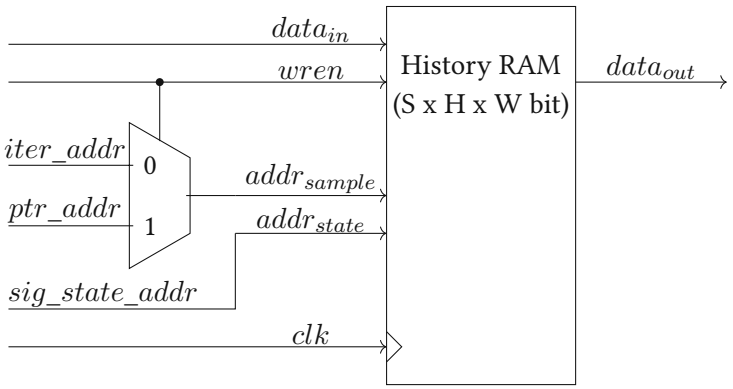


Figure 4.27: Sample-serial History

permanently, these registers have to be saved to memory before a signal state change. After the signal state address (*sig_state_addr*) has been updated, these addresses or indexes have to be loaded from memory in case of a state reentrance. If a new signal state has been created, both registers are initialized with zero. A *hist_full* flag would certainly have to be set to *false*. An overview on how to handle the ptr_addr and the $fill\_level$ is shown in Figure 4.28.

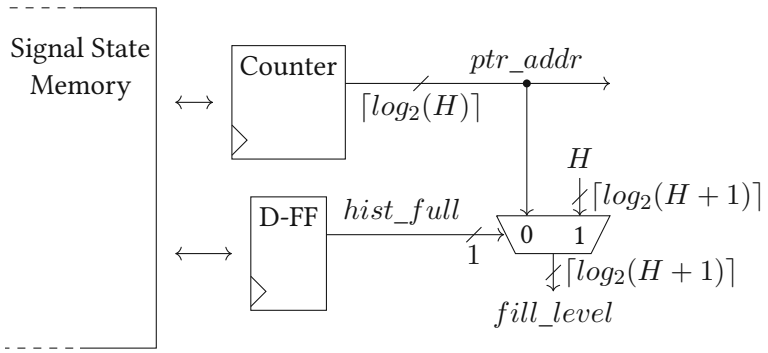


Figure 4.28: Circuitry for ptr_addr and $fill\_level$

As the history RAM has two input addresses at the same time, the *(signal) state address* and the index of the sample, the history RAM has internally 2 address decoders. The *select* bits could then, for example, be combined with AND gates as shown in Figure 4.29. The rows from different Signal States could be stacked up, however, stacking up too many rows will increase the memory latency and therefore it might make sense to have memory blocks per Signal State. The outputs of these memory blocks have to then be multiplexed.

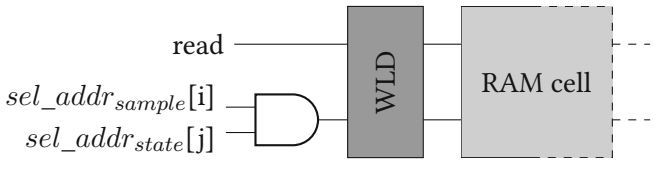


Figure 4.29: Select signal for History RAM

For the estimation of the device count of a History RAM with data bit with W, history length H and up to S possible Signal States, the RAM cells, wordline drivers, 2 binary encoders for address encoding and AND gates are considered (Equation 4.23). The required memory has $S * H * W$ bits.

$$\begin{aligned} D_{HistoryRAM}(S,H,W) &= S*H*W*D_{RAM\_Cell} + S*H*D_{WLD} \\ &+ D_{Binary\_Decoder(S)} + D_{Binary\_Decoder(H)} + S*H*D_{AND} = \\ &= (5*W+8)*S*H + D_{Binary\_Decoder(S)} + D_{Binary\_Decoder(H)} \end{aligned} \tag{4.23}$$

The complete history logic can be estimated to be the result of Equation 4.24. It counts the whole Signal State Memory to the device count although it might also contain other information. With the simplifications to the control logic, however, the only data that has to be stored per Signal State are the samples and the history meta-data. Moreover, the circuitry shown in Figure 4.28 and additional multiplexers to select FF-register inputs have been considered.

$$\begin{aligned} D_{History}(S,H,W) &= D_{HistoryRAM}(S,H,W) + 2*\lceil log_2(H)\rceil * D_{MUX} + \lceil log_2(H+1)\rceil * D_{MUX} + \\ &+ D_{CTNR,DYN}(H) + 2*D_{MUX} + 1*D_{DFF,rst} + D_{RAM}(S, \lceil log_2(H)\rceil + 1) = \\ &= D_{HistoryRAM}(S,H,W) + D_{RAM}(S, \lceil log_2(H)\rceil + 1) + 4\lceil log_2(H)\rceil + \\ &+ 2\lceil log_2(H+1)\rceil + 4 + 38 + 46*\lceil log_2(H)\rceil - 2 = D_{HistoryRAM}(S,H,W) + \\ &+ D_{RAM}(S, \lceil log_2(H)\rceil + 1) + 50*\lceil log_2(H)\rceil + 2\lceil log_2(H+1)\rceil + 40 \end{aligned} \tag{4.24}$$

4.10.2 History with Parallel Read-Out

Instead of writing the samples element-wise into the history, the whole history of a Signal State could be addressed with the *Signal State Address* only. As a consequence, the FIFO behavior has to be implemented with a shift register. New samples are shifted into the shift register but are read in parallel from all registers. When a the Signal State is exited, the whole content of the shift register must be saved into the History RAM and the content of the entered Signal State has to be retrieved from the memory as well. The shift register is reset if a new state is created. Figure 4.30 shows a sketch of the history structure.

The flip-flop registers inside the shift register require a MUX in front of their data inputs to switch between shifting and loading from the History RAM. The parallel output of the registers is wired to the data input of the RAM and the datapath. To keep track of the fill level of the history, an additional *valid* bit for each sample in the history could be stored in the RAM and the shift registers. The flip-flops of

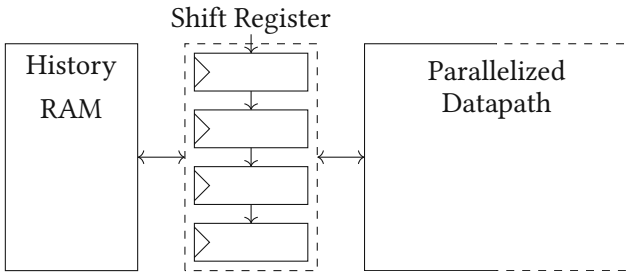


Figure 4.30: Sketch of the Sample-parallel History

holding these *valid* bits require a synchronous reset to deal with the creation of a new Signal State. The synchronous reset adds one MUX per resettable FF.

This architecture allows for a parallelized Matching Confidence datapath at the cost of device count. The History RAM stores $S * H * (W + 1)$ bits due to the additional $valid$ bit per sample. The estimated device count formula is written below (Equation 4.25 for the History RAM and Equation 4.26 for the complete device count of the history).

$$D_{HistoryRAM}(S,H,W) = S * H * (W+1) * D_{RAM\_Cell} + S * D_{WLD} + D_{Binary\_Decoder(S)} \tag{4.25}$$

$$\begin{aligned} D_{History}(S,H,W) = D_{HistoryRAM}(S,H,W) + H*(W+1)*D_{FF} + H*(W+1)*D_{MUX} + H*D_{MUX} = \\ = D_{HistoryRAM}(S,H,W) + 26H*W + 26H + 2H*W + 2H + 2H = \\ = D_{HistoryRAM}(S,H,W) + 28H*W + 30H \end{aligned} \tag{4.26}$$

4.10.3 Comparison

The decision on which history to take is mainly based on the requirements of the datapaths. Regarding the device count, both history designs are viable options, however, the history with the sample-serial read-out has the lower device count as shown in the diagram in Figure 4.31.

4.11 Summary

In this chapter, relevant building blocks of CCAM and, respectively, TCAM have been discussed. At first, the underlying technology with the available standard cells containing logic gates and memory cells is presented. Then logic optimization towards 4-T devices is mentioned, followed by sections about

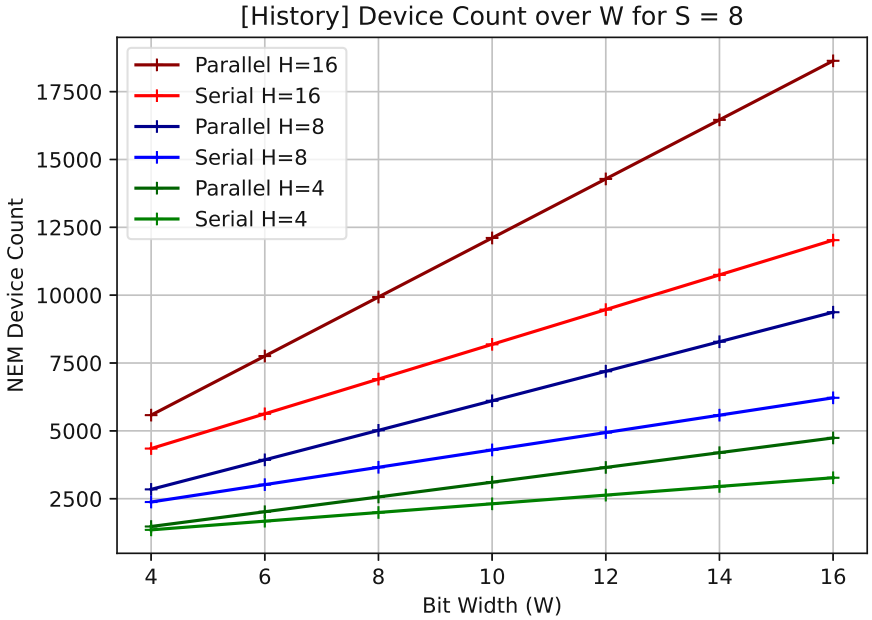


Figure 4.31: Device Count of Histories over Bit Width

the discussed building blocks. To support the design exploration, their device count has been calculated or estimated, including synthesis results. If more than one design was considered, comparisons have been made. For modules containing sequential logic, their cycle count is estimated a well. Covered modules are binary encoders and decoders, a Less-Than comparison, an absolute subtractor, binary counters, fuzzifcation, sorting unit, minimum and maximum operations, and histories for samples and confidences. Table 4.4 summarizes formulas for device count estimation. Components that have only been synthesized and the device count estimation of components including RAM like the history and the fuzzification have not been added to keep the table clean.

Table 4.4: Device Count Estimation Formulas

Component	Notation	Device Count
Less-Than	$D_{LT}(n)$	$6n-4$
Greater-Than	$D_{GT}(n)$	$6n-4$
Absolute Subtactor	$D_{ASUB}(n)$	$16n-4$
Minimum (2 Operands)	$D_{Min2}(n)$	$8n-4$
Maximum (2 Operands)	$D_{Max2}(n)$	$8n-4$
Minimum (k Operands) serial	$D_{Min_k}(n)$	$36n-4$
Maximum (k Operands) serial	$D_{Max_k}(n)$	$36n-4$
Minimum (k Operands) parallel	$D_{Min_K}(n)$	$(K-1)*8n$
Maximum (k Operands) parallel	$D_{Max_K}(n)$	$(K-1)*8n$
Counter	$D_{CNTR}(n)$	$42n-2$
Counter + Set	$D_{CNTR,SET}(n)$	$44n-2$
Counter w. arbitrary range	$D_{CNTR,DYN}(k)$	$46\lceil log_2(k)\rceil-2$
Counter w. arbitrary range + Set	$D_{CNTR,DYN,SET}(k)$	$48\lceil log_2(k)\rceil-2$

Chapter 5

Architecture and Design Space Exploration

5.1 Exploration Strategy

Optimization of a design architecture in general is known to be an NP-hard problem. The bigger the overall design, the more options have to be considered and the more design decisions have to be made. To apply a structured methodology that helps to not get lost under all possible designs, it may seem a good idea to take a look at the field of high-level synthesis (HLS). HLS has been an active field of research over the last decades and can therefore be considered to have well-understood concepts. Although the Verilog code and the architecture are not generated automatically via HLS, the methodology behind an HLS process is still applicable. This may lead to the question, of why HLS has not been used in the first place, but designing with an HDL allows for a more fine-grained optimization. Moreover, the gained insight is greater if the design is not partitioned automatically and the original CCAM algorithm would have to be adapted anyway to be synthesizable via HLS.

Especially, the topics of resource allocation, scheduling, and resource binding, are of special interest([9] [40]). These steps can be translated to the following methodology:

- **Allocation**: Identify necessary functional blocks
- **Scheduling**: Consider serialization, parallelization, pipelining, resource sharing, and adaption of control logic
- **Binding**: Compare trade-offs of equivalent functional blocks and design decision outcomes and select the most promising ones.

