Journal of Computer and Communications, 2025, 13(10), 147-186
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

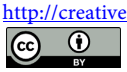
The Development of Digital Chip

Di Wu

School of Electronics and Information, South China University of Technology, Guangzhou, China
Email: 354309763@qq.com

How to cite this paper: Wu, D. (2025) The Development of Digital Chip. *Journal of Computer and Communications*, **13**, 147-186.
https://doi.org/10.4236/jcc.2025.1310009

Received: August 31, 2025
Accepted: October 24, 2025
Published: October 27, 2025



Abstract

The golden age of digital chips seems to be coming to an end. For decades, we have relied on making transistors smaller and increasing clock speeds to improve performance. However, when chip sizes shrink below 90 nanometers, this approach becomes untenable—power consumption spirals out of control, and heat issues threaten to burn the chips. Meanwhile, processors are getting faster, but memory cannot keep pace, widening the gap between the two. Faced with these challenges, engineers have had to explore new avenues. This article outlines the four major breakthrough directions in digital chip design in recent years: leveraging parallel computing to enable multiple cores to collaborate and divide tasks, using pipeline and hyper-threading technologies to maximize the value of each clock cycle; adopting clock gating, low-swing signals, and other technologies to optimize energy efficiency down to every joule; designing specialized accelerators tailored for specific tasks, sacrificing versatility but achieving energy efficiency improvements of tens or even hundreds of times; and constructing latency-insensitive designs and on-chip networks to ensure more stable and reliable communication within the chip. These technologies do not exist in isolation but complement and support one another, collectively shaping the new landscape of chip design in the post-Moore's Law era. The shift from solely pursuing faster and smaller chips to balancing power consumption, performance, and area marks a profound paradigm shift in chip design. Therefore, the purpose of this paper is to provide a quick introduction for beginners entering the digital chip industry to the challenges encountered in chip development and some cutting-edge technologies for addressing these issues.

Keywords

Computer Architecture, Post-Moore's Law, Low-Power Design, Heterogeneous Computing, Network-on-Chip, System-on-Chip

1. Introduction

Once upon a time, the life of a chip designer was simple—wait for the next generation of process nodes, and transistors would naturally shrink, with performance naturally improving. This golden age lasted nearly half a century. Gordon Moore's prediction in 1965 and Robert Dennard's scaling law proposed in 1974 served as two reliable compasses, guiding the entire semiconductor industry forward. Every two years, the number of transistors doubled, while power density remained constant—this almost became a matter of faith.

However, the laws of physics are relentless. When we shrunk transistors below 90 nanometers, problems began to arise. First, Dennard's scaling law ceased to hold—threshold voltage could no longer be reduced, as further reduction would cause leakage current to grow exponentially, causing the chip to consume power wildly even when idle. Next came the power wall issue, with Intel's Pentium 4 being a classic example. In pursuit of higher frequencies, power consumption and heat generation reached unbearable levels. Meanwhile, while processor speeds increased by 60% annually, memory speeds only improved by 7%, creating a "tortoise and hare" scenario that led to the memory wall problem, forcing even the fastest CPUs to frequently pause and wait for data. Even worse, once a chip integrates billions of transistors, how to enable efficient communication between them and how to verify the correctness of the design become enormous challenges.

Faced with these challenges, chip designers have embarked on an innovation race. Since a single core can no longer run fast enough, multiple cores are used together, thus giving rise to multi-core processors and various parallel technologies. Since overall power consumption cannot be reduced, inactive parts are put to rest—leading to the emergence of clock gating and dynamic voltage regulation. Since general-purpose processors are inefficient, let's customize hardware for specific tasks—GPUs handle graphics, NPUs handle AI, and VCU handle video, each with its own role. Traditional bus architectures can't handle complex on-chip communication, so let's borrow ideas from the internet and build networks within the chip. These innovations aren't isolated fixes but a systematic transformation.

This article aims to paint a comprehensive picture of this transformation. Unlike previous studies that focused on specific technologies, we seek to highlight the intrinsic connections between these innovations—how they influence and constrain one another, and how they collectively shape today's chip design landscape. We will explore this transformation from four dimensions: The first section examines how parallel technologies break through performance bottlenecks, including pipeline, multi-core, and hyper-threading technologies, and how Amdahl's Law helps us understand the limits of parallelization; The second part analyzes the three pillars of power management: clock gating, low swing design, and dynamic voltage and frequency scaling; the third part discusses the trend toward specialization, examining how abandoning general-purpose functionality can yield remarkable

efficiency gains; the fourth part focuses on communication issues within chips, introducing latency-insensitive design and on-chip networks to ensure system stability. The fifth part takes a broader perspective to explore the revolutionary communication architecture of on-chip networks. In the sixth part, we will discuss several challenges that run throughout: the memory wall problem and its countermeasures, how to manage increasing complexity with high-level design methods, and the impact on the economy and the environment. Finally, we will discuss the future of digital chip.

Through this overview, we hope readers will understand that chip design in the post-Moore's Law era is no longer a simple race for process technology, but a systematic engineering endeavor that requires careful balancing and innovation across multiple dimensions.

2. High Parallelism

2.1. The Background of Neoteric Development of Digital Chip

Several decades ago, Moore's law successfully predicted the development of digital systems [1]. According to the paper, as the number of components per integrated circuit increases, the unit cost decreases accordingly. At the same time, the number of transistors on integrated circuits roughly doubles every two years. The document indicates that as the number of components per integrated circuit increases, the unit cost decreases accordingly. Moore's 1965 paper predicted that by 1975, it might be possible to integrate as many as 65,000 components onto a single silicon chip. This prediction has been widely cited and has formed what is known as "Moore's Law," which states that the number of transistors on integrated circuits roughly doubles every two years. Moore mentioned in the document that the cost of individual integrated circuits is decreasing, and as more functions are integrated into circuits, the cost-effectiveness continues to improve. This point reflects that Moore's Law is not only a technological prediction but also concerns economic efficiency. With the constrained area, if we pursue high performance, it's absolute that we need to reduce the size of the components, and then we will get a sophisticated system that can achieve many functions. Dennard's scaling is used for this, so we can accurately calculate the proportion of the components with its help [2]. The core idea is elegantly simple yet profound: as transistor dimensions shrink by a consistent scaling factor κ (kappa), other key parameters—including voltages and doping concentrations—can be adjusted proportionally so that the internal electric fields remain constant, preserving device behavior while achieving dramatic improvements in performance, density, and power efficiency. The practical implementation of Dennard scaling follows a systematic approach. When all linear dimensions (channel length L, width W, and gate oxide thickness tox) are reduced by factor κ, the physical benefits are immediate—shorter channels mean charge carriers travel shorter distances between source and drain, enabling faster switching. However, applying the same operating voltage across a shorter channel would dramatically increase electric field strength ($E = V/L$), po-

tentially causing reliability catastrophes such as hot-carrier injection and gate oxide breakdown. To prevent this, all operating voltages must scale down proportionally: supply voltage V_{DD} becomes V_{DD}/κ, and threshold voltage Vth scales to V_{th}/κ. This voltage scaling provides a double benefit—it maintains electric field integrity while delivering quadratic power reduction, since dynamic power scales with the square of voltage ($P \propto V^2$).

The genius of Dennard scaling lies in its holistic approach to parameter optimization. Achieving proper threshold voltage scaling requires increasing substrate doping concentrations by factor κ, which compensates for the reduced depletion region width and maintains switching characteristics. Meanwhile, the scaling of various capacitances works favorably: while gate capacitance remains roughly constant, junction capacitances decrease, and combined with shorter transit times, this enables frequency improvements roughly proportional to κ. The area per transistor scales as $1/\kappa^2$, allowing transistor density to increase by κ^2 for the same chip area, creating the foundation for Moore's Law's continued progress.

For over three decades, from the 1970s through the early 2000s, Dennard scaling delivered on its promises spectacularly. Intel's progression from 10 μm processes in 1971 to 130 nm in 2001 largely followed Dennard principles, achieving simultaneous improvements in speed (frequencies increased from MHz to GHz), density (transistor counts grew exponentially), and power efficiency (power per operation decreased consistently). This golden age of scaling enabled the entire digital revolution, making possible everything from personal computers to smartphones.

However, with the development of technologies and the limited area on the chip, billions of components exist on the chip. However, the smaller and smaller components make Dennard's Scaling invalid. To be specific, when the chip size is within 90 nm, Dennard's scaling fails. The first reason is that real designs became more aggressive—longer pipelines, larger caches, and more transistors per chip—pushing power requirements beyond what air cooling or battery-based systems could manage. Besides this, according to the formula for subthreshold leakage current:

$$I_{SUB} = \frac{W}{L}\mu v_{th}^2 C_{sth} \mathrm{e}^{\frac{V_{GS}-V_T+\eta V_{DS}}{nv_{th}}}\left(1-\mathrm{e}^{\frac{-V_{DS}}{v_{th}}}\right)$$

Around the 90-nm node, the threshold voltage V_T cannot continue to fall further, because lowering V_T further would significantly degrade circuit performance and cause exponential increases in leakage current, which will result in a dramatic increase in power. In the meantime, as laptops and mobile devices became more popular, thermal and power constraints became stricter. Designers could no longer simply convert additional power into higher performance. With constant or only slightly reduced voltages and aggressively scaled transistor dimensions, chips ran hotter, and cooling solutions became costlier and more complex. Hence, Dennard's original assumptions—synchronous scaling of all dimensions and volt-

ages—were not maintained in modern processes. Leakage currents, thermal limitations, and design aggressiveness (e.g., deeper pipelines) caused the industry to deviate from ideal constant field scaling. As a result, power densities rose, frequencies grew faster than predicted, and we eventually hit the "power wall," marking the end of Dennard's scaling [3]. Therefore, we should find other ways to optimize our chips and improve our performance.

2.2. Temporal Parallelism

2.2.1. Pipelining

"Pipelining" is a technique for splitting an instruction or data processing task into multiple successive stages for parallel processing [4]. Each stage completes a portion of the overall task and then passes the intermediate results to the next stage. In this way, different instructions or data can run in parallel on different stages at the same time, dramatically increasing the overall throughput. The key to achieving the pipeline is that registers are inserted between stages to hold intermediate results and isolate stages, allowing each stage to start the next operation independently on the upper edge of the clock. Ideally, if the original system had N stages, its throughput could be increased by a factor of N, at the cost of only having a delay for the final output. Although the pipelining seems such a perfect thing, problems also exist in the design. Three main kinds of problems will strongly exacerbate our efficiency: Data Hazard, Control Hazard, and Structural Hazard.

To be specific, Data Hazard is when instruction stalling occurs when a subsequent instruction needs to use the result of a previous instruction, but the previous instruction has not yet been completed. How to solve it? There are many measures, and I will only introduce several common ways. The first way is Forwarding and Bypassing. Adding a path between execution units directly "bypasses" the result of a previous stage (e.g., execution completion) to a subsequent stage that requires it, eliminating the need to wait for a write-back. The second way is using Out-of-Order Execution [5]. Subsequent instructions are allowed to override previous instructions as long as they do not violate data dependencies. The third way is using Software Scheduling. The compiler rearranges the order of instructions as it fires them, filling the delay slots with unrelated instructions and reducing hardware insertion bubbles.

Then, Control Hazard is that when branch and jump instructions are executed, the address of the next instruction to be fetched cannot be determined until the branch resolution is completed, resulting in the fetch phase not knowing which instruction to grab [6]. It's a little abstract, so I will use an example to explain. Look at **Figure 1**, in cycle 1: the IF stage fetches the BEQ at address 100. In cycle 2: since the branch outcome isn't known until the EX stage, the IF stage speculatively fetches the instruction at address 104 (ADD R3, R4, R5). In cycle 3: in the EX stage, the CPU evaluates BEQ. If it turns out the branch should have been taken, both the ADD and any subsequent instructions fetched are on the wrong path and must be flushed.

```
100:  BEQ R1, R2, LABEL   ; If R1 == R2, jump to LABEL (address 200)
104:  ADD R3, R4, R5     ; (Assuming "not taken," continue sequentially)
108:  SUB R6, R7, R8
...
200: LABEL: MUL R9, R10, R11
```

Figure 1. A simple sample of instruction.

There are also several solutions to solve it. The first way is Static Prediction. Fixedly always assuming "always jump" or "never jump", or depending on the probability of forward/backward branching of hotspots analyzed at compile time. The second way is Dynamic Prediction. The hardware utilizes two-bit saturation counters, local/global history tables (BHT, BTB), etc., to predict branch directions and targets in real time based on past execution results. The third way is a Delayed Branch. The compiler ensures that one or more "harmless" instructions (slot fillers) are placed after the branch instructions so that the pipeline is not interrupted.

