Analysis of Sequential Logic Using Finite State Machine Representation in VLSI Design

Blaine Khalil
Texas Tech University
Master of Engineering Report
Completed May 2022

Abstract: Modern engineering systems rely on digital hardware that behaves correctly across time. Communication devices, aviation systems, power-distribution networks, medical instruments, and computing platforms all depend on integrated circuits whose internal state must be controlled, verified, and synchronized. This report examines sequential logic in very-large-scale integration (VLSI) design and explains why registers, latches, flip-flops, timing constraints, and finite state machine (FSM) models are essential to dependable digital systems. The discussion introduces sequential timing elements, describes design trade-offs among static and dynamic latches, master-slave flip-flops, C2MOS, true single-phase clocking, and pulsed latch approaches, and connects these circuit-level elements to FSM-based reasoning. The report also reviews challenges created by the slowing of traditional Moore-law scaling and the growing importance of formal verification. A simplified verification problem is then presented using ordered binary decision diagrams (OBDDs), reachability analysis, and a product-machine construction to evaluate whether two sequential machines produce equivalent input/output behavior. The central result is that global behavioral equivalence can be reduced to checking a local property over reachable states. This approach illustrates why FSM representation remains a practical bridge between discrete mathematics, algorithms, and VLSI design verification.

Keywords: Sequential logic; VLSI design; flip-flop; register; latch; binary decision diagram; finite state machine; formal verification

I. Introduction

Why use sequential logic? What are sequential elements? Are timing parameters of sequential elements vital to design? Can flip-flop implementations mitigate timing constraints in practical circuits? These questions are fundamental in VLSI design because sequential circuits are functions of both the present input and the previous state. In other words, sequential circuits contain memory. A combinational logic circuit implements a Boolean function, but when its output is stored in a memory element and later fed back into the design, the circuit begins to depend on history as well as on current input conditions [1], [16].

Most sequential circuits used in digital VLSI are synchronous. They use a clock to coordinate the movement of data through logic paths and registers. This synchronization is one reason sequential logic appears throughout modern digital systems. Consider an accumulator circuit. An accumulator sums a list of values, and its output is fed back into the input path for the next addition. Without a register to control that feedback, races can occur and the output can be sampled before the intended calculation is complete. In that case, the circuit may produce an erroneous value even when the combinational function itself is correct [16].

The motivation for synchronous circuit design is similar to the motivation for traffic lights. Without a shared timing structure, vehicles may enter an intersection at the wrong time and interrupt one another. In a sequential circuit, the clock and registers create a disciplined timing structure. They hold values long enough for slower paths to settle and prevent faster paths from corrupting downstream logic. If every path in a design could be made to have exactly the same delay, sequential timing would be less critical. In practice, equalizing all paths is either impossible or uneconomical, so designers must align fast and slow paths using sequential elements.

Pipelining provides a second reason for using sequential logic. A common example is the laundry analogy. If one washer takes 30 minutes, one dryer takes 40 minutes, and one operator takes 20 minutes to fold, then a single load takes 90 minutes from start to finish. Four loads performed strictly one after another require about six hours. In a pipelined process, the washer begins the next load as soon as it becomes available. The drying stage is the slowest stage, so it determines the throughput, but total completion time improves substantially because the available resources are used concurrently.

The same idea applies to data paths. If one instruction takes 10 time units to pass through a logic path, a non-pipelined design completes one instruction every 10 time units. If the path can be divided into five stages that each take approximately two time units, registers can separate the stages and allow a new instruction to enter the pipeline every two time units. The latency of one instruction may still be 10 time units, but the throughput is improved because multiple instructions are in progress simultaneously [16].

In VLSI design, pipeline stages do not always have equal delay. Therefore, each stage must hold its input constant until the preceding stage is complete. Registers provide this function. A classic five-stage RISC pipeline illustrates the concept with instruction fetch, instruction decode/register fetch, execute or address calculation, memory access, and write-back stages. Registers between stages shorten the slowest combinational path between clock edges and allow higher operating frequencies. Overhead from the registers and clock distribution must be considered, but the general throughput advantage remains important.

II. Motivation: Sequential Logic Design Elements and Registers

The VLSI design cycle begins with system specification and moves through functional design, logic design, circuit design, physical design, design verification, fabrication, packaging, testing, and debugging. This report focuses on the verification side of that process while still connecting the discussion to partitioning, placement, routing, and compaction decisions in physical design.