The functional blocks have already been identified and discussed in the previous chapter (Chapter

4), along with a comparison of different implementations of them. Therefore, the focus in this chapter will be the scheduling and the combination of the discussed sub-components to form the desired system.

Resource sharing includes sharing of registers, functional units, and communication channels [9]. In this work, the sharing of functional units is focused as it could lead to high savings in device count if an expensive functional unit could be shared between different parts of the system.

Sharing a logic block between two datapaths or different stages within a datapath could be done by multiplexing the inputs of the shared block as shown in a general example in Figure 5.1. This requires a *control* signal that defines which module uses the shared logic block and the block can only be assigned to one location at a time. If the block should be shared between more than two locations, a bus system with some bus arbitration logic might be an option.

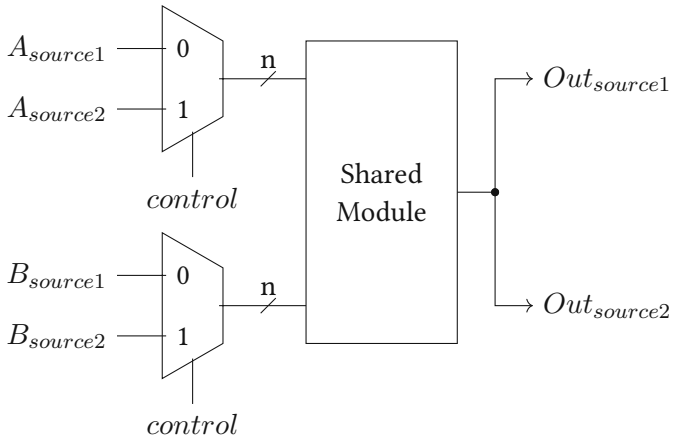


Figure 5.1: Sharing a 2-input module between 2 modules [9]

To estimate if it would save device count to share a common module between 2 locations, the multiplexers at the inputs have to be taken into account. As most arithmetic blocks have two input vectors with the same bit width, this general example can be applied to adders, subtractors, binary comparisons, multipliers, and so on. Under the assumption that the *control* signal is already available, the decision can be based on the question if the inequation $D_{shared}(m,n) + m * n * D_{MUX} < 2 * D_{shared}(m,n)$ is true, with m being the number of input vectors and n being their bit width.

Another scheduling-related design decision is to balance serialization and parallelization. Parallelization allows for more performant designs that can do calculations in a shorter time, however, at the cost of higher area requirements. Serial implementations, however, have an overhead in control logic compared to their parallel counterparts that is usually negligible. Pipelining separates a serial datapath into pipeline stages that allow for the next execution in a stage when the previous execution result has been passed to the next stage. The decision on how many pipeline stages are necessary is fine-grained and heavily depends on the clock frequency and the delay of the critical path of the datapath. For this

case study of CCAM on NEMS, such fine-grained design decisions will not be made. Further pipeline registers can be easily added to the datapaths of the presented designs if necessary.

5.2 Overview

Figure 5.2 depicts the first naive structure of the CCAM hardware implementation. Each signal under observation needs its own *Signal State Detector* state machine with its own histories for sample and confidence values, a DAB, and other memories. The resulting confidences, state flags and, most important, the IDs of the signal states are passed to the System State Detector. When all signal states have been updated, the System State Detector will start its execution. The relevant outputs of it are the *functioning confidence*, the System State ID and maybe the overall confidence.

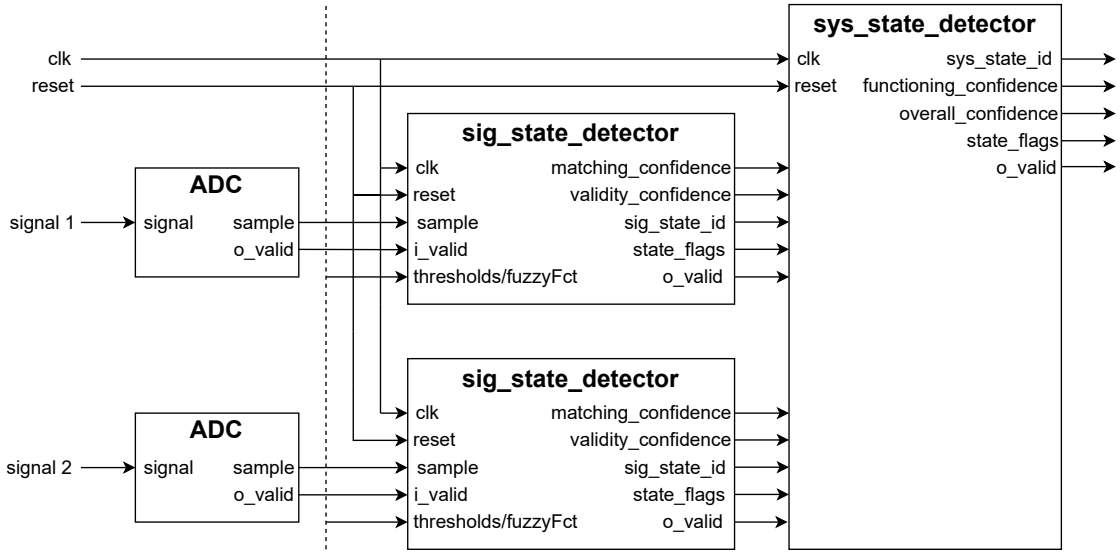


Figure 5.2: Naive Structure of CCAM Implementation

Modules of the datapath like arithmetic blocks may be shared among the CCAM design but this makes allocation logic for these resources necessary.

As the execution times of the Signal State Detectors can overlap, either waiting cycles for allocation would have to be introduced, or no resources are shared between multiple Signal State Detector instances. Depending on the use case, the number of SuO signals to monitor varies and so the number of required Signal State Detectors.

The sharing of control logic and counters, however, does not make a lot of sense as the overhead of the allocation logic is not in a reasonable relation. Moreover, states of stateful logic that are not part of a datapath have to be preserved until the next execution. Also, slow serialized computations are more difficult to share between different modules.

The identified design parameters of a CCAM or TCAM design are listed in Table 5.1. The ranges that seemed to be relevant for the tradeoff analysis are also attached. CCAM also requires a confidence bit width as shown in Table 5.2. Design parameters that were originally considered but discarded are presented in Table 5.3. The bit width of the State IDs became redundant as the IDs are the same as the state addresses with the bit widths $\lceil log_2(S) \rceil$ and $\lceil log_2(Y) \rceil$. The size of the DAB has to be configured only if drift detection is implemented.

Table 5.1: Design Parameters of CCAM / TCAM

Symbol	Name	Description	Range
N	Signal Number	Number of monitored SuO signals	2-4
W	Signal Bit Width	Bit width of sampled sensor data	4-16
H	History Length	Number of storable samples per Signal State	4-16
S	Signal States	Number of storable Signal States	4-16
Y	System States	Number of storable System States	4-16

Table 5.2: Additional Design Parameters for CCAM

Symbol	Name	Description	Range
C	Confidence Bit Width	Bit width of confidence values	1-8

Table 5.3: Removed Design Parameters

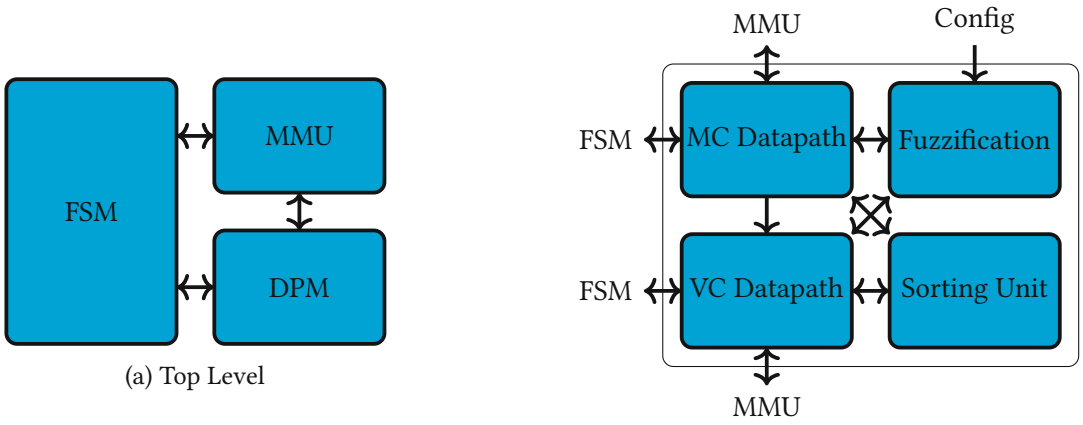
Symbol	Name	Description	Range
D	DAB Size	Number of samples stored in a DAB	4,8,16
SW	Signal State ID Bit Width	-	3-5
YW	System State ID Bit Width	-	3-5

5.3 Signal State Detector

As Figure 5.3a shows, the Signal State Detector consists of the main state machine, the memory management and a module which contains the datapaths for computation of the confidence values.

While the Memory Management Unit (MMU) contains the Signal State Memory, the histories, the control logic for loading and storing them and the state address counter, the Datapath Module (DPM) contains the Matching Confidence Datapath (MCDP) and the Validity Confidence Datapath (VCDP), together with possibly shared resources (Figure 5.3b). The MCDP may directly share $c_{sl,i,new}$ and $c_{dh,i,new}$ with the VCDP if it calculates them for it. Both datapaths are interfacing with the main state machine and return their results to it. Access to the histories is given by the MMU.

The Signal State address, respectively the Signal State ID, for the output of the Signal State Detector can be taken from the MMU. The resulting confidence values from the DPM do not actually have to be



(a) Top Level

(b) Datapath Module

Figure 5.3: Signal State Detector Architecture

exchanged with the main state machine since only the decision flags are relevant for the FSM of the detector. If desired as detector outputs, the confidence values can also be forwarded from the DPM to the outputs.

5.3.1 Matching Confidence Datapath

Original MCDP in Serial

Figure 5.4 shows an overview of the MCDP in serial. The samples are fed from the history cycle by cycle into the absolute subtractor (ASUB) to calculate the distances which are then passed to the sorting unit. The presented sorting modules take up to H cycles for sorting and can then shift out the sorted distances to the fuzzification blocks. The index of the sorted elements are also fuzzified. On the resulting confidences values, the fuzzy algebra can be applied and the comparison the the matching confidence $c_{b,i}$ against the co-confidence $c_{n,i}$ yields the decision flag $match$.

The formula for the device count is written in Equation 5.1. It also includes two fuzzy operations for the VCDP and a counter to iterate over the history. The device count of the serial history has not been included.

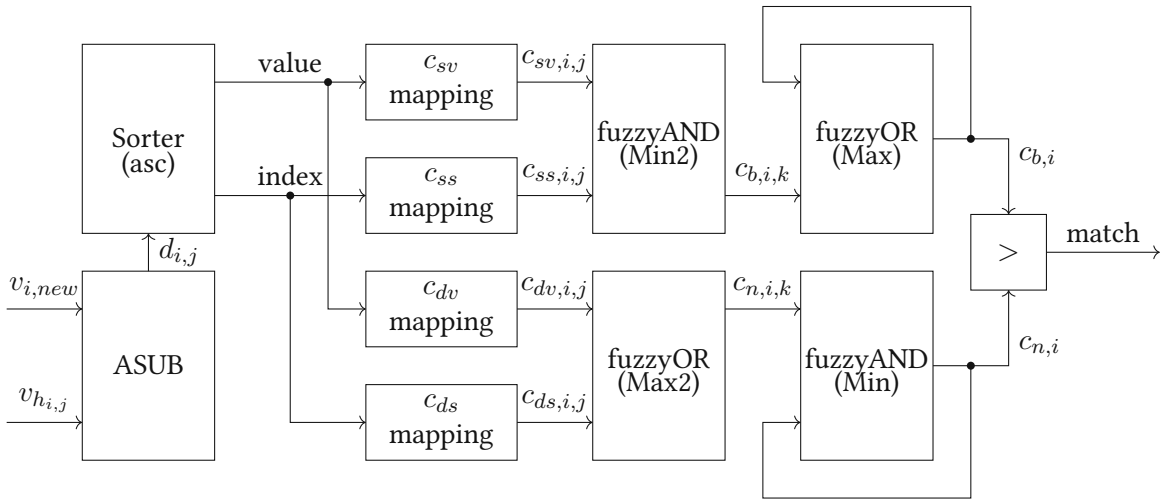


Figure 5.4: Matching Confidence Datapath (Serial)

$$
\begin{aligned}
D_{MCDP} &= D_{ASUB}(W) + D_{SORTER}(H,W) + 2 * D_{FUZZ1}(R,C) + 2 * D_{FUZZ2}(H,C) + \\
&+ D_{Min2} + D_{Max2} + 2 * D_{Max_k}(C) + 2 * D_{Min_k}(C) + D_{GT} + D_{CNTR,DYN,SET}(H) = \\
&= 16W - 4 + D_{SORTER}(H,W) + 2 * D_{FUZZ1}(R,C) + 2 * D_{FUZZ2}(H,C) + \\
&+ 16C - 8 + 32C - 16 + 6C - 4 + 4C * D_{FF} + 4C * D_{MUX} + 48\lceil log_2(H) \rceil - 2 = \\
&= D_{SORTER}(H,W) + 2 * D_{FUZZ1}(R,C) + 2 * D_{FUZZ2}(H,C) + 16W + 48\lceil log_2(H) \rceil + 166C - 34
\end{aligned}
\tag{5.1}
$$

It takes n cycles to read the sample values from the history. The selected sorting unit (LED Sorter) will start to push out the sorted distances when the sorter has been filled with all distances. Since the fuzzification consists of RAM LUTs, they do not add additional cycles to the execution cycles. The Min and Max operations at the end contain FF-registers. Therefore, after $2n$ cycles, the result(s) will be ready at the outputs of the datapath.