The last problem is Structural Hazards. It is a structural conflict that occurs when multiple instructions simultaneously require the use of the same hardware resource (e.g., arithmetic logic units, register file ports, access ports, pipeline registers, etc.) that cannot be supported concurrently. It's easy to understand, and several solutions can be used to tackle it. The first way is Hardware Resource Replication. Increase the number of arithmetic units, access channels, or register ports so that concurrent instructions can each have exclusive access to resources. The second way is Scheduling and Renaming. In the disordered architecture, register renaming is used to avoid mapping multiple instructions to the same physical register port; through dynamic scheduling, instructions with available resources are executed first. The third way is the Command Flow Limit. The compiler or hardware limits the number of instructions that can be fired at the same time, or the ISA is designed to ensure at least dual-port access to critical resources.

2.2.2. Time Slicing

If we only have one core on the chip, how can we accomplish multiple tasks at the same time? As you can see, when a single core system encounters a very time-consuming task, it's highly inefficient to end this task only when the task has been completed, unless it's most significant. It's because if some tasks are completed too late will cause the chip turmoil and the chip won't provide the correct function for us. Therefore, we use some trikes to run several tasks at the "same" time. The technique is time slicing(it can also be used in multi-core chips), which isn't really making the tasks run synchronously, but it is just alike.

Time slicing is a scheduling technique that "cuts" CPU time into equal or variable length segments (time slices) and distributes execution rights among ready tasks (threads/processes) in turn [7]. In single-core or multi-core systems, multiple tasks are made to seemingly execute in parallel by rotating the use of tiny slices of time, thus improving system responsiveness and fairness.

To be specific, how can you achieve time-slicing in practical design? First, we need to "cut" the time into many slices. For example, we should divide the available time of the processor into "time quantum" of equal length, e.g., 10 ms or 20 ms per time quantum. Within a time slice, the CPU serves only the currently scheduled thread or process; when the time slice runs out, the OS forces an interrupt (via a clock interrupt). Then, in the interrupt handling routine, the operating system first saves the current task's register status, the program counter (PC), stack pointer, etc. into the PCB (process control block) of memory, and then loads the next task's context so that it can continue to run "from wherever it was last hijacked". How to apportion the assignments? There are several scheduling strategies. The first one is the round robin, the most classic time-slicing algorithm, where the next task in the ready queue takes turns getting a time slice, which is simple and fair. The second one is prioritized scheduling. It means that each task is prioritized, with high-priority tasks getting time slices more frequently or for longer periods of time; to avoid "starvation", priority aging is generally done. The last one is a multilevel feedback queue. We need to split the ready queue into multiple tiers with time slice length and priority inversely proportional to each other, with the lower tier queue having longer time slices but lower priority, dynamically adjusted to balance responsiveness and throughput. In addition, modern operating systems also monitor process behavior and dynamically adjust the length of time slices for each process during scheduling-interactive processes get shorter time slices to improve response speed, while batch processes get longer time slices to reduce switching overhead. The entire implementation process is interlocked: hardware timer → interrupt → context save → scheduling decision → context recovery → return to user state, enabling a single-core CPU to rapidly switch between multiple tasks, taking into account both system throughput and user experience.

2.3. Spatial Parallelism

2.3.1. Multi-Core

A multi-core processor is a design that integrates multiple independent computing cores on the same chip (SoC/CPU). Each Core contains its own ALU, register set, and sometimes a private cache (L1 Cache), and shares the on-chip bus, L2/L3 Cache, and memory controller with the other Cores [4] [8]. How can this technique improve the performance? First, it's true simultaneous execution. Each core is a complete set of execution resources. When the system has N cores, ideally, there are N instruction streams/threads that can run in parallel at the same time, independently of each other. Then, it reduces competition for resources. Unlike superscalar or hyper-threading parallelism, where "functional units are multiplexed within the same core," multicore duplicates the functional units of a processor without interfering with each other. Reduced Resource Contention. In this way, each core can access its private L1 Cache independently, dramatically reducing conflicts and waits caused by memory access. Unlike superscalar or hyper flu-

idized parallelism, where "functional units are multiplexed within the same core", multi-core duplicates the functional units of a processor without interfering with each other. Reduced Resource Contention. What's more, the different cores share cache and on-chip interconnect. Multicore chips typically share data at the L2 or L3 Cache level, and information can be exchanged faster between cores through Network-on-Chip (NoC, I will introduce it in the last part) or high-bandwidth buses. This ensures data consistency between parallel threads and also improves overall throughput when collaborating on multiple cores.

For instance, ARM is big.LITTLE heterogeneous-multicore architecture [9], as instantiated in the Samsung Exynos 5 Octa SoC, couples four out-of-order, high-performance Cortex-A15 "big" cores with four in-order, energy-efficient Cortex-A7 "LITTLE" cores [10]. At runtime, the system employs cluster migration and global task scheduling to direct compute-intensive threads to the A15 cluster while relegating background or latency-tolerant tasks to the A7 cluster, thus maximizing throughput under a fixed thermal-design-power (TDP) constraint. Dynamic voltage-frequency scaling (DVFS) per cluster and fine-grained power gating of inactive clusters further suppress both dynamic and leakage power, effectively compensating for the loss of Dennard scaling in sub-90 nm processes and mitigating the dark-silicon challenge on modern mobile SoCs. The likely technology also exists in iPhone's chips.

Table 1 highlights the evolution of Apple's chips from A4 to A12 in terms of process technology, die size, frequency, and the number of CPU and GPU cores, while reflecting advancements in parallelization, cache design, and specialization technologies. From A4 to A12, the CPU architecture transitioned from single-core to six-core designs (four high-performance cores + two high-efficiency cores), showcasing the application of parallelization to improve performance while optimizing power consumption. GPUs evolved from single-core to quad-core designs, focusing on enhanced graphical performance to support more complex tasks like rendering and machine learning. In terms of cache, technological advancements enabled larger integrated caches to reduce DRAM access latency and energy consumption. The A12 chip incorporates four small and two large cores architecture, combining parallelization and energy-saving strategies. Additionally, by integrating specialized hardware such as image processors and neural engines, the A12 chip is optimized for specific tasks like AI inference and AR operations, demonstrating significant progress in specialization technology. These advancements greatly improved computational efficiency and enhanced performance for targeted applications.

Table 1. The relevant information about iPhone chips.

Version	Dimension	Frequency/Hz	CPU	GPU
A4	53.3 mm^2 (45 nm)	800 M	one core	one core

Continued

A5	first: 122 mm^2 (45 nm) Improve: 69 mm^2 (32 nm)	800 M - 1 G	double cores	double cores
A6	96.71 mm^2 (32 nm)	1.3 G	double cores	three cores
A12	83.27 mm^2 (7 nm)	2.49 G	six cores (4 + 2)	four cores

2.3.2. Hyper-Threading Technology

Normally speaking, a single core can only execute one task during a period. However, the pursuit of high parallelism never stops. We have introduced a temporal parallelism technique—Time Slicing, which can achieve parallelism in a core, although it is not really parallel. Therefore, in this part, we will talk about another spatial parallelism technique—Hyper-Threading Technology. Before knowing what SMT(hyper-threading) is, we should know what multithreading is. Multithreading is a form of parallelization, or splitting work so that it can be processed simultaneously. Instead of giving a large amount of work to a single core, a threaded program splits the work into multiple software threads. These threads are processed in parallel by different CPU cores to save time. Hyper-Threading is the technique of Simultaneous Multi-Threading (SMT) to execute multiple threads in parallel on a single physical core [11] [12]. A physical core presents multiple "logical processors" to which the operating system can assign multithreading/multiprocessing to achieve hardware-level multitasking concurrency. A physical core that supports Hyper-Threading appears to the outside world as 2 (or more) "logical processors", which the operating system schedules as separate cores.

The core principle of Hyper-Threading is to replicate a minimal architectural state—such as program counters and register sets—to present two or more logical processors within a single physical core. These logical processors share most of the physical execution resources, including the instruction fetch and decode units, front-end pipeline stages, scheduling windows, functional units, and cache hierarchies. Each cycle, the pipeline is fed by dedicated thread-selection logic that fetches instructions from multiple threads. The resulting thread-tagged micro-operations are placed into a unified scheduling window, where all ready instructions from different threads dynamically compete for execution ports. This allows the processor to better utilize idle pipeline slots that may occur due to branch mispredictions, data dependencies, or memory stalls in one thread. As a result, other threads can fill these "bubbles", significantly improving resource utilization and throughput, with minimal increase in silicon area or power consumption. Intel's empirical measurements show that Hyper-Threading provides a 15% - 30% performance improvement for multi-threaded workloads with only a 5% increase in

die area. Specifically, on SPEC CPU2000 benchmarks, SMT achieved throughput improvements of 22% for integer and 28% for floating-point workloads [13].

2.4. Estimate the Performance of Parallelism

After we optimize our system, we usually want to estimate our improvement. Amdahl's law is a very useful way to predict the improvement [14]. To begin with, I want to derive this formula intuitively. When we own a system, and we want to improve it in some aspects. Use Fe to indicate the percentage of the original system's execution time that can be improved. And use Re to indicate the degree of improvement in the performance of the improvement component. Use the T and T0 to represent the time taken before and after the improvement, respectively. Therefore, T*(1 − Fe) is the part which can't be changed. T*Fe is the part that can be improved, and it can be optimized to T*Fe/Re. We can get T0 = T*(1 − Fe + Fe/Re). And acceleration rate is 1/(1 − Fe + Fe/Re). When the consumption time of the Fe part can be improved to 0. Then the acceleration rate is nearly 1/(1 − Fe). It reveals that when we optimize a part of the system, there will be a limit, and the performance will not go on endlessly. In addition, when we use this law in our actual system, Fe not only represents the percentage of the original system's execution time that can be improved, but also can be used to represent the proportion of the total number of parts that can be optimized from serial to parallel [4]. Then, the acceleration rate will indicate the increase in performance that can be obtained by optimizing a serial computing module with a parallel computing module. When we really think about a system, Amdahl's law is mainly used for the part of the pipeline with the longest delay. Because the pipeline is controlled by the longest delay, when we optimize other parts, the performance of the system will not be improved. Instead, when we improve the main part of the pipeline, the performance of the system will increase, and Amdahl's law is used to predict the degree of improvement.

All in all, when we finish designing a system, and we want to optimize it, the first thing we can do is to increase the pipelines, which directly improves the system. Then, we can optimize for the most time-consuming parts of the pipeline. For instance, use better algorithms, or use a multi-core processor to give serial computation to parallel computation. After improving the system, Amdahl's law can help us predict the performance after optimization. If we can't reach our goal, we can do other things to improve. After knowing the details of the law, we can understand that the law doesn't help us to optimize the system, instead, it only helps us know the result before we really produce the system.

2.5. Conclusion of High Parallelism

This chapter explores how parallel technologies overcome performance bottlenecks after Dennard scaling's failure. The core problem emerged below the 90 nm node, where threshold voltage could no longer be reduced without causing exponential leakage current growth, causing single-core frequency scaling to hit the

power wall. Solutions unfold across temporal and spatial dimensions: Temporal parallelism employs pipelining to decompose instruction execution into multiple stages, achieving instruction-level parallelism despite challenges like data hazards, control hazards, and structural hazards; time-slicing enables single-core "pseudo-parallelism" through rapid context switching. Spatial parallelism achieves true parallel execution through multi-core architectures, exemplified by ARM's big.LITTLE heterogeneous design combining high-performance and high-efficiency cores; Hyper-Threading replicates minimal architectural state to present multiple logical processors on a single physical core, delivering 15% - 30% performance improvement with only 5% area overhead. Amdahl's Law reveals parallelization's fundamental limit: even with infinite cores, 75% - 90% parallelizable code yields only 4 – 10 × speedup.

3. Low Power Consumption

Chip power consumption comprises dynamic and static components. Dynamic power arises from charging and discharging capacitive loads on every clock edge, while static power stems from leakage currents even when transistors are idle. As process nodes shrink and clock frequencies climb, dynamic power density and leakage escalate, leading to thermal limits and reduced battery life for portable systems. This situation is referred to as the power wall, where the increasing power consumption becomes the primary bottleneck, limiting further performance improvements due to thermal dissipation and energy consumption. Addressing these issues demands circuit- and system-level techniques that selectively reduce unnecessary toggling, lower voltage swings, and adapt voltage and frequency to workload demands. This chapter surveys three cornerstone methods—clock gating, low-swing signaling, and dynamic voltage & frequency scaling (DVFS)—that together strike a balance between performance and energy efficiency.

3.1. Clock Gating

To begin with, we will talk about the clock gating. Before knowing its principle, we should know what it is. To be specific, Clock gating is a power-saving technique in which a gate or small control circuit is inserted into the clock path to selectively enable or disable the clock signal for certain logic blocks [15]. The basic idea is to shut off the clock to modules that are idle or inactive, thereby eliminating unnecessary switching activity. And it has been invented because we want to reduce the power consumption. Dynamic power consumption is closely tied to clock toggles. Every clock edge can trigger charging and discharging in registers and combinational paths, thus consuming power. After knowing what it is and the background, we need to know the principle about how it works in order to better use it. First, we should use a gated flip-flop. An AND gate or a specialized gating cell is placed before the flip-flop, just like **Figure 2**. The clock can pass through only if an enable signal is active; otherwise, the flip-flop receives no toggling clock.

