Athens Journal of Technology & Engineering 2026, 12: 1-26
https://doi.org/10.30958/ajte.X-Y-Z

Automata Decomposition and Pipeline Synthesis based on Krohn-Rhodes Theory for Asynchronous Dual-Rail Domino Logic

By Florian Deeg[*] & Sebastian M. Sattler[±]

This paper examines the application of Krohn–Rhodes decomposition for the systematic synthesis of cascaded asynchronous control automata. By decomposing finite automata into hierarchically organized permutation and reset components, directed transition behaviors can be efficiently modularized. This cascade structure provides a formal basis for the construction of single-step, hazard-free circuits, which are of central importance in self-timed or Globally Asynchronous Locally Synchronous systems in particular. Based on the Tree Subset Automaton method, complex state machines are gradually converted into structured cascades. Using various machine components (e.g., oscillators, reset logics, stabilization units), the decomposability and synthesizability are demonstrated. We examine two different types of decomposition: inhibition and complementary decomposition. In addition, we show how these cascades can be directly converted into pipeline architectures to accelerate serial logic structures and enable parallel processing of asynchronous data flows. Implementation on an FPGA confirms the practical feasibility of the proposed method and demonstrates its potential in terms of modularity, single-step operation, and deterministic signal processing in completely clock-free digital systems. The various types of decomposition are compared in the FPGA in terms of their resource requirements and performance.

Keywords: *Krohn-Rhodes decomposition, asynchronous circuits, cascade synthesis, self-timed, pipeline synthesis*

Introduction

The ever-increasing demands on energy efficiency, clock independence, and modulating capability of digital systems have led to a growing interest in asynchronous circuit technology in recent years. In contrast to classic synchronous designs, which rely on a global clock, asynchronous systems operate clockless or self-clocked and use local handshake protocols for coordination. This not only allows for better adaptation to local runtimes, but also reduces dynamic losses, electromagnetic interference, and clock distribution problems, which are key advantages in highly integrated system-on-chip architectures or globally asynchronous locally synchronous (GALS) approaches [1]. A central design principle of such systems is cascading, i.e., the structured composition of complex controllers from simpler machine units. Particularly in the context of directional transition behavior—as occurs in functional

[*]PhD Student, Friedrich-Alexander-University Erlangen-Nuremberg, Germany.
[±]Chair of Reliable Circuits and Systems, Friedrich-Alexander-University Erlangen-Nuremberg, Germany.

state machines with reset or permutation components—the cascade enables clear decoupling and modularization of the control logic. This is important not only for step-by-step design, but also for subsequent optimization. The decomposition of finite automata into elementary cascade components can be elegantly described by Krohn–Rhodes theory. This provides a mathematically sound method for representing arbitrary deterministic finite automata as a composition of simple reset and permutation automata. This cascade structure is not only conceptually useful, but also significant from a practical point of view: its hierarchical feed-forward topology creates a natural single-step nature of the transitions, which is a property that contributes significantly to the avoidance of hazards and races in asynchronous designs [2]. Another advantage of the cascade structure is its direct transferability to pipeline architectures. By placing explicit buffers for each stage, serial state sequences can be converted into parallel processing paths, which is a key mechanism for accelerating data-driven processes in asynchronous systems. In contrast to extant approaches for asynchronous control synthesis, which are predominantly based on Petri nets, handshake protocols, or high-level behavioral descriptions, the approach presented in this work commences from an algebraic automaton-theoretic perspective. While Krohn–Rhodes theory has been extensively studied in a theoretical context, its systematic application to the synthesis of implementable, hazard-free asynchronous control circuits has received little attention so far. To the best of the authors' knowledge, there is no recent work that applies Krohn–Rhodes decomposition as a synthesis methodology for implementable asynchronous control circuits, which underlines the exploratory character of the present contribution. This work addresses this gap by demonstrating how Krohn–Rhodes decomposition, in combination with Tree Subset Automata, can be utilized as a practical synthesis methodology that directly maps to cascaded and pipeline-capable hardware structures. The aim of this work is therefore to investigate the Krohn–Rhodes decomposition as a methodological tool for synthesizing cascaded and pipeline-capable asynchronous control automata. Using concrete automata examples, we show how directed transition behavior can be controlled by structured cascades and systematically implemented by targeted subset compositions. In summary, this work contributes to establishing the Krohn–Rhodes decomposition as a synthesis tool for asynchronous control logic, systematically representing directed transition behavior in modular automaton structures, and making cascades specifically usable as a basis for self-clocked pipeline architectures.

Recent research in asynchronous control synthesis and formal hardware generation has predominantly focused on behavioral and concurrency-oriented design approaches. Petri-net- and Signal Transition Graph–based synthesis methods and handshake-based design methodologies derive asynchronous circuits from behavioral specifications while ensuring properties such as speed independence, deadlock freedom, or protocol correctness [3,4]. In parallel, automata-based control and reactive synthesis methods generate controllers from formal specifications by enforcing safety and liveness constraints through algorithmic construction [5].

In contrast to these approaches, the method presented in this work follows an algebraic and structural perspective. Rather than starting from behavioral descriptions or temporal specifications, the synthesis process is derived directly from the algebraic decomposition of the transition monoid of a finite automaton using Krohn–Rhodes

theory. The resulting cascade structure provides an inherently hierarchical and feed-forward organization of control logic, which naturally supports single-step transitions and facilitates hazard-free implementation in asynchronous environments. The proposed methodology should therefore be understood as complementary to existing synthesis paradigms, providing a structural decomposition-based pathway from automata theory to implementable asynchronous hardware.

Structure of the Paper

In the further course of this work, the theoretical foundations of Krohn–Rhodes decomposition and the properties of reset and permutation automata as elementary building blocks are first described, and general parts of an automaton are discussed. Building on this, the systematic methodology for subset composition from [6] is discussed, which can be used to construct automaton hierarchies that are both cascadable and pipelined. In the main part, exemplary automata with different transition behaviors (reset, permutation, stabilization, oscillation) are analyzed, decomposed, and gradually converted into a cascaded form. Subsequently, it is shown how this cascade structure can be converted into a pipeline architecture to enable temporally and logically correct parallel processing. This is followed by practical implementation on an FPGA, in which the cascade and pipeline are realized in a comparable logic structure and evaluated in terms of their area requirements and performance.

Theoretical Basis of the Krohn–Rhodes Decomposition: Concepts and Applications in Digital Systems

In this section, we explain the theoretical foundations of the fundamental Krohn–Rhodes theorem from the perspective of digital automata.

Algebraic Foundations: Automata, Monoids, and Groups

The Krohn–Rhodes theorem states that every automaton A can be decomposed, up to homomorphism, into a cascade of simpler automata [7,8]. A deterministic finite automaton can be described algebraically by the tuple $A = \{X, Y, Z, \delta, \lambda\}$. Here, δ is the transition function and λ is the output function of a Mealy automaton. Each input sequence causes a state transformation or confirmation; the set of all possible state transitions forms a finite semigroup (with the identity element, a monoid). This so-called transition monoid completely describes the internal structure of the automaton. Of central importance is whether this monoid is aperiodic (i.e., has no non-trivial inverses) or contains group elements. A simple group is a finite group without non-trivial normal subgroups and thus represents a fundamental building block in group theory. The central insight of Krohn and Rhodes was that every finite transition monoid (and thus every finite automaton) can be converted into a composition of such elementary building blocks, analogous to the decomposition of groups. These

simpler automata are minimal, i.e., they cannot be further reduced (cf. prime factor decomposition). The decomposition is carried out in two basic building blocks: primes (groups) and units (aperiodic monoids, in particular the reset monoid). Their combination is described by so-called semi-direct products or iterated wreath products.

Permutation and Reset Automata as Elementary Building Blocks

The Krohn-Rhodes decomposition is based on two fundamental types of automata: permutation automata and reset automata, also referred to as primes and units, respectively. In a permutation automaton, each input operation corresponds to a permutation of the state set—all transitions are completely invertible:

- S is a group with a unique multiplication structure (permutation), i.e., each element has an inverse (there is a reverse transition for every transition).
- S has no proper normal subdivisions except for the trivial subgroup {e} and itself (except for renaming). Such automata have no reducible substructures.