Co-Confidenceless MCDP in Serial

Due to the redundancy of co-confidences in CCAM as described in Section 3.2.1, the MCDP can be reduced to the design shown in Figure 5.5. The order of the fuzzification to $c_{sv,i,j}$ and the sorting unit can be changed, when the sorting order is changed to descending. As a consequence, the confidence values $c_{sv,i,j}$ will be sorted instead of the distances, but the fuzzification of the sorting index still has to be done after the sorter.

The execution of this datapath still needs $2n$ clock cycles, however, the device count has been greatly

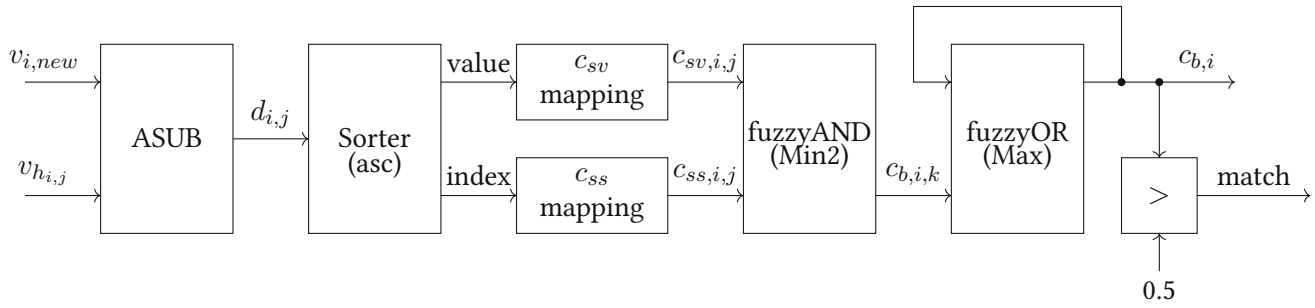


Figure 5.5: Matching Confidence Datapath without Co-Confidences (Serial)

reduced.

$$\begin{aligned} D_{MCDP} &= D_{ASUB}(W) + D_{SORTER}(H,W) + D_{FUZZ1}(R,C) + D_{FUZZ2}(H,C) + \\ &+ D_{Min2} + D_{Max_k}(C) + D_{Min_k}(C) + D_{GT} + D_{CNTR,DYN,SET}(H) = \\ &= 16W - 4 + D_{SORTER}(H,W) + D_{FUZZ1}(R,C) + D_{FUZZ2}(H,C) + \\ &+ 8C - 4 + 16C - 8 + 6C - 4 + 2C * D_{FF} + 2C * D_{MUX} + 48\lceil log_2(H)\rceil - 2 = \\ &= D_{SORTER}(H,W) + D_{FUZZ1}(R,C) + D_{FUZZ2}(H,C) + 16W + 48\lceil log_2(H)\rceil + 86C - 22 \end{aligned} \tag{5.2}$$

Besides halving the number of requires memory bits for the fuzzification, the device count of the logic that scales with the confidence bit width is also nearly halved by a reduction of $80C - 12$ devices.

Co-Confidenceless MCDP in Parallel

Figure 5.6 shows the first part of CCAM's MCDP in parallel. A parallel datapath only makes sense if the sample history can be read in parallel and the resulting confidences can be passed to the sorting unit in parallel. This includes that the sorting unit does not expect the data to arrive in serial).

If the sorting unit pushes the sorted elements out in parallel, the second half of the Matching Confidence Datapath could be realized in parallel, as shown in Figure 5.7. Since the indexes of the sorted elements are fixed in a parallel implementation, the fuzzification could be simplified, however, the confidence c_{ss} depends on the fill level of the history and is therefore not fixed.

Equation 5.3 shows the device count formula for the whole parallized datapath.

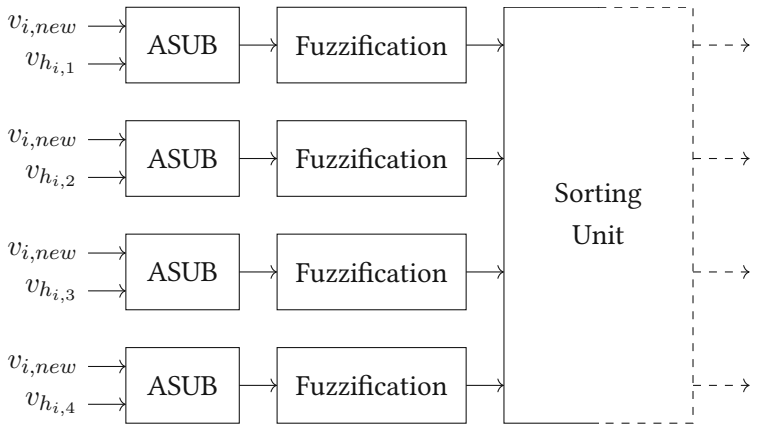


Figure 5.6: First Part of the MC Datapath in parallel

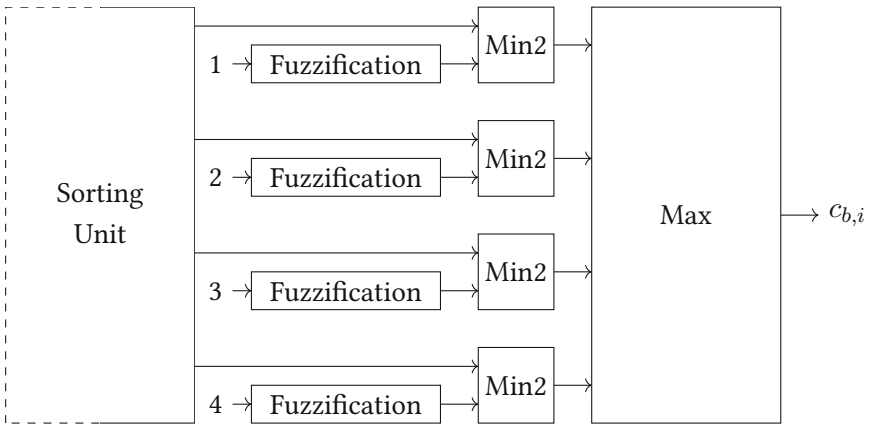


Figure 5.7: Second Part of the MC Datapath in parallel

$$
\begin{aligned}
D_{MCDP} = H * D_{ASUB}(W) + D_{SORTER}(H,W) + H * D_{FUZZ1}(R,C) + D_{FUZZ3}(H,C) + \\
+ H * D_{Min2}(C) + D_{Max_K}(H,C) + D_{Min_K}(H,C) + D_{GT} = \\
D_{SORTER}(H,W) + H * D_{FUZZ1}(R,C) + D_{FUZZ3}(H,C) + \\
+ 16HW - 4H + 8HC - 4H + 2 * (H-1) * 8C + 6C - 4 = \\
= D_{SORTER}(H,W) + H * D_{FUZZ1}(R,C) + D_{FUZZ3}(H,C) + 16HW + 24HC - 10C - 8H - 4
\end{aligned}
\tag{5.3}
$$

Since the fuzzification of the sample distances is one of the most expensive blocks in device count or memory bits, a duplication of it is not desired. However, the sorting unit kind of cuts the datapath in two halves, and parallelization of the second half seems to be more feasible if the sorter outputs all sorted elements in parallel. Equation 5.4 estimates only the device count of the second half of the MCDP.

$$D_{MCDP2} = D_{FUZZ3}(H,C) + H * D_{Min2}(C) + D_{Max_K}(H,C) + D_{Min_K}(H,C) + D_{GT} =$$
$$= D_{FUZZ3}(H,C) + 8HC - 4H + 2 * (H-1) * 8C + 6C - 4 =$$
$$D_{FUZZ3}(H,C) + 24HC - 10C - 4H - 4 \quad (5.4)$$

Threshold-based MCDP in Serial

When deciding on a threshold-based design, the datapaths become much simpler. Memory requirements are greatly reduced and the logic of the datapaths transforms into a combination of counters and Boolean operations. Figure 5.8 depicts the threshold-based MCDP.

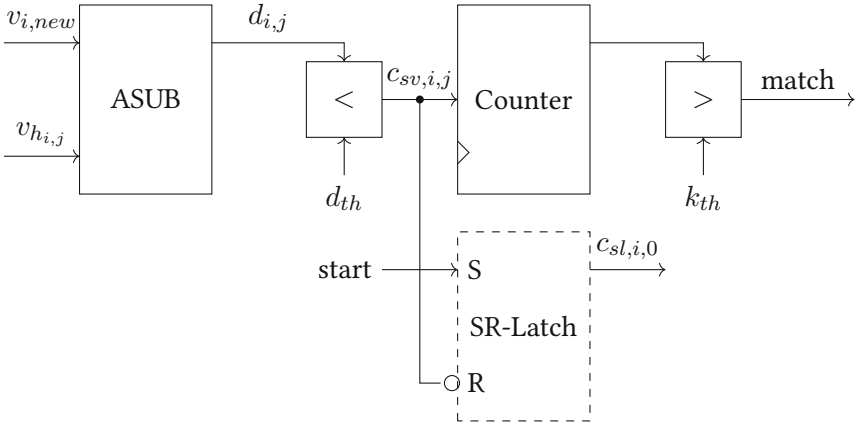


Figure 5.8: Threshold-based Matching Confidence Datapath

If the sample distances are less than a configurable threshold d_{th}, the 1-bit "confidence" $c_{sv,i,j}$ is true. A counter counts the number of matching distances and in the end, this number has to surpass the threshold k_{th} which is the half of the current number of samples in the history. If there was a sample distance that exceeded the threshold, $c_{sl,i,0}$ is set to $false$. Since the sorter is no longer required, the execution of the MCDP takes n clock cycles. The formula of the estimated decive count is written in Equation 5.5.

$$D_{MCDP} = W * D_{REG} + D_{CNTR,DYN,SET}(H) + D_{ASUB}(W) + D_{LT}(W) +$$
$$+ D_{CNTR,SET}(\lceil log_2(H+1) \rceil) + D_{GT}(\lceil log_2(H+1) \rceil) + D_{DFF} + D_{MUX} =$$
$$= 5W + 48\lceil log_2(H) \rceil - 2 + 16W - 4 + 6W - 4 + 44\lceil log_2(H+1) \rceil - 2 +$$
$$+ 6\lceil log_2(H+1) \rceil - 4 + 26 + 2 = 48\lceil log_2(H) \rceil + 50\lceil log_2(H+1) \rceil + 27W + 12 \quad (5.5)$$

It takes the register for the threshold d_{th}, the counter for iterating over the history, and the calculation of $c_{sl,i,0}$ into account.

5.3.2 Validity Confidence Datapath

Original VCDP in Serial

The unsimplified serial version of the VCDP is shown in Figure 5.9 and Figure 5.10. The first part should be combined with the MCDP as it would only be an addition of one fuzzy AND and one fuzzy OR operation instead of the whole circuit in Figure 5.9.

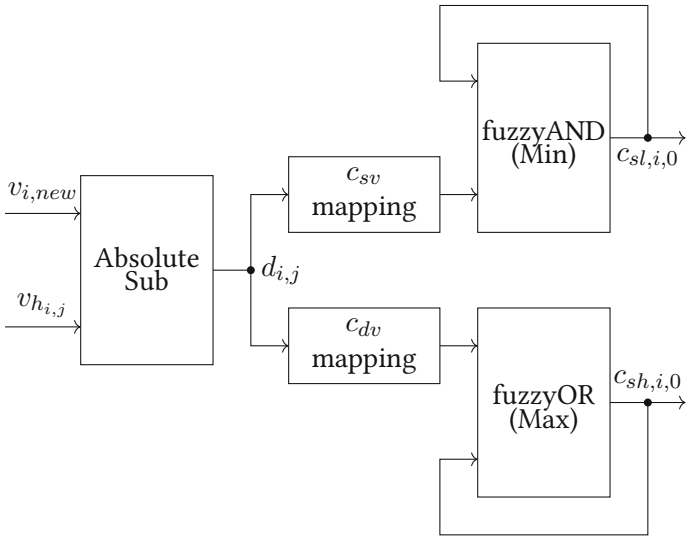


Figure 5.9: First Part of the Validity Confidence Datapath (Serial)

In addition to fuzzification and the histories, the second part of the VCDP consists of a counter, four fuzzy operations, and a Greater-Than comparison.