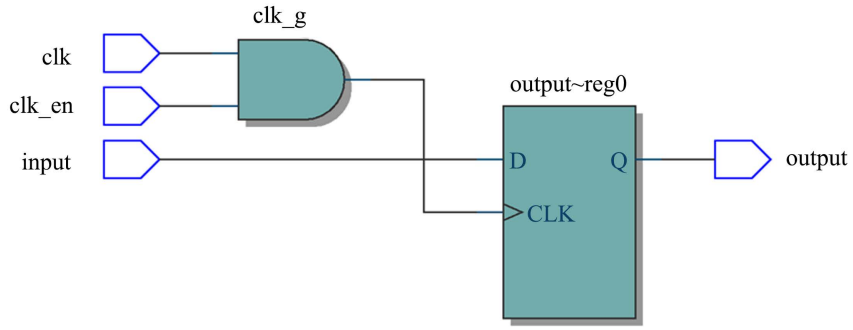


Figure 2. Simple clock gating application.

However, a glitch may be generated in the clock provided to the FF. Therefore, A modified version is shown in Figure 3.

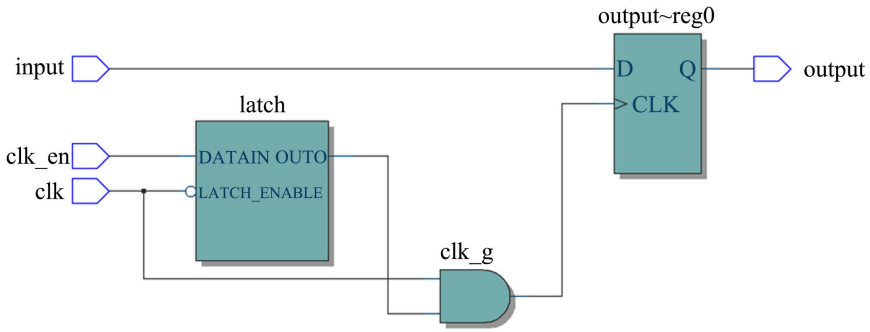


Figure 3. Improved clock gating application.

Besides this, we can also use model-level gating. A central controller (or local gating logic) monitors whether a functional module (e.g., VDE, DSP, DEU) needs clock signals at a given moment. Once a module completes its task or has no input data, the controller deactivates the clock to that module. Clock gating can be applied at various levels of granularity: individual registers, functional blocks, or entire IP cores. Finer granularity usually means higher complexity but can offer better power savings. M. Ohashi *et al.* successfully developed an MPEG-4 video decoding ASIC that achieves an optimal balance between power consumption and performance by employing low-power design techniques such as clock gating and power supply gating [16]. This demonstrates the feasibility and significant advantages of using dedicated hardware (ASIC) to address video playback challenges in mobile devices.

3.2. Low-Swing Flip-Flop

After talking about the clock gating, we will talk about the low-swing flip-flop/latch. First of all, we should know the logic of the low-swing flip-flop/latch. When we switch the CMOS, the capacitors will charge and discharge, and this will consume the power. The dynamic power is:

$$P_{dynamic} = \alpha * C * V^2 * f$$

α denotes the "Flip Factor" or "Activity Factor", which indicates the probability of a flip during a clock cycle. C denotes the capacity of the capacitor. V indicates the voltage swing for signal switching (usually V_{dd}). f indicates the switching frequency (usually related to the clock frequency). In the formula, the α C, f usually is a constant, which is decided by the circuit and design. Therefore, we can change V to reduce the dynamic power. However, if we directly lower the V, the circuit may produce many burrs, and the low Voltage can't be used for the next logic judge or other core logic. As a result, some technologists create some methods to solve it. For instance, in the paper "Dual supply voltage clocking for 5 GHz 130 nm integer execution core," they make some methods to achieve this idea. First, they use Dual Supply Voltage Clock Distribution. Lower the supply voltage (e.g., 0.8 V) on the clock grid, which is responsible for a significant portion of switching activity, while keeping core logic and latches at a higher voltage (1.2 V) for performance. Since clock toggles dominate dynamic power, reducing the voltage on the clock network substantially cuts down switching energy, including gate leakage and subthreshold leakage. Then, they divide the circuit into two types, G + L and G-only. As regards G + L, both the global clock grid and local clock buffers (LCBs) operate at the low voltage domain. To maintain robust latch operation at low voltage, the authors introduce a pass-transistor dual-V_cc latch that eliminates unnecessary inverters and DC paths, thus avoiding glitches and reducing static/dynamic losses. As regards G-only, the clock grid is at low voltage, but the local latch and buffer remain at a higher voltage; a split-output level-converting LCB is used at the end to shift from the low-voltage clock to the high-voltage latch domain [17].

Notwithstanding, though the low-swing flip-flop/latch is very fantastic, we should have a rule to choose an appropriate voltage (**Figure 4**). I will briefly outline how to select an appropriate voltage based on the paper by J. Cai *et al.* [18]. First, according to the paper, we are supposed to partition the circuit into high-activity and low-activity blocks. High-activity blocks (e.g., clock networks) switch very frequently, so dynamic power dominates. Low-activity blocks (e.g., most logic cells, caches) have lower switching rates, and thus static power can be significant. Next, we need to model power and performance. The dynamic power is: $P_{dynamic} = \alpha * C * V^2 * f$, and the static power is: $P_{DC} = I_{off} * V_{DD}$. Then, derive optimal voltages for different activity factors. For 100% activity circuits (e.g., clocks), optimal supply voltage is around 0.4 - 0.5 V. For moderate/low-activity circuits (10% and 1% activity), optimal supply voltages are around 0.9 V and 1.2 V, respectively. High-activity circuits gain significantly from low supply voltages (since dynamic power is proportional to V^2. Meanwhile, for low-activity circuits, static power, and noise margin concerns limit how far voltage can be reduced. Therefore, when we use it, we can reduce dynamic power, lower electromagnetic interference, and get the potential for higher working speeds. Quantitative studies demonstrate that fine-grained clock gating can reduce dynamic power by 20% - 40% in typical processors. For example, the ARM Cortex-A9 achieves 28% total

power reduction through hierarchical clock gating, with register-level gating contributing 12%, functional-unit gating 9%, and module-level gating 7% [19]. However, it will decrease noise margin, increase design complexity, and increase the difficulty of timing optimization.

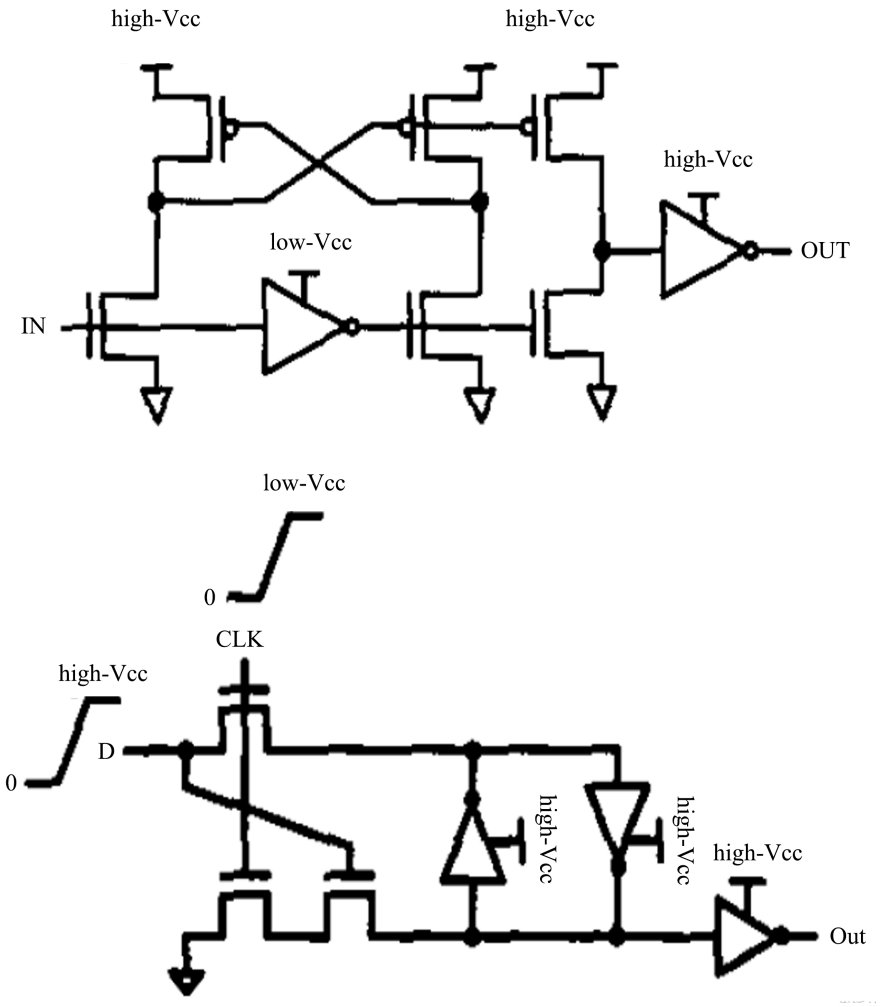


Figure 4. The principle of the low-swing flip-flop/latch [17].

3.3. Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) is a fundamental and core power management technology in modern processors and System-on-Chip (SoC). Its fundamental goal is to dynamically and synergistically adjust the operating voltage (Voltage) and clock frequency (Frequency) according to the real-time computing load of the chip. The goal is to minimize power consumption while ensuring mission performance, thereby maximizing Energy Efficiency. Focus on the formula:

$$P_{dynamic} = \alpha * C * V^2 * f$$

$$P_{leak} = I_{leak}(V) \times V$$

The first formula has been introduced in Part 2.2 Low-swing Flip-Flop. Besides, the second formula indicates the static power consumption. Leakage current is related to process, temperature, and supply voltage, and it typically decreases exponentially with V decreases exponentially. Therefore, DVFS is reduced by V and f simultaneously; these two components of power consumption can be reduced at the same time. However, unlike Swing Flip-Flop, which is a microscopic swing optimization affecting only the unit circuitry, DVFS is a macroscopic power and clock regulation affecting the entire chip or part of the core.

The DVFS implementation is a sophisticated closed-loop control system spanning software and hardware [20]. The first is at the software decision level. We need to design a Governor and OPPs. For the Governor, the DVFS scheduler in the operating system kernel (e.g., Linux) is at the center of decision making. It accurately senses the amount of computation required for the current task and thus predictively computes the best matching target frequency. The choice of target frequency is not arbitrary. To ensure stability, chip manufacturers pre-define a series of discrete, stable and reliable "frequency-voltage" combinations, known as OPPs, through rigorous wafer characterization tests, and Governor's decision-making process involves selecting the most appropriate one of these OPPs. The other is at the hardware execution level. After the Governor chooses the appropriate OPPs, the hardware layer will begin to implement the OPPs that reach the corresponding. The centerpiece here is the power supply to the Voltage Islands. Modern SoC chips are physically divided into multiple areas that can be powered and controlled independently, known as "voltage islands" (e.g., CPU cores, GPUs, NPUs, etc.). The power supply to these islands is based on an advanced two-stage hybrid power architecture: the first stage is a switching regulator for global coarse regulation, and the second stage is a low dropout linear regulator (LDO) for local fine regulation. For the first stage, this is usually performed by a power management unit (PMIC) external to the chip, which integrates several high-efficiency DC-DC converters whose main task is to efficiently (>90% efficiency) convert a higher voltage (e.g., 3.7 V) from the battery or from the motherboard into one or more lower intermediate voltages (e.g., 1.8 V or 1.1 V) that are required by the chip. It has the advantage of being highly efficient and capable of carrying high currents. For the second stage, one or more miniature LDOs (Low-Dropout Linear Regulator) are integrated directly on the silicon of the SoC chip, next to critical voltage islands like the CPU core. These LDOs receive intermediate voltages from the PMIC and perform the final, most precise voltage regulation. Its role is critical: when the CPU load changes drastically in nanoseconds, the LDO stabilizes the voltage with an extremely fast transient response, preventing it from dropping (Vdroop) and causing a system crash. At the same time, it filters out noise from the upstream DC-DC converter, providing extremely "pure" power to the CPU core. The advantages are fast response and low noise. Through these three steps—from rigorous testing at design time to set OPP, to intelligent software decisions at run time, to sophisticated implementation of the hybrid power architecture at

the hardware level—the DVFS system operates reliably and efficiently, realizing the perfect balance between dynamic performance and power consumption. power consumption.

Despite the sophisticated voltage regulation mechanisms in DVFS systems, they still face significant challenges from voltage droop during actual operation. When a processor suddenly switches from light-load to heavy-load conditions—for example, instantly launching intensive computational tasks from standby mode—current demand increases dramatically within nanoseconds. Due to parasitic resistance and inductance in the Power Delivery Network (PDN), this sudden current surge causes instantaneous voltage drops. More critically, if voltage droop causes the supply voltage to fall below the transistor's minimum operating voltage (typically slightly above the threshold voltage), the processor may experience timing violations, data errors, or even system crashes. Modern processors mitigate this risk through multiple approaches: integrating numerous on-chip decoupling capacitors as "charge reservoirs" to provide instantaneous current during load transients; adopting more aggressive voltage margin designs to ensure normal operation even under worst-case voltage droop scenarios; and implementing intelligent load management strategies to prevent all cores from simultaneously jumping from idle to full-load states. While these measures increase design complexity and cost, they are crucial for ensuring DVFS system stability across various workload conditions.

