Proceedings of the CONF-MLA 2026 Symposium: Explainable Computing, Modeling & Data Science in Complex Systems
DOI: 10.54254/2755-2721/2026.GF37590

Heterogeneous Computing Systems: A Review of Architectural Evolution and Design Methodologies

Zhengyuan Xia

School of Communication & Information Engineering, Shanghai University, Shanghai, China
xxxzyswd@qq.com

Abstract. Heterogeneous computing has emerged as an important architectural response to the slowdown of conventional semiconductor scaling, particularly in the post-Moore era. This review follows the shift from homogeneous multicore processors toward integrated platforms that place CPUs, GPUs, FPGAs, and domain-specific accelerators on the same die. The transition is closely related to two long-standing technology constraints, namely the slowing of Moore's Law and the breakdown of Dennard scaling, while Amdahl's Law still limits the speedup that can be obtained from parallel acceleration. From the system perspective, the review focuses on three recurring obstacles: the memory wall, cache coherence in shared-memory heterogeneous architectures, and on-chip communication supported by Network-on-Chip (NoC) designs. I also reviewed low-power VLSI approaches, especially clock gating and low-swing flip-flop designs, together with design methodologies such as SystemC-based high-level synthesis (HLS), the Catapult HLS flow, and latency-insensitive design. Based on these 24 research contributions, I organize the discussion into a multi-level technical framework for heterogeneous SoC design in AI and IoT applications.

Keywords: heterogeneous computing, Network-on-Chip, memory wall, high-level synthesis, latency-insensitive design

1. Introduction

When I began reading the 24 references used in this review, I was struck by how closely the story of computing performance has been tied to two simple trends. For about fifty years, Moore Law made transistors more dense, roughly doubling every two years, whereas Dennard scaling allowed voltage and device size to shrink together. That combination caused higher clock frequencies, which are feasible since each new technology node has been introduced. Around 2005， this comfortable pattern became hard to maintain. Threshold voltages are approaching practical limits, and leakage current is increasing rapidly. Moreover, Dennard scaling has not been delivering the same benefits. Process improvements and higher frequencies alone can't sustain performance growth as before. Moore Law also started to slow down, so further miniaturization resulted in smaller gains [1, 2].

These changes caused computer architecture to take a new direction. Initially, the industry tried to employ homogeneous multicore designs. This required adding more cores to enhance throughput. But as workloads became more diverse, this approach had limitations [3]. Heterogeneous designs

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2026.GF37590

are popular because they combine different processing units on a single chip: general-purpose cores, graphics processors, programmable logic, and accelerators for specific tasks. Biglari et al. [4] points out that energy efficiency is a major reason behind this shift. Different units are good at different kinds of computation, so tasks do not all have to run on the same type of core. I found this point especially important for the post-Moore era. NVIDIA, AMD, and Intel have all invested heavily in heterogeneous integration [1, 3], largely because AI training, inference, and large-scale analytics demand both more performance and lower energy consumption [5].

The benefit, however, comes with new design problems. Units may differ in memory hierarchy, communication rules, timing needs, and power behavior. Putting them together is not just a matter of connecting blocks. Hardware/software interfaces, shared-resource scheduling, and system-level verification all become part of the design problem. Several system-level constraints then become hard to ignore. Processor speed and memory access latency are moving apart, which is the memory wall. Inter-core bandwidth is constrained, communication introduces latency, cache coherence demands overhead, and both static and dynamic power must be managed [6]. The traditional RTL-based manual design flows have their limits, and as heterogeneous SoCs grow larger and more diverse, these limits become more apparent. Larger designs development and verification are longer, so the engineering effort needs to match the increasing chip complexity.

This review examines heterogeneous computing from three distinct perspectives. The first reason is that the architecture has changed: the end of Dennard scaling [7], and the limit described by Amdahl's Law [8, 9], and the role of vector processing units in data-parallel work [10] are mentioned. Second, what limits the system: memory wall, cache coherence [11, 12], and on-chip interconnect architectures [4] are discussed. Third, how to build it: low-power circuit techniques, high-level synthesis [13-15], and latency-insensitive design [16-18] are presented. All these together, the 24 references I reviewed offer a way to organize the discussion across architecture, circuit implementation, and design methodology. The framework mentioned is meant to serve as a practical reference for grasping the design principles of heterogeneous SoC. It seeks to determine the problems that are essential for the future research and engineering progress.