In contrast, a reset automaton has at least one input that transfers the automaton to a fixed target state regardless of the current state. These reset transitions are not invertible and destroy state information. The semigroup U3 is a three-element semigroup with certain multiplication laws:

$$U3 = \{u2, u1, u0\}$$

The possible units are the substructures of U3, specifically:

- U0 = {1}: The trivial semigroup with only one element, which corresponds to an automaton with a single state.
- U1 = R(∞): A semigroup with an absorbing state. Once reached, the automaton remains permanently in that state.
- U2: A two-element semi-group representing two stable states, e.g., a flip-flop.

These two types of automata can be roughly characterized as permutative (information-preserving, combinational in their transition behavior) or reset-like (memory-reducing, with information loss). Permutation transitions have full rank (bijective), reset transitions have rank 1 (not
bijective). The representation of a permutation automaton can be seen in Figure 1 ($\bar{X}$ meaning complement of X), that of a reset automaton in Figure 2.

Figure 1. *Permutation Automaton with State Z and Input X*

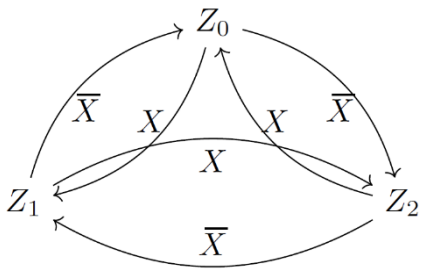
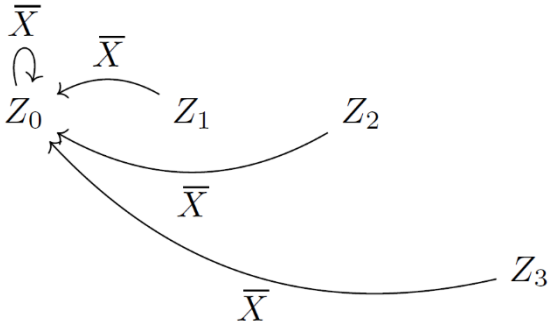


Figure 2. *Reset Automaton*



The Krohn–Rhodes Theorem: Cascade Decomposition

As already mentioned, the Krohn–Rhodes theorem states that every finite automaton A can be realized as a homomorphic image of a cascade A0 ◦ A1 ◦ · · · ◦ Ak whose transitions consist exclusively of permutations and resets. The cascade has a hierarchical (feed-forward) structure and does not allow feedback. Algebraically, this corresponds to iterated semi-direct products. Group structures only occur in the decomposition if the original automaton has non-trivial group components. For purely aperiodic automata, a decomposition into pure resets is sufficient. The decomposition typically alternates between permutation groups (primes) and aperiodic semigroups (units), always beginning and ending with a unit. The minimum number of groups occurring in such a decomposition is referred to as the complexity θ(S) of the semi-group S. This cascade structure represents a coordinate system of the original automaton: The states are represented by a structured combination of sub-automata, which allows for a modular structure and systematic analysis.

Relevance to digital systems

The Krohn-Rhodes decomposition opens up several perspectives for electrical engineering and technical computer science:

- Modularization: Complex control systems can be broken down into smaller, manageable modules, e.g., as flip-flop stages or counters.

- Hierarchization: The structure promotes a design and control hierarchy in which higher-level automata take over coarse control.
- Interpretation: Permutative parts correspond to memoryless logic (combinatorics), reset-like parts store states (e.g., flip-flops).
- Analysis: The decomposition makes group freedom (counter freedom) explicitly visible and provides clues for implementation using digital logic.

Overall, the Krohn-Rhodes decomposition offers a mathematically sound method for structured analysis and modular design of digital systems. We want to use the Krohn-Rhodes decomposition to convert general automata, see Figure 3, into cascades, see Figure 4, since cascades generate single-step execution due to their structure and thus no hazard problems can occur.

Figure 3. *General Automaton*

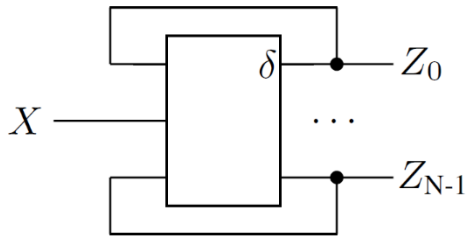
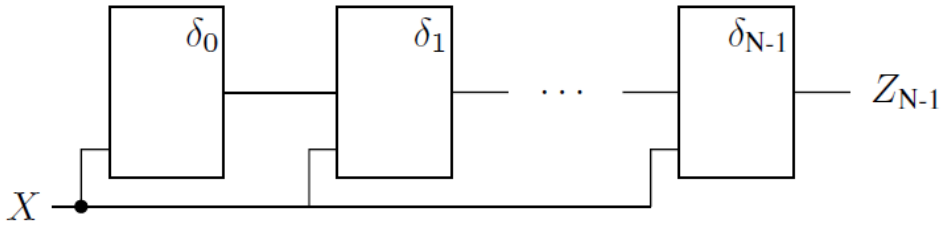


Figure 4. *General Cascaded Automaton*



Parts of Automata

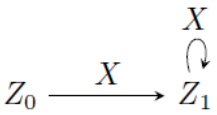
State transition functions can consist of different categories. To consider the general applicability of decomposition, let us briefly list the different parts of an automaton categorically.

Classification of Transition Behavior

- **Reset**: $\exists Zi \in Z: \forall x \in X, f(Zi, x) = Z0$
- **Permutation**: $\delta(Z1, a) = Z0 \wedge \delta(Z0, a) = Z1$ bijective
- **Stabilization**: $\delta(Z0, x) = \delta(Z1, x) = Z1$, incoming edge leads to a self-loop
- **Oscillation**: $\delta(Z1, x) = Z0 \wedge \delta(Z0, x) = Z1$ permutates subset $S \subset Q$ cyclic
- **Isolated State**: Z0 is not reachable if there is no incoming edge $x \in \Sigma*$ with $\delta(Z, x) = Z0$

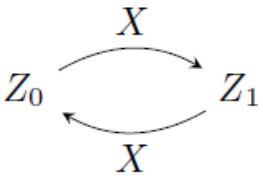
All other automaton parts are combinations of these. Of course, there are also non-deterministic automaton parts (two output edges with the same assignment in different states), but such non-deterministic cases are excluded, since only deterministic behavior is intended. The following section shows example automaton graphs that visualize the parts of automata. As already mentioned, in the reset automaton part, the edge x leads from each state to a defined state, see Figure 2. This is also called the state-independent or combinatorial part. The permutation, on the other hand, leads from one state with X and, with the complementary edge X from the subsequent state, back to the state, see Figure 1. The permutation therefore still contains the state information, since the complementary edge clearly reaches the previous state. Function stabilization can be seen as an intrinsic edge in the automaton graph, see Figure 5.

Figure 5. *Function Stabilization*



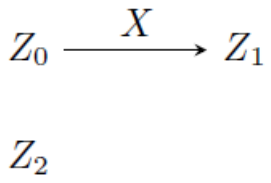
Oscillation, on the other hand, jumps back and forth between at least two states with a constant input edge, see Figure 6. It is the dual component to stabilization, not self-stabilizing.

Figure 6. *Oscillating Automaton*



An isolated state is a state that cannot be reached because it has no incoming edge, see Figure 7.

Figure 7. *Isolated Node Z2*



It is advisable to secure isolated states so that they contain, for example, a 1-edge in a defined state in order to prevent deadlocks.

Subset Composition

In this section, we want to show how composition occurs as a cascade by setting up a tree subset automaton (TSA) [6]. The setup of the TSA for an automaton proceeds in the following steps.