The device count is estimated as written in Equation 5.6. It includes a counter for iterating over both confidence histories, the confidence histories, the fuzzy operations, the SAR counter, multiplexers to use the fuzzification from the MCDP and the Greater-Than comparison. The SAR counter does not have to count beyond the history length.

$$\begin{aligned} D_{VCDP} &= D_{CTNR,DYN,SET}(H) + 2 * D_{CHISTORY}(H,C) + D_{Min_k}(C) + D_{Max_k}(C) + \\ &D_{Min2}(C) + D_{Max2}(C) + D_{CNTR,SET}(\lceil log_2(H) \rceil) + 2 * 2 * \lceil log_2(H) \rceil * D_{MUX} + D_{GT}(C) = \\ &= 48\lceil log_2(H) \rceil - 2 + 2 * D_{CHISTORY(H,C)} + 2 * (36C - 4) + 2 * (8C - 4) + \\ &+ 44\lceil log_2(H) \rceil - 2 + 8\lceil log_2(H) \rceil + 6C - 4 = \\ &= 2 * D_{CHISTORY(H,C)} + 100\lceil log_2(H) \rceil + 94C - 24 \end{aligned} \quad (5.6)$$

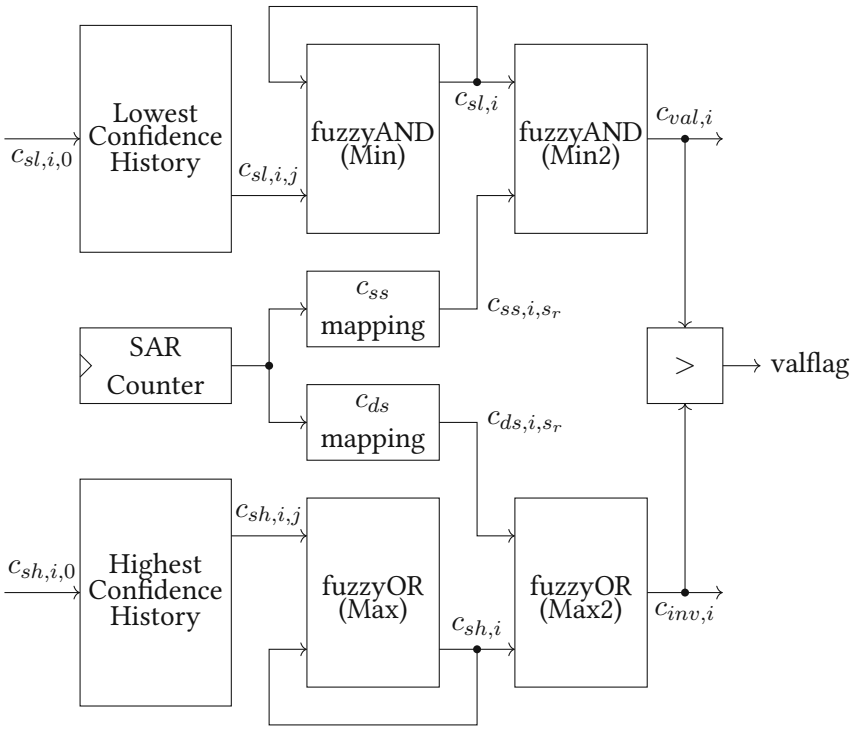


Figure 5.10: Second Part of the Validity Confidence Datapath (Serial)

The execution of the VCDP takes n cycles to iterate over n confidence values in the confidence histories.

Co-Confidenceless VCDP in Serial

Figure 5.11 shows the reduced Validity Confidence Datapath. Also here, almost half of the circuit was removed as it computed the co-confidence $c_{inv,i}$.

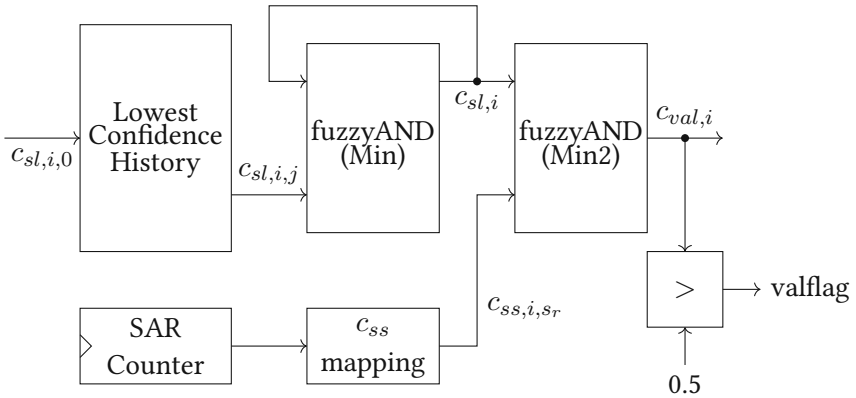


Figure 5.11: Co-Confidenceless Validity Confidence Datapath (Serial)

Equation 5.7 estimates the device count. The execution still takes n cycles for $n \leq H$ values in the confidence history.

$$
\begin{aligned}
D_{VCDP} &= D_{CTNR,DYN,SET}(H) + D_{CHISTORY}(H,C) + D_{Min_k}(C) \\
&D_{Min2}(C) + D_{CNTR,SET}(\lceil log_2(H) \rceil) + 2 * \lceil log_2(H) \rceil * D_{MUX} + D_{GT}(C) = \\
&= 48\lceil log_2(H) \rceil - 2 + D_{CHISTORY(H,C)} + 36C - 4 + 8C - 4 + \\
&+ 44\lceil log_2(H) \rceil - 2 + 4\lceil log_2(H) \rceil + 6C - 4 = \\
&= D_{CHISTORY(H,C)} + 96\lceil log_2(H) \rceil + 50C - 16 \quad (5.7)
\end{aligned}
$$

Threshold-based VCDP in Serial

The threshold-based implementation does not need a confidence history but a counter that counts H cycles when $c_{sl,i,0}$ is below 0.5 (or rather 0 since it became a 1 bit flag). As for the SAR counter that counts the number of samples since the last state reentrance, it only matters if the counter value exceeds the threshold. Therefore, the counter bit width can be set to $\lceil log_2(H/2+1) \rceil$.

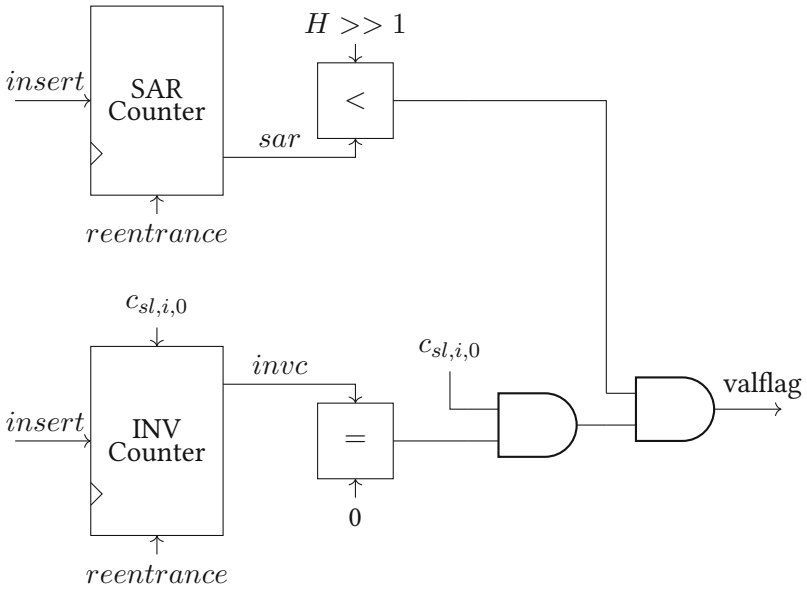


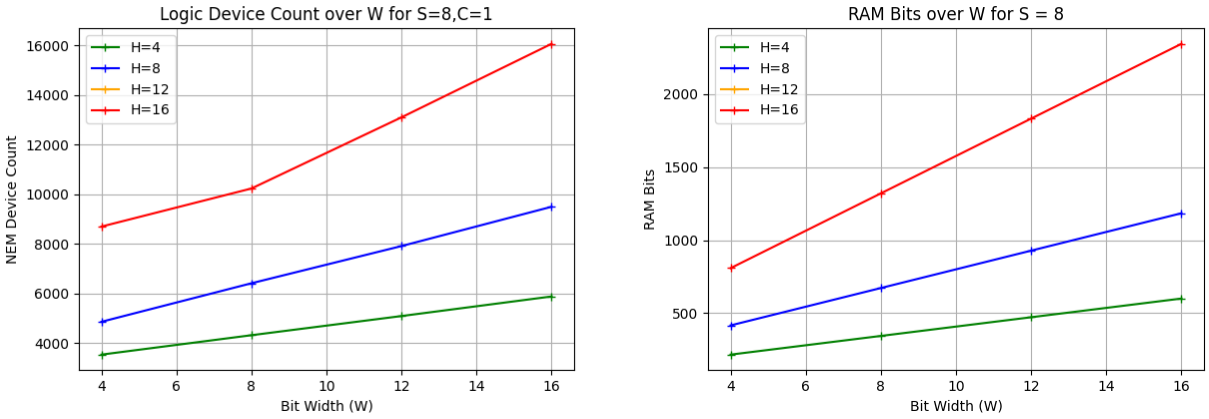
Figure 5.12: Threshold-based Validity Confidence Datapath

The estimated device count is written in Equation 5.8. The execution takes 1 cycle since the counters just have to be updated once per execution.

$$
\begin{aligned}
D_{VCDP} = D_{CNTR,DYN,SET}(H/2+1) + D_{LT}(\lceil log_2(H/2+1)\rceil) + \\
+ D_{CNTR,DYN,SET}(H) + \lceil log_2(H)\rceil * (D_{MUX} + D_{XOR}) + 2 * D_{AND} = \\
= 48\lceil log_2(H/2+1)\rceil + \lceil log_2(H/2+1)\rceil * 6 - 4 + 52\lceil log_2(H)\rceil + 4 = \\
= 54\lceil log_2(H/2+1)\rceil + 52\lceil log_2(H)\rceil \quad (5.8)
\end{aligned}
$$

5.3.3 Synthesis Results with Original Datapaths

The design has been implemented in Verilog with the same simplified control logic as the threshold-based version but without any simplification of the datapaths. Moreover, the extended cell library has also not been used. Figures 5.13 and 5.14 show the logic device count from the synthesis results and the memory requirements with a fixed number of storable Signal States S of 8 and a confidence bit width C of 1. The device count and the memory size seem to grow linearly with increasing W or H. A major difference from the results of the threshold-based Signal State Detector in the next subsection is the linear dependence of the device count on H. The results are also listed in Table 5.4. The LUT RAMs for fuzzification have not been added to these numbers and have to be considered additionally.



(a) Logic Device Count (b) Memory Bits

Figure 5.13: Device Count and Memory for CCAM Signal State Detector over W, S=8, C=1

Figure 5.15 clearly shows that the device count grows linearly with increasing C. The growth of the memory size over C has a higher gradient. The precise numbers are also written in Table 5.5.

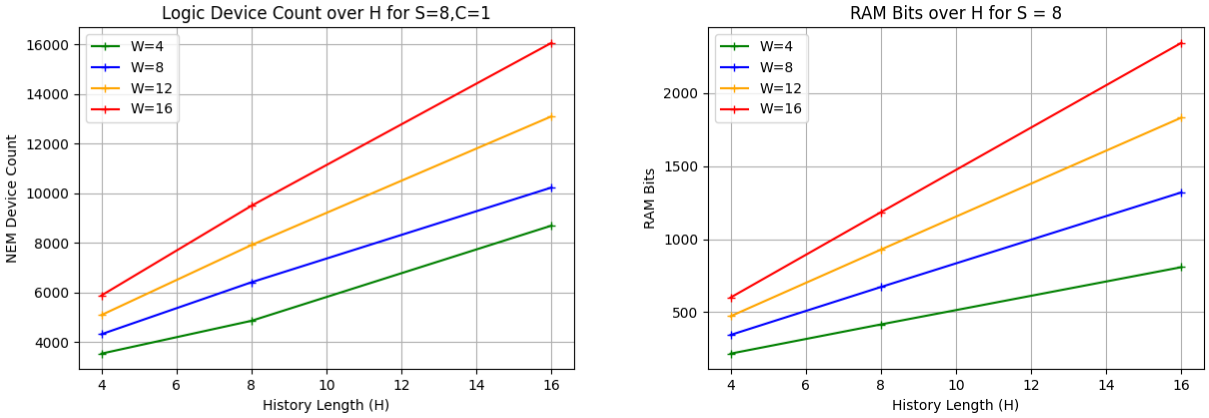
Thw worst case execution cycles can be calculated as $(2H+2) + (2H+3)S + (H+2) + 4 = (2H+3)S + 3H + 8$ when all histories are full and only the last Signal State at the last slot is matching.