1.1. Dennard scaling failure and the rise of heterogeneous architectures

When I first studied this topic, I didn't just accept the conclusion that Dennard scaling had failed. I also looked at a table of Dennard scaling parameters, which listed a variety of quanties, like Vdd, delay time, and power/area across different technology generations. Seeing these parameters together, I realized the original logic of Dennard scaling: supply voltage and device dimensions were scaled together, which made delay decrease and power per unit area remain roughly stable. So, higher frequency didn't automatically cause power density to explode. Although the table is straightforward, it made the subsequent breakdown more straightforward to understand, because Vdd couldn't scale in the same manner, so the balance was no longer held.

After reading the literature, I found out that this breakdown was a big change in how computer architects handled performance growth. Esmaeilzadeh et al. [7] describes how power constraints increase, which causes a significant part of on-chip transistors to become inactive at a specific time, a phenomenon commonly termed as the "dark silicon" phenomenon. Under these conditions, just adding more homogeneous cores won't provide a proportional benefit, because the available power budget restricts the number of cores that can operate simultaneously. This why hardware specialization is appealing [2]: the accelerator logic is designed to handle specific computational tasks, thereby enabling the system to achieve significantly higher performance per watt.

DOI: 10.54254/2755-2721/2026.GF37590

Heterogeneous multicore processors to handle this issue, they distribute cores with different computational features on the same die. One of the commonly used arrangements features high-performance "big" cores with energy-efficient "little" cores allow the handling of serial and parallel workloads in different ways. ARM's big LITTLE architecture is a well-known example [2], it enables workloads to be relocated between core types according to their current requirements, thus providing a practical balance between performance and energy efficiency in mobile devices. As specialization kept going, heterogeneous systems expanded beyond just the basic CPU–GPU combinations. More recent architectures also include FPGAs, DSPs, and AI accelerators, which are mapped to the same computing platform [3, 19], so the range of workloads that can be mapped to hardware with different performance and energy characteristics is expanded.

1.2. Amdahl's law in heterogeneous systems

Since 1967, Amdahl's *Law has* given a *basic* method to *describe* the *limits* of *parallel performance*. Its main point is that the serial fraction of a program imposes an upper bound on speedup; even if the number of available cores is very large, the maximum speedup approaches the reciprocal of that serial fraction. For heterogeneous systems, however, the classical expression does not fully describe the effects of having different processor types, so extensions are needed [8].

A number of extensions have therefore been proposed for heterogeneous configurations. Standard and multi-fraction formulations cover a wider range of system organizations and have been evaluated on many-core platforms with realistic workloads [8]. The Hill–Marty model was one of the earlier efforts to apply Amdahl's Law to heterogeneous system analysis, while later work has also shown that the performance and energy-efficiency benefits of heterogeneous platforms remain constrained by the serial portion of an application.

The usefulness of Amdahl's Law is not limited to performance alone. It has also been applied to lifetime-reliability modeling and energy-efficiency scaling in heterogeneous multicore processors [9]. In these kinds of models, factors like processor composition, task scheduling strategy, and how serial and parallel parts of a workload scale, affect reliability, performance, and energy consumption. This perspective also affects how chip resources are distributed. Under the constraints of Amdahl's Law, designers must balance the allocation of area for "big" cores for serial acceleration and "little" cores for parallel throughput, rather than maximizing either type individually.

1.3. Vector processing units and data-level parallelism

Vector Processing Units (VPUs) are essential components in heterogeneous architectures, due to their ability to efficiently leverage data-level parallelism. Under a single instruction, multiple data (SIMD) execution model, a VPU performs the same operation several data elements simultaneously, by using vector registers to store the corresponding data. Compared with scalar execution, this arrangement can enhance throughput for workloads such as image processing, scientific computing, and machine learning [5].