Understanding how chips are packaged, such as through printed circuit boards, multichip modules, and wafer-scale integration, also helps explain why density, cost, timing, and time to market are coupled engineering concerns.

My interest in this topic developed during graduate coursework in VLSI design. Prior exposure to discrete mathematics and computer science made the finite state machine representation feel directly connected to digital logic. Sequential logic provides the physical implementation, while finite state machines provide a mathematical model for reasoning about state transitions, reachability, and correctness.

A level-sensitive latch is transparent during one phase of the clock and opaque during the other phase. For a positive latch, the latch is opaque when the clock is low, meaning data is not transferred from input to output. When the clock is high, the latch becomes transparent and the input can be mirrored to the output. A flip-flop, by contrast, is edge-triggered. A positive-edge flip-flop samples the input at the rising edge of the clock and holds that value until the next qualifying edge.

During the remainder of the clock cycle, a flip-flop blocks changes at the input from immediately appearing at the output. Textbooks and industry practice sometimes use different naming conventions. Some academic texts refer to any bistable element as a flip-flop, including the SR latch. In practice, engineers often use the broader term register when referring to banks of flip-flops or latch-based storage elements. Clear naming is important because timing behavior depends on whether a device is level-sensitive, edge-triggered, static, dynamic, pulsed, synchronous, or asynchronous.

There are three common categories of sequential timing elements. The first is the conventional flip-flop, which is widely used in synchronous logic and samples data on a defined clock edge. The second is the transparent latch. Latches can support time borrowing when half-cycle timing is intentionally used, which makes them valuable in some high-performance designs. The third is the pulsed latch. A pulsed latch uses a short clock pulse so that the latch is transparent only briefly and behaves similarly to an edge-triggered flip-flop. Pulsed latches can be smaller and less expensive, but pulse generation and distribution must be carefully controlled [16].

Static and dynamic latch implementations illustrate the area-speed-reliability trade-off. A static latch maintains its output through feedback and can restore its value despite limited leakage or noise. A common static CMOS latch uses a multiplexer or transmission-gate structure with feedback from the output. When the clock selects the input, data propagates to the output; when the clock selects the feedback path, the stored value is retained. A dynamic latch stores state temporarily on capacitance. It can be fast and compact, but it is more sensitive to leakage and requires careful timing [16].

Ratioed and non-ratioed latch designs show another design trade-off. A static latch can be built with feedback inverters, but the write path must be strong enough to overcome the feedback path. Non-ratioed latch structures are often more robust because correct operation does not depend as heavily on device sizing ratios. The cost is usually additional area or capacitance. In CMOS VLSI design, this trade-off appears frequently: a design that is smaller or faster may require more careful verification, while a more robust design may consume more area.

A master-slave edge-triggered flip-flop can be formed by combining two latches with opposite clock phases. When the master is transparent, the slave is opaque; when the master becomes opaque, the slave becomes transparent and transfers the sampled value to the output. This structure is conceptually simple but can require many transistors, multiple clocked nodes, and additional clock loading. Large clock loads increase power consumption and may require stronger clock drivers.

Resettable flip-flops can be designed with asynchronous or synchronous reset behavior. An asynchronous reset forces the storage element into a reset state independent of the clock edge. A synchronous reset is evaluated on the clock edge and is often more cost-effective and easier to integrate into a synchronous timing methodology. Both approaches appear in real designs, but the reset strategy must be consistent with the system-level timing and verification plan.

Clock overlap and clock skew create additional implementation concerns. If true and complement clock phases overlap or arrive at different times, data may feed through unintentionally. Clocked CMOS, or C2MOS, reduces sensitivity to certain forms of clock overlap by using clocked tri-state or inverter-based structures in the master and slave stages [6]. True single-phase clocked registers (TSPC) reduce the need for explicit complementary clock phases and can support higher speed with fewer devices. Pulsed latch designs offer another alternative by using a short duty-cycle clock pulse to emulate edge-sensitive behavior. Each of these approaches is useful in selected design contexts, but each also introduces a verification burden.

These circuit-level ideas lead naturally to state machines. A sequential circuit has a state, a transition rule, an input alphabet, and an output behavior. The research problem in this report uses that connection to reason about a resettable finite state machine and to determine whether its reachable behavior satisfies a desired output property.