1) For each state: Split outgoing edges
2) Arrange steps according to the number of states, starting at the head of the automaton
3) Each level consists of at least two states, which must be complete (the conjunction of all outgoing edges must equal 1)
4) Move subsets under states that have the same initial behavior (Multiset)

Let's briefly review the decomposition shown in [6] and then look at an example with more edges.

Example Decomposition

Given is the automaton graph [6] in Figure 8 with its truth table in Table 1.

Figure 8. *Automaton Graph of given Example*

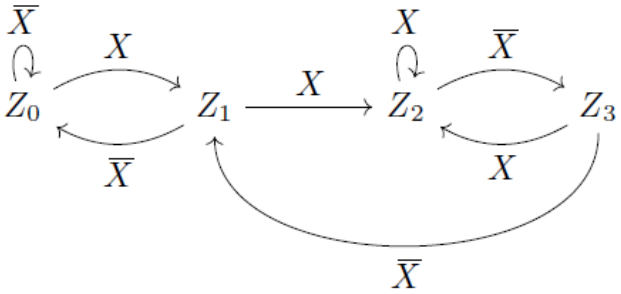
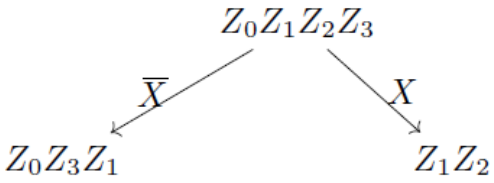


Table 1. *Truth Table of given Automaton*

Z	z1	z0	x	z1	z0
Z0	0	0	0	0	0
	0	0	1	0	1
Z1	0	1	0	0	0
	0	1	1	1	0
Z2	1	0	0	1	1
	1	0	1	1	0
Z3	1	1	0	0	1
	1	1	1	1	0

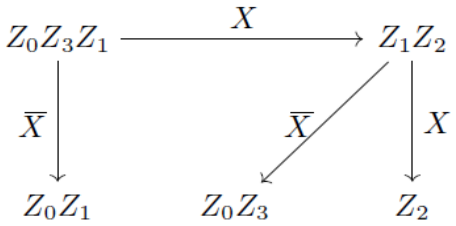
We now want to set up the TSA by summarizing all states as a set and dividing the possible transitions over the two edges X and X, i.e., performing step 1.

Figure 9. *First Stage of example TSA*



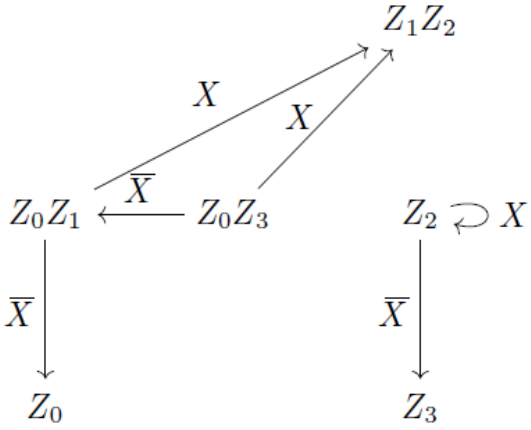
Only states Z0, Z1, and Z3 have incident edges for X, and only states Z1 and Z2 have incident edges for X. In the second step, we again divide the subsets into X and X. This time, however, we use the starting points of the subsets from the first level (i.e., there are two starting points), see Figure 10.

Figure 10. *Second Stage of the TSA*



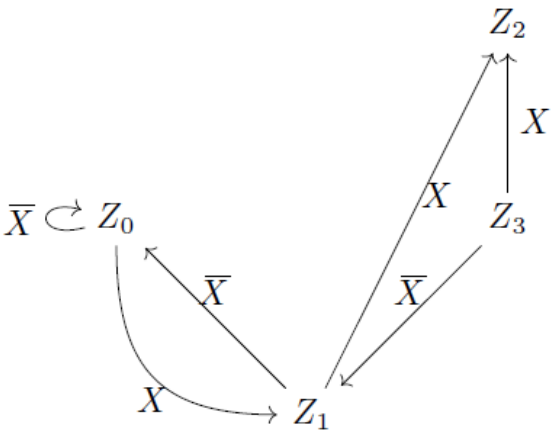
It is divided until only subsets with one state remain, i.e., another level is created by dividing again according to X and X, see Figure 11.

Figure 11. *Third Stage of the TSA*



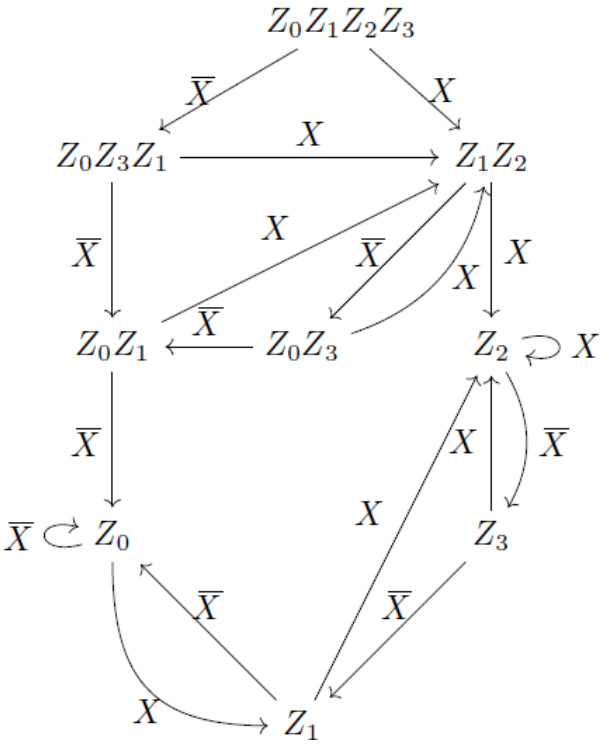
The final step is now to divide the last stage again into X and X, see Figure 12.

Figure 12. *Last Stage of the TSA*



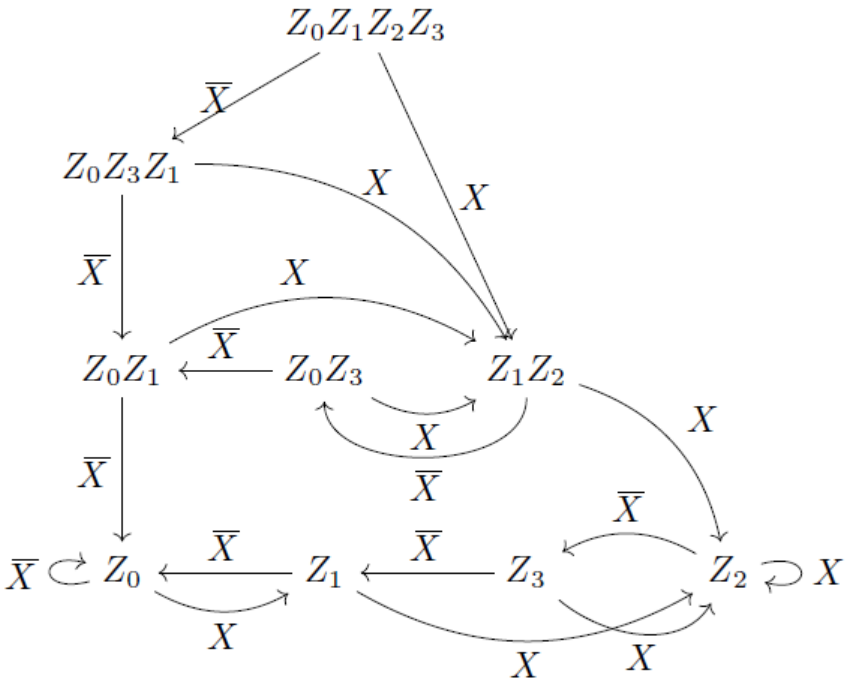
The final stage shows the original automaton. The entire tree can be seen in Figure 13.