As vector architectures have developed, decoupled VPU structures have also been gaing more attention. In this organization, VPU operates separately from the scalar core, so the two units can use different frequencies and perform computation and control tasks in parallel. Long-vector operations and more intricate vector instructions can be optimized without the need for the scalar core to operate under the same conditions. Additionally, power management can be handled separately for both units. Wang et al. [10] reviewed a number of decoupled VPU processors and mentioned the benefits in performance, energy efficiency, scalability, and flexibility. Their experimental results

DOI: 10.54254/2755-2721/2026.GF37590

indicate that adding a single-channel VPU to a heterogeneous multicore system can increase multicore performance by more than 2.4× compared with scalar processors, while single-core performance can increase by more than 36× [10].

2. System-level performance bottlenecks

2.1. The memory wall

In my reading of this topic, the memory wall is not only a memory problem. It is also a consequence of the slowing of Moore's Law and the breakdown of Dennard scaling, because these two trends changed both how processors are built and how many computing units are placed on a single chip.

When Dennard scaling still worked, supply voltage and device dimensions could shrink together, so higher frequency did not automatically cause power density to rise out of control. Moore's Law also supplied more transistors each generation. Once these trends weakened, performance growth increasingly depended on adding more cores and, later, on integrating heterogeneous units such as GPUs, FPGAs, and domain-specific accelerators. This response can raise throughput, but it also increases the number of concurrent computation and data-access streams on the same die.

That is the place where the memory wall becomes more intense. CPU capability has increased by roughly 60% annually, whereas DRAM access latency has improved by about 7% annually. In a heterogeneous architecture, CPUs, GPUs, and accelerators might share the same memory hierarchy, compete for bandwidth, and there are several access streams that can operate concurrently, which adds pressure on limited off-chip memory bandwidth [6].

Semiconductor scaling also raises another architectural concern: the global interconnect propagation delay is not improving as rapidly as the size of the transistors. Signals need more clock cycles to travel a chip, so it creates a large SoC that looks like a distributed system from a physical implementation perspective. This development puts pressure on traditional synchronous design assumptions, especially the idea that a signal can pass through the entire chip in one clock cycle. That assumption was supported by computer-aided design toolchains for a long time, but the scale of modern SoCs has changed a lot, making them from designs with thousands of transistors to systems with billions. Design methodologies thus require explicit consideration of communication delay and physical distance, rather than treating the chip as an ideal synchronous structure.

2.2. Cache coherence and memory consistency

In this section, I realized that cache coherence and memory consistency are important not just in general shared-memory systems, but especially because a single SoC might have multiple accelerators sharing the same memory addresses. When a number of units maintain local copies of the same data, and at least one of them writes to that address, then the copies may diverge. If the system doesn't maintain units consistency, then different units might see different values, and program behavior becomes unpredictable.

Cache coherence protocols are used to control how writes are propagated and observed. They usually use finite-state machines at the cache-line level, so shared-data accesses from different cores follow a well-defined coherence model [1, 12]. From this, I comprehend that the protocol is more than just about performance; it's the definition of the shared memory system.

To make this more concrete, I practiced with a simple multiprocessor architecture, which is based on a shared bus. I created a table and tried to write out the changes in the state of a given byte under different access sequences. In the process, I encountered events and states such as read miss, write

DOI: 10.54254/2755-2721/2026.GF37590

hit, invalid, and modified. Working through these cases made me grasp why the protocol demands explicit states, and why a write by one processor can't just remain invisible to others. It also made me realize why coherence is a correctness requirement, not an optional optimization.

When CPUs, GPUs, and domain-specific accelerators are present together, the design becomes more challenging. These units can have varied cache hierarchies and access patterns, like data-access stride, spatial locality, and required coherence strength. A coherence protocol was developed for a homogeneous multicore processor, so it can't be directly transferred to a heterogeneous system without any modifications. Nagarajan et al. [11] in their 2023 book discusses memory consistency and cache coherence systematically, saying that modern computer systems, either homogeneous or heterogeneous, already have many implementations with native hardware support for shared memory [12]. This existing hardware support offers a useful foundation, but heterogeneous systems still require the processing units to account for their differences.