III. Literature Review: The Future of Computing Beyond Moore's Law

When engineers design electronic systems, the number of interactions among components can become extremely large. Over the past several decades, theoretical computer science has influenced VLSI design and layout through data structures, algorithms, synthesis methods, and formal verification. Manual design without computer-aided tools is no longer practical for modern chips. At the same time, continued performance scaling has become harder because traditional transistor scaling no longer provides the same predictable improvements in performance, power, and cost.

One well-known example of the cost of hardware errors is the 1994 Intel Pentium division error. The issue resulted from incorrect entries in a lookup table used by divider circuitry and caused incorrect numerical results in rare cases. Although the error was limited, the public and economic consequences were significant. The event remains an important reminder that hardware verification is not optional; it is a central engineering requirement.

Another major verification challenge is the number of possible combinations of individual chip components. Representing and manipulating the functional behavior of a full design can be extraordinarily difficult. This is where algorithms, graph representations, binary decision diagrams, satisfiability solving, model checking, and formal methods become important to VLSI design practice [3], [14], [15], [19].

Moore's law, introduced by Gordon Moore and associated with the semiconductor industry for decades, described a trend in which the number of transistors available at a practical cost increased on a regular schedule. That trend provided the computing industry with a predictable foundation for cost, power, area, tools, compilers, simulators, and emulators. Shalf argues that the classical scaling drivers that supported this trend are weakening and that continued

performance growth requires investment in architecture, packaging, materials, and new models of computation [25].

In this context, the end of traditional scaling does not mean the end of computing performance. Instead, it changes the design space. Heterogeneous architectures, accelerators, monolithic three-dimensional integration, advanced packaging, and new device physics all become more important. Figure D and Figure E in the appendix summarize the transition toward extreme heterogeneity and the broad design space beyond conventional CMOS scaling [25].

Artificial intelligence, machine learning, image processing, and accelerator-based workloads have increased demand for specialized architectures. Many such systems are sequential at their core because they depend on internal state, memory, and controlled data movement. Even when the functional blocks are highly specialized, designers still need predictable timing, verified state transitions, and reliable input/output behavior.

The central analogy is physical space. When a city becomes dense and ground-level space is exhausted, architects build upward. Similarly, as transistor scaling becomes more difficult, computer engineers increasingly build through specialization, packaging, heterogeneous integration, and architecture-aware algorithms. Sequential logic and FSM-based verification remain valuable because they provide a way to reason about these more complex systems.

IV. Research Problem and Assumptions

Binary decision diagrams are data structures for representing Boolean functions. Ordered binary decision diagrams (OBDDs) provide a canonical representation for a fixed variable ordering and have been widely used in symbolic model checking and hardware verification [14], [15]. Although many modern verification flows also use conflict-driven clause learning (CDCL) satisfiability solvers, BDD-based methods remain useful for explaining reachability and state-space reasoning in sequential systems.

Many problems in discrete mathematics, computer science, and integrated-circuit design involve manipulating objects over a finite domain. When elements of that domain are encoded as bit vectors, many operations can be expressed as switching functions that map bit vectors to single-bit outputs. This is useful because the behavior of finite systems can often be represented compactly as Boolean functions, and algorithms can operate directly on those representations [18].

A binary decision diagram can be viewed as a graph of binary-valued decisions that eventually terminate in either a true or false result. In hardware verification, this structure supports questions about reachability and equivalence. Reachability asks which states can be reached from an initial state. Equivalence asks whether two systems produce the same output sequences for the same input sequences. Both questions are essential when a design must be proven correct before fabrication.

To connect the formal and graphical views, consider a directed graph G with a finite vertex set V and an edge set E. The edge set is a subset of V x V. If V = {1, 2, 3, 4, 5} and E = {(1, 2), (1, 3), (2, 3), (2, 4), (5, 2), (5, 3)}, then the graph can be visualized as shown in Figure A. Two nodes u and v are connected by an edge from u to v if and only if (u, v) is in E. Conversely, a binary relation over a finite set can be interpreted as a graph whose vertices are the elements of the set and whose edges are the ordered pairs in the relation [4].

Modern electronic systems make verification difficult because the number of states can grow exponentially with the number of storage elements. A traffic-light controller may be straightforward to analyze, but medical equipment, communication hardware, and processor subsystems contain many more states and timing interactions. For this reason, the quality of a verification method should be judged by how well it represents the state space and how efficiently it supports the necessary operations.

The research problem in this report is intentionally restricted so that the verification structure is clear. Assume a customer asks whether two sequential systems have the same input/output behavior. Each system is modeled as a finite state machine, and each finite state machine is described by a netlist of gates. The goal is to determine whether both machines produce identical output sequences for every possible input sequence.