Figure 13. *Complete TSA of the Example Automaton*



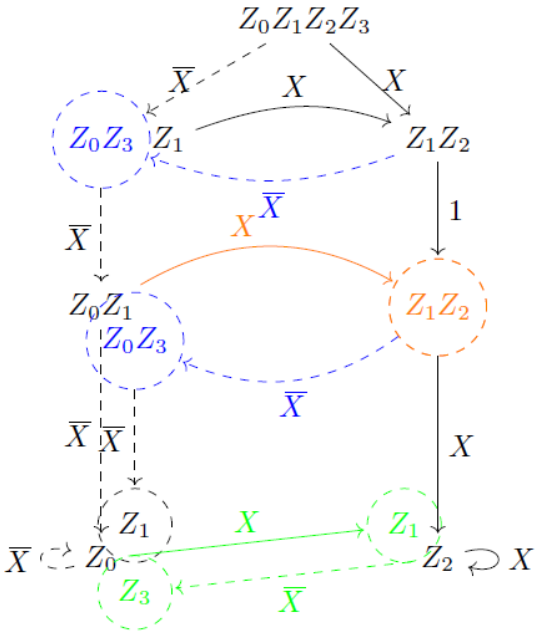
Now we move on to step 2, which states that we arrange the tree so that subsets with the same number of states are placed on the same level, see Figure 14.

Figure 14. *Reordering of TSA*



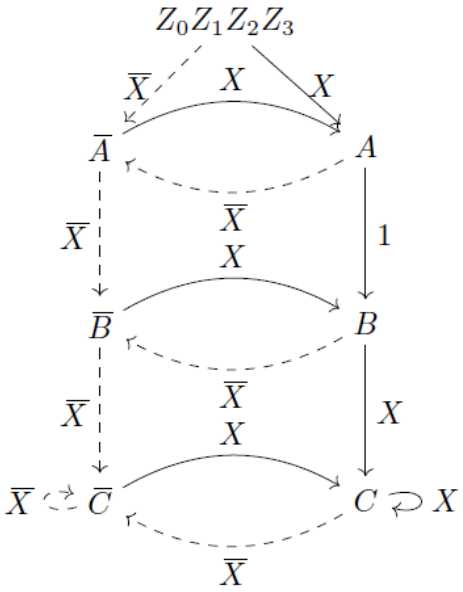
In the following, we will make the cascade total per level by superimposing states in the MS, see step 3. Here, states are pulled out of the multiset to make the graph total in the levels. We therefore pull state Z1Z2 from the multiset into the first level. In step 4, an X edge is added to Z0Z3, since you can get from state Z1 or Z2 to the left side in Z0Z3, see Figure 15.

Figure 15. *Tree Subset Automaton - Power Set Construction*



If the states are now recoded, the following graph is created, see Figure 16.

Figure 16. *Encoded Tree Subset Automaton*



The resulting formulas are:

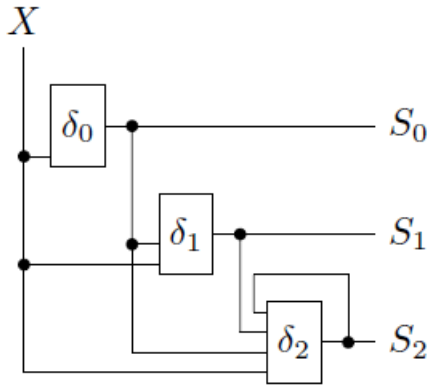
$$\bar{A} := \bar{X} \qquad A := X$$
$$\bar{B} := \bar{X}(\bar{A}, B) \qquad B := (A, X\bar{B})$$
$$\bar{C} := (\bar{X}, \bar{X}\bar{B}) \qquad C := (X, XB)$$

The resulting circuits in the module view are given in Figure 17.

Figure 17. *Module View of C and* $\bar{C}$

If the circuits were designed to be complementary and total, both could be dual-composed, resulting in the circuit shown in Figure 18.

Figure 18. *Resulting Cascaded Automaton*

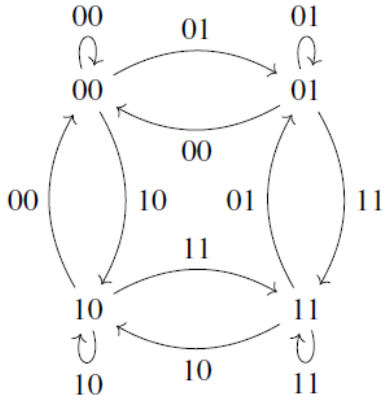


It can be seen that |N−1| stages have been created; in general terms, we can speak of a complexity of 2^(N−1), where N is the number of states. We can therefore see that the structure contains no feedback loops except in the last stage, meaning that the general automaton has been implemented as a cascade. Now that we understand the decomposition from [6], let us consider more general automata (with different parts of automata and more than one input variable).

Example Transformation of Automaton with Two Input Variables

In the following, a new example is considered. This example consists of two z-variables (z1, z0), i.e., four states (Z3, Z2, Z1, Z0), and two x-variables (x1, x0), i.e., four possible input edges (X3, X2, X1, X0), see Figure 19.

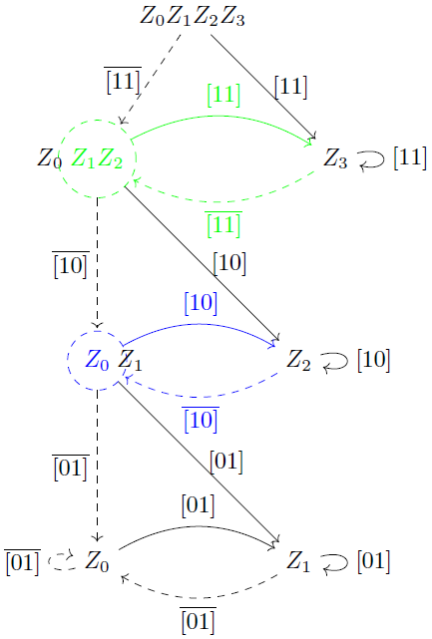
Figure 19. *Automatongraph of the New Example*



It is a one-step (in the z- and x-variables) and function stable incomplete automaton. As in the previous example, we now want to construct the Tree Subset Automaton.

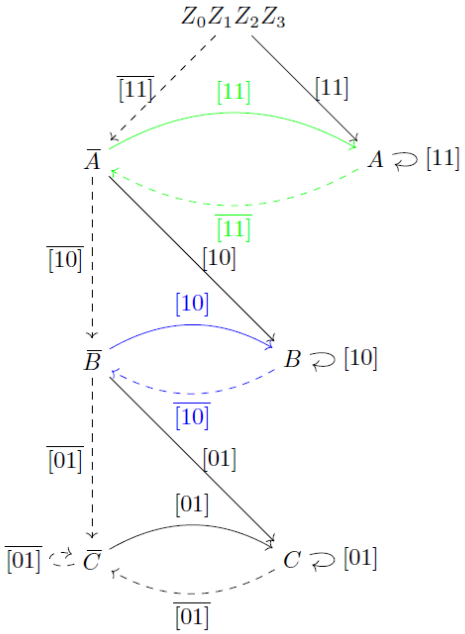
Construction of the TSA with inhibition of individual orthogonal transitions. We want to decompose the automaton into individual outgoing edges, but in such a way that each event is considered disjointly (outgoing edges are taken and a disjoint portion is generated for them), see Figure 20.

Figure 20. *TSA under Inhitibition of Disjoint Edges*



Since the automaton is constructed such that each state is assigned an incoming edge, the result is a very elegant TSA that clearly reveals the subset structure. Each stage is a subset in the states as well as in the input assignments (since, in each stage, one edge is taken from the set). If the states are now recoded, the structure shown in Figure 21 emerges.