New interconnect protocols also become important for this problem, such as CXL, NVLink-C2C, and AMD Infinity Fabric [3]. These technologies enable cache-coherent links, and can be employed to construct heterogeneous systems with varied processors and accelerators, sharing resources via memory pooling, distributed shared memory, and unified address spaces. When hardware takes care of coherence across components, part of the burden can be removed from software. In practice, this method can simplify the complexity of memory management, and reduce the typical overhead in software stack.

2.3. On-chip communication architecture and Network-on-Chip

As the number of cores on a chip keeps increasing, the data moving between cores contributes more to both system latency and power consumption. Traditional shared-bus structures and point-to-point interconnects are hard to scale when there are many cores that need to communicate, especially because bandwidth and arbitration efficiency don't increase with system size. Network-on-Chip (NoC) architectures provides a more organized and structured way to address this communication issue. They have been adopted in large-scale multicore and many-core SoCs as an alternative to conventional shared buses and point-to-point interconnect structures [4], where the network provides organized paths for data movement among processing elements.

Heterogeneous systems has technical requirements for NoC design, which are different from those of homogeneous systems. GPU cores usually employ high-bandwidth, data stream communication, and demands high throughput; CPU cores, on the other hand, exhibit point-to-point, low-latency control interactions, and are more characterized by these. Biglari et al. [4] analyzed the NoC design space for heterogeneous systems, and pointed out a few issues that need to be considered together—topology selection, routing algorithm design, virtual-channel allocation, and QoS support. Their analysis indicates that a NoC tailored for homogeneous CPU traffic might not meet all communication requirements found on a heterogeneous platform at the same time. Current work therefore gives attention to adaptive routing that can react to changes in traffic patterns, as well as heterogeneous NoC topologies that assign different service levels to different traffic classes [20].

3. Low-power VLSI design techniques for heterogeneous systems

Power constraints have become the core design metric in modern VLSI design, second only to functional correctness. Data indicates that clock distribution networks and timing units, including various flip-flops and latches, together can account for 30%–60% of the chip's total power consumption [6]. In heterogeneous systems, multiple computing units might be active

DOI: 10.54254/2755-2721/2026.GF37590

simultaneously under power budget constraints, which would impose higher demands on power optimization technologies.

3.1. Clock gating

To grasp clock gating more directly, I once constructed a simple clock-gating circuit diagram. The diagram was not too complicated, but working it through allowed me to grasp the essence of low-power design under a constrained power budget: power is not just saved by making active operations faster, but also by avoiding unnecessary activities. In the case of clock gating, this implies that the clock is stopped, so it can't reach modules that don't need to update their state.

From the literature, I found out that clock gating reduces dynamic power by blocking the clock signal to idle circuit modules. When a module doesn't need to update its sequential elements, it avoids unnecessary clock transitions, and also avoids switching activity in the associated clock buffers and internal nodes. The basic operation is quite straightforward. If the input data of a sequential element is expected to stay the same during the next clock cycle, then the gating logic can block the clock path instead of allowing another transition through the circuit. I also found out that researchers have directly incorporated clock-gating logic into the flip-flop itself, resulting in embedded clock-gating schemes [21, 22]. Under conditions with relatively low data-toggle activity, test results show that these approaches can reduce dynamic power consumption by roughly 41%–56% compared to conventional designs [22]. Seeing this result after working through the simple diagram made the idea more concrete for me: even a straightforward gating structure can have a meaningful effect when the power budget is tight.

3.2. Low-swing flip-flop and latch techniques

Low-swing clocking reduces the charging and discharging energy of the clock network by using a smaller voltage swing. Among the approaches developed in this area are pulse-triggered latches based on a single-clock-transistor structure and edge-triggered flip-flops that use capture mechanisms under low-voltage-swing conditions; both are intended to reduce the power associated with clock operation [6, 21].