V. Methodology Used and Results

The general customer requirement is stated as follows. Given two sequential systems, or more precisely two finite state machines M1 and M2 with the same number of output bits and represented by gate-level netlists, determine whether M1 and M2 have the same input/output behavior. Equivalently, determine whether both machines produce the same output sequence for every possible input sequence [21].

The key idea behind the OBDD-based approach is to reduce verification of a global property to verification of a local property over all reachable states. This matters because a formal property does not need to be checked over states that cannot occur from the initial condition. Reachability analysis computes a representation of the reachable set R, and the characteristic function X_R of that set can be represented using an OBDD when the function is compact under a suitable variable ordering [14], [15], [21].

A restricted verification problem can now be stated. Given a sequential finite state machine M with a single output lambda(x, e) over the output alphabet {0, 1}, determine whether M always produces the output value 1 for every possible input sequence. If the reachable-state characteristic function X_R has been computed, the verification problem can be reduced to checking whether lambda(x, e) = 1 for every reachable state x in R and every input e.

This restricted problem is useful because the equivalence test for two machines can be transformed into one machine whose output reports whether the two original machines agree. Let M1 = (Q1, I, O, delta1, lambda1, q1) and M2 = (Q2, I, O, delta2, lambda2, q2) be two finite state machines with p input bits and m output bits. Define a product machine M with state space Q = Q1 x Q2 and initial state q0 = (q1, q2). For an input e, the transition function is delta((x1, x2), e) = (delta1(x1, e), delta2(x2, e)).

The output of the product machine is 1 precisely when the corresponding output bits of M1 and M2 match. In compact form, lambda((x1, x2), e) = 1 if and only if lambda1,j(x1, e) = lambda2,j(x2, e) for every output bit j from 1 to m. Otherwise, lambda((x1, x2), e) = 0. The product machine therefore simulates both original machines in parallel and converts equivalence into a single-output verification problem.

The result is that M1 and M2 are equivalent if and only if the product machine produces output 1 for every reachable product state and every input. If any reachable state and input produce output 0, then the two machines differ, and the corresponding input sequence provides a counterexample. This is the same conceptual pattern used in symbolic model checking: compute or represent reachable states, evaluate the required property over that set, and produce a counterexample if the property fails.

This methodology demonstrates why reachability analysis is central to formal verification of sequential systems. It avoids checking

unreachable states, uses Boolean functions to represent transition and output behavior, and provides a direct connection between gate-level logic and finite-state mathematical reasoning.

VI. Discussion

Representing sequential logic as a finite state machine is a practical way to think about design behavior. In computer science, the related term finite automaton is often used, and deterministic finite automata describe systems that have a single next state for each state/input pair. In digital design, the FSM model links the physical implementation of registers and combinational logic to a mathematical structure that can be analyzed, tested, and verified.

If a design can be described as a finite state machine, then the designer can reason about time by reasoning about states and transitions. This makes it possible to prove properties, identify unreachable states, find states where the machine may become stuck, and determine whether the transition structure is complete. The same concept appears in hardware description languages, where case statements and conditional logic often implement the transition rules of an FSM.

A simple passcode block illustrates the concept. Suppose a system has a Locked state, an Entering Code state, and an Unlocked state. When a key is pressed, the system moves from Locked to Entering Code. If the entered keys match the required passcode, the system moves to Unlocked. If the keys are incorrect, it returns to Locked. If the lock key is pressed while the system is unlocked, it returns to the Locked state. Figure B shows this state transition diagram.

A hardware implementation of the same model contains a register bank to hold the current state, next-state logic to compute the next state from the current state and inputs, and output logic to generate the outputs. The register bank is commonly built from flip-flops. The next-state logic uses the current state and input signals to determine which state will be loaded at the next clock edge. The output logic may depend only on the current state, which is the Moore-machine case, or it may depend on both the current state and current inputs, which is the Mealy-machine case. Figure C summarizes this architecture.

This structure is not limited to passcode blocks. Counters, shifters, registers, traffic-light controllers, vending machines, garage-door controllers, and many communication protocols can be modeled as FSMs. A two-bit binary counter has four possible output states and regular transitions between those states. A shift register has a state determined by the stored bit pattern, and its next state depends on both the current state and the incoming bit. In this way, many digital sequential circuits can be analyzed as finite state machines.

