Teaching Collaborative SoC Engineering with HLS-Based Accelerator Design and Agile System-Level Integration

Gabriele Tombesi*, Joseph Zuckerman*, Marian Abuhazi, Sai Vittal Battula, Maico Cassel dos Santos, Kuan-Lin Chiu, Guy Eichler, Srivatsan Raveendran, Biruk Seyoum, Pei-Huan Tsai, Ajay Vanamali, Je Yang, and Luca P. Carloni
Department of Computer Science — Columbia University in the City of New York
New York, New York, USA

ABSTRACT

System-on-Chip Platforms is an advanced course in the computer engineering curriculum at Columbia University that focuses on system-level design principles to expose students, both theoretically and practically, to the main challenges of SoC engineering. The course is divided into two complementary (principles and practice) tracks. In recent years, we have revamped the practice track to give students hands-on experience with SoC design and programming. A key component of this effort is the introduction of object-oriented high-level synthesis, which enables students to acquire the skills necessary to design their own hardware accelerators. In the collaborative final project, students optimize an accelerator and combine it with accelerators designed by other students to create a complete SoC for a target application. The project is based on open-source hardware and can be replicated at other universities.

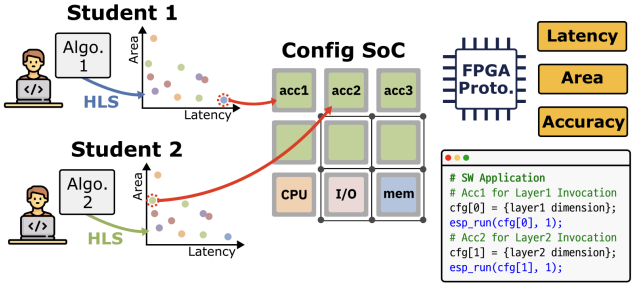

Figure 1: Overview of the SoC Platforms final project. Students design an accelerator with HLS, build an SoC with multiple accelerator instances, and deploy an application invoking the accelerators on an FPGA prototype of the SoC.

1 INTRODUCTION

State-of-the-art system-on-chip (SoC) architectures are becoming more and more heterogeneous by hosting a growing number of domain-specific accelerators [14, 37] on a single die [5, 15, 24]. Heterogeneity improves energy-efficient performance as much as it increases design complexity; consequently, the addition of capabilities to an SoC is increasingly limited by the effort and size of the SoC engineering team [26]. Since a single team cannot afford to design every single component, the reuse of *intellectual property blocks*, or simply *IPs*, is critically important in complex SoC design.

Training new generations of university students to design these types of SoC architecture is a challenge for multiple reasons, including the limited availability of real-world design examples and the cost of purchasing third-party IPs. The emergence of the *open-source hardware (OSH)* movement [23] offers a unique opportunity to address this challenge by enabling the realization of SoCs with realistic properties through *collaborative SoC engineering* across different institutions. Different research groups in academia, as well as in some companies and government labs, have started releasing a variety of SoC components (e.g. CPUs, GPUs, accelerators, network-on-chip) as open-source artifacts in the public domain. These components have enabled the design and tape-out of prototype chips of heterogeneous SoC of remarkable complexity [10, 12, 20, 35, 36] that can now be used in the classroom to educate and develop the SoC design workforce.

Traditional curricula in computer engineering have been built around courses on processor architecture, register-transfer level (RTL) design, and VLSI design. Although these remain important topics, we believe that preparing students for the new era in SoC design necessitates the addition of new types of course that train them to think at the system level, develop IPs for broad reusability, and build complete systems by integrating newly developed IPs with existing and third-party ones. To fill this gap, our research group, the System-Level Design Group, has developed ***System-on-Chip Platforms (SoCP)***, a course for senior undergraduate and graduate students in computer science and electrical engineering that has been offered at Columbia for the past 12 years [18].

SoCP consists of two complementary tracks: the *principles* and the *practice* track. The principles track equips students with the tools to reason about complex SoC designs by evaluating performance trade-offs and understanding the hardware-software interface. The practice track, which we have recently revamped to reflect new offerings from the electronic design automation (EDA) industry and leverage new capabilities developed by our research group, now teaches students to design accelerators with *object-oriented high-level synthesis (OOHLS)* and integrate them in an SoC with the help of an open-source SoC platform. Combined, the two tracks give students with a stronger software background the confidence to design hardware and equip students with a stronger hardware background with the skills to design at a higher level of abstraction.

*Equally credited authors

WCAE '25, June 21–25, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-2007-9/2025/06
https://doi.org/10.1145/3743646.3750013

The course culminates in a new, exciting final project, where students accelerate a given application by conducting *design-space exploration (DSE)* at both the component and the SoC level, as shown in Figure 1. In recent years, for example, the students first optimize an accelerator for a single layer of a convolutional neural network (CNN) using the OOHLS techniques learned throughout the course; then, in the second part of the project, they can combine their accelerator with accelerators for other layers, designed by other students in the class, to build a complete SoC for CNN inference. Leveraging ESP [30], an open-source platform for SoC design developed by our research group, the students can seamlessly design a complete SoC with multiple accelerator instances, rapidly derive FPGA prototypes of their selected design, and deploy real software applications that invoke the accelerators in the SoC. The project unlocks a new level of design complexity that is achievable in the classroom and, thanks to the open-source nature of ESP, can easily be replicated at other universities.

After a brief overview of the course (Section 2), the rest of the paper focuses on the recent innovations int the practice track. Section 3 details how we gradually introduce students to OOHLS design through a series of increasingly more complex coding assignments. Sections 4 and 5 describe in detail the final project of the course.

2 COURSE OVERVIEW

SoCP was primarily developed for advanced undergraduate and graduate students in computer science or electrical engineering [18]. Recent enrollment has been approximately 40–60 students per semester. The only prerequisites for enrollment are 1) an introductory programming course in C or C++ and 2) an introductory class on computer systems organization. The course meets twice a week for 75-minute lectures; 2-3 graduate student TAs provide weekly office hours and occasional review/focus sessions. The two main tracks of the course run in parallel throughout the semester: approximately 60% of the courses consists of more traditional lectures devoted to the principles track, while the practice track comprises the remaining 40% lectures, which follow a more hands-on approach