For low-swing flip-flops implemented with MTCMOS techniques and a reduced clock swing, experimental results report about a 5% improvement in operating speed together with roughly a 10% improvement in power efficiency when compared with full-swing designs [22]. These techniques fit naturally with the structure of heterogeneous systems because a heterogeneous SoC commonly contains several clock domains and voltage islands. Low-swing techniques can therefore be applied separately within individual domains, and when they are combined with dynamic voltage and frequency scaling (DVFS), power consumption can be adjusted more selectively for different processing units.

4. Design methodologies for heterogeneous systems

4.1. Systemc-based high-level synthesis

Before I read the literature for this section, I had tried writing some simple C++ code myself, including MAC and Adder .cpp and .h files. At that moment, I mainly treated them as ordinary classes and functions. I focused on addition, multiplication, and how one module called another. Later, while studying SystemC and high-level synthesis together, I realized that SystemC is not a

DOI: 10.54254/2755-2721/2026.GF37590

complete separate language. It adds the basic semantics need for hardware description, which is done on top of C++. This experience made it easier for me to understand why HLS can move the entry point of hardware design from the register-transfer level to the behavioral or algorithmic level, and how a C++ class library can represent hardware modules.

From the literature, I found out that high-level synthesis (HLS) is a change in the starting point of hardware design. Instead of starting at the register-transfer level, it starts with a behavioral or algorithmic description. As I read further, I found out that this change isn't just reducing the time taken for development. It also enables designers to consider a variety of implementation strategies, thereby simplifying the process of design-space exploration [13, 14]. SystemC, according to my understanding, is the core of this methodology. It is a C++ class library which provides hardware-description capabilities to the C++ programming environment. These are the constructs for modules and module hierarchies, ports, signals, clocks, resets, and timing. What I learned from this that designers can describe hardware behavior while maintaing a familiar C++ development environment [13]. This point is closely related to my own experience with MAC and Adder. Initially, they were just ordinary C++ classes. But when ports, signals, and clocks were introduced, the way the code is organized started to resemble hardware modules instead of plain software functions.

Another feature I found very useful is that SystemC supports the object-oriented organization of synthesizable hardware models. Class inheritance, polymorphism and encapsulation are ways to structure designs. From what I read, this usually results in a more compact description compared to an equivalent implementation written in a traditional hardware description language. Grötker et al. [15] analyzed the modeling capabilities of SystemC, and found out that SystemC can represent both combinational and sequential processes. From this, I learned that the same language can be used at various abstraction levels, like algorithmic behavior, detailed hardware modules, etc. So, related descriptions can be kept in a common syntax. With the support of transaction-level modeling extensions, it can perform system-level simulation. Before RTL implementation is completed, a design team can employ this capability to verify the anticipated functionality and performance of an SoC at an earlier stage. From my perspective, this is a valuable aspect because it allows the identification of expensive design changes before the implementation is completed.

4.2. Catapult high-level synthesis design flow

Reading about Catapult, I found it to be a concrete example of how the HLS ideas discussed above are turned into a hardware design flow. Catapult HLS was introduced by Mentor Graphics, and it was first released in 2004. From the literature, I learned that it has since been used in the tape-out and delivery of thousands of chip projects. I understood its basic role as follows: it accepts functional models written in C++ or SystemC, and then carries out several synthesis and optimization stages, including compilation and parsing, scheduling, resource allocation, and operator binding, before producing synthesizable RTL code [15].

What I found particularly helpful in understanding this flow was the engineering sequence it implies. An embedded algorithm is first implemented in a high-level language on a processor platform and verified functionally. Once the performance-critical parts have been identified, those parts can be moved to hardware accelerators through Catapult, instead of rewriting the entire application as RTL from the beginning. In the reading, this is what makes the flow practical: it does not require designers to abandon the high-level description once the algorithm is verified.