The value of the FSM model is that it offers a common language across mathematics, software, and hardware. It is possible to draw a state diagram, write HDL code, construct a state table, synthesize the circuit, and then verify properties using reachability or equivalence checking. This connection is especially important in VLSI design because the physical implementation may contain thousands or millions of storage elements, but the desired behavior must still be stated in a form that can be checked.

VII. Conclusion

Sequential logic can be modeled effectively using finite state machines. Unlike purely combinational circuits, sequential circuits depend on both current inputs and previous state. Flip-flops, latches, registers, and clocking strategies provide the physical mechanisms that allow a circuit to store state and move from one state to another. When a clock controls the movement of state, the circuit is synchronous; when no global clock controls the transition, the circuit is asynchronous.

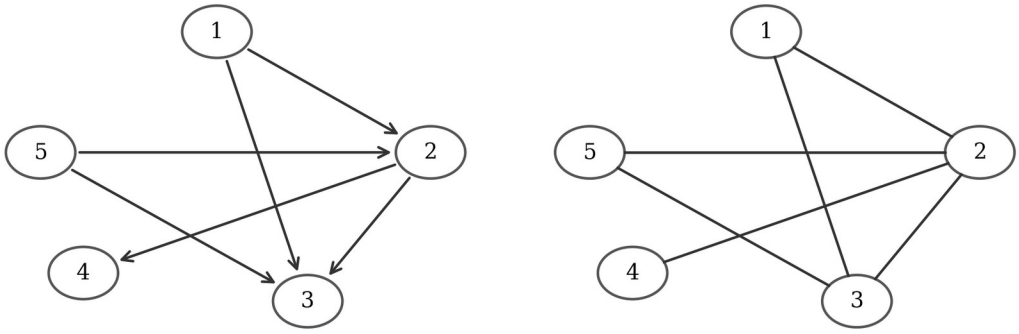
The FSM representation is valuable because the behavior of a sequential circuit can be described by a finite number of states, transition rules, inputs, and outputs. Practical systems such as traffic lights, passcode locks, vending machines, communication controllers, and processor pipelines all rely on this state-based behavior. The general implementation includes input or next-state logic, a register bank, and output logic. The stored values in the register bank represent the state of the circuit, and the transition logic determines how that state changes on each clock event.

The number of flip-flops required by an FSM depends on the number of states needed to implement the intended behavior. The output logic can be designed as a Moore machine, where outputs depend only on the current state, or as a Mealy machine, where outputs depend on both the current state and the primary inputs. Edge-triggered flip-flops are commonly used to ensure that only one state transition occurs per clock cycle and to prevent unstable or erroneous outputs.

Although many equivalence and property-checking tasks are now performed by electronic design automation tools, understanding the underlying method remains valuable. A designer who understands reachability, product machines, OBDDs, and FSM structure can better interpret tool results, diagnose verification failures, and explain why a circuit satisfies or violates a specification. Engineering is a systematic discipline, and formal state-based analysis provides a systematic way to define constraints, test behavior, and reduce risk before fabrication [20].

Moore-law scaling pressures have made verification and architecture-aware design even more important. As integrated circuits become more complex and increasingly heterogeneous, the ability to represent behavior, reason about state, and prove properties remains essential. Finite state machine representation is therefore not merely a classroom concept; it is a practical foundation for analyzing sequential logic in VLSI design.

VIII. Figure Appendix



(a) Directed graph (b) Undirected graph

Figure A. Directed and undirected graph representations used to explain finite relations [4].

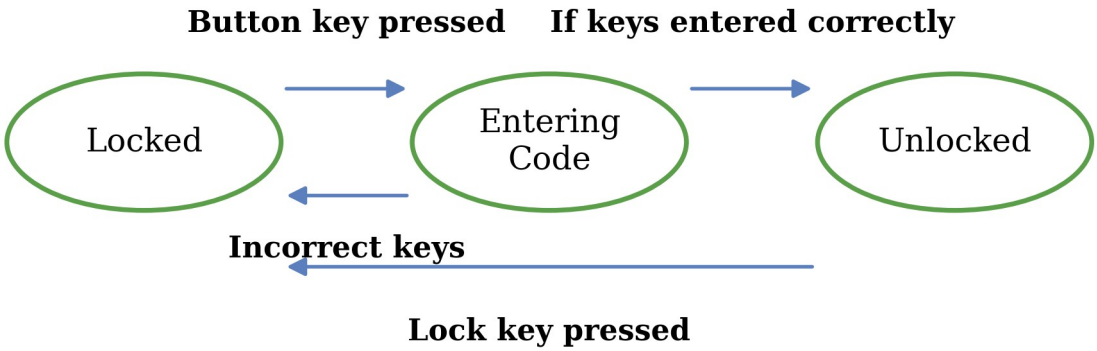


Figure B. Passcode-lock finite state machine example.