Table 5.4: Device Count Results of CCAM Signal State Detector with C=1

W	H	S	C	Complete DC	Logic DC	RAM Bits
4	4	8	1	5974	3542	216
8	4	8	1	7436	4324	344
12	4	8	1	8888	5096	472
16	4	8	1	10356	5884	600
4	8	8	1	9486	4864	416
8	8	8	1	12358	6416	672
12	8	8	1	15176	7914	928
16	8	8	1	18078	9496	1184
4	16	8	1	17670	8698	808
8	16	8	1	21804	10232	1320
12	16	8	1	27270	13098	1832
16	16	8	1	32830	16058	2344

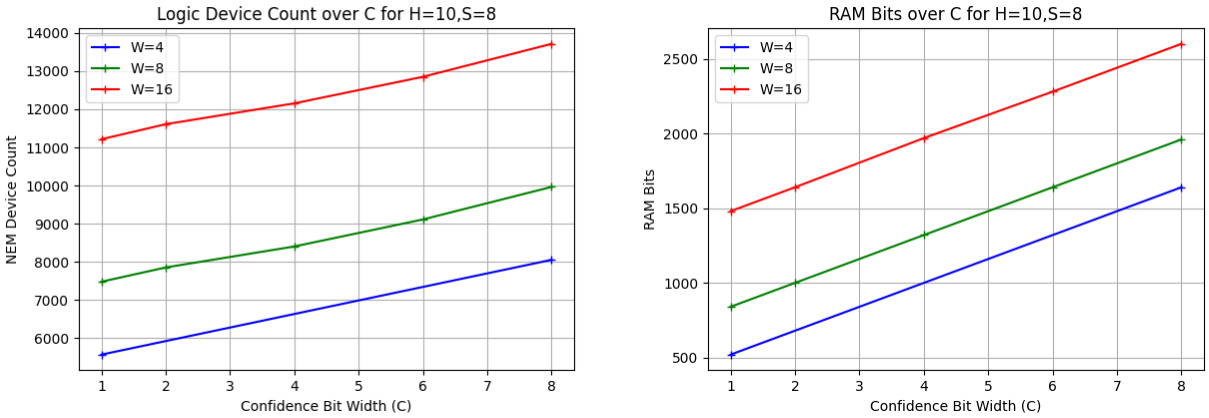
Table 5.5: Device Count Results of CCAM Signal State Detector with H=10, S=8

W	C	Complete DC	Logic DC	RAM Bits
4	1	11326	5570	520
4	8	19620	8054	1640
8	1	14876	7480	840
8	2	16080	7854	1000
8	4	18290	8404	1320
8	6	20658	9112	1640
8	8	23170	9964	1960
16	1	21890	11214	1480
16	2	23114	11608	1640
16	4	25320	12154	1969
16	6	27676	12850	2280
16	8	30200	13714	2600



(a) Logic Device Count (b) Memory Bits

Figure 5.14: Device Count and Memory for CCAM Signal State Detector over H, S=8, C=1



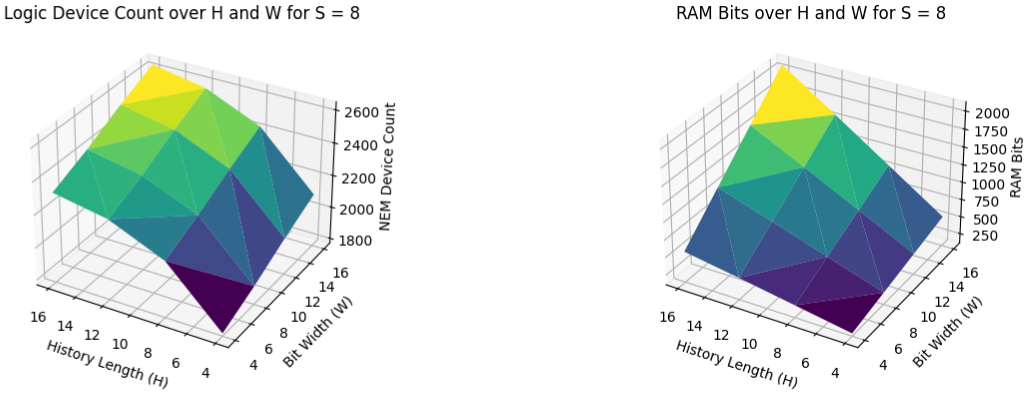
(a) Logic Device Count (b) Memory Bits

Figure 5.15: Device Count and Memory for CCAM Signal State Detector over C, H=10,S=8

5.3.4 Synthesis Results with Threshold-based Datapaths

The TCAM Signal State Detector contains both threshold-based datapaths and was synthesized with the extended gate library. Figure 5.16 shows synthesis results of the logic design excluding memory and the required memory bits in a 3-dimensional plot. The complete device count is depicted in the 3D plot in Figure 5.17. Since there are three design parameters (S,H and W), the number of storable Signal States S has been set to 8 in these diagrams to create 3-dimensional plots.

Figures 5.18 and 5.19 show the same information in a 2-dimensional plot for a more detailed inspection. Apparently, the logic device count grows logarithmically by the history length but linearly by the sample bit width. The logarithmic shape of the curves is probably due to the counting and addressing logic with adress bit width of $\lceil log_2(H) \rceil$. The memory size, however, scales linearly on both parameters, H and W.



(a) Logic Device Count (b) Memory Bits

Figure 5.16: Device Count and Memory for TCAM Signal State Detector, S = 8

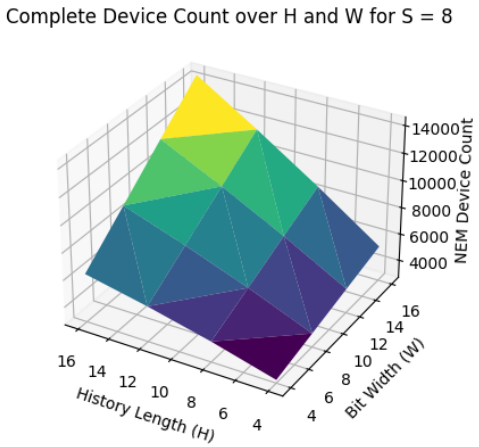
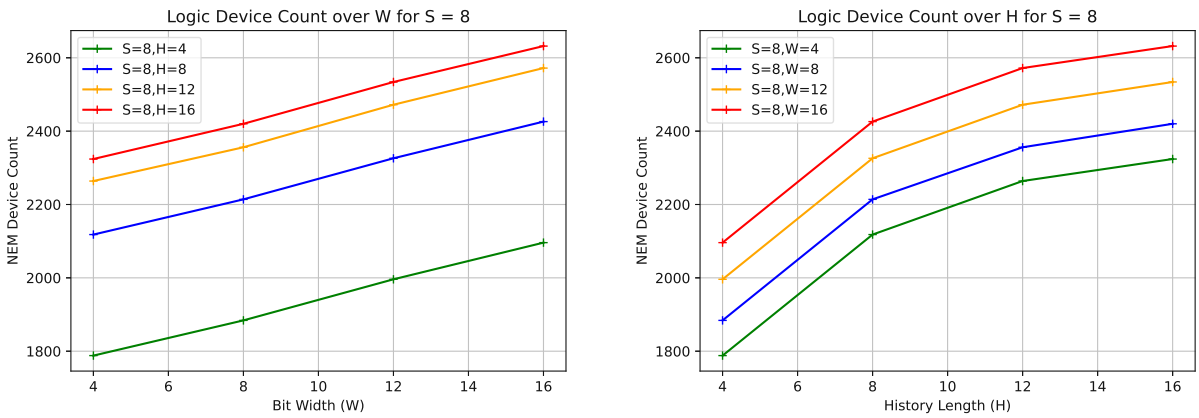
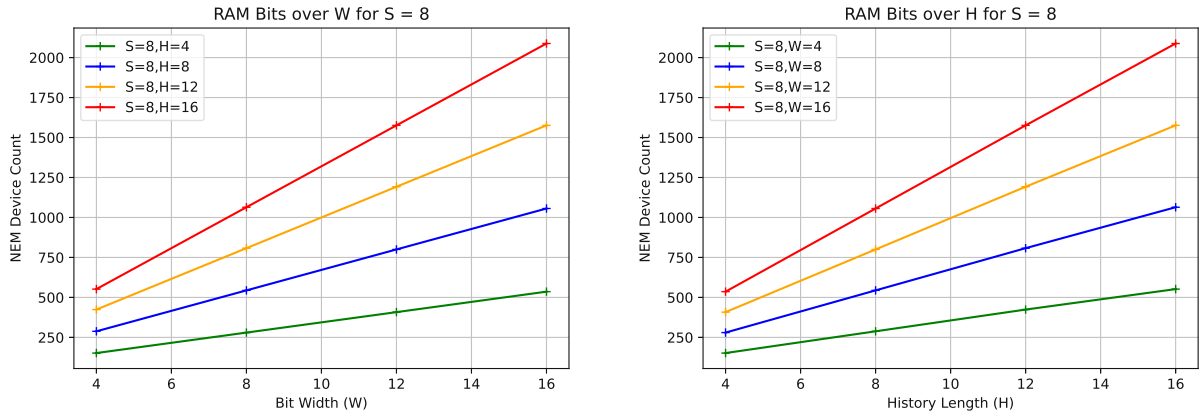


Figure 5.17: Total Device Count of TCAM Signal State Detector, S = 8



(a) Over Bit Width (b) Over History Length

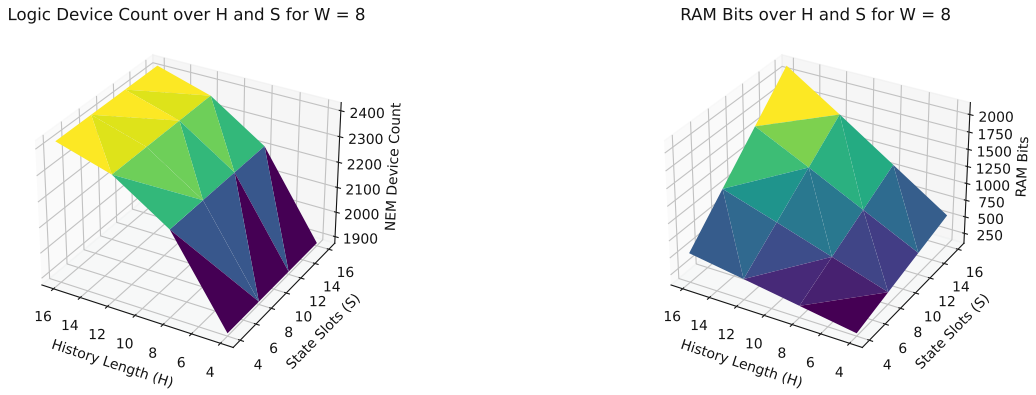
Figure 5.18: Logic Device Count for TCAM Signal State Detector, S = 8 (2D)



(a) Over Bit Width (b) Over History Length

Figure 5.19: RAM Bits for TCAM Signal State Detector, S = 8 (2D)

Figures 5.20 and 5.21 show that S almost does not affect the logic device count. The control logic includes a counter that computes the address of the Signal State but the state machines and datapaths are not affected at all. The number of memory bits, however, grows linearly with increasing S. The sample history has clearly a much stronger influence on the device logic than the remaining control logic and datapaths.



(a) Logic Device Count (b) Memory Bits

Figure 5.20: Device Count and Memory Bits for TCAM Signal State Detector, W = 8

The synthesis results and the calculated memory size are shown in Table 5.6. Some columns have been removed from the table to make it more compact.

The worst case of execution cyles of this implementation amounts to $(H+2)+(H+3)S+4$ when all Signal State Slots contain Signal States and only the last one matches.