The aspect I came to appreciate most is the support for design-space exploration. By changing constraints and compiler directives in synthesis scripts while keeping the same high-level specification, designers can generate multiple RTL implementations with different area, timing, and

DOI: 10.54254/2755-2721/2026.GF37590

power characteristics in a relatively short period. This rapid iteration is very relevant to heterogeneous system design, since accelerators usually need to be adjusted to meet the throughput and latency requirements of their target applications. From my understanding, HLS tools provide a multi-objective trade-off space, thus, they require some manual work to write and verify separate RTL implementations. Recent literature [23] gives a relatively comprehensive classification of the HLS ecosystem for heterogeneous FPGA systems.

4.3. Latency-insensitive design methodology

Latency-insensitive design (LID) is a technique developed to handle the increasing global interconnect latency in nanoscale technologies, while preserving the core advantages of synchronous design [16, 17]. Rather than requiring every module to meet a fixed communication delay, LID introduces a "protocol-shell" design paradigm, and it's a way to handle communication behavior via an interface surrounding the functional module. My reading of these sources made me categorize the core advantages of this paradigm into four distinct dimensions. This grouping is in line with the general presentation of these advantages in literature: modularity, scalability, flexibility, and efficiency. For the first part, global interconnect latency-induced timing effects can be included in the synchronous design framework, so functional modules can maintain a relatively independent relationship with the exact physical delay. In terms of scalability, during the design phase, interface synthesis can be employed to ensure the correctness of critical properties as the system evolves. When it comes to flexibility, communication can be separated from computation, so it's easier to adapt an existing module to different interconnect conditions or system organizations. Similarly, for efficiency, pre-designed modules can be reused in various systems, which is not redesigned when the surrounding communication structure changes [16, 18].

Under this method, a synchronous module is considered latency-insensitive since it can operate correctly even with arbitrary delays in its input and output channels. The main idea is that functional correctness isn't determined by a fixed communication latency. If a communication channel is temporarily out of service, then clock-gating mechanisms can make modules that depend on this channel be suspended until communication is restored. The LID protocol stack therefore allows SoC designers to develop the internal logic of individual computing cores separately from their communication interfaces while retaining the synchronous design paradigm, which can reduce some of the difficulty associated with timing closure during system integration [17, 24].

Carloni in this area [16, 17] has a systematic work, which connects the early theoretical foundations of latency-insensitive design with later developments in system-level design methodologies. His contributions were also mentioned as a key factor in his selection as an ACM Fellow in 2025, indicating the ongoing importance of this research in both academic and industrial settings. LID provides a conceptual connection between the system-level functional design and the timing requirements that occur during the physical implementation process. SoC platforms are becoming more complicated, this methodology is a good way to manage communication-related design complexity and handle timing and interconnect delay issues [18].

I learned LID most clearly by doing a small exercise. I used an input signal trace to examine the behavior of an on-chip logic circuit, focusing on the timestamps 1 to 11. I also tried to write out the changes that would happen to the internal and output signals under different inputs. Although I made a few mistakes and found it very hard to understand how it works initially, working through the internal and output transitions step by step made the idea more concrete for me: the module can behave correctly even with communication delays, and its internal behavior can be analyzed separately from the timing of the surrounding interface. This experience has made me realize that

DOI: 10.54254/2755-2721/2026.GF37590

LID is more than just a theoretical model; it also serves a practical approach to thinking about how a module behaves when its inputs and outputs are not available at fixed times.

5. Conclusion and future perspectives

After reading and comparing 24 references, my main conclusion is that heterogeneous computing isn't just about putting different processors onto one chip. I came to see it as a response to the end of easy scaling: Dennard scaling no longer gives us free frequency gains, Moore's Law is slowing, and Amdahl's Law still limits the benefit of parallel acceleration. Across the papers, I noticed a recurring pattern. When process and frequency can no longer drive performance, architects turn to specialization; as specialization increases, memory, cache coherence, and on-chip communication become the real bottlenecks; and as those bottlenecks grow, power consumption and design complexity become first-order constraints. This is why the memory wall, NoC design, low-power VLSI techniques, high-level synthesis, and latency-insensitive design belong to the same story. They are different layers of one problem. My own understanding is that heterogeneous system design is less about choosing the fastest core and more about balancing computation, data movement, energy, and verification effort together. Looking forward, I believe the focus will continue to move from individual compute units toward the memory and interconnect system that connects them. AI workloads will keep pushing accelerator design, open coherent interconnects such as CXL will make memory pooling and disaggregation more realistic, and security, reliability, and machine-learning-assisted design exploration will become increasingly important. In the end, what I learned is that the future of heterogeneous computing depends less on adding more types of cores than on making the whole platform coherent, efficient, and manageable.