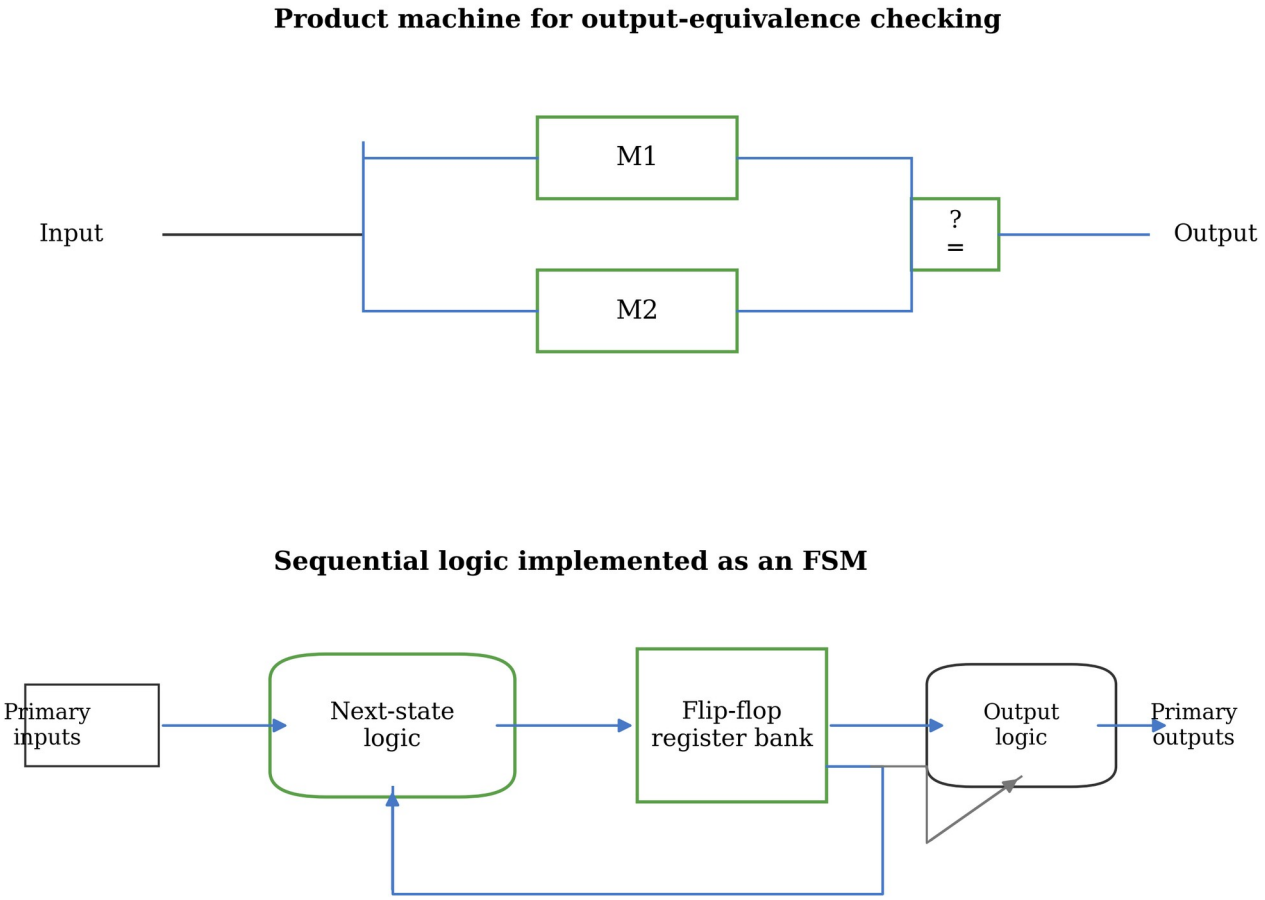


Figure C. Product-machine equivalence check and generic FSM implementation pattern.