Figure 21. *Recoded TSA under Inhibition of Disjoint Edges*

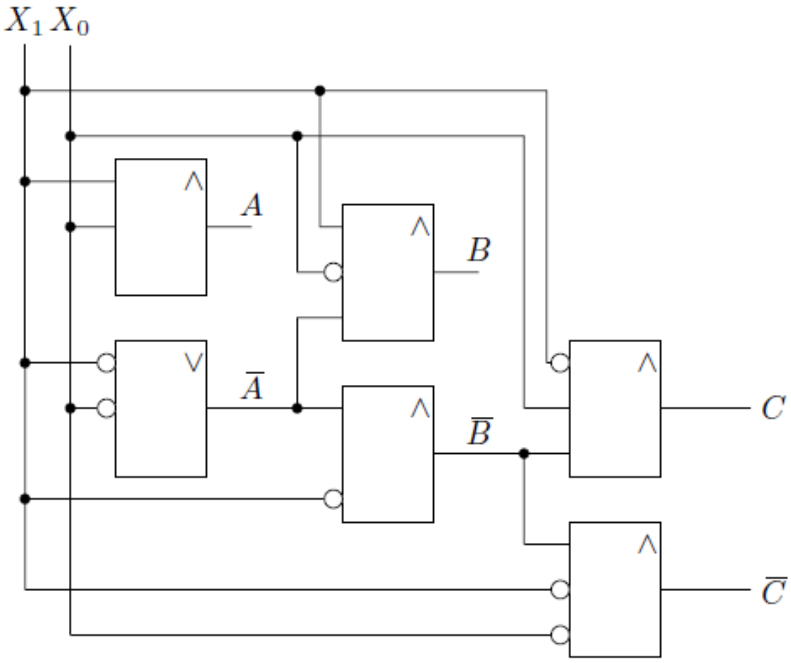


The back transformation to the original automaton is as follows:

$$Z0 := \bar{A}\,\bar{B}\,\bar{C}$$
$$Z1 := \bar{A}\,\bar{B}\,C$$
$$Z2 := \bar{A}\,B\,\bar{C}$$
$$Z3 := A\,\bar{B}\,\bar{C}$$

The cascade structure is shown in Figure 22.

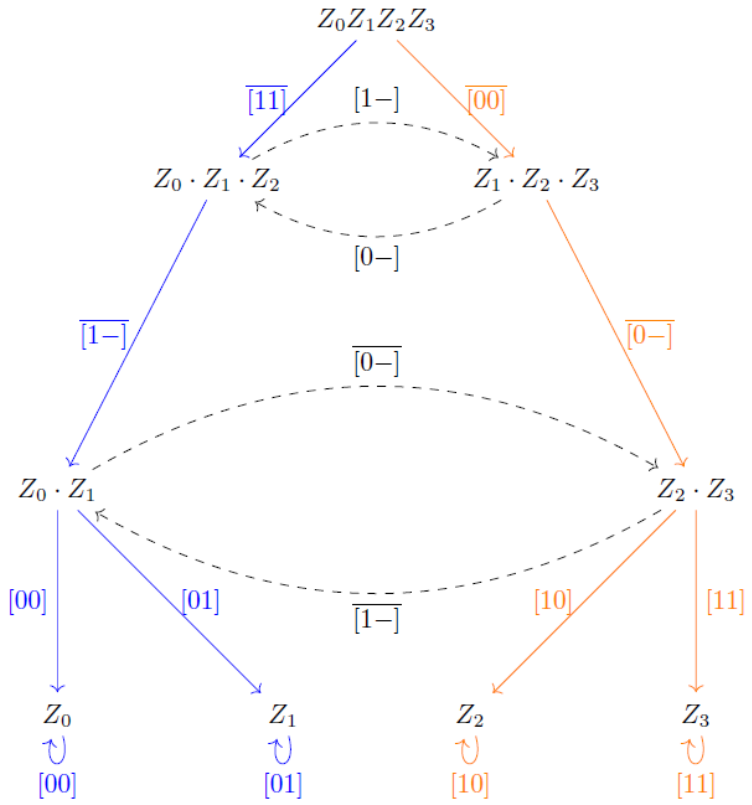
Figure 22. *Module View of Cascade for C and* $\bar{C}$



Construction of the TSA with inhibition of individual (non-orthogonal) transitions

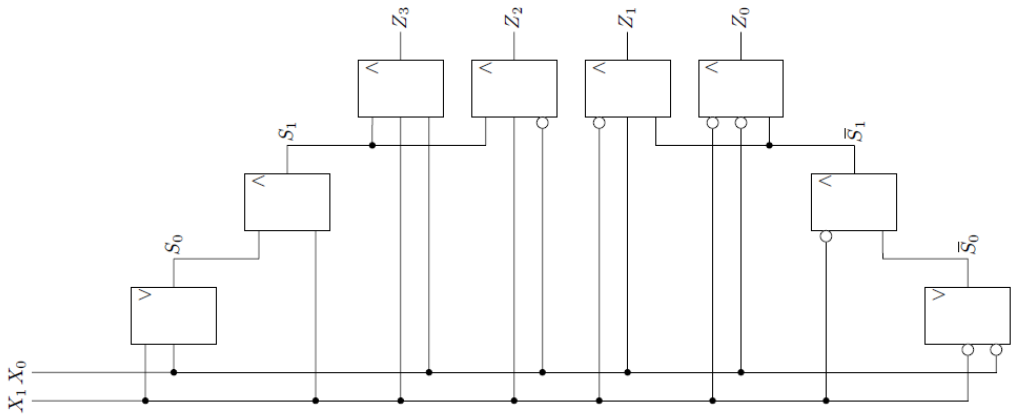
Next, we do not want to proceed with disjoint edges, but instead allow overlaps as well. To this end, we construct inhibitions of individual edges, which result in three edges being removed per side. The resulting TSA for the example automaton from Figure 19, after applying the inhibitions, corresponds to the TSA shown in Figure 23.

Figure 23. *Resulting TSA with Non-disjoint Edges*



The corresponding gate level is shown in Figure 24.

Figure 24. *Gate Level of the TSA*

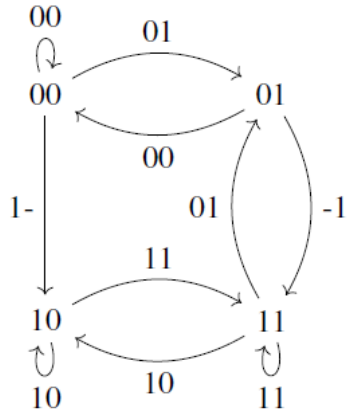


However, since the example automaton was, as mentioned, designed in a very specific way (unique edges to the states, single-step), a new example will be developed that does not represent a special case, in order to examine the general applicability of the transformations.

Partial Example with two Input Variables

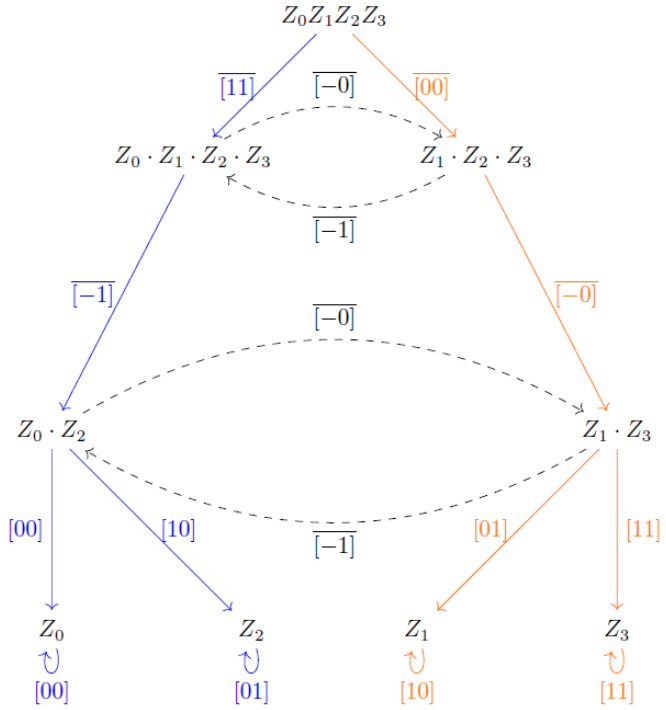
In the following, an automaton is presented that transitions into an oscillation with [01] and ends in a stable state with [10], [11], and [00], see Figure 25.