Now that we understand the above three approaches, we should know that in a real system, these three approaches coexist and assist each other. DVFS belongs to the System-Level, which is a macroscopic and dynamic strategy to regulate the operation of the whole functional module. Clock Gating is Architecture-Level, which dynamically shuts down temporarily unused portions of the chip at each clock cycle to eliminate invalid transistor flip-flops. Low-Swing Clocking, such as Swing Flip-Flop, is Circuit-Level, a microscopic, static design technique that saves power by reducing the voltage swing of the clock network signals themselves. These three techniques are not mutually exclusive, and a good low-power chip is inevitably the result of the synergistic optimization of these different levels.

3.4. Conclusion of Low Power Consumption

This chapter addresses the power wall problem impeding performance improvements. Dynamic power stems from capacitive charging/discharging each clock cycle, while static power originates from transistor leakage currents during idle states. Three core techniques collaborate at different levels: Clock gating operates at the architectural level by selectively disabling clock signals to idle modules, eliminating unnecessary switching activity and reducing dynamic power by 20% - 40%. Low-swing signaling works at the circuit level by reducing clock network voltage swing (e.g., from 1.2 V to 0.8 V), leveraging the $P \propto V^2$ relationship for substantial power reduction while requiring optimal voltage selection based on activity factors. DVFS functions at the system level, dynamically adjusting voltage

and frequency according to real-time workload through software Governors and hardware voltage islands, achieving 30% - 70% power savings. While these techniques increase design complexity (such as timing uncertainty from voltage drooping), they collectively achieve a crucial balance between performance and energy efficiency.

4. Specialism

General-purpose cores are no longer the one-size-fits-all solution. With Dennard scaling broken and the dark-silicon budget tightening, each incremental GHz delivers fewer performance gains per watt. Meanwhile, modern cloud and edge workloads—video transcoding, deep-learning inference, cryptography, genomics—exhibit orders-of-magnitude arithmetic intensity and predictable dataflow patterns that CPUs cannot exploit efficiently. Domain-specific accelerators (DSAs) therefore emerge as a pragmatic path forward: they sacrifice universality for tailored datapaths, memory hierarchies, and control logic that map precisely to an algorithm's hotspots. By eliminating the instruction-decode overhead and minimizing off-chip traffic, DSAs can boost energy efficiency by 10 - 100× compared with commodity CPUs or even GPUs [4]. Yet specialism is not monolithic—its design space spans tightly coupled on-die units that shave microseconds off real-time tasks, to loosely coupled cards that stream terabytes per second through dedicated DRAM. This chapter surveys that spectrum:

1) Video Coding Unit (VCU)

A warehouse-scale case study showing how ASIC hard macros, high-bandwidth NoCs, and multi-layer scheduling co-design can slash transcoding costs per stream.

2) Tightly- vs Loosely-Coupled Accelerators (TCA/LCA)

A taxonomy that contrasts latency-critical in-core engines with throughput-oriented DMA devices, and discusses when to choose LLC-DMA over DRAM-DMA.

4.1. VCU

First, let's see the VCU(Video Coding Unit). A Video Coding Unit (VCU) is a specialized hardware accelerator designed to overcome the challenges of large-scale video transcoding in data centers—namely heterogeneous device standards, real-time versus on-demand workflows, and stringent cost, throughput, and scalability requirements. Its specialization stems from dedicated ASIC cores for key encoding stages (motion estimation with 64 × 64 block search, rate-distortion optimization, CABAC/VLQ entropy coding, loop and temporal filtering, and frame-buffer compression), all interconnected via a high-bandwidth on-chip network (NoC) and DRAM interfaces on a PCIe-attached card. By offloading these compute-intensive kernels to a multilayer architecture (warehouse-scale scheduler → regional clusters → VCU-equipped servers), employing FIFO buffering to match pipeline stages, and using dual-voltage low-swing clock distribution to cut switching power, the

VCU delivers real-time or near-real-time multi-stream transcoding throughput that is up to an order of magnitude higher and a fraction of the energy consumption of conventional CPU/GPU solutions, thereby greatly improving system performance and reducing total cost of ownership. According to research by P. Ranganathan *et al.*, systems deploying VCU achieved a 20 to 33 times improvement in performance-to-power ratio compared to the then-optimized approach of transcoding using software on traditional servers. This means more video transcoding work was accomplished with fewer servers and lower power consumption, significantly reducing YouTube's operational costs [21].

4.2. TCA and LCA

Then, let's see the typical hardware accelerator that can be classified into two kinds: TCA(Tightly-coupled accelerator) and LCA(Loosely-coupled accelerator). Through the research of E. G. Cota *et al.*, we can understand the fundamental trade-offs between various "coupling" methods in terms of communication performance, design complexity, and application scenarios [22]. As for TCA, it is a specialized hardware unit that is physically and architecturally integrated with a host processor (e.g., CPU) to accelerate specific computational workloads, so it's designed in the CPU and cooperates with the CPU to tackle tasks. When the system needs to tackle specialized tasks like real-time encoding, the CPU can transform the task to the TCA because the TCA has a high performance on calculating some specialized tasks. TCA shares the information with the CPU and directly accesses CPU caches or registers, avoiding costly off-chip data transfers. Therefore, the TCA has a low delay, which can be used for AI inference, cryptography, and signal processing. However, the disadvantage of the TCA is the low throughput. When TCA shares the resources like a cache with the CPU, the data transfer limit is restricted (**Figure 5**).

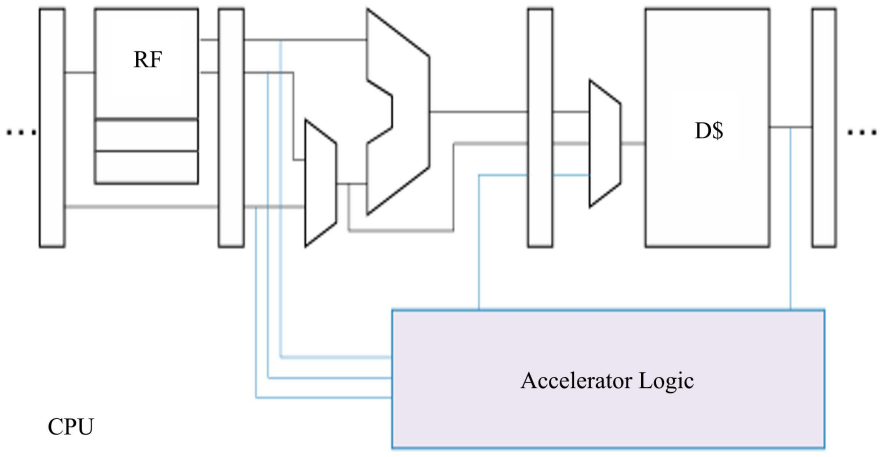


Figure 5. Tightly-coupled accelerator (TCA) model. The accelerator shares key resources (register le, MMU and L1) with the CPU [22].

As for LCA, it's not designed in the CPU, instead, it's a separate model in the

system, and it can be divided into two kinds: LLC-DMA and DRAM-DMA. The LLC-DMA means it has direct access to the last level of the cache, and the DRAM-DMA means it has direct memory access. Neither of them don't needs the CPU's permission to get the data. LCA has a high delay compared with TCA and high throughput. The key is that it has its memory. It can store the information in its memory and can use the data from the memory to tackle the tasks. In most situations, the limited throughput is because of the limited bandwidth. When the accelerator shares the memory sources with other models, it won't have a large bandwidth to get data, resulting in the consequence of low throughout. Instead, with its memory, the accelerator can use the whole storage to choose the data it needs and improve its performance.

Therefore, after knowing about the advantages and shortages of the TCA and LCA, we can design the appropriate accelerator to achieve our system. For instance, if the system needs to deal with some online tasks, the main goal is low delay, so we should use the TCA, like live video encoding. By contrast, if the systems need a high throughput, like calculating a Fourier transform, we'd better use the LCA. Then, we will pay attention to the difference between the LLC-DMA and DRAM-DMA (**Figure 6**). They both are the LCA, but they have different ways to get the data. The former one is using the LLC to store the data, and the latter one is using the DRAM to store the data. To begin with, we should know that the LLC has a faster speed and a higher efficiency. It's because LLC has faster lines of communication than the DRAM, and it's known to all that the main consumption power for a system is sharing the information with the memory. According to the paper "An analysis of accelerator coupling in heterogeneous architectures," when DRAM access is reduced by 80%, energy consumption is reduced by 3 - 5 times, and Gas pedal response time is reduced by 30% when the LLC hit rate is >90%. But does it mean that DRAM doesn't have any advantage? The answer surely is no. DRAM's storage capacity far exceeds that of LLC. Therefore, when it needs to tackle large data that LLC can't store, like 4K video stream processing, DRMA-DMA does a better job than the LLC-DMA, because when LLC frequently changes the information from the memory, it will cost more time and power, which is not the designer wants. To be specific, LLC-DMA (Last-Level Cache Direct Memory Access) optimizes data transfers between the CPU's last-level cache (LLC, e.g., L3 cache) and main memory using a DMA controller, bypassing CPU involvement to reduce latency and improve cache coherence in multi-core systems. It focuses on minimizing software overhead (e.g., interrupts) and is ideal for real-time tasks requiring low-latency cache-to-memory synchronization. DRAM-DMA LCAS (Line Cache Address Select), on the other hand, targets DRAM physical access efficiency by leveraging row buffer locality. It reduces row activation latency (tRAS/tRP) by reusing open DRAM rows (via address selection logic), maximizing sequential bandwidth for bulk data transfers (e.g., video streaming). While LLC-DMA enhances cache-memory synergy, DRAM-DMA LCAS optimizes raw DRAM throughput through hardware-level row management.

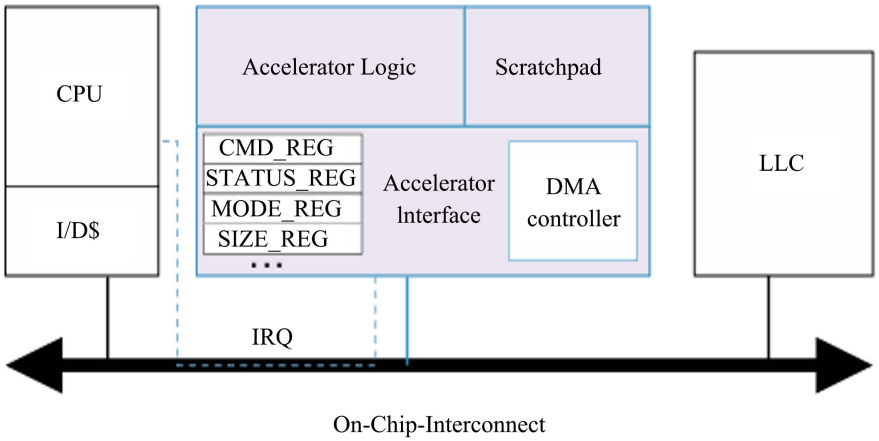


Figure 6. Loosely-coupled accelerator (LCA) model. The integrated DMA controller transfers data between the accelerators scratchpad and either the LLC or DRAM [22].

4.3. Conclusion of Specialism

This chapter examines how domain-specific accelerators break through general-purpose processor efficiency bottlenecks in the dark silicon era. The key insight is that modern workloads (video transcoding, deep learning, cryptography) exhibit predictable dataflow patterns that CPUs cannot efficiently exploit. Domain-specific accelerators achieve 10 - 100× energy efficiency improvements by customizing datapaths, memory hierarchies, and control logic tailored to algorithm hotspots, eliminating instruction-decode overhead and minimizing off-chip traffic. The VCU case study demonstrates how specialized ASICs achieve 20 - 33× performance-per-watt improvements through dedicated hardware macros, high-bandwidth NoCs, and multi-layer scheduling. Tightly-coupled accelerators (TCAs) integrate within CPUs, sharing caches for low latency but limited throughput; loosely-coupled accelerators (LCAs) operate as independent modules with dedicated memory, trading higher latency for greater throughput. LLC-DMA suits real-time tasks requiring fast cache synchronization, while DRAM-DMA excels at bulk data streaming. The price of specialization is inflexibility and rapid obsolescence as algorithms evolve.