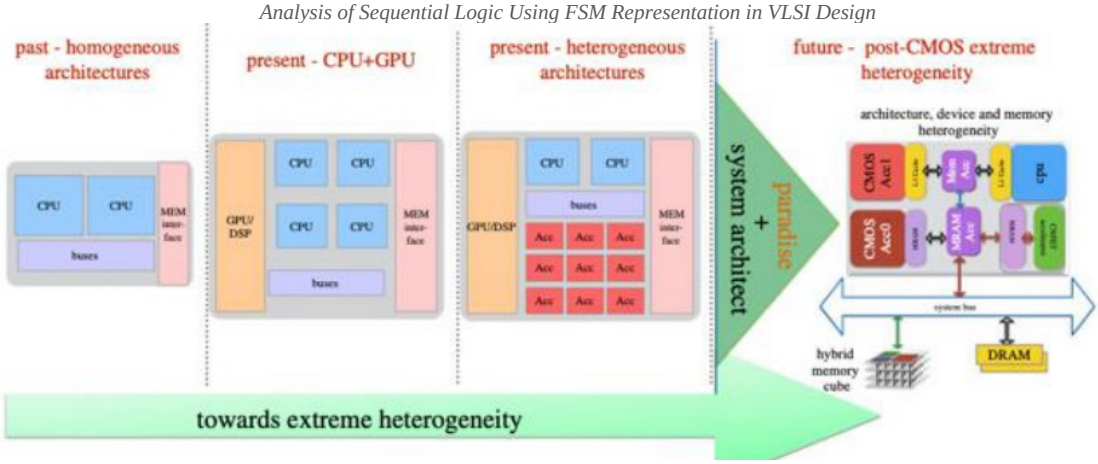


Figure D. Evolution toward heterogeneous computing architectures [25].

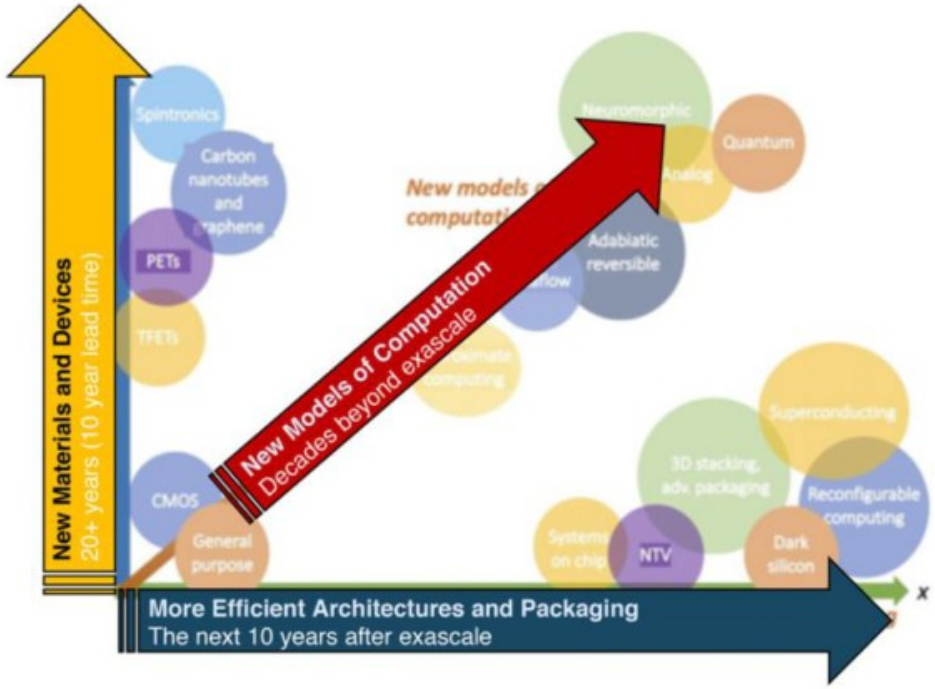


Figure E. Candidate directions beyond conventional scaling, including architecture, packaging, materials, and new models of computation [25].

IX. Master's Report Appendix

Report Element	Description
Title block	Report title, author, institution, and completion date.
Abstract and keywords	One-paragraph technical summary and search terms.
Introduction	Purpose of sequential logic, timing, registers, and pipelining.
Motivation	Sequential timing elements, latches, flip-flops, reset behavior, and clocking methods.
Literature review	Computing beyond Moore's law, verification pressure, and heterogeneous architecture.
Research problem	OBDDs, graph interpretation, reachability, and assumptions.
Methodology and results	Product-machine equivalence checking and reachable-state verification.
Discussion	FSM design patterns, state diagrams, Moore and Mealy behavior, and implementation.
Conclusion	Summary of FSM representation as a practical VLSI verification tool.
Figure appendix and bibliography	Supporting diagrams and references.