Table 5.6: Synthesis Results of the TCAM Signal State Detector

W	H	S	Complete DC	Logic DC	RAM Bits
4	4	4	2380	1788	76
8	4	4	2836	1884	140
16	4	4	3768	2096	268
4	8	4	3194	2118	144
8	8	4	3970	2214	272
16	8	4	5542	2426	528
4	16	4	4340	2324	276
8	16	4	5756	2420	532
16	16	4	8608	2632	1044
4	4	8	2928	1788	152
8	4	8	3704	1884	280
16	4	8	5276	2096	536
4	8	8	4210	2118	288
8	8	8	5626	2214	544
16	8	8	8478	2426	1056
4	16	8	6272	2324	552
8	16	8	8968	2420	1064
16	16	8	14380	2632	2088
4	4	16	4016	1788	304
8	4	16	5432	1884	560
16	4	16	8284	2096	1072
4	8	16	6234	2118	576
8	8	16	8930	2214	1088
16	8	16	14342	2426	2112
4	16	16	10128	2324	1104
8	16	16	15384	2420	2128
16	16	16	25916	2632	4176

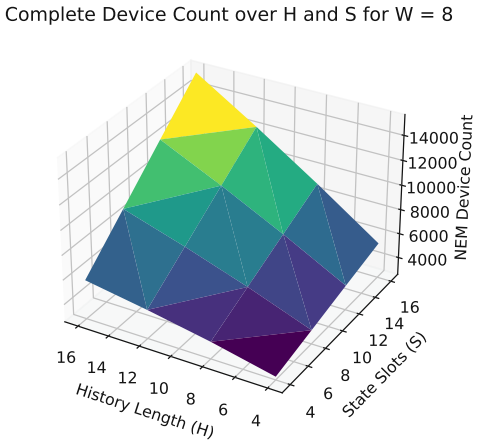


Figure 5.21: Total Device Count of TCAM Signal State Detector, W = 8

5.4 System State Detector

The System State Detector receives the outputs of all Signal State Detectors, including the Signal State IDs, the Signal State flags and the calculated confidences. Figure 5.22 depicts the topology of this module. The first component is a synchronization stage that stores all relevant Signal State Detector outputs until the computation can start. This happens when all Signal State Detectors have updated their estimation. The data is then passed to the state machine and the aggregator. As the Signal State IDs are the most important information for the System State Detector, it passes them to the memory management unit (MMU) to store them when a new System State is created. The ID checker compares the current Signal State IDs to the moored State IDs that the System State has been created with. If one of the input Signal State IDs or one of the output Signal State IDs does not match, this information is fed back to the FSM. Since the datapath of the *functioning confidence* consists only of a counter and a LUT RAM for fuzzification, no separate datapath module has been depicted in the topology.

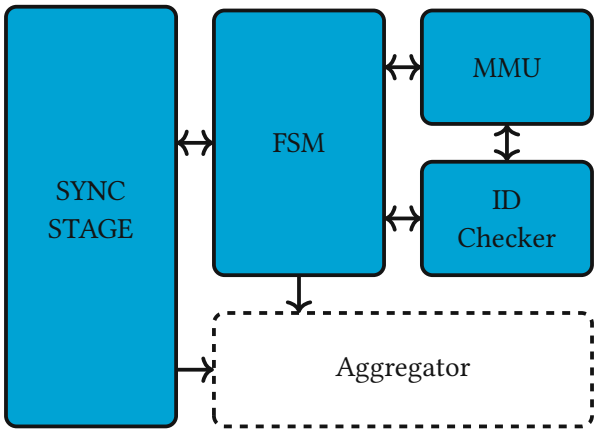


Figure 5.22: System State Detector Topology

As the focus of the research was on the more important Signal State Detector, the System State Detector was only implemented without confidence values. It supports threshold-based and confidence-based Signal State Detectors because it only needs the Signal State IDs and the Signal State flags. Therefore, the fuzzification of the *functioning confidence* has not been implemented. The information is more compact in the form of the disparity counter before fuzzification or as a flag that compares the counter value against a threshold.

In principle, the aggregation would use a fuzzy AND operation across all kind of confidences from the Signal State Detectors to get one common confidence value across the system. Co-confidences are aggregated by a fuzzy OR operation, respectively. The exact calculations to get the overall confidence have been mentioned in Section 3.2.6.

5.4.1 Synthesis Results

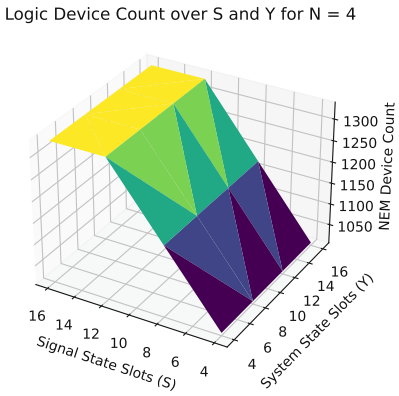
The relevant design parameters of the System State Detector are the number of observed SuO signals N, the bit width of the Signal State ID $SW = \lceil log_2(S) \rceil$ and the number of storable System States Y. If the System State Detector would also handle confidence values, the bit width of confidence values C would also have to be considered. Therefore, the design of the System State Detector is independent of the sample bit width W and the history length H. Of course, the implementation of the Signal State Detectors affects the detection of the System State but does not affect the device count and the number of required execution cycles.

Table 5.7 shows the synthesis results. Figures 5.23 and 5.24 depict them in diagrams for a fixed number of SuO signals, whereas Figures 5.25 and 5.26 show diagrams with a fixed number of System State slots. Since not the number of Signal State slots is relevent to the System State Detector but the bit width of Signal State IDs, the device count does not increase with every increase of S. The device count and the memory size have a linear dependence on the number of monitored SuO signals N. The number of available System State slots only affects the memory size linearly but has hardly any effect on the logic device count.

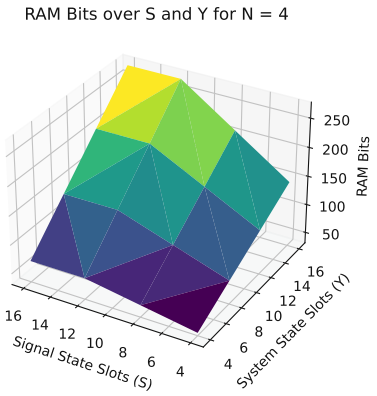
The worst case of execution cyles of the System State Detector is $2Y + 5$ cycles. Before the System State Detector starts its execution, however, all Signal State Detectors have to update their results.

Table 5.7: Synthesis Results of the TCAM System State Detector

N	S	Y	Complete DC	Logic DC	RAM Bits
2	4	4	624	490	20
2	4	8	764	490	40
2	4	12	912	490	60
2	4	16	1036	490	80
2	8	4	744	570	28
2	8	8	924	570	56
2	8	12	1112	570	84
2	8	16	1276	570	112
2	12	4	864	650	36
2	12	8	1084	650	72
2	12	12	1312	650	108
2	12	16	1516	650	144
2	16	4	864	650	36
2	16	8	1084	650	72
2	16	12	1312	650	108
2	16	16	1516	650	144
3	4	4	934	760	28
3	4	8	1114	760	56
3	4	12	1302	760	84
3	4	16	1466	760	112
3	8	4	1114	880	40
3	8	8	1354	880	80
3	8	12	1602	880	120
3	8	16	1826	880	160
3	12	4	1294	1000	52
3	12	8	1594	1000	104
3	12	12	1902	1000	156
3	12	16	2186	1000	208
3	16	4	1294	1000	52
3	16	8	1594	1000	104
3	16	12	1902	1000	156
3	16	16	2186	1000	208
4	4	4	1226	1012	36
4	4	8	1446	1012	72
4	4	12	1674	1012	108
4	4	16	1878	1012	144
4	8	4	1466	1172	52
4	8	8	1766	1172	104
4	8	12	2074	1172	156
4	8	16	2358	1172	208
4	12	4	1706	1332	68
4	12	8	2086	1332	136
4	12	12	2474	1332	204
4	12	16	2838	1332	272
4	16	4	1706	1332	68
4	16	8	2086	1332	136
4	16	12	2474	1332	204
4	16	16	2838	1332	272



(a) Logic Device Count



(b) Memory Bits

Figure 5.23: Device Count and Memory Bits for TCAM System State Detector, N = 4

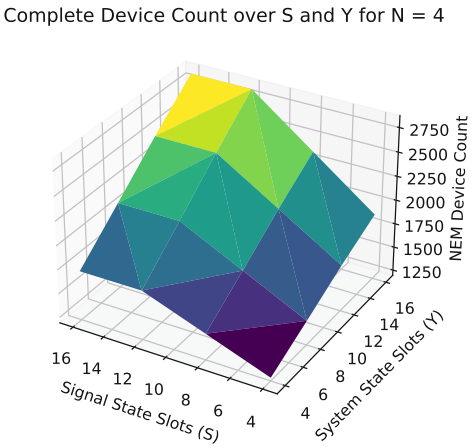
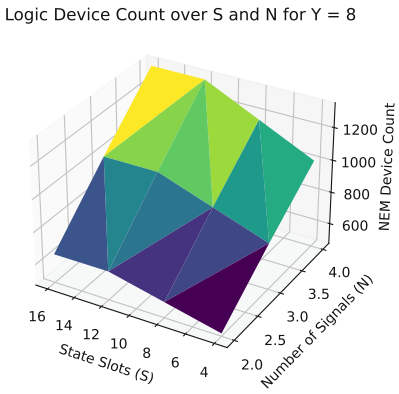
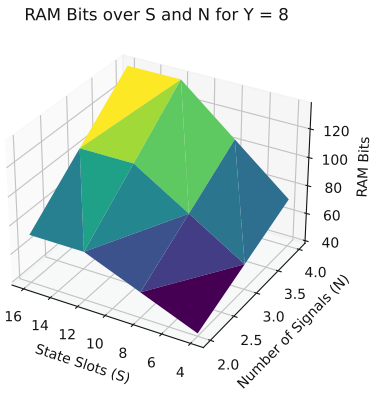


Figure 5.24: Total Device Count of TCAM System State Detector, N = 4



(a) Logic Device Count



(b) Memory Bits

Figure 5.25: Device Count and Memory Bits for TCAM System State Detector, Y = 8

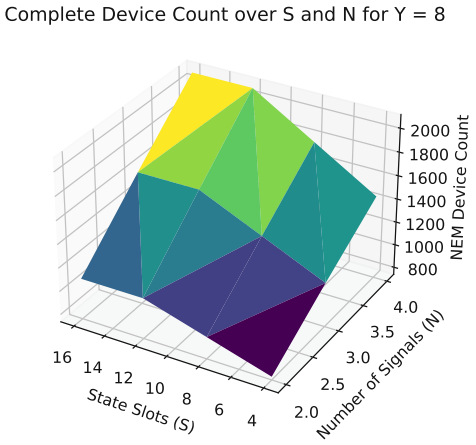


Figure 5.26: Total Device Count of TCAM System State Detector, Y = 8

5.5 CAM Accuracy Results

When configuring the fuzzification Look-Up Tables of CCAM correctly, it provides the same accuracy at an arbitrary confidence bit width as a threshold-based solution. Therefore, the results for the accuracy have been shared. Unless the confidence values are not needed in a subsequent system, there are no disadvantages of using the threshold-based TCAM instead of CCAM. The timeseries used for testing originate from the CCAM case study mentioned earlier in Section 3.1. The selected timeseries snippets are:

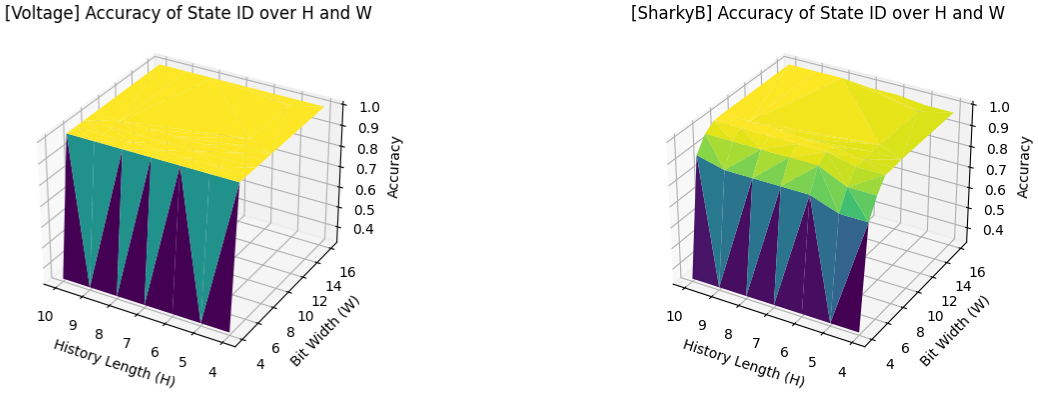
1. 20180529_NormalModeSeveralChanges
2. 20180529_BrokenSystem
3. 20180529_Drift
4. 20180529_NormalTwoTimesSameState

Since some of the signals to be monitored do not change effectively in any of the selected test cases, the number of monitored signals can be reduced. In the following measurements, the control input $Voltage$ and the SuO outputs $SharkyS$, $SharkyB$ and $Dyna$ have been observed. In the following subsections the results of the mentioned test cases are presented. The diagrams and results of test case 20180529_$NormalTwoTimesSameState$ did not add additional value and have been skipped.