Figure 25. *Automatongraph of Partial Example with Oscillation*



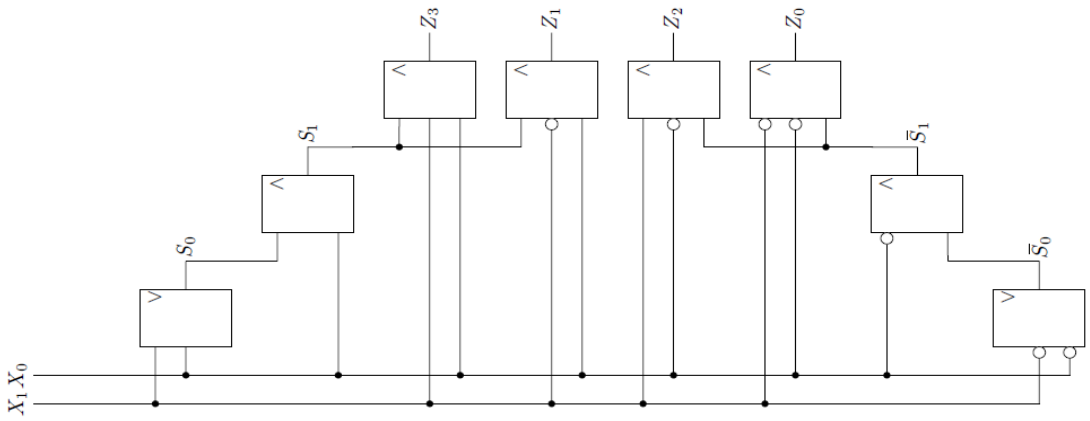
We will now once again construct the TSA with the inhibitions [−]\[11] and [−]\[00], see Figure 26.

Figure 26. *TSA with (non-disjoint) Inhibitions*



The circuit at the gate level is given in Figure 27.

Figure 27. *Gate Level of new Example TSA*

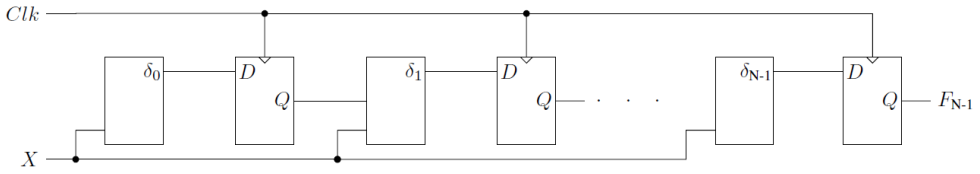


Having now understood the design methodology under inhibition and complementary edges, we will briefly address the direct transformation of the resulting cascades into pipelines in the following.

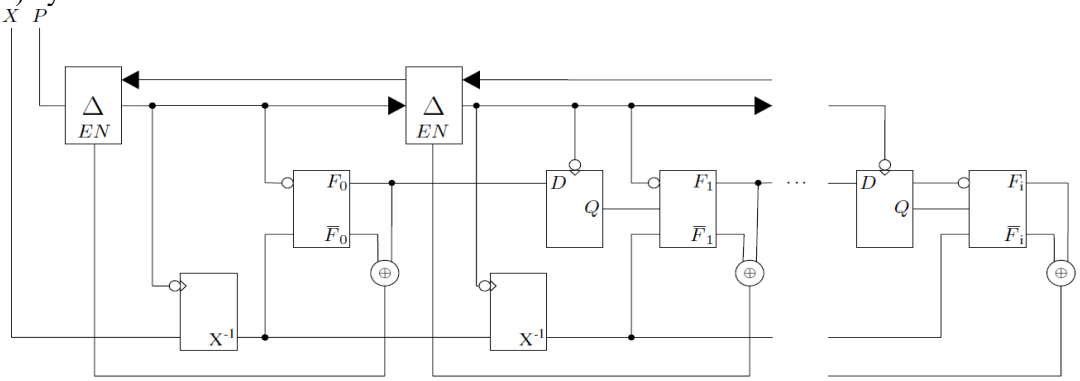
Transformation of Cascades into Pipelines

Cascades can be directly transformed into pipelines by implementing a register at each stage that stores the intermediate values. The general pipeline is shown in Figure 28a. If handshaking is also implemented, asynchronous pipelines can be realized, see Figure 28b.

Figure 28. *Pipeline Transformation of the Cascade*



a) Synchronous

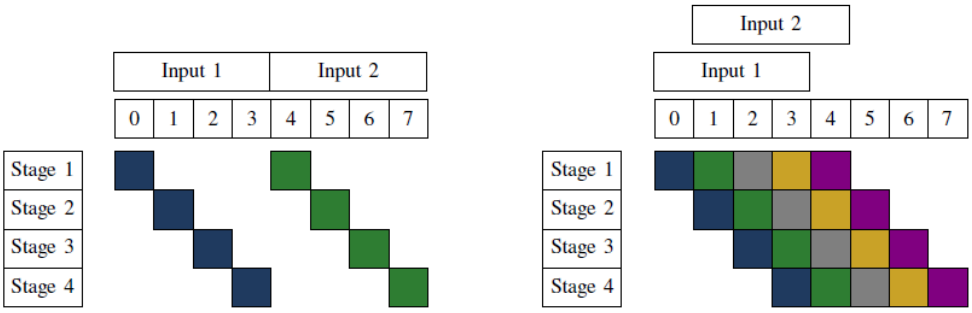


b) Asynchronous

It is important to note that pipeline design offers advantages, since cascades require stabilization at the input and the data must remain stable until the signal has propagated through the entire circuit, whereas pipeline structures allow new inputs to be applied as soon as the individual stages have finished processing the data.

The advantages of pipelining compared to cascading are shown in Figure 29.

Figure 29. *Comparison of Normal Cascade to Pipelined Cascade*



While the cascade structure requires 8 cycles to process the input data, the pipeline implementation needs only five. The advantages become even clearer when one realizes that the pipeline stage processes a new datum after each cycle of clock, whereas the cascade requires 4 cycles.

Implementation on FPGA

We want to implement and compare the structures with overlapping and orthogonal edges. For this purpose, a total automaton is chosen, since FPGA design, using totally defined look-up tables (LUTs), enforces total structures. We will realize the automata as pipelines and compare them. In the example, we have an automaton that is total, i.e., it is complete in the states, the disjunction of all states equals 1, expressed as the equation:

$$\left(\bigvee_{i=0}^{2^n-1} Z_i(x) = 1\right)$$

and complete in the outgoing edges per state, the disjunction of all outgoing edges equals 1, expressed as:

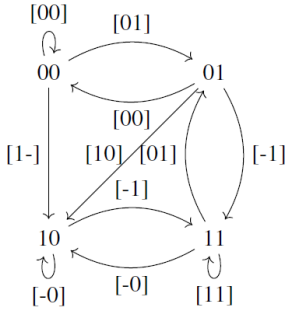
$$\forall i \left(\bigvee_{j=0}^{2^n-1} h_{ij}(x) = 1\right)$$

The automaton is also transient and oscillating. It is not single-step, but it is conflict-free and therefore deterministic, each outgoing edge is present only once per state, expressed as:

$$\forall i \left(\bigvee_{j,k=0;j\neq k}^{2^p-1} h_{ij}(x) \wedge h_{ik}(x) = 0 \right)$$

The automaton is bijective except for the multiset and is shown as an automaton graph in Figure 30.