Bibliography

[1] R. Jenila and K. Pappa, "VLSI implementation of error detection and correction codes for space engineering," 2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks, pp. 569-573, 2021.

[2] R. I. Bahar, E. A. Frohm, C. M. Gaona, et al., "Algebraic decision diagrams and their applications," Formal Methods in System Design, vol. 10, pp. 171-206, 1997.

[3] J. L. Bormann and M. Payer, "Model checking in industrial hardware design," in 32nd Design Automation Conference, San Francisco, CA, 1995.

[4] C. Meinel and T. Theobald, "OBDD foundations and applications," in Data Structures and VLSI Design, Berlin, Heidelberg: Springer-Verlag, 1998, pp. 173-197.

[5] S. Jahanir and J. Rahaman, "High-speed and low-power preset-able modified TSPC D flip-flop design and performance comparison with TSPC D flip-flop," International Symposium on Devices, Circuits and Systems, vol. 2, no. 3, pp. 1-4, 2018.

[6] Z. Sandhie and F. Ahmed, "Design of ternary master-slave D-flip flop using MOS-GNRFET," 2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 554-557, 2020.

[7] W. Merrill, "Sequential neural networks as automata," Yale University, New Haven, CT, 2019.

[8] A. Fern and A. Koul, "Learning finite state representations of recurrent policy networks," ICLR, Corvallis, OR, 2019.

[9] R. Krishnan, "Pattern search automation for combinational logic analysis," International Symposium for Testing and Failure Analysis, pp. 86-92, 2019.

[10] S.-i. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," pp. 272-277, 1993.

[11] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," ACM/IEEE Design Automation Conference, Dallas, TX, 1993.

[12] P. Hitzler and M. K. Sarker, Neuro-Symbolic Artificial Intelligence: The State of the Art, vol. 1, Amsterdam: IOS Press BV, 2022, pp. 146-158.

[13] J. Cavanagh, Sequential Logic: Analysis and Synthesis, Boca Raton, FL: CRC Press, 2018.

[14] R. E. Bryant, "Binary decision diagrams: An algorithmic basis for symbolic model checking," School of Computer Science, Pittsburgh, PA, 2018.

[15] R. E. Bryant, "Binary decision diagrams," in Handbook of Model Checking, Cham: Springer, 2018, pp. 191-217.

[16] A. Teman, "Digital VLSI design - sequential logic," Bar-Ilan University, Ramat Gan, Israel, 2021.

[17] R. Brayton and H. K., Logic Minimization Algorithms for VLSI Synthesis, vol. 2, Springer Science & Business Media, 1984, pp. 160-174.

[18] S. Kumar, Design and Analysis of Algorithms: A Contemporary Approach, vol. 1, New York, NY: Cambridge University Press, 2019, pp. 40-201.

[19] R. K. Brayton, "VIS: A system for verification and synthesis," International Conference on Computer Aided Verification, pp. 428-432, 1996.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach," Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 117-126, Jan. 24-26, 1983.

[21] D. Geist and I. Beer, "Efficient model checking by automated ordering of transition relation partitions," International Conference on Computer Aided Verification, vol. 818, June 7, 2005.

[22] I. Beer and D. S. B., "RuleBase: An industry-oriented formal verification tool," in 33rd Design Automation Conference Proceedings, Las Vegas, NV, 1996.

[23] W. Meng, "Automated design of search algorithms: Learning on algorithmic components," Expert Systems with Applications, vol. 185, no. 1, pp. 1-13, Sept. 15, 2020.

[24] Z. Zhang and X. Chen, "Reachability analysis of networked finite-state machines with communication losses: A switched perspective," IEEE Journal on Selected Areas in Communications, vol. 38, no. 5, pp. 845-853, May 2020.

[25] J. Shalf, "The future of computing beyond Moore's law," in Numerical Algorithms for High-Performance Computational Science, The Royal Society, 2020, pp. 1-13.

[26] L. Sai and H. Jianwei, "Study on the single-event upset sensitivity of 65-nm CMOS sequential logic circuit," IEICE Electronics Express, 2020.