5. Robust Communication Design

Modern digital integrated circuits are evolving towards high frequency and high density at an unprecedented rate. However, this progress has also exposed the inherent fragility of system communications. Transistor switching speed has been reduced to nanoseconds or faster, but the physical delay of on-chip interconnect (on-chip interconnect) has become a persistent performance bottleneck due to the RC effect. This “fast gate, slow line” contradiction, coupled with dynamic uncertainties such as process, voltage and temperature (PVT) fluctuations, signal crosstalk, and power supply noise, makes the traditional design approach of relying on a globally synchronized clock unsustainable, and the stable operation of the system faces a serious threat [23]. Therefore, the construction of a robust communi-

cation mechanism has become the core issue and primary goal of modern chip design. The so-called "robustness" refers to the ability of a system to maintain its functional correctness and data integrity in the face of timing variations and noise disturbances at the physical level as described above. In this chapter, we will discuss in depth the multilevel design strategies to achieve this goal. First, we will focus on Latency-Insensitive Design (LID), which radically copes with timing uncertainty, and analyze how it frees system functionality from dependence on precise delays by decoupling computation and communication. Subsequently, we extend our perspective to data interactions in multi-core architectures, illustrating how storage coherence and cache consistency protocols can serve as key techniques to guarantee robustness at the data level, ensuring that all communicating participants in parallel processing environments are provided with a correct and consistent view of the data. Through the discussion in this chapter, we will reveal how these design approaches work together to construct a robust defense system to ensure reliable communication in modern high-performance chips.

5.1. Latency Harms the Robustness

To begin with, we should know why the robustness collapses and then we can use specific ways to deal with it. The key to the problem is latency. According to the paper "Will physical scalability sabotage performance gains?", we know that with the Dennard scaling, the physical size will be scaled by a factor of k. The time constant RC will be amplified by factor $1/k$, because $R = \rho ls$, L is scaled by k, and S is scaled by k^2. Therefore, by intuition, the wire length should be reduced to k. However, we ignore the gate performance in this situation. The gate delay will also be improved, which leads to the occasion when the gate delay will not match the line delay. In conclusion, if the gate performance is improved by the factor w, and the Dennard scaling factor is k, then the length of the wire should be scaled by w^*k. It means that when we scale a design, the reachable area in one clock cycle will be deeply reduced, which is unfavorable to global clock synchronization. When the gate delay matches the line delay, we can deepen the pipeline to improve the clock cycle. However, it's limited, because in a clock cycle, it's must exist a gate at least. According to the paper, if the 0.6 μm process can accommodate about 25 gate delays per cycle, then the clock frequency can go up to 166 MHz; by the 0.06 μm process (again assuming 25 gate delays), the frequency can go up to 2.5 GHz. "How many gate delays per cycle" also determines how far a signal can travel in one beat (one clock cycle). As the process shrinks and the clock frequency increases, the distance the signal can travel per cycle becomes shorter. This results in "the whole chip being divided into multiple small logic islands", which further increases the complexity of the design by requiring constant insertion of registers for long-distance signal transmission at high frequencies. When the different signals' latency can't match each other, there may be a problem with the functionality of one of the modules. If we want to maintain the robustness of our system, we must find a solution.

5.2. Latency-Insensitive Design

The LID (Latency-Insensitive Design) was invented to handle this problem, and it plays an important role in all methods. LID is a digital circuit design methodology whose core principle is to enable reliable collaboration between circuit modules through a protocol-based communication mechanism, independent of specific signal propagation delays [24]. Even if the physical interconnect delays between modules are uncertain or dynamically varying (e.g., due to process variations, temperature fluctuations, or layout differences), the system can still guarantee functional correctness.

In traditional globally synchronous design, designers assume that all modules share a perfect, zero-delay global clock. Designers must ensure that the longest path delay between any two registers is less than one clock cycle. As chip size and frequency have increased dramatically, this assumption has long since collapsed. Long wire delays and process/voltage/temperature (PVT) variations have turned global timing closure into a nightmare. The core idea of LID is to abandon the obsession with global synchronization and instead adopt a "locally synchronous, globally asynchronous" approach. It views the system as composed of multiple independently synchronized "islands" (computational modules), with communication between these islands handled through a set of "bridges" (communication channels) that are independent of specific delays. As long as the timing converges within each "island," the overall system functionality will definitely be correct once they are connected together, regardless of how long the "bridges" are or how uncertain their delays may be. We can give a vivid example: when we get dressed after taking a shower, putting on underwear and pants forms one "island," putting on undershirt and shirt forms another "island," and putting on socks and shoes forms a third "island." The execution order within each island must be strictly maintained—for instance, you cannot put on shoes before socks. Meanwhile, the sequence between islands is fixed, but the intervals are flexible. For example, after putting on underwear and pants, you can immediately proceed to put on your undershirt and shirt, or you can take a break before continuing. Ultimately, this doesn't affect whether you end up properly dressed.

To achieve Latency-Insensitive Design (LID), we rely on two key components: "shells" and "stallable modules." This architecture uses shells that encapsulate modules to implement handshake protocols (such as "void-stop" signals), enforcing synchronization between modules and ensuring data integrity and ordered delivery despite arbitrary delays [25].

Consider modern processors like the A18 chip. A single core and cache cannot meet performance expectations due to insufficient parallelization and limited bandwidth. Therefore, the A18 incorporates multiple cores and numerous caches. However, this distributed design creates timing challenges where maintaining a global clock becomes problematic. LID addresses this by using stop and void signals to manage timing variations effectively.

The foundation of LID lies in the concept of latency equivalence. As long as two

data streams maintain the same sequence, they are considered equivalent regardless of timing differences. For example, "1 s, A, 1 s, B, 1 s, C" and "2 s, A, 1 s, B, 3 s, C" represent the same logical sequence—only the intervals between data elements differ. This equivalence principle ensures functional correctness independent of specific timing.

Stallable modules are the heart of LID implementation. A stallable module can be paused at any time and at any state, allowing the system to handle timing variations gracefully. Instead of trying to predict exact delays and control clocks precisely—which is nearly impossible in complex systems—we use control signals to manage data flow dynamically. Each stallable module is surrounded by a "shell" that acts as a control interface. **Figure 7** shows the detailed design of a shell. This shell monitors data flow and rectifies timing mismatches. For instance, when Module A sends data but Module B cannot receive it immediately, Module B sends a stop signal to A. Module A then pauses transmission and buffers the remaining data until Module B is ready. Consider a scenario where Module B has two inputs, but due to different delays, only one input is ready when the clock edge arrives. The shell inserts a "void" signal for the missing input while storing the valid data. The system waits until both inputs are available before proceeding. Since latency equivalence is preserved, temporarily stopping or buffering data doesn't affect the module's functional correctness.

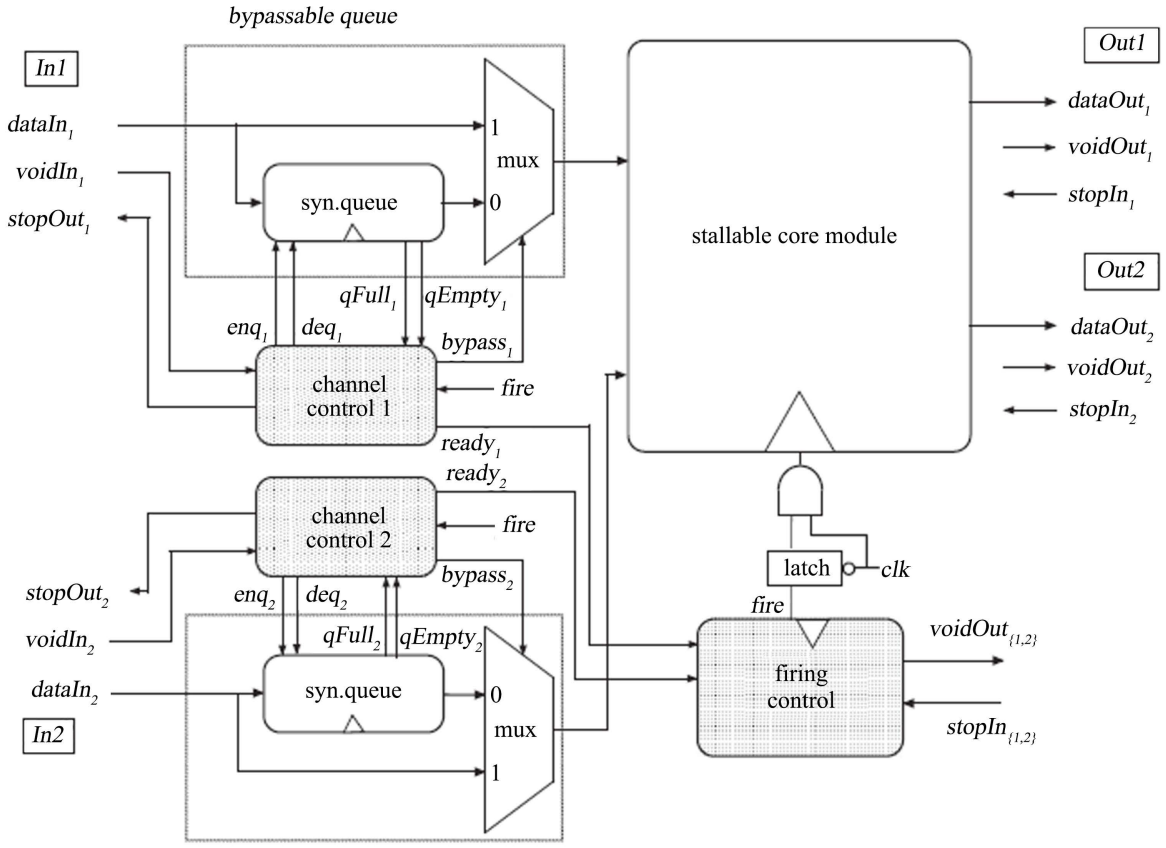


Figure 7. The details about how to design a shell [25].

While LID solves timing issues, it introduces throughput overhead. Since the system now transmits control signals alongside data, the percentage of effective data decreases. To mitigate this, LID incorporates channel pipelining. Channel pipelining divides long wires into multiple segments using sequential repeaters (relay stations). Each segment's delay is constrained within one clock period, while relay stations synchronize and buffer data flow following latency-insensitive protocols. Frist, by segmenting global connections and inserting relay stations, each segment's delay is limited to a single clock cycle, avoiding timing convergence problems. Second, relay stations act as "buffer agents," enabling automated pipeline insertion within the protocol framework without modifying core module logic. This adapts to high-frequency design requirements while responding to backpressure signals to pause upstream data flow when downstream blocking occurs. Third, LID uniquely allows adaptation to different process nodes or performance goals by simply adjusting the number of repeaters, making it highly flexible across various design contexts. This approach transforms the traditionally rigid globally synchronous design into a flexible, scalable architecture that maintains correctness while accommodating the realities of modern chip design.

5.3. Memory Consistency and Cache Coherence

Besides LID, assuring memory consistency and cache coherence is critical for stable communication in modern multiprocessor systems, yet these two concepts address fundamentally different aspects of shared memory correctness. Memory consistency defines the observable ordering constraints on memory operations across processors—essentially establishing a "contract" between hardware and software about which reorderings are permissible [26]. Unlike the simplified view of sequential execution, real processors aggressively reorder operations: a store buffer may delay writes, allowing subsequent loads to bypass them; speculative execution may fetch data before confirming branch outcomes; and out-of-order pipelines may complete younger loads before older stores to different addresses. The consistency model determines which of these optimizations remain invisible to programmers.

The spectrum of consistency models represents a fundamental tension between programmability and performance. Sequential Consistency (SC), the most intuitive model, mandates that all processors observe a single total order of operations matching program order—effectively serializing the entire system's memory accesses. However, SC's strict ordering prohibits critical optimizations: every load must stall until all prior stores complete globally, and no speculative reads can proceed past pending writes, resulting in 10% - 40% performance degradation on modern processors. Total Store Order (TSO), adopted by x86 architectures, relaxes SC by allowing loads to bypass earlier stores to different addresses through store-buffer forwarding, preserving most programmer intuitions while enabling crucial latency-hiding techniques. More aggressive models like Partial Store Order (PSO) and Relaxed Memory Order (RMO) permit extensive reordering—stores

to different locations may become visible out-of-order, and even causality can appear violated when processor A's write influences processor B's write, yet C observes B's update before A's. These relaxed models achieve performance through selective synchronization: the hardware provides memory fence instructions (e.g., MFENCE, LFENCE, SFENCE on x86, or DMB, DSB on ARM) that act as reordering barriers. A full fence forces all prior operations to complete globally before subsequent ones begin, while specialized fences target specific operation types—a store-load fence prevents loads from bypassing stores, while a load-load fence maintains read ordering. The programmer or compiler strategically places these fences at synchronization points: acquiring a lock requires a load-acquire fence to prevent critical section operations from moving before the lock check, while releasing needs a store-release fence to ensure updates complete before unlock. This creates a two-tier memory system: within fence-delimited regions, aggressive reordering maximizes instruction-level parallelism, while fence boundaries establish globally observable synchronization points where all processors agree on memory state.

Cache coherence, operating at a lower abstraction level, ensures that multiple private caches holding copies of the same memory line converge to a consistent value when any processor modifies that line [4]. While memory consistency governs the ordering of operations across different addresses, coherence maintains the illusion of a single value per address despite distributed caching. The coherence protocol must enforce two invariants: write propagation ensures that updates eventually reach all caches (preventing indefinite staleness), and write serialization guarantees all processors observe writes to the same location in identical order (preventing conflicting views of history). Modern coherence protocols implement these invariants through carefully orchestrated state machines. The MESI protocol (Modified, Exclusive, Shared, Invalid) optimizes beyond basic MSI by distinguishing between exclusive-clean and exclusive-dirty states: a cache line in Exclusive state can transition to Modified without bus traffic, eliminating unnecessary invalidations for private data. Directory-based protocols further optimize by tracking sharers precisely—instead of broadcasting invalidations to all processors, the directory maintains a bit-vector or limited pointer list identifying exactly which caches hold copies [27]. This reduces interconnect traffic from $O(P^2)$ to $O(P)$ for P processors, crucial for scaling beyond 8 - 16 cores. Advanced protocols like MOESI add an Owned state where one cache serves other sharers without writing back to memory, reducing memory bandwidth pressure by 20% - 30% in producer-consumer patterns.

The interaction between consistency and coherence introduces subtle complexities. While coherence ensures eventual visibility of writes, the consistency model determines when that visibility becomes architecturally guaranteed. Under TSO, even after coherence propagates a store to all caches, local load reordering might still observe stale values from the store buffer. Conversely, a strict consistency model cannot compensate for slow coherence—if invalidation messages are de-

layed in the network, even SC cannot make writes instantly visible. This interplay necessitates careful co-design: modern systems often implement eager release consistency, where cache line invalidations carry fence semantics, or transactional memory extensions that provide both atomicity and ordering guarantees through coherence protocol modifications. Furthermore, emerging architectures challenge traditional boundaries. Non-uniform cache architectures (NUCA) introduce variable latencies based on physical distance, making the coherence protocol timing-dependent. Heterogeneous systems mixing CPUs, GPUs, and accelerators must reconcile different consistency models—GPUs typically implement weaker models than CPUs, requiring explicit synchronization at boundaries. Persistent memory adds durability requirements, where consistency must extend beyond volatile caches to include persist barriers that ensure crash consistency.

In summary, memory consistency and cache coherence form complementary layers in the parallel memory hierarchy: consistency defines the legal software-visible orderings while coherence provides the hardware mechanisms for maintaining per-location consistency. Their co-design involves navigating trade-offs between programmer productivity (favoring stronger models), performance (favoring weaker models with selective synchronization), scalability (favoring directory-based coherence), and complexity (affecting verification and debugging). Together, they enable the illusion of shared memory while permitting the aggressive optimizations necessary for modern multi-core performance.

5.4. Communication on Chips

As Moore's Law-driven transistor integration continues to approach its physical limits, the "power wall" problem has put an end to the traditional paradigm of relying solely on increasing processor clock frequency to achieve performance gains. To continue the pace of performance improvement, the semiconductor industry has turned to parallel computing by integrating a large number of processing cores on a single chip, giving rise to complex multi-core systems-on-chip (SoCs). However, this high degree of integration makes data exchange between cores a new bottleneck. The traditional bus architecture is not scalable due to its shared and serial characteristics; the point-to-point connection has superior performance, but its wiring complexity and area overhead grow exponentially with the number of cores, both of which are unable to satisfy the needs of the many-core era. Therefore, Network-on-Chip (NoC), which draws on the idea of macro-networks, has emerged as a key technology for modern high-performance SoC design with its excellent scalability, parallel processing capability, and high energy efficiency. NoC design is not only a matter of architecture selection, but also a systematic engineering challenge involving traffic modeling, topology, routing algorithms, QoS guarantee, functional verification and physical implementation, etc. [28]. The direction of its development is evolving towards the support of cutting-edge technologies such as DSA and Chiplet.

In this part, we will discuss a kind of effective communication architecture:

NoC, and compare it with P2P and Bus-based communication. The Network-on-Chip (NoC) architecture organizes IP cores into a structured network interconnected via routers and shared links. Data is transmitted as packets through predefined paths, eliminating the need for dedicated global wires [29]. This modular design enables scalability, reusability, and predictable electrical characteristics. For instance, routers and network interfaces can be standardized and reused across multiple designs, significantly reducing development effort. In contrast, Point-to-Point (P2P) architectures rely on dedicated communication channels between each pair of IP cores. While this approach minimizes contention and maximizes bandwidth utilization, it introduces significant area and energy overhead due to redundant links. For example, an MPEG-2 encoder implemented with P2P requires ten dedicated links, most of which remain idle during operation, resulting in an average utilization of just 4%. Finally, Bus-Based architectures, the simplest of the three, connect all IP cores through a single shared channel managed by arbitration logic. While this reduces wiring complexity and area overhead for small-scale systems, the shared medium becomes a bottleneck as the number of cores increases. Arbitration delays and fixed bandwidth limit performance, make buses unsuitable for high-throughput applications like real-time video encoding. Then, I will talk about the difference between the three kinds of communication, especially on the aspects of efficiency, area, and performance and recovery capabilities. The first one is about performance and scalability. Performance metrics, particularly throughput and latency, reveal stark differences among the architectures. The P2P implementation achieves the highest throughput due to its contention-free dedicated links. Latency is minimized, as data transfer time depends solely on link bandwidth W:

$$L_{Data}^{P2P} = \frac{N_{Data}}{W}$$

However, this performance comes at a cost: scaling the design by adding cores requires exponential increases in links and interfaces. For instance, duplicating the motion estimation (ME) module in the MPEG-2 encoder necessitates four additional links and eight new network interfaces in the P2P architecture. The NoC implementation, while slightly slower, matches P2P in scalability. By employing wormhole routing and modular routers, NoC avoids the area explosion seen in P2P. Communication latency in NoC depends on hop count H and header processing time LR:

$$L_{Data}^{NoC} = H \cdot L_R + \frac{N_{Data} - W}{\rho_{NoC} W}$$

When the ME module is replicated, NoC achieves a 93% throughput improvement, nearly matching P2P's 88% gain but with far less area overhead. The Bus-Based architecture performs poorly in both throughput and scalability. Arbitration delays LR and bandwidth contention degrades performance as cores are added:

$$L_{Data}^{Bus} = L_R + \frac{N_{Data}}{\rho_{Bus} W}$$

For example, adding a second ME module to the bus design yields minimal throughput gains due to persistent contention. The second one is the area efficiency and interconnect complexity. Area overhead is a critical factor in SoC design. The P2P architecture, with its redundant links and interfaces, occupies 24.7% more area than NoC in an 8-ME implementation. Each additional core in P2P requires redesigning interfaces and adding dedicated links, leading to quadratic growth in wiring complexity. NoC's structured network reduces interconnect complexity. Routers occupy a modest area (e.g., 161 slices per router), and shared links minimize redundant wiring. While NoC requires 4.6% more area than the Bus architecture, its scalability justifies this trade-off. For instance, integrating eight ME modules in NoC increases area linearly, whereas P2P's area grows exponentially. The Bus architecture excels in area efficiency for small designs, requiring only 67 slices for its control unit. Beyond basic metrics, architectural differences manifest in implementation complexity. Bus architectures require sophisticated arbitration mechanisms—priority encoders, round-robin schedulers, and fairness counters—that become critical path bottlenecks at high frequencies. Modern buses like ARM's AXI employ split-transaction protocols and out-of-order completion to mitigate this, adding 30% - 40% controller complexity. P2P architectures face routing congestion challenges: in a 16-core system, the crossbar switch alone requires 256 multiplexers (16 × 16) with each multiplexer having 16:1 selection ratio, consuming more area than all processing cores combined. Physical implementation requires 6 - 8 metal layers just for routing, increasing manufacturing cost by 25%. NoC routers, while complex individually, benefit from regularity and locality. A typical 5-port virtual-channel router with 4 VCs per port, 4-flit buffers per VC, and dimension-ordered routing occupies 0.05 mm^2 in 14 nm technology. The regular mesh structure enables automated place-and-route with predictable timing closure—critical for achieving 5 GHz + operation frequencies. Advanced features like adaptive routing and quality-of-service add 15% - 20% area but improve worst-case latency by 40%. However, its simplicity becomes a liability in larger systems, as congestion and arbitration overhead negate area advantages. The third one is energy consumption and power dynamics. Energy efficiency is paramount for battery-powered devices. The NoC implementation consumes the least energy per frame (37.6 μJ for 1 ME, 23.4 μJ for 2 ME), owing to its optimized communication pathways and reduced static power. Communication energy in NoC constitutes only 19% of total consumption, even in scaled designs. P2P's energy consumption is significantly higher (42.8 μJ/frame for 1 ME) due to its numerous active interfaces and underutilized links. Communication components account for 34% of total power, a figure that worsens with scaling. The Bus architecture's energy profile is dominated by prolonged encoding times. Despite lower static power, its 41.2 μJ/frame consumption for 1 ME reflects inefficiencies in shared resource management. In conclusion, network-on-chip architectures rep-

resent the future of scalable SoC design. The final distinction lies in failure modes and resilience. Bus architectures present a single point of failure—bus controller failure disables all communication. However, their simplicity enables comprehensive formal verification and built-in self-test (BIST) with 99.9% fault coverage. P2P's dedicated links provide natural redundancy for unaffected paths, but a single link failure completely isolates the connected pair. Implementing spare links for fault tolerance increases already-prohibitive wiring overhead by 20% - 30%. NoC's path diversity enables graceful degradation: dimension-ordered routing can automatically route around failed links/routers with 10% - 15% performance impact. Error-correcting codes (ECC) on links and replay mechanisms in routers achieve $<10^{-15}$ bit error rates. For safety-critical applications, NoC's fault containment—where router failures affect only local traffic—provides superior isolation compared to bus's global impact or P2P's binary connectivity. By offering near-P2P performance, Bus-like area efficiency, and superior energy scalability, NoC addresses the limitations of traditional approaches. As SoC complexity grows, NoC's modularity and efficiency will cement its role as the cornerstone of next-generation communication frameworks.

Despite NoC's excellent performance in scalability and energy efficiency, its implementation brings significant design challenges. First is the area cost issue of routers. Each NoC router must integrate complex functional units, including input buffers, switching matrices, routing computation units, and arbitration logic. Taking a typical 5-port router as an example, its area may consume hundreds of logic gate units. In large-scale multi-core systems, the cumulative area overhead of dozens or even hundreds of routers can reach 10% - 15% of the total chip area. More critically, to guarantee performance, routers typically require deep buffer configurations to handle network congestion, and these SRAM buffers further increase area and power costs. Second is the software programming burden imposed by weak memory models. The distributed nature of NoC and packetized transmission make memory consistency complex—accesses to the same memory location from different cores may arrive out of order due to varying network delays and routing paths. This weak memory consistency model requires programmers to explicitly use memory barriers, atomic operations, and synchronization primitives to ensure program correctness, significantly increasing the complexity of parallel programming. In contrast, traditional bus architectures naturally provide strong memory consistency due to their serialized characteristics, allowing programmers to ignore these low-level details. Therefore, while NoC provides excellent performance and scalability at the hardware level, the maturity and programming friendliness of its software ecosystem remain important issues that the industry needs to continuously address and improve.

Nevertheless, the choice between Bus, P2P, and NoC architectures should be driven by specific application requirements and system constraints. For real-time embedded systems with 2 - 4 cores and predictable traffic patterns, bus architectures provide the optimal solution with minimal overhead—for instance, automo-

tive ECUs achieve deterministic 10 μs response times using simple AMBA buses. For high-performance computing requiring ultra-low latency between specific core pairs, hybrid approaches work best: critical paths use P2P connections (achieving sub-2-cycle latency) while general communication uses NoC. GPU architectures exemplify this—texture units connect to caches via dedicated P2P links for 1-cycle access, while shader cores communicate through a packet-switched NoC tolerating 10 - 15 cycle latencies. For scalable many-core systems (>16 cores) with dynamic workloads, NoC becomes essential. Consider Intel's Mesh Interconnect in Xeon processors: a 28-core configuration would require 378 P2P links (consuming ~40% chip area) versus a 2D mesh NoC using only 112 bidirectional links (consuming ~8% area) while achieving 85% of P2P's theoretical bandwidth through spatial locality optimization.

5.5. Conclusion of Robust Communication Design

This chapter addresses severe communication robustness challenges in high-frequency, high-density chips. The core problem is the "fast gates, slow wires" contradiction: transistor switching reaches nanosecond speeds while on-chip interconnects become performance bottlenecks due to RC effects, compounded by PVT fluctuations making global synchronous clock design untenable. Latency-Insensitive Design (LID) adopts a "locally synchronous, globally asynchronous" approach, using shells and stallable modules to implement protocol-based handshake communication, ensuring functional correctness despite arbitrary delays. Memory consistency defines observable ordering constraints for memory operations across processors, ranging from strict Sequential Consistency (SC) to relaxed TSO/PSO/RMO models, balancing programmability and performance through selective synchronization fences. Cache coherence protocols like MESI ensure distributed caches maintain consistent views of shared memory locations. Network-on-Chip (NoC) demonstrates superior scalability beyond 16 cores compared to bus and point-to-point architectures: linear area growth, 85% of P2P bandwidth, and excellent fault tolerance, establishing it as the key technology for the many-core era.