5.5.1 Test Case 1: 20180529_NormalModeSeveralChanges

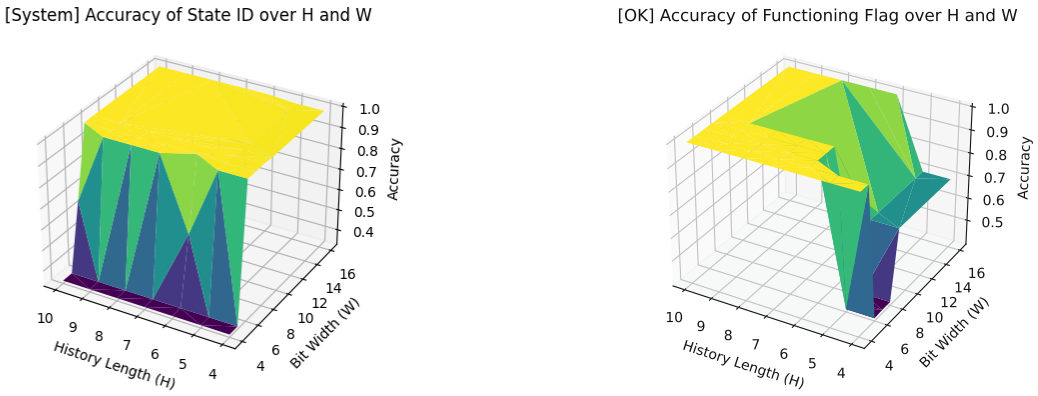
Figures 5.27, 5.28 show the results on the timeseries snippet *20180529_NormalModeSeveralChanges*. The accuracy of the detection of a Signal State drops significantly when the sample bit width is decreased too much. A bit width of 8 is sufficient for all of the relevant Signal State Detectors in the hydraulic

system data set. A bit width of 7 still gives good results but a decrease in accuracy is already noticeable for some of the observed signals. When all Signal States are derived correctly, also the System State will. Although history length does not appear to have an important effect on accuracy in Figure 5.27, the *functioning* flag produces incorrect outputs below a history length of 9. This means that the System State stays in a *potential* state that is the correct System State but detects a mismatch.



(a) Voltage Signal State ID (b) SharkyB Signal State ID

Figure 5.27: Accuracy of Signal State Detection on (1)



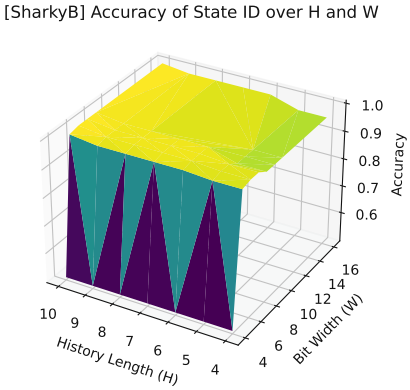
(a) System State ID (b) Functioning Flag

Figure 5.28: Accuracy of System State Detection on (1)

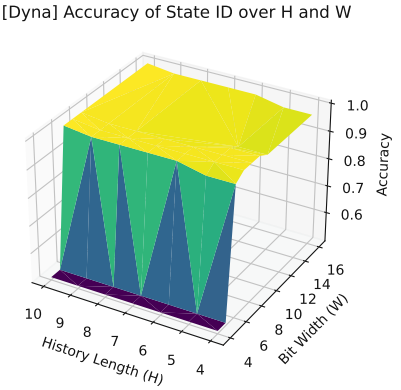
5.5.2 Test Case 2: 20180529_BrokenSystem

Figures 5.29 and 5.30 show results from a run on the timeseries *20180529_BrokenSystem* in which the SuO was indeed not working as intended. The accuracy of the *functioning* flag drops below a history length of 8. For lower sample bit widths, shorter history lengths also produce correct results, however, in this case the quantization of the samples probably mitigates the effect of outliers and noise to some

extent.

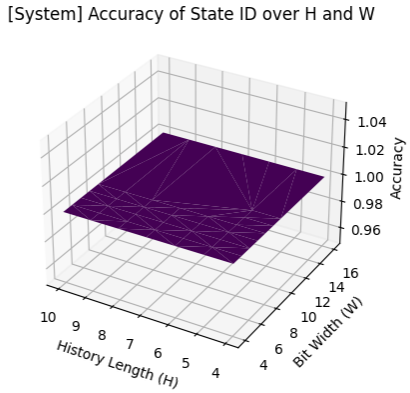


(a) SharkyB Signal State ID

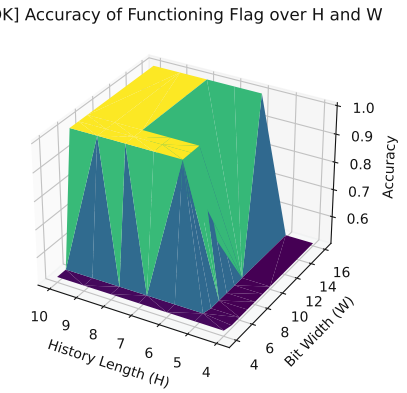


(b) Dyna Signal State ID

Figure 5.29: Accuracy of Signal State Detection on (2)



(a) System State ID

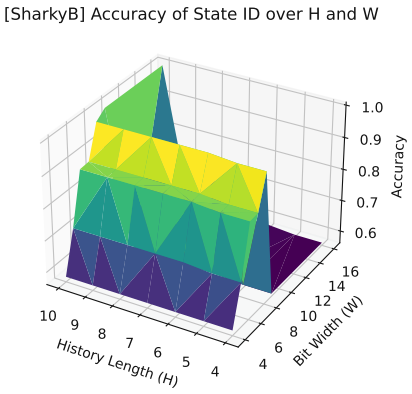


(b) Functioning Flag

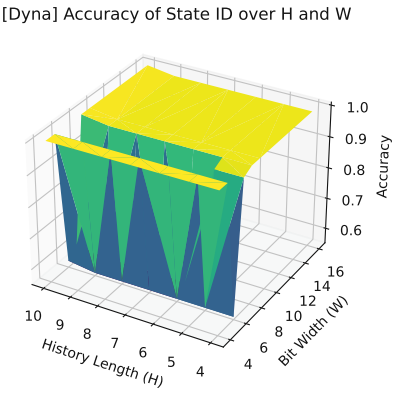
Figure 5.30: Accuracy of System State Detection on (2)

5.5.3 Test Case 3: 20180529_Drift

Figures 5.31 and 5.32 show the accuracy of the results on the *20180529_Drift* snippet. The drift detection of CCAM has not been implemented, however, the drift detection can be seen as an optional module that is able to *detect* drift but is not mitigating its effect on the state detection. This is where the sample history comes into effect. Only a slightly shorter history length of 9 already yields wrong results on this test case.

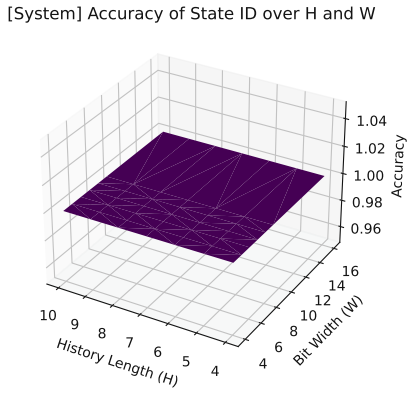


(a) SharkyB Signal State ID

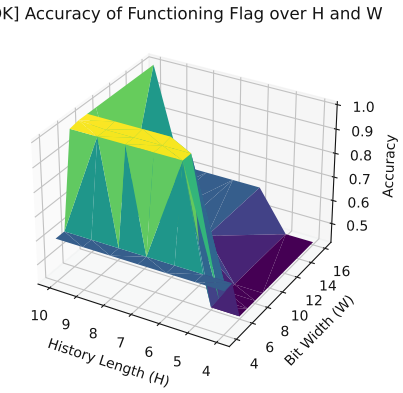


(b) Dyna Signal State ID

Figure 5.31: Accuracy of Signal State Detection on (3)



(a) System State ID



(b) Functioning Flag

Figure 5.32: Accuracy of System State Detection on (3)

5.6 Discussion

The results of the accuracy measurements show that the minimum design parameters that are acceptable for the hydraulic system data set are a sample bit width of 8 and a history length of 9 when no drift occurs. Otherwise, the history length has to be kept at 10.

Table 5.8: Maximum Used State Slots for Working Configurations

Test Case	Voltage	SharkyS	SharkyB	Dyna	System
20180529_NormalModeSeveralChanges	3	1	4	4	3
20180529_BrokenSystem	1	2	3	3	1
20180529_Drift	1	5	3	2	1
20180529_NormalTwoTimesSameState	2	1	4	3	2
Maximum	3	5	4	4	3

The maximum numbers of signal state slots and system state slots required for the working con-

figurations are shown in Table 5.8. The total maximum number of stored Signal States was 5 and the maximum number of System States was 3. As further states may be created with longer time series, at least $S = 8$ and $Y = 4$ are suggested, however, even these numbers would be most probably too low in the real application. Table 5.9 summarizes the selected design parameters.

Table 5.9: Selected Design Parameters

W	H	S	Y
8	10	8	4

The following device count requirements (Table 5.10) can be derived from these design parameter settings:

Table 5.10: Results for selected Design Parameters

Design	Opt	C	Complete DC	Logic DC	RAM Bits	WC Cycles
CCAM Signal State Detector	-	4	18290	8404	1320	222
CCAM Signal State Detector	✓	4	15312	6474	1320	222
CCAM Signal State Detector	-	1	14876	7480	840	222
CCAM Signal State Detector	✓	1	12222	5874	840	222
TCAM Signal State Detector	-	-	7818	3172	680	120
TCAM Signal State Detector	✓	-	6602	2364	680	120
TCAM System State Detector	-	-	1520	1214	52	13
TCAM System State Detector	✓	-	1466	1172	52	13

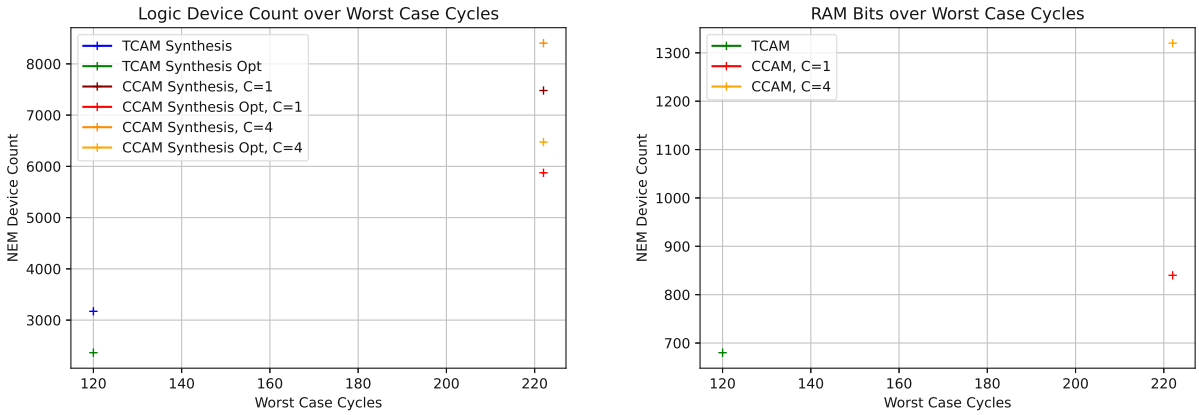
Results from synthesis with the basic and extended standard cell library are shown (Opt column). The confidence-based design has been synthesized with a confidence bit width of 4 and a confidence bit width of 1. However, a confidence value of one transforms CCAM in fact already into a threshold-based design but without the simplifications that were made possible. Fuzzification has not yet been considered and has to be added to the resulting numbers. The LUT RAM is not necessary for $C = 1$, but for $C = 4$. If an LUT RAM address range of $R = \lceil \frac{1}{5} * 2^W \rceil$ for containing the configurable slope of the fuzzy functions c_{sv} and c_{dv} are assumed, the additional memory requirement would be as shown in Table 5.11.

Table 5.11: RAM Bits of fuzzification with C = 4

Fuzzy Function	Equation Reference	RAM Bits
c_{sv}	4.12	208
c_{dv}	4.12	208
c_{ss}	4.14	55
c_{ds}	4.14	55
Total	Σ	526

Figures 5.33 and 5.34 depict the results from Table 5.10. Without surprise, the threshold-based Signal State Detector performs best in means of device count and the worst case cycle count. The device count of the confidence-based Signal State Detector with $C = 4$ amounts to about 21000 devices when

including the fuzzification. The optimized threshold-based design has only 31.4% of this device count which is approximately a reduction of 68.6%. The extension of the standard cell library reduced the logic device count of the TCAM Signal State Detector by 25.5 % which leads to an overall device count reduction of 15.5 % when including memory.