Figure 30. *Automatongraph of the New Example*



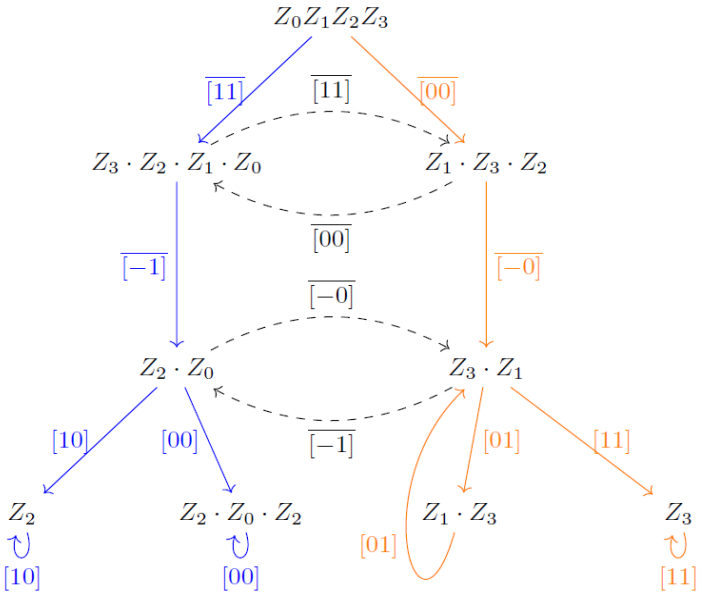
The previously discussed parts of an automaton are implemented, except for the isolated state. State 10 is a reset state due to edge 10; we have an oscillation between 01 and 11 for edge 01; there are transient states, e.g., the transition from 00 with edge [11] into state 10 and then into state 11; and the function stable states 00, 10, and 11. The truth table is listed in Table 2.

Table 2. *Truth Table of the Total Automaton*

z1	z0	x1	x0	z1	z0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	1	1	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	1	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	1	1

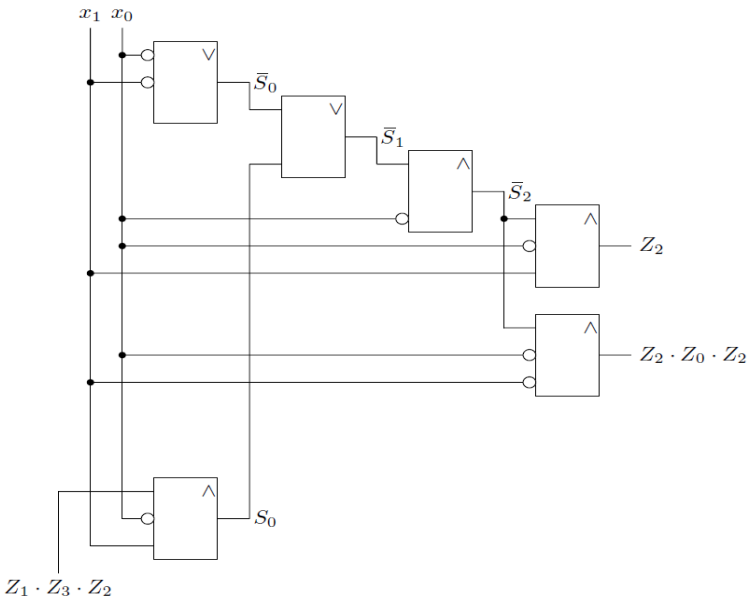
The TSA is now constructed with inhibitions, see Figure 31.

Figure 31. *TSA constructed with Inhibitions in Category Level (+, ·, ∧)*



Edge [11] on the left side as well as edge [00] on the right side are excluded. The part for the inhibition of [11] of the circuit in Category Level is shown in Figure 32.

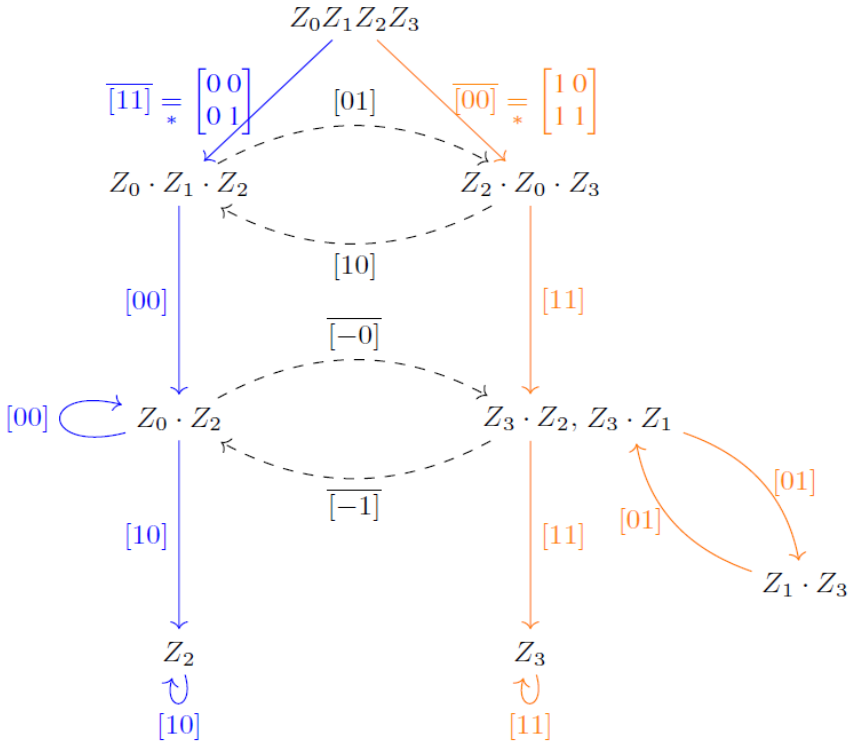
Figure 32. *Circuit in Category Level*



Implementation of the TSA in the Ring

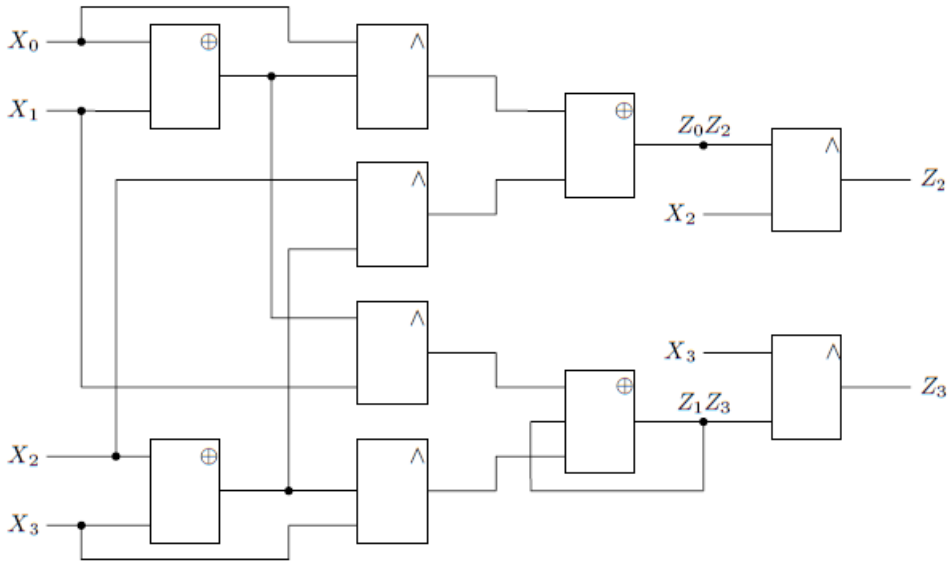
From the category level, the TSA can be constructed in the ring, see Figure 33.

Figure 33. *TSA in Ring* $(\oplus, \cdot, \wedge)$



The corresponding circuit is shown in Figure 34.