References

[1] A survey on heterogeneous CPU–GPU architectures and simulators. (2024). *Dimensions*.
[2] An architecture for heterogeneous high-performance computing systems: Motivation and requirements. (2024). *IEEE Xplore*.
[3] Heterogeneous integration technologies for artificial intelligence applications. (2024). *IEEE Xplore*.
[4] Biglari, S., Hosseini, F., Upadhyay, A., & Zhao, H. (2024). Survey of network-on-chip (NoC) for heterogeneous multicore systems. In *Proc. IEEE 17th Int. Symp. Embedded Multicore/Many-core Syst.-on-Chip (MCSoC)* (pp. 155–162). Kuala Lumpur, Malaysia.
[5] Chen, Y.-H., Krishna, T., Emer, J., & Sze, V. (2016). Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *Proc. ISSCC* (pp. 262–263).
[6] Comprehensive review of low-power VLSI design techniques: Challenges, cryptographic implementations, and future prospects. (2025). *IEEE Xplore*.
[7] Esmaeilzadeh, H., Blem, E., Amant, R., Sankaralingam, K., & Burger, D. (2011, June). Dark silicon and the end of multicore scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (pp. 365–376).
[8] Amdahl's law in the context of heterogeneous many-core systems—A survey. (2020). *IET Computers & Digital Techniques, 14*(4).
[9] Amdahl's law for lifetime reliability scaling in heterogeneous multicore processors. (2016). *IEEE Transactions on Computers*.
[10] Wang, R. X., Zhang, D. B., Lang, Q. J., Xie, D. H., Wang, Z. W., Gao, Z. Y., & Shen, L. (2025, September). Decoupled vector processing unit: Past, present, and future. *Journal of Computer Science and Technology, 40*(5), 1368–1385.
[11] Nagarajan, V., Sorin, D. J., Hill, M. D., & Wood, D. A. (2023). *A primer on memory consistency and cache coherence* (2nd ed.). Springer.
[12] Cache coherence protocols in multiprocessor systems: A survey and comparative analysis. (2024).
[13] Object-oriented high level synthesis based on SystemC. (2001). In *Proc. IEEE Int. Conf. Electron., Circuits Syst.*.
[14] SystemC for high-level synthesis. (2026). Cadence Training.

DOI: 10.54254/2755-2721/2026.GF37590

[15] Coussy, P., & Morawiec, A. (Eds.). (2008). *High-level synthesis: From algorithm to digital circuit*. Springer.
[16] Carloni, L. P. (2015). From latency-insensitive design to communication-based system-level design. *Proc. IEEE, 103*(11), 2133–2151.
[17] Carloni, L. P., McMillan, K. L., & Sangiovanni-Vincentelli, A. L. (2001). Theory of latency-insensitive design. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., 20*(9), 1059–1076.
[18] Generalized latency-insensitive systems for single-clock and multi-clock architectures. (2004). *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*.
[19] Asanovic, K., et al. (2016). The Rocket Chip generator. Tech. Rep. UCB/EECS-2016-17, Univ. California, Berkeley.
[20] A scalable FPGA-based RISC-V many-core platform using a lightweight NoC. (2024).
[21] A comprehensive survey of clock-gating techniques for low-power VLSI design. (2026). *IEEE Xplore*.
[22] Power-efficient dual-edge implicit pulse-triggered flip-flop with an embedded clock-gating scheme. (2025). *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*
[23] A taxonomy of the high-level synthesis ecosystem for heterogeneous FPGA systems. (2025). *ACM Trans. Reconfigurable Technol. Syst., 18*(4).
[24] Superconnectors: A latency insensitive approach to SFQ design. (2025). *IEEE Xplore*.