(a) Logic Device Count vs. Worst Case Cycles

(b) RAM Bits vs. Worst Case Cycles

Figure 5.33: Signal State Detector Design Comparison

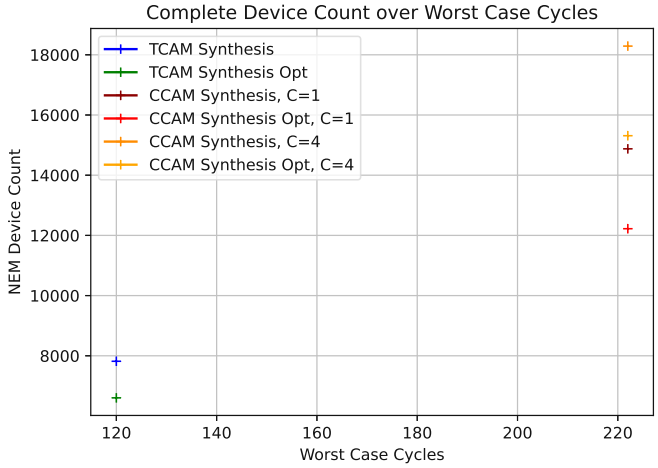


Figure 5.34: Total Device Count vs. Worst Case Cycles of Signal State Detector

5.7 Summary

This chapter presented the basic architecture and explored different datapath designs of hardware implementations of confidence-based CCAM and threshold-based TCAM. Then, synthesis and simulation results are discussed and the device count and worst case cycle count for a design parameter setting that yields satisfying accuracy for out test dataset are given.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

By going through the ASIC design flow of CCAM targeting NEMS, insight has been gained for both CCAM and NEMS. Due to the hardware restrictions and the lack of a NEMS processor, the whole algorithm has been synthesized to hardware instead of following a hardware/software co-design.

At first, the original CCAM algorithm (A) has been adapted to be more hardware-friendly by removing division operations (B). After thorough simplification, the algorithm was reduced further to a threshold-based algorithm called TCAM (C). Versions B and C have both been implemented in Verilog and synthesized to a netlist of gates constructed from NEM switches. By an extension of the standard cell library for synthesis, the synthesis results have been considerably improved. The device count of the logic design of the Signal State Detector has been decreased by 25.5 %. Furthermore, custom NEM designs for basic components like counters or an absolute subtractor have been introduced that are optimized to the capabilities of 4-T NEM devices.

In the end, the trade-off between accuracy and the device count of implementations B and C has been explored by using the example of a use case that monitors a hydraulic system. The results show an astonishing decrease in the device count of 68.6% from CCAM to TCAM for the setting of the selected design parameters.

6.2 Future Work

Future work could include applying CCAM to an actual NEMS use case in a harsh environment. The dataset that has been used for the validation of the results was captured from a hydraulic system for previous work on CCAM. As fuzzy logic operations have been implemented, the acquired knowledge could also be propagated to different NEMS fuzzy logic designs.

With regard to the underlying NEMS technology, during the design process for the CCAM case study, the strengths and weaknesses of the current state of the NEMS design flow have become apparent. A big advantage of NEMS designs is that they require a lower device count than conventional CMOS designs due to the 4-T and 7-T devices, however, the potential is still not utilized to the full extent. Therefore, one of the next steps is to adapt a synthesis tool for optimization toward NEMS.

Another possibility is to add more standard cells to the NEMS standard cell library mentioned in [2] and to further optimize the ones already present. Especially the flip-flops in the standard cell library do not use 4-T devices although they are by far the most area-demanding standard cells. This leads also to the preference for asynchronous design techniques. Synchronous designs require a clock signal that is still difficult to provide in NEMS and this results in high switching activity, and thus in increased power consumption and faster contact degradation of the NEMS devices. Therefore, asynchronous design suits the technology and implementing basic building blocks for asynchronous designs in NEMS would pave the way for an asynchronous CCAM implementation in the future.

Bibliography

[1] T. J. Ross, *Fuzzy logic with engineering applications.* John Wiley & Sons, 2005.

[2] E. Worsey, "Nanoelectromechanical technology for radiation and temperature harsh environments," Ph.D. dissertation, University of Bristol, 2024.

[3] D. Hauer, M. Götzinger, A. Jantsch, and F. Kintzler, "Context aware monitoring for smart grids," in *2021 IEEE 30th International Symposium on Industrial Electronics (ISIE)*, 2021, pp. 1–6.

[4] M. Götzinger, D. Juhász, N. Taherinejad, E. Willegger, B. Tutzer, P. Liljeberg, A. Jantsch, and A. M. Rahmani, "Rosa: A framework for modeling self-awareness in cyber-physical systems," *IEEE Access*, vol. 8, pp. 141 373–141 394, 2020.

[5] H. Watanabe, W. Dettloff, and K. Yount, "A vlsi fuzzy logic controller with reconfigurable, cascadable architecture," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 2, pp. 376–382, 1990.

[6] I. Hamzaoglu, B. Ayrancioglu, and H. Azgin, "Low error approximate absolute difference hardware," in *2022 IEEE International Conference on Consumer Electronics (ICCE).* IEEE, 2022, pp. 1–6.

[7] K. Ø. Arisland, A. C. Aasbø, and A. Nundal, "Vlsi parallel shift sort algorithm and design," *Integration*, vol. 2, no. 4, pp. 331–347, 1984.

[8] S. S. Ray, D. Adak, and S. Ghosh, "Worst case o (n) comparison-free hardware sorting engine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 10, pp. 3332–3345, 2021.

[9] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded system design: modeling, synthesis and verification.* Springer Science & Business Media, 2009.

[10] M. Götzinger, N. Taherinejad, H. A. Kholerdi, A. Jantsch, E. Willegger, T. Glatzl, A. M. Rahmani, T. Sauter, and P. Liljeberg, "Model-free condition monitoring with confidence," *International Journal of Computer Integrated Manufacturing*, vol. 32, no. 4-5, pp. 466–481, 2019.

[11] G. H. Bazan, P. R. Scalassara, W. Endo, A. Goedtel, W. F. Godoy, and R. H. C. Palácios, "Stator fault analysis of three-phase induction motors using information measures and artificial neural networks," *Electric Power Systems Research*, vol. 143, pp. 347–356, 2017.

[12] J. D. Reynolds, S. Rana, E. Worsey, Q. Tang, M. K. Kulsreshath, H. M. Chong, and D. Pamunuwa, "Single-contact, four-terminal microelectromechanical relay for efficient digital logic," *Advanced Electronic Materials*, vol. 9, no. 1, p. 2200584, 2023.

[13] L. Boodhoo, Y. P. Lin, H. M. H. Chong, Y. Tsuchiya, T. Hasegawa, and H. Mizuta, "Energy reversible si-based nems switch for nonvolatile logic systems," in *The 8th Annual IEEE International Conference on Nano/Micro Engineered and Molecular Systems*, 2013, pp. 558–561.

[14] J. Ng, W. Zhang, Y.-H. Xie, Q. Chen, R. Ma, and A. Wang, "Optimization of suspended graphene nems devices for electrostatic discharge applications," in *2017 IEEE 12th International Conference on Nano/Micro Engineered and Molecular Systems (NEMS)*, 2017, pp. 364–369.

[15] T.-S. Kim, Y.-B. Lee, S.-Y. Lee, S.-J. Lee, and J.-B. Yoon, "A reliable release method for a back-end-of-line nems switch of a monolithic three-dimensional integrated cmos-nems circuit," in *2023 IEEE 36th International Conference on Micro Electro Mechanical Systems (MEMS)*, 2023, pp. 76–79.

[16] T. Qin, S. J. Bleiker, S. Rana, F. Niklaus, and D. Pamunuwa, "Performance analysis of nanoelectromechanical relay-based field-programmable gate arrays," *IEEE Access*, vol. 6, pp. 15 997–16 009, 2018.

[17] M. Götzinger, N. TaheriNejad, H. A. Kholerdi, and A. Jantsch, "On the design of context-aware health monitoring without a priori knowledge; an ac-motor case-study," in *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2017, pp. 1–5.

[18] M. Gotzinger, E. Willegger, N. TaheriNejad, A. Jantsch, T. Sauter, T. Glatzl, and P. Lilieberg, "Applicability of context-aware health monitoring to hydraulic circuits," in *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2018, pp. 4712–4719.

[19] M. Götzinger, "Self-aware reliable monitoring," Doctoral dissertation, University of Turku, Turku, Finland, October 2021.

[20] D. Schnoell, "Analysing and extending model-free condition monitoring with confidence," Diploma Thesis, Vienna University of Technology (TU Wien), 2021.

[21] L. A. Zadeh, "Fuzzy sets," *Information and control*, vol. 8, no. 3, pp. 338–353, 1965.

[22] W. Van Leekwijck and E. E. Kerre, "Defuzzification: criteria and classification," *Fuzzy sets and systems*, vol. 108, no. 2, pp. 159–178, 1999.

[23] V. G. Marot, M. K. Kulsreshath, Q. Tang, M. B. Krishnan, and I. Pamunuwa, "Reconfigurable non-volatile 4-way routing switch with zero standby power," 2024.

[24] E. Worsey, Q. Tang, M. B. Krishnan, M. K. Kulsreshath, and D. Pamunuwa, "Nanoelectromechanical analog-to-digital converter for low power and harsh environments," in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2024, pp. 1–5.

[25] T.-J. K. Liu, N. Xu, I.-R. Chen, C. Qian, and J. Fujiki, "Nem relay design for compact, ultra-low-power digital logic circuits," in *2014 IEEE International Electron Devices Meeting*. IEEE, 2014, pp. 13–1.

[26] S. Dutta *et al.*, "Floating-point unit (fpu) designs with nano-electromechanical (nem) relays," Ph.D. dissertation, Massachusetts Institute of Technology, 2013.

[27] K. Dwan and D. Markovic, "Logic synthesis of mem relay circuits," Ph.D. dissertation, Citeseer, 2011.

[28] D. Lee, W. S. Lee, C. Chen, F. Fallah, J. Provine, S. Chong, J. Watkins, R. T. Howe, H.-S. P. Wong, and S. Mitra, "Combinational logic design using six-terminal nem relays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 5, pp. 653–666, 2013.

[29] W. Haaswijk, L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Nem relay design with biconditional binary decision diagrams," in *Proceedings of the 2015 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH´ 15)*. IEEE, 2015, pp. 45–50.

[30] L. G. Amaru, *New Data Structures and Algorithms for Logic Synthesis and Verification*. Springer, 2017.

[31] Liberty reference manual (version 2007.03). [Online]. Available: https://people.eecs.berkeley.edu/~alanmi/publications/other/liberty07_03.pdf

[32] M. Abdelrasoul, A. S. Shaban, and H. Abdel-Kader, "Fpga based hardware accelerator for sorting data," in *2021 9th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC)*. IEEE, 2021, pp. 57–60.

[33] A. F. Alif, S. M. R. Islam, and P. Deb, "Design and implementation of sorting algorithms based on fpga," in *2019 International Conference on Computer, Communication, Chemical, Materials and Electronic Engineering (IC4ME2)*. IEEE, 2019, pp. 1–4.

[34] A. H. Jalilvand, F. S. Banitaba, S. N. Estiri, S. Aygun, and M. H. Najafi, "Sorting it out in hardware: A state-of-the-art survey," *arXiv preprint arXiv:2310.07903*, 2023.

[35] T. Nakatani, S.-T. Huang, B. Arden, and S. Tripathi, "K-way bitonic sort," *IEEE Transactions on Computers*, vol. 38, no. 2, pp. 283–288, 1989.

[36] R. Ayoubi, S. Istambouli, A.-W. Abbas, and G. Akkad, "Hardware architecture for a shift-based parallel odd-even transposition sorting network," in *2019 Fourth International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*. IEEE, 2019, pp. 1–6.

[37] M. H. Najafi, D. J. Lilja, M. Riedel, and K. Bazargan, "Power and area efficient sorting networks using unary processing," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 125–128.

[38] G. Campobello, G. Patanè, and M. Russo, "On the complexity of min–max sorting networks," *Information Sciences*, vol. 190, pp. 178–191, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025511006360

[39] A. H. Jalilvand, S. N. Estiri, S. Naderi, M. H. Najafi, and M. Imani, "A fast and low-cost comparison-free sorting engine with unary computing: late breaking results," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1390–1391.

[40] P. Eles, K. Kuchcinski, and Z. Peng, *System synthesis with VHDL*. Springer Science & Business Media, 2013.