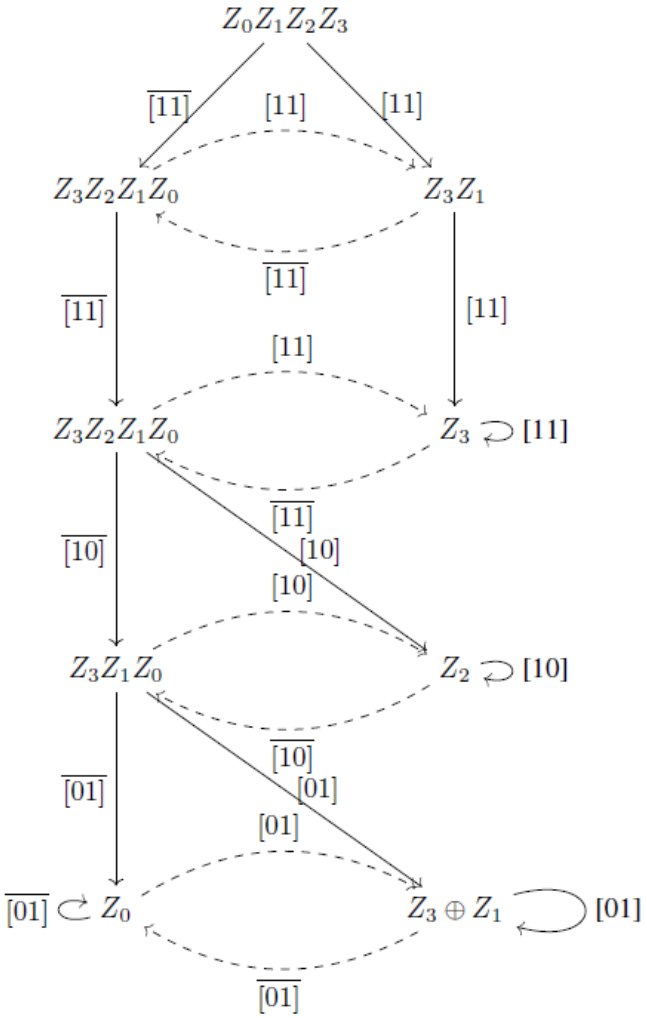
Figure 34. *Ring-Circuit of TSA under non-disjoint Inhibition*



TSA with Complementary Edges

For comparison, we want to construct the TSA with complementary edges, see Figure 35.

Figure 35. *TSA with Complementary Edges*



Realization on FPGA

We implement the TSAs as pipelines at the low-level primitives stage in order to avoid modifications during the synthesis step. The utilization reports are listed in Table 3.

Table 3. *Utilization Report of TSAs*

Pipeline	LUTs	Slice Registers	Slices
Inhibition	9	3	4
Complementary Edges	11	4	4

The complementary-edge pipeline required approximately 22% more LUTs compared to the inhibition-based design (11 vs. 9 LUTs). This is due to the fact that an additional stage is introduced, since in the first stage, when applying the algorithm, a redundant intermediate stage is created. The inhibition implementation in the final stages, however, was realized in a clever way: by exploiting the complementary edges in the last stage, the states Z2 and Z2·Z0·Z2 could be implemented within a single LUT, just as the final stage was able to accommodate the states Z1·Z3 and Z3. As expected from the structural properties, the complementary pipeline is somewhat slower (approximately 0.75 $f$clk,inh). The issues of higher resource utilization and lower performance can, however, be compensated for by careful optimization. The complementary decomposition structure generates stage-wise state detection and resembles the principle of one-hot encoding. However, it typically results in a larger number of states. In contrast, the inhibition decomposition structure primarily generates state detection only in the final stage. The states in this last stage are not always distinguishable, although further compaction may be possible. Finally, we briefly address how to deal with the complexity that arises in such designs.

Dealing with Complexity

We observed that the pipeline must be traversed once before new data is delivered at every clock cycle. From a performance perspective, this transformation into a TSA is therefore not problematic. However, for large FPGA projects, resource utilization plays a crucial role, and complexity increases exponentially with 2^n. It is thus advisable to reduce the state space of the automaton before its transformation. The following strategies can contribute to a reduction in complexity:

- Compaction: Since overall complexity grows rapidly, compaction plays a central role. Redundant or overlapping stages should be overlaid and optimized to minimize structural overhead.
- Prefer Mealy over Moore machines: Every Moore automaton can be transformed into an equivalent Mealy automaton. Because a Mealy machine can produce multiple outputs from a single state depending on its inputs, it usually requires fewer states. This transformation therefore reduces the size of the state space.
- State merging and decomposition: As suggested by Krohn–Rhodes theory, complex automata can be decomposed into simpler components. For instance, transitive closures can be formed around permutations with the same input and output, reducing the complexity of the TSA. During implementation, these closures can later be unfolded again.

Conclusion and Future Work

This work has systematically examined the application of the Krohn–Rhodes decomposition for the synthesis of cascaded control automata. The objectives enumerated in the introduction, including the systematic synthesis of cascaded

control automata, their conversion into pipeline-capable structures, and the demonstration of practical feasibility, have been achieved as follows. A structured methodology was presented, rooted in the Krohn–Rhodes decomposition and the Tree Subset Automata. This methodology facilitates the modular and single-step implementation of asynchronous control logic. In particular, the transformation of these cascades into pipeline structures has been demonstrated to reduce latency and enable parallel processing, while preserving deterministic behavior.

The exemplary FPGA implementation confirms the practical feasibility of the proposed approach and enables a quantitative comparison of different synthesis variants (inhibition vs. complementary edges). The comparison demonstrates that, contingent on the selected structure, disparities in performance and resource utilization may emerge. Complementary coding facilitates straightforward stage coupling, whereas inhibition enables a more compact realization when applied selectively. It has been observed that complementary realization does not necessarily reduce the number of states in every stage, such that the maximum number of stages may be greater than or equal to the number of original states. Furthermore, complementary realization bears a striking resemblance to one-hot encoding, a technique that can attain equivalent objectives through more streamlined approaches. By contrast, inhibition-based decomposition has proven to be more resource-efficient and higher-performing.

Future work may extend the presented methodology to larger, real-world control automata, for instance in communication protocols or low-synchronization microarchitectures. In particular, the combination of Krohn–Rhodes decomposition and TSA pipelines holds great potential for domain-specific ASIC designs and ultra-low-power applications. Furthermore, integrating the approach into automatic synthesis tools would represent an important step toward embedding the methodology into established design flows. Finally, coupling the approach with formal verification techniques, such as hazard- and race-freedom checks as well as glitch robustness, offers a promising direction for future research.

References

Krstic M, Grass E, Gurkaynak FK, Vivet P (2007) Globally asynchronous, locally synchronous circuits: Overview and outlook. *IEEE Design & Test of Computers*, 24(5), 430–441. https://doi.org/10.1109/MDT.2007.164

Deeg F, Zhu J, Sattler SM (2020) Asynchronous design. In: *AmE 2020 – Automotive meets Electronics; 11th GMM Symposium*, pp 1–5.

Cortadella J, Kishinevsky M, Burns SM, Stevens K (1999) Synthesis of asynchronous control circuits with automatically generated relative timing assumptions. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, pp 324–331. https://doi.org/10.1109/ICCAD.1999.810669

Bloem R, Jobstmann B, Piterman N, Pnueli A, Sa'ar Y (2012) Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3), 911–938. https://doi.org/10.1016/j.jcss.2011.08.007

Kushnerov A, Bystrov S (2023) Signal transition graphs for asynchronous data path circuits. *Modeling and Analysis of Information Systems*, 30, 170–186. https://doi.org/10.18255/1818-1015-2023-2-170-186

Maler O (2010) On the Krohn–Rhodes cascaded decomposition theorem. In: Manna Z, Peled DA (eds) *Time for verification: Essays in memory of Amir Pnueli*. Springer, Berlin–Heidelberg, pp 260–278. https://doi.org/10.1007/978-3-642-13754-9_12

Krohn K, Rhodes J (1965) Algebraic theory of machines I: Prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116, 450–464. https://doi.org/10.1090/S0002-9947-1965-0183773-6

Rhodes J, Nehaniv CL (2009) *Applications of automata theory and algebra: Via the mathematical theory of complexity to biology, physics, psychology, philosophy, and games*. World Scientific, Singapore. https://doi.org/10.1142/6977