6. Relevant Problems in Digital Chips' Design

The semiconductor industry is facing a profound transformation in a new era defined by artificial intelligence and data-intensive computing. On the one hand, the never-ending demand for computing power is in fierce conflict with a long-standing physical bottleneck, the "memory wall." This problem, caused by the growing performance gap between processors and memory, has become a key barrier limiting the performance of all systems, from edge devices to supercomputers. On the other hand, to break through this "wall," the industry is turning to disruptive system architectures such as chiplets and heterogeneous integration. However, while these new architectures unleash tremendous potential, they also bring unprecedented design complexity. In order to navigate this complexity, designing from a

higher level of abstraction is no longer an option, but a necessity. This summary explores these two interrelated core topics: first, we will analyze the serious performance challenge of the "memory wall" and its modern solutions; second, we will introduce advanced design methodologies represented by SystemC and High-Level Synthesis (HLS), which are key tools for engineers to build the complex chips of the future. The next step is to introduce advanced design methodologies represented by SystemC and High-Level Synthesis (HLS), which are key tools for engineers to build complex chips of the future.

6.1. Memory Wall

At last, let's see the memory wall that seriously harms our system performance. What's the memory wall? The Memory Wall is a critical concept in computer system design, referring to the growing gap between processor speed and memory speed. This gap has become increasingly significant in modern computer systems and serves as a primary bottleneck limiting overall performance improvements. The root cause of the memory wall is the mismatch between processor performance and the rate of growth of memory performance. When the processor performance is increased by a factor of two, the memory performance is increased by a factor of about 1.2. Why does it happen? According to the paper "Latency lags bandwidth", there are six reasons to explain the phenomenon [3]. Firstly, Moore's Law favors bandwidth over latency. Moore's Law drives an increase in transistor density, allowing more transistors to fit on a single chip. This increase in density not only allows for more parallel processing operations but also supports higher bandwidth through increased pin counts. Although faster transistors help reduce latency, the increase in chip size and the growth in the length of wires on the chip limit the latency improvement. The second reason is that the distance limits delay. The speed of propagation of signals in wires is limited by the speed of light, and the improvement in latency is extremely limited, especially over long distances. For instance, due to the laws of physics, the theoretical minimum delay for a signal traveling 300 meters is 1 microsecond. In a real copper or fiber optic link, this delay usually exceeds 1.4 microseconds, taking into account the effect of the medium on the signal speed. Therefore, latency can be a big problem when we need to communicate over long distances. Moreover, the bandwidth is easier to sell than the latency. Improvements in bandwidth are easier for users to understand and accept, and are better marketed. For example, 10 Gbps of Ethernet bandwidth is easier to impress customers than 10 microseconds of latency. At the same time, improvements in latency help bandwidth, but not vice versa. It's easy to understand when the latency is low, we can get more information at the same time, which leads to indirectly improving the bandwidth. However, when the bandwidth improves, it's through more bandwidth that we can get more information, in this situation where the latency doesn't change at all. Improvements in bandwidth do not directly improve latency, and this unidirectional relationship exacerbates the lag in latency performance. In addition, bandwidth gains may come at

the cost of latency. For instance, queuing theory shows that an increase in buffers contributes to bandwidth, but may lengthen task wait time, which means that the latency will be impaired. Last but not least, operating system overhead has a greater impact on latency. Overheads at the operating system and driver layers account for a much higher percentage of latency performance, especially when short messages are processed. Therefore, how can we specialize our components to solve this question? The first method is to leverage capacity to help latency. How does it work? CPU execution is very fast and memory (DRAM) is relatively much slower to access; if memory is accessed directly every time, it will cause the processor to idle and wait too long. The cache is like a "buffer": frequently used data is "kept" in the cache, so that the next time you access the same data, you don't have to access the slower main memory or the disk, thus significantly reducing access times. It's based on the premise of the locality of reference, which consists of two parts. The one is temporal locality, programs will access the same piece of data multiple times in a short period. And the other one is spatial locality, programs usually access data that is close to the current data address. When the system satisfies the two principles, using the cache can help reduce the latency. The second method is to leverage capacity to again help latency. The goal is similar to the previously discussed caching: to shorten access paths and reduce waits, but using "replication", like keeping multiple copies of the data at the same time. When there is only one copy of the data, all accesses can only be made to this original data; if this data is geographically distant (network latency) or slow to be accessed on physical hardware, this can lead to longer wait times. If data is replicated in multiple locations, the system can select the "closest copy" to serve, thereby shortening access paths and reducing latency. The last method is to leverage bandwidth to again help latency [3] [30]. The core of this method is prediction. When the system waits for the arrival of data, this time is wasted, instead, the system can predict what the data will be or what the instruction will be executed, and then it can work in this time after the real data arrives, the system can judge the prediction whether is right or not. If it's right, the system can go on and save the latency time. On the contrary, if it's a fault, the system will use the real data to repeat. Therefore, this method needs a relatively high performance of correct prediction.

6.2. SystemC

Previously, technologists always used VHDL or Verilog HDL to design the chip's gate-level circuits, and then, when they used the completed chip to test and found the chip was incorrect, they needed to design again and test again. It's time-consuming, and it also wastes a lot of cost. However, soon after, SystemC was invented, it was really helpful for the designer to create the chip practically. Why can SystemC have such a difference? SystemC is a C++-based modeling language and simulation framework designed for describing and verifying hardware/software systems at a higher level of abstraction [31]. It offers a more flexible approach than traditional RTL (Register-Transfer Level) by allowing developers to describe

system behavior using processes (SC_METHOD, SC_THREAD, SC_CTHREAD) and modules, and to handle events, timing, concurrency, and communication interfaces more conveniently. Because it supports a range of modeling—from the algorithmic level down to the register-transfer level—SystemC enables fast functional validation and performance evaluation at the system architecture level, which is why it is called a system-level design tool. In contrast to traditional Verilog/VHDL, SystemC not only describes precise signal-level behavior but also provides Transaction-Level Modeling (TLM) for faster system simulation and easier hardware-software co-design [32]. Thus, it helps designers perform architectural exploration, functional verification, and performance analysis before completing the RTL implementation of a chip or system, allowing crucial design decisions to be made at the system level. This is the core meaning of its designation as a "system-level" tool. To be contrasted with Verilog and VHDL, the best improvement of SystemC is that we can directly use many algorithms to achieve our goals with C + + , but not use Verilog or VHDL to control every signal or flip-flop, etc. to achieve the algorithm and then indirectly achieve the same function, because the synthesis tool – catapult will help us to translate the systemC into the RTL circuit. What we should do is try our best to use any algorithm to improve our system. After learning about the basic knowledge of SystemC, the next thing is to understand why we need to use TLM, and how it works to improve efficiency. So firstly, what is the TLM?TLM (Transaction-Level Modeling) is a methodology for modeling and simulating digital systems at a higher level of abstraction. Unlike traditional RTL (Register-Transfer Level) modeling, TLM treats a complete "data transfer" or "operation" as a single transaction and passes it between modules via function or interface calls, rather than simulating every cycle of each signal. This approach significantly reduces the number of events and process switches required in the simulation, thus improving efficiency. The main idea of the TLM is that we use a transaction instead of controlling hundreds of signals. For instance, when a model produces data and the next model needs to use it, in Verilog or VHDL, we need to change the signal to transport the data, instead, when we use the TLM, we can use the function to transport the information, use interface pointer to access the data, reduce the time consuming by transporting. In conclusion, the advantages are obvious. By reducing signal-level triggers and scheduling overhead, you can conduct rapid system-level validation. By focusing on complete data or command transfers, the complexity of design and verification is greatly simplified. Even before the RTL design is finalized, a TLM-based virtual prototype can be used for functional verification and software development. Because system behavior is described at an abstract level, it is easier for software and hardware teams to collaborate and verify designs in the same environment. At last, we will talk about the Catapult (a kind of software), how does this tool work to optimize our program? During the synthesis process, catapult first parses and analyzes the high-level code, identifying loops, function calls, and data dependencies. Then, based on user-defined constraints such as target clock frequency and resource lim-

its, Catapult performs scheduling and resource binding, assigning hardware operators (like adders and multipliers) to each operation and determining the timing within clock cycles. The tool also applies loop unrolling and pipelining optimizations to enable parallel or staged execution in hardware, improving throughput or saving resources. Additionally, Catapult provides design space exploration (DSE) features, which automatically or semi-automatically try different levels of parallelization, timing constraints, and resource-sharing strategies to balance performance, area, and power, eventually generating an optimal hardware implementation. The final output is standard RTL that downstream synthesis and place-and-route tools can process, with results cross-validated against the original high-level simulations [33].

6.3. The Impact on the Economy and the Environment

First is the economic impact. Modern SoC development costs have reached $500 million for 7 nm nodes and $1.5 billion for 3 nm, a 100× increase since 2000. Only products with >100 million unit volumes can amortize these costs, creating unprecedented market concentration. The top 5 semiconductor companies now control 77% of advanced node capacity, compared to 35% in 2000. This oligopolization raises concerns about innovation bottlenecks and supply chain vulnerabilities—as evidenced by the 2020-2023 chip shortage that cost the automotive industry $210 billion. What's more, The shift from hardware scaling to architectural innovation demands different expertise. Traditional circuit designers face obsolescence while parallel programming specialists command $200,000+ salaries. Universities report 40% unfilled positions in computer architecture programs while industry demands grow 15% annually. This skill gap costs the global economy an estimated $450 billion in unrealized productivity gains. In addtion, Advanced chip design capability has become a national security priority. The US CHIPS Act ($280 billion), EU Chips Act (€43 billion), and China's semiconductor fund ($150 billion) represent unprecedented peacetime industrial interventions. Technology export restrictions on EUV lithography and advanced design tools have fragmented the global supply chain, potentially adding 20% - 30% to chip costs through duplicated infrastructure.

Second is the impact on the environment. The semiconductor industry's environmental footprint extends beyond operational power consumption to encompass the entire lifecycle. Producing a single 300 mm wafer requires 2000 gallons of ultra-pure water, 20 kWh of electricity, and generates 2 kg of hazardous waste. Taiwan region's TSMC alone consumes as much water as a city of 1.8 million people. Advanced nodes exacerbate this—3 nm fabrication uses 50% more water and 35% more energy than 10 nm. With global chip production doubling every 7 years, semiconductor manufacturing will consume 3% of global electricity by 2030. Moreover, The pursuit of specialization creates an e-waste tsunami. Data centers replace GPU accelerators every 2 - 3 years, generating 2.5 million tons of electronic waste annually. Only 17% of chip materials are recyclable due to toxic

dopants and complex packaging. Rare earth element extraction for semiconductors devastates ecosystems—producing 1 ton of rare earth oxides generates 2000 tons of toxic waste and 1000 tons of wastewater containing radioactive thorium. While efficient chips reduce operational emissions, their manufacturing carbon footprint often exceeds lifetime operational savings. A 7 nm chip's production emits 100 kg CO_2-equivalent—requiring 3 - 4 years of operation to offset through efficiency gains. For specialized accelerators replaced every 2 years, the carbon payback never occurs, making them net carbon-negative despite superior efficiency.

7. Future Prospect

The future of digital chip design extends far beyond incremental improvements to existing architectures, converging toward three revolutionary paradigms that fundamentally reimagine computation. Quantum computing, currently demonstrated in systems like IBM's 433-qubit Osprey and Google's 70-qubit Sycamore, exploits quantum superposition and entanglement to achieve exponential speedup for specific problems—Google's random circuit sampling took 200 seconds versus an estimated 10,000 years on classical systems—though practical applications remain limited by decoherence times of mere microseconds and error rates requiring 1000 - 10,000 physical qubits per logical qubit. Neuromorphic computing mimics the brain's extraordinary efficiency (20 W for human cognition versus megawatts for inferior AI), with chips like Intel's Loihi 2 implementing event-driven computation through 1 million neurons and 120 million synapses, while emerging memristor and phase-change memory technologies enable in-memory computing with 1000× better energy efficiency than GPUs for inference tasks. Chiplet architectures revolutionize system economics by disaggregating monolithic SoCs into modular dies—AMD's EPYC Rome combines eight 7 nm compute chiplets with a 14 nm I/O die, reducing costs by 50% while enabling unprecedented heterogeneous integration, as demonstrated by Intel's Ponte Vecchio GPU combining 47 chiplets across five process nodes. These paradigms will synergistically converge: quantum-classical hybrid systems will use neuromorphic processors for real-time quantum error correction, chiplet technology will enable packaging of cryo-CMOS quantum control circuits alongside classical processors, and future heterogeneous platforms might integrate CPU, GPU, neuromorphic, and quantum control chiplets within single packages connected by silicon photonic interconnects providing terabit-per-second bandwidth. While challenges remain—quantum decoherence, neuromorphic programming models, chiplet standardization—this convergence promises to transcend the fundamental limits of silicon and von Neumann architectures, ushering in an era where computing adapts to problems rather than problems adapting to computing, fundamentally transforming how we approach challenges from drug discovery to artificial intelligence.

8. Conclusion

In summary, the digital chip industry is at a profound and exciting turning point.

The slowdown of Dennard's and Moore's Laws did not signal the end of computing, but rather the birth of a dramatic architectural revolution that ushered in the "new golden age" predicted by Hennessy and Patterson. Two fundamental challenges, the power wall and the memory wall, forced the industry to leave the single growth path of transistor miniaturization and explore a more diverse and complex innovation space. This exploration process shows a clear evolutionary lineage: from the initial parallel computing strategy to deal with frequency bottlenecks, to the emergence of domain-specific architectures (DSAs) to overcome the diminishing returns of parallelization, and to aim at extreme efficiency; from the physical realization level, to break through the economic and technological limits of the monolithic chip born of the chiplet and the heterogeneous integration revolution, to the development of high-level design methodologies to manage the unprecedented complexity of the system; from the physical realization level, Chiplet and heterogeneous integration revolution to break through the economic and technological limits of single chip, to the development of high-level design methodology and hardware-software co-design to manage the unprecedented system complexity. These architectural, physical, and methodological innovations do not exist in isolation, but are interconnected and mutually reinforcing, and together they have created a new paradigm for computing in the post-Moore's Law era. This new paradigm, with modularity, heterogeneity, and specialization as its core features, brings higher design complexity but also opens up unprecedentedly broad prospects for performance, energy efficiency, and security enhancement, marking the arrival of a truly colorful renaissance in computer architecture.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Moore, G.E. (1965) Cramming More Components onto Integrated Circuits. *Electronics*, **38**, 114-117.

[2] Dennard, R.H., Gaensslen, F.H., Yu, H., Rideout, V.L., Bassous, E. and LeBlanc, A.R. (1974) Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, **9**, 256-268. https://doi.org/10.1109/jssc.1974.1050511

[3] Patterson, D.A. (2004) Latency Lags Bandwidth. *Communications of the ACM*, **47**, 71-75.
Horowitz, M. (2014) 1.1 Computing's Energy Problem (and What We Can Do about It). 2014 *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (*ISSCC*), San Francisco, 9-13 February 2014, 10-14.

[4] Hennessy, J.L. and Patterson, D.A. (2017) Computer Architecture: A Quantitative Approach. 6th Edition, Morgan Kaufmann.

[5] Jones, D.W. (2004) Operand Forwarding and Out-of-Order Execution in Pipelined Architectures. University of Iowa, Department of Computer Science, Technical Report.

[6] McFarling, S. (1993) Combining Branch Predictors. WRL Technical Note TN-36,

Digital Western Research Laboratory.

[7] Tanenbaum, A.S. and Bos, H. (2014) Modern Operating Systems. 4th Edition, Pearson.

[8] Kumar, A., *et al.* (2013) Single-Chip-Multiprocessor: A Multi-Core Processor For Embedded Systems. *International Journal of Advanced Research in Computer Science and Software Engineering*, **3**, 1-27.

[9] Wikipedia: ARM Big. LITTLE. Wikimedia Foundation.

[10] ARM Limited (2013) Big. LITTLE Technology: The Future of Mobile. White Paper.

[11] Marr, D.T., *et al.* (2002) Hyper-Threading Technology: Architecture and Microarchitecture. *Intel Technology Journal*, **6**, 4-15.

[12] Tullsen, D.M., Eggers, S.J. and Levy, H.M. (1995) Simultaneous Multithreading: Maximizing on Chip Parallelism. *Proceedings of the* 22*nd Annual International Symposium on Computer Architecture—ISCA*'95, Santa Margherita Ligure, 22-24 June 1995, 392-403. https://doi.org/10.1145/223982.224449

[13] Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L. and Stamm, R.L. (1996) Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. *ACM SIGARCH Computer Architecture News*, **24**, 191-202. https://doi.org/10.1145/232974.232993

[14] Amdahl, G.M. (1967) Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *Proceedings of the April* 18-20, 1967, *Spring Joint Computer Conference*, Atlantic, 18-20 April 1967, 483-485. https://doi.org/10.1145/1465482.1465560

[15] De, V. and Borkar, S. (1999) Technology and Design Challenges for Low Power and High Performance. *Proceedings of the* 1999 *International Symposium on Low Power Electronics and Design*, San Diego, 16-17 August 1999, 163-168. https://doi.org/10.1145/313817.313908

[16] Ohashi, M., Hashimoto, T., Kuromaru, S.I., Matsuo, M., Mori-iwa, T., Hamada, M., *et al.* (2002) A 27 MHz 11.1 mW MPEG-4 Video Decoder LSI for Mobile Application. 2002 *IEEE International Solid-State Circuits Conference. Digest of Technical Papers* (*Cat. No.*02*CH*37315), San Francisco, 7 February 2002, 366-474 https://doi.org/10.1109/isscc.2002.993084

[17] Krishnamurthy, R.K., *et al.* (2002) Dual Supply Voltage Clocking for 5 GHz 130 nm Integer Execution Core. *Proceeding of Symposium on VLSI Circuits*, 13-15 June 2002, 172-175.

[18] Cai, J., *et al.* (2002) Supply Voltage Strategies for Minimizing the Power of CMOS Processors. *Proceeding of Symposium on VLSI Circuits*, 11-13 June 2002, 154-155.

[19] Kathuria, J., Ayoubkhan, M. and Noor, A. (2011) A Review of Clock Gating Techniques. *MIT International Journal of Electronics and Communication Engineering*, **1**, 106-114.

[20] The Linux Kernel Archives (2003) CPU Frequency and Voltage Scaling. Kernel.org documentation.

[21] Ranganathan, P., *et al.* (2021) Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild. *Proceeding of ASPLOS*, 2021, 12-23 April 2021, 600-615.

[22] Cota, E.G., Mantovani, P., Di Guglielmo, G. and Carloni, L.P. (2015) An Analysis of Accelerator Coupling in Heterogeneous Architectures. *Proceedings of the* 52*nd Annual Design Automation Conference*, San Francisco, 7-11 June 2015, 1-6. https://doi.org/10.1145/2744769.2744794

[23] Ho, R., Mai, K.W. and Horowitz, M.A. (2001) The Future of Wires. *Proceedings of*

the IEEE, **89**, 490-504. https://doi.org/10.1109/5.920580

[24] Carloni, L.P., McMillan, K.L. and Sangiovanni-Vincentelli, A.L. (2001) Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **20**, 1059-1076. https://doi.org/10.1109/43.945302

[25] Carloni, L.P. (2015) From Latency-Insensitive Design to Communication-Based System-Level Design. *Proceedings of the IEEE*, **103**, 2133-2151. https://doi.org/10.1109/jproc.2015.2480849

[26] Adve, S.V. and Gharachorloo, K. (1996) Shared Memory Consistency Models: A Tutorial. *Computer*, **29**, 66-76. https://doi.org/10.1109/2.546611

[27] Culler, D. and Singh, J.P. (1999) Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann.

[28] Wu, Y., *et al.* (2017) A Survey of Routing Algorithms for Mesh Network-On-Chip. *Frontiers of Computer Science*, **11**, 16-30.

[29] Lee, H.G., Chang, N., Ogras, U.Y. and Marculescu, R. (2007) On-chip Communication Architecture Exploration: A Quantitative Evaluation of Point-to-Point, Bus, and Network-On-Chip Approaches. *ACM Transactions on Design Automation of Electronic Systems*, **12**, 1-20. https://doi.org/10.1145/1255456.1255460

[30] Przybylski, S.A. (1990) Cache and Memory Hierarchy Design: A Performance-Directed Approach. Morgan Kaufmann.

[31] OSCI (2011) IEEE Standard for SystemC Language Reference Manual. IEEE Standard 1666-2011.

[32] Ghenassia, F. (2010) Transaction-Level Modeling with SystemC: TLM-2.0 in Practice. Springer.

[33] Siemens, E.D.A. (2020) Catapult High-Level Synthesis. White Paper.

Appendix

Correspondence Table of Related Terms

Term	Implication
Moore's Law	An observation predicting that the number of transistors on an integrated circuit roughly doubles every two years. This principle is not only a technological prediction but also concerns economic efficiency, as the cost-effectiveness of circuits improves as more functions are integrated.
Dennard Scaling	A design principle stating that as transistor dimensions shrink, other parameters like voltage can be adjusted proportionally to keep the internal electric fields constant. This approach preserves device behavior while achieving significant improvements in performance, density, and power efficiency. However, it ceased to be valid below the 90 nm node as leakage current began to grow exponentially.
Power Wall	A critical bottleneck in chip design where increasing power consumption and thermal dissipation limit further performance improvements, particularly from increasing clock frequencies. It marks the end of the era where performance was gained simply by making transistors faster.
Memory Wall	The growing performance gap between the speed of processors and the speed of memory. This disparity has become a primary bottleneck that limits overall system performance because even the fastest CPUs must frequently pause and wait for data from slower memory.
System-on-Chip (SoC)	A design that integrates multiple independent computing cores and other components onto a single chip. Modern mobile SoCs, for instance, must mitigate challenges like the dark-silicon problem that arises from having more transistors than can be simultaneously powered on.
Pipelining	A technique that splits an instruction or data processing task into multiple successive stages to be processed in parallel. This allows different instructions to be in different stages at the same time, dramatically increasing overall throughput.
Multi-core	A processor design that integrates multiple independent computing cores onto the same chip. Each core contains its own ALU and register set, and sometimes a private L1 Cache, while sharing resources like the L2/L3 Cache with other cores.
Hyper-Threading (SMT)	Also known as Simultaneous Multi-Threading (SMT), this is a technique to execute multiple threads in parallel on a single physical core. The physical core presents multiple "logical processors" to the operating system, allowing it to better utilize idle pipeline slots and improve throughput with a minimal increase in silicon area. Intel's measurements show it provides a 15% - 30% performance improvement for multi-threaded workloads for only a 5% increase in die area.
Amdahl's Law	A formula used to predict the theoretical maximum improvement in performance when only part of a system is improved. It reveals that any performance gain is ultimately limited by the portion of the task that cannot be parallelized.
Clock Gating	A power-saving technique where a gate is used to selectively disable the clock signal for logic blocks that are idle or inactive. This eliminates unnecessary switching activity, which is a major source of dynamic power consumption.
Low-swing Signaling	A circuit-level design technique that saves power by reducing the voltage swing of signals, especially on the clock network. Since dynamic power is proportional to the square of the voltage ($P \propto V2$), reducing the voltage swing is highly effective at cutting down switching energy.
Dynamic Voltage and Frequency Scaling (DVFS)	A core power management technology that synergistically adjusts a chip's operating voltage and clock frequency in real-time according to its computational load. The goal is to minimize power consumption while ensuring the required performance is met.
Domain-Specific Accelerator (DSA)	Specialized hardware that sacrifices universality for datapaths, memory hierarchies, and control logic tailored to a specific algorithm or domain. By doing so, DSAs can boost energy efficiency by 10 - 100× compared to general-purpose CPUs or GPUs.

Continued

Video Coding Unit (VCU)	A specialized hardware accelerator designed to handle the challenges of large-scale video transcoding in data centers. It contains dedicated ASIC cores for key encoding stages like motion estimation, rate-distortion optimization, and entropy coding.
Tightly-Coupled Accelerator (TCA)	A specialized hardware unit that is physically and architecturally integrated with a host processor like a CPU. TCAs have low-latency access to the CPU's caches or registers, making them suitable for tasks like real-time AI inference and cryptography.
Loosely-Coupled Accelerator (LCA)	An accelerator that functions as a separate module in the system, not integrated directly with the CPU. LCAs have higher communication delay but also higher throughput, often because they have their own dedicated memory, making them suitable for tasks with large datasets like 4K video stream processing.
Network-on-Chip (NoC)	An on-chip communication architecture that organizes cores and other modules into a structured network connected by routers and links. It is a scalable solution for many-core SoCs, overcoming the bottlenecks of traditional bus architectures.
Latency-Insensitive Design (LID)	A design methodology that ensures reliable collaboration between circuit modules, independent of the signal propagation delay between them. It uses a "locally synchronous, globally asynchronous" approach, where "shells" and "stallable modules" use handshake protocols to manage data flow and guarantee functional correctness despite timing uncertainty.
Cache Coherence	A mechanism in multiprocessor systems that ensures multiple private caches holding copies of the same memory line converge to a consistent value when any processor modifies that line. It provides the illusion of a single, unified memory value per address.
Memory Consistency	A set of rules that defines the observable ordering constraints on memory operations across different processors. It acts as a "contract" between hardware and software, specifying which reorderings of memory operations (like a load bypassing a store) are permissible.
SystemC	A C + + -based modeling language and simulation framework used for describing and verifying hardware/software systems at a high level of abstraction. It enables fast functional validation and architectural exploration before the detailed RTL (Register-Transfer Level) implementation is complete.