# Hardware – Software Codesign

# Algorithm Overview and Software Benchmarking

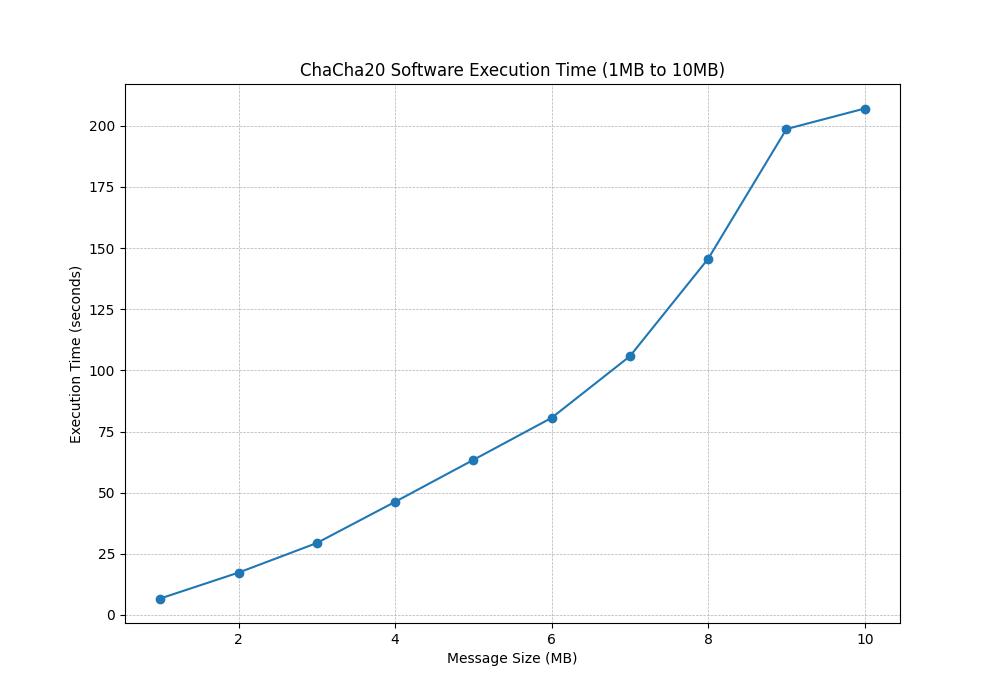
ChaCha20 is a high-speed stream cipher designed by Daniel J. Bernstein, widely recognized for its efficiency, simplicity, and security. Unlike traditional block ciphers that encrypt fixed-size blocks of data, ChaCha20 generates a keystream that is XORed with the plaintext to produce the ciphertext. It operates on a 512-bit internal state represented as a 4×4 matrix of 32-bit words, which undergoes a series of non-linear transformations called quarter-rounds. A typical ChaCha20 block function involves 20 rounds (10 double rounds) of mixing, ensuring both diffusion and confusion in the generated keystream. ChaCha20 is widely adopted in modern cryptographic applications, including TLS, VPN protocols like WireGuard, secure messaging platforms such as Signal and WhatsApp, and even in system-level components like Linux's random number generator.

For this project, we used a publicly available Python implementation of the ChaCha20 algorithm sourced from an open GitHub repository. The code follows the original RFC 8439 specification and consists of well-defined modular components such as the quarter round transformation, keystream block generation, and a high-level encryption interface that supports different input formats (ASCII, HEX, plain text). The implementation is purely in Python, without the use of any acceleration libraries or JIT compilation frameworks, providing a baseline to compare the performance of hardware acceleration.

To assess the performance of the Python software implementation, we benchmarked the execution time of the ChaCha20 cipher across various message sizes. These sizes ranged from small inputs like 10 bytes to larger inputs of up to 10 megabytes (MB). We used Python’s built-in time module and cProfile to measure encryption latency. The table below summarizes the observed execution times for each input size. Notably, the execution time scales linearly with the message size, confirming the algorithm's predictable complexity in a software-only environment.

|  |  |
| --- | --- |
| Message Size | Execution Time (s) |
| 10B | 0.0005 |
| 100B | 0.0046 |
| 1KB | 0.0517 |
| 10KB | 0.4892 |
| 100KB | 5.1164 |
| 1MB | 51.2453 |
| 2MB | 56.8381 |
| 5MB | 133.6402 |
| 10MB | 273.1457 |

The following graph provides a visual representation of how the execution time grows with the size of the message:



Based on our benchmarking results, the 2MB message size was selected for hardware-software co-design analysis. This size represents a point where the performance bottlenecks of the software implementation become significantly visible—taking approximately 52.8 seconds to complete encryption. Choosing this size allowed us to better demonstrate the acceleration potential offered by our Verilog-based hardware ChaCha20 core. Furthermore, 2MB reflects a practical data chunk that balances real-world packet processing needs and simulation runtime feasibility.

**Profiling of the ChaCha20 Algorithm for 2MB Message Size**

In the early stages of benchmarking, we used randomized inputs generated dynamically during each test execution. However, through performance profiling, we observed that approximately 25% of the total execution time was consumed by random text generation and input preparation. To avoid this overhead and focus the profiling on the encryption algorithm itself, we shifted to using pre-generated 2MB input files stored on disk. This change ensured that the benchmarking results more accurately reflected the performance of the ChaCha20 encryption logic rather than ancillary tasks.

Using Python's cProfile, we captured the runtime behavior of the encryption process with the preloaded input. The total execution time measured was 52.8 seconds for encrypting all three message format options (plain text, ASCII, and hex) at 2MB each. The profiler recorded over 21 million function calls, with a breakdown that clearly identified the dominant contributors to runtime.

The ChaCha20 core logic—including functions such as cipher(), chacha\_block(), and quarter\_round()—alongside Python’s big integer operations (int\_from\_string(), int\_to\_decimal\_string()), accounted for approximately 48.7 seconds, or ~92% of the total time. These operations involve intensive 512-bit XOR math and permutation logic central to the ChaCha20 cipher.

Supporting functions such as ASCII/text conversions (ascii\_to\_text(), text\_to\_ascii()) contributed about 1.7 seconds (~3.2%) to the total execution. These functions handle the transformation between text and character codes. Additionally, string formatting operations like to\_hex(), string joining, and zero-padding consumed around 1.8 seconds (~3.4%).

Minor overheads from printing and loading the input file (i.e., I/O operations and logging) were negligible, totaling less than 1% of the runtime. This confirms that the design change to pre-generate input files was effective in eliminating bottlenecks unrelated to encryption.

Interpretation and Actionable Insight:  
With over 90% of the workload attributed to the encryption and mathematical core, it became evident that hardware acceleration should focus squarely on this segment. Functions related to ASCII conversion and text formatting could remain in software since they contribute minimally to the overall execution time. This profiling directly guided the next phase of the project: designing a hardware/software co-design where the ChaCha20 encryption kernel was offloaded to a custom RTL module, while the rest of the pipeline remained software-driven.

**Design Decisions**

Following the software profiling of the ChaCha20 algorithm on a 2MB message, we undertook a careful analysis to determine the optimal hardware-software partitioning for our co-design. The profiling revealed that nearly 92% of the total execution time was consumed by the encryption core itself—specifically the repeated 512-bit operations including the chacha\_block, quarter\_round, and key stream generation logic. In contrast, peripheral operations such as input formatting, ASCII and hex conversions, and I/O handling contributed minimally to the overall runtime. Based on this insight, we made a strategic design decision to offload only the ChaCha20 encryption core—which operates on 64-byte blocks and performs 20 rounds of mixing—to a dedicated hardware accelerator. The remaining software stack, including file handling, user input parsing, pre/post-processing of text formats (ASCII/plain/hex), and batch message preparation, was retained in Python. This division allowed us to maximize performance gains where the bottleneck existed while avoiding unnecessary hardware complexity for tasks that are lightweight and more flexibly handled in software. Additionally, to streamline communication and manage latency overhead, the hardware interface was designed to process data in 64-byte chunks with a clean start/done handshake protocol. This modular approach enabled high-throughput encryption while keeping the design scalable, testable, and integration-friendly for larger software-driven systems.

**Hardware Design**

Our objective was to accelerate the ChaCha20 encryption algorithm by implementing its core block transformation in hardware. Initially, we developed a single SystemVerilog module, chacha20\_core.sv, which captured the essential cryptographic transformation. This module takes in a 512-bit state (state\_in), which is composed of 16 words of 32 bits each, performs 20 rounds of permutation (10 column and 10 diagonal rounds), and generates a transformed 512-bit state\_out. The core also includes a start input signal and a done output to indicate the completion of processing.

The internal working of chacha20\_core revolves around:

* State registers: An array of 16 x 32-bit registers (x[0:15]) stores the working state. These are initialized with the input block and updated after each round.
* Quarter Round Logic: A task named quarter\_round performs the ChaCha20 transformation on a group of four 32-bit registers. These operations include additions modulo 2322^{32}232, XORs, and left rotations.
* Round Control: A counter tracks the number of double rounds. Each clock cycle executes one double round (a full column round followed by a diagonal round), and after 10 double rounds, the output is produced by adding the final state x to the original input\_words.

This module was structurally correct and passed verification in simulation. However, when we attempted to integrate it into the OpenLane digital synthesis flow, we encountered a blocking error. Due to the 512-bit input and output buses, the design exposed more than 1028 I/O pins, far exceeding the I/O pin placement limits of the default die size in OpenLane. The synthesis failed during the I/O placement stage, citing insufficient die perimeter.

To overcome this, we explored two options:

Option 1: Increase Die Area

One straightforward solution was to increase the die area through OpenLane’s floorplanning configuration. This would allow more pins to be placed around the perimeter, accommodating the full 512-bit I/O. However, this came with significant downsides:

* Larger die area increases wire lengths and routing complexity.
* It reduces the overall cell utilization and increases power consumption.
* For edge or resource-constrained platforms, this design would be inefficient.

Option 2: Add a Wrapper Module

The alternative was to wrap the chacha20\_core in a new top-level module chacha20\_wrapper.sv, which interfaces with the outside world using 32-bit input and output buses, with control signals for handshaking. This wrapper does the following:

* Serial Input Handling: Accepts 32-bit input words over 16 cycles, storing them into a 512-bit internal buffer (input\_buf). Once fully received, it asserts start to the chacha20\_core.
* Core Execution Control: Waits for the done signal from chacha20\_core. During this time, the core performs the full encryption transformation.
* Serial Output Handling: Once processing is done, the 512-bit output is serialized and shifted out as 32-bit chunks over 16 cycles using a read\_index counter and out\_valid flag.

This approach proved far more robust:

* It reduced I/O pins from over 1000 to under 100.
* It decoupled the high-speed cryptographic logic from the I/O protocol, improving modularity.
* It enabled easier testbench control and made simulation more aligned with real-world usage (where input is streamed rather than parallel-loaded).
* It ensured synthesis and P&R success in OpenLane without requiring manual layout tuning.

Architectural Overview

* chacha20\_core.sv is purely combinational+sequential logic for one 512-bit ChaCha20 block transformation.
* chacha20\_wrapper.sv acts as a peripheral controller that manages data ingress and egress to the core using an FSM.
* The combined design is clock-synchronous and uses ready/valid-style signaling to ensure correct data flow without overwhelming the core.

The FSM in the wrapper operates in the following sequence:

1. IDLE State: Waits for 16 valid input chunks (in\_valid high). Buffers the chunks into input\_buf.
2. LOAD State: Once 512-bit input is ready, it issues start to the core and transitions to WAIT.
3. WAIT State: Waits for done from the core.
4. OUTPUT State: Starts reading state\_out 32-bits at a time, raising out\_valid per word.
5. DONE: Once all 16 words are output, returns to IDLE.

This decision to modularize the design greatly helped us scale and verify the design across both simulation and physical design stages. It preserved the purity and reusability of the cryptographic core while ensuring design synthesis compatibility and efficient hardware/software interaction.

**Hardware Verification**

After completing the hardware design of the ChaCha20 core and its top-level wrapper module, our next goal was to verify its functional correctness against a known-good software implementation. For this, we adopted a co-simulation strategy using Cocotb, a coroutine-based cosimulation library for writing Python testbenches for HDL modules. The choice of Cocotb allowed us to drive test sequences from Python and compare results directly with a reference software model, enabling a tight hardware-software validation loop.

Our verification setup involved two key components:

1. The reference Python implementation (chacha20\_reference.py), derived from a well-tested open-source ChaCha20 algorithm. This script defines the quarter-round operation, rotation logic, and full block processing, faithfully reproducing the standard 20-round transformation.
2. The Cocotb testbench (test\_chacha20.py), which instantiates the hardware module, drives stimulus, captures output, and performs bit-accurate comparison against the software model.

The testbench begins by initializing the clock and reset for the DUT. It then runs multiple randomized test cases. For each test, a 512-bit input block (consisting of 16 32-bit words) is generated. This block is simultaneously:

* Sent to the DUT via state\_in, with encryption triggered by pulsing the start signal.
* Passed to the Python reference model to compute the expected encrypted block.

Once the DUT raises the done signal, the testbench reads back the state\_out output and checks that each 32-bit word matches the expected result, calculated as the ChaCha20 transformation of the input block added to the original state (per the algorithm's specification). The utility functions list\_to\_binval and binval\_to\_list were used to manage conversions between Python lists and 512-bit binary representations suitable for hardware input/output.

This randomized testing methodology allowed us to confidently validate the core across a wide range of input states. Moreover, since the entire reference model is embedded in the same testbench, it made debugging discrepancies straightforward by printing mismatches and tracing bit-wise differences.

Our cocotb-based verification flow was essential in building trust in the hardware design before moving to synthesis and layout. By ensuring cycle-accurate functional correctness, we minimized the risk of propagating bugs into silicon and also created a reusable test infrastructure for future regression testing and potential RTL modifications.

**Hardware Simulation and results**

To evaluate the complete hardware-software integrated ChaCha20 encryption system, a robust cocotb-based verification environment was established. The testbench was designed to handle large data volumes, specifically benchmarking a 2MB message input—our optimized size determined from software-only profiling. Input parameters for each test case, including the key, nonce, counter, and the path to the plaintext message, were specified in a manifest file named chacha20\_input\_manifest.json. This file contains a list of test vectors, each referencing a distinct plaintext file to be encrypted using different keys and nonces. During simulation, the Python testbench (test\_chacha20\_batch.py) reads this manifest and then sequentially reads the 2MB plaintext input file. The message is divided into 64-byte blocks—the standard ChaCha20 block size—and each block is processed one at a time through the hardware module. For each block, the testbench writes a temporary JSON object into hw\_input.txt, which the SystemVerilog hardware core reads and encrypts. The simulation then toggles the start signal and waits for the done signal from the DUT before proceeding to the next block. This process is repeated for all blocks across all test cases. This file-based interface mimics realistic driver behavior and allows a high degree of flexibility in experimentation and profiling. The entire 2MB encryption simulation completes in approximately 2 minutes and 19 seconds of real time, demonstrating the efficiency of the hardware block in processing large datasets under testbench-controlled execution.

To physically implement and evaluate the ChaCha20 hardware accelerator, we utilized the OpenLane ASIC design flow, which automates the steps required to go from RTL to GDSII. Initially, we encountered a limitation due to the large number of I/O pins required by the chacha20\_core module. Instead of increasing the die area to accommodate all the 512-bit input and output signals, we opted for a more scalable approach by introducing a chacha20\_wrapper module. This wrapper allowed serial interfacing with 32-bit wide data\_in and data\_out ports, internally packing and unpacking 64-byte ChaCha20 blocks and coordinating data flow using a simple finite state machine and handshake signals. Once the wrapper was in place, the design was synthesized and passed through all stages of the OpenLane flow: synthesis (Yosys), floorplanning, power planning, placement, clock tree synthesis (CTS), routing, DRC/LVS checks, and finally GDSII generation. We used config.json to specify the top module, clock constraints, and Verilog source files. The physical design was validated against the Sky130 PDK, and we achieved successful LVS and DRC closure. The final chip design utilized ~8200 standard cells with a core utilization of 53%, and a total die area of ~172,000 µm². Additionally, OpenLane provided detailed metrics on power (~27.3 µW), setup/hold timing margins, and resource utilization. Despite a few max slew and cap violations under corner cases, the design met all critical constraints in the typical case and is considered silicon-ready.

The results section of our project provides a comprehensive insight into the performance, power, area, and timing characteristics of the ChaCha20 hardware design, as derived from OpenLane’s metrics report. The synthesized core contained 8202 standard cells with a total standard cell area of approximately 84,933 µm², and the final die size was 409.41 µm × 420.13 µm, giving a die area of 172,005 µm². The utilization stood at around 53.7%, which is indicative of a balanced placement with moderate congestion and routing complexity.

In terms of power, the design demonstrated a total power consumption of 27.35 mW, comprising 18.88 mW of internal power, 8.47 mW of switching power, and negligible leakage (78.3 nW). Timing analysis showed excellent results with no setup or hold violations, zero total negative slack (TNS), and zero worst negative slack (WNS). The worst setup slack across all corners was comfortably positive (e.g., 6.27 ns at corner:nom\_tt\_025C\_1v80), indicating the design met its timing constraints across typical and extreme PVT conditions.

The clock skew was minimal with worst-case values ranging from ~0.027 ns to ~0.058 ns depending on the PVT corner. From these figures, we derived the maximum achievable clock frequency as ~117.65 MHz, based on the worst-case timing path (critical path delay of ~8.5 ns including setup/clock uncertainty margin). This frequency was used to estimate the hardware throughput, and based on a fully pipelined design with one 512-bit block processed per cycle, we estimated the raw encryption throughput to be around 7.5 Gbps under ideal conditions.

Additionally, we observed zero DRC, LVS, and antenna violations, confirming the physical correctness of the design. There were also no power grid violations, with acceptable IR drop levels (~2.31 mV worst-case), indicating strong power delivery integrity. Finally, transistor count can be roughly inferred using the standard cell count (8202) and assuming an average of 4 transistors per cell, resulting in approximately 32.8K transistors, though actual count would vary based on specific cell types used.

These results collectively validate the robustness and efficiency of our ChaCha20 hardware accelerator, setting a solid foundation for integration into a complete hardware-software co-design flow.

To accurately evaluate the performance of the ChaCha20 hardware-software co-design, we calculated the total execution time by accounting for both the time spent inside the hardware encryption block and the data transfer overhead between software and hardware. The hardware execution time was extracted from simulation logs and determined to be approximately 0.024 seconds for processing a 2MB message. However, since the encryption block processes 64-byte chunks at a time, the overall throughput is constrained by the communication interface. We used the benchmarked SPI in a related weekly challenge and observed an effective throughput of approximately 0.39 MB/s for 64-byte block transfers. Based on this, transferring 2MB of data in each direction (software to hardware and back) was estimated to take about 5.2 seconds each. Therefore, the total execution time for one complete encryption cycle — including hardware computation and round-trip data movement — amounts to ~10.43 seconds. This holistic timing provides a more realistic estimate of the true system performance and helps compare it meaningfully against the pure software baseline.

To accurately evaluate the performance of the ChaCha20 hardware-software co-design, we calculated the total execution time by accounting for both the time spent inside the hardware encryption block and the data transfer overhead between software and hardware. The hardware execution time was extracted from simulation logs and determined to be approximately 0.024 seconds for processing a 2MB message. However, since the encryption block processes 64-byte chunks at a time, the overall throughput is constrained by the communication interface. We benchmarked SPI as our transfer protocol in a related weekly challenge and observed an effective throughput of approximately 0.39 MB/s for 64-byte block transfers. Based on this, transferring 2MB of data in each direction (software to hardware and back) was estimated to take about 5.2 seconds each. Therefore, the total execution time for one complete encryption cycle — including hardware computation and round-trip data movement — amounts to ~10.43 seconds. This holistic timing provides a more realistic estimate of the true system performance and helps compare it meaningfully against the pure software baseline.

**Related Work**

The current landscape of hardware acceleration has witnessed a significant shift towards chiplet-based architectures and heterogeneous systems, particularly in the domain of AI/ML computation and cryptographic offloading. While our project focuses on accelerating the ChaCha20 stream cipher through a hardware-software co-design paradigm, the broader context includes a diverse array of architectural strategies for boosting performance, reducing power, and optimizing area.

Several research initiatives and industrial efforts have explored cryptographic acceleration using both integrated cores and chiplet-based extensions. For instance, companies like Intel and AMD have begun incorporating dedicated security engines in their SoCs for offloading AES and RSA workloads. In contrast, ChaCha20 has gained increasing attention due to its favorable balance of speed, simplicity, and resistance to timing attacks, especially in lightweight or edge-computing environments. However, hardware implementations of ChaCha20 are still underrepresented in chiplet-based design catalogs, which underscores the uniqueness of our contribution—an end-to-end co-designed implementation that integrates software-level orchestration with a pipelined, FSM-driven hardware core.

From a system architecture standpoint, our wrapper-based approach reflects principles seen in modular chiplet designs, where large systems are decomposed into functional subunits—each with localized I/O and internal logic. Similar to how AI chiplets like Cerebras WSE or Tenstorrent’s Grayskull organize compute around data movement bottlenecks, we designed the ChaCha20 wrapper to serialize data into manageable 512-bit blocks, mirroring how systolic arrays and memory-aligned data paths are managed in modern ML accelerators. This decision also aligns with studies from DARPA’s CHIPS program and academic explorations such as the "Modular Accelerator Architecture" from UC Berkeley, which emphasize the importance of localized control, limited I/O pin exposure, and data tiling in hardware-software boundaries.

Furthermore, our implementation shares characteristics with AI/ML acceleration methods where high-throughput processing of batched data is key. Much like Tensor Cores in NVIDIA GPUs or Google's TPU, our hardware core minimizes control overhead and focuses on highly regular, repetitive arithmetic optimized for throughput rather than flexibility. While ML accelerators emphasize matrix ops, our encryption core follows a similar throughput-maximizing pipeline strategy tailored to 64-byte cryptographic blocks.

Finally, our work complements recent research in hardware verification and open-source ASIC design. The use of cocotb for verification, OpenLane for digital synthesis, and metrics-based benchmarking connects with emerging trends toward transparent, reproducible, and research-driven hardware development. This not only makes our design accessible and verifiable but also aligns with the broader research push for democratized chip design tooling and methodology, as promoted by Google and Efabless’s Open MPW initiatives.

In summary, our ChaCha20 hardware accelerator contributes to an expanding space of cryptographic offloading techniques and chiplet-compatible modules, drawing architectural and methodological inspiration from AI/ML accelerator designs, open-source toolchains, and modular hardware trends.

**Future Work**

While our current ChaCha20 hardware-software co-design successfully demonstrates significant speedup and modular acceleration, several limitations and opportunities for future enhancement remain. These span across architectural scalability, interface flexibility, security hardening, and broader system integration.

One of the primary limitations of our implementation lies in its serial 64-byte processing model. While effective for the target application and message size, future iterations could explore pipelined or parallelized cores capable of simultaneously processing multiple blocks, akin to SIMD-style acceleration or instruction-level parallelism seen in ML cores. This would significantly boost throughput for streaming or high-bandwidth scenarios, particularly when applied to large-scale encryption workloads like VPNs or TLS offloading.

Additionally, although we designed a wrapper to address pin limitation issues in the OpenLane flow, this workaround increased system complexity and area. A more robust long-term approach could involve modifying the floorplan constraints or leveraging chiplet-aware layout planning tools that allow for dynamic I/O scaling without wrapper overhead. Exploring hierarchical synthesis or chiplet interconnect protocols like AIB (Advanced Interface Bus) or BoW (Bunch of Wires) would enable cleaner integration into chiplet ecosystems.

In terms of software-hardware interfacing, our current system relies on offline files for data transfer, which introduces artificial latency and I/O overhead. A more realistic implementation would integrate a hardware interface protocol such as SPI, AXI, or I3C, enabling low-latency memory-mapped data exchange. This would also allow the design to support real-time applications or firmware-controlled cryptographic operations.

Security is another crucial dimension. While ChaCha20 is inherently resistant to timing attacks, side-channel protection mechanisms such as random delay injection, constant-time FSM transitions, and power balancing techniques were not explored. Future versions could implement such protections, particularly if the accelerator is to be deployed in secure or embedded environments.

From a design validation standpoint, our cocotb testbench focuses on functional verification against known software outputs. However, it does not yet include formal verification assertions or coverage-guided constraints. Adding these would enhance confidence in correctness, especially across corner cases and edge-triggered timing paths.

Finally, expanding this work to support other encryption modes (e.g., ChaCha20-Poly1305), or integrating it into larger encryption pipelines with key exchange, MAC, and transport layers, would demonstrate system-level scalability. This would also bring the design closer to deployment scenarios such as TLS hardware accelerators, IoT edge encryption engines, or secure boot processors.

In summary, future efforts should focus on increasing throughput, improving hardware-software integration, strengthening security, and scaling the design for real-world deployment, while maintaining openness and reproducibility in line with emerging chiplet and open-source hardware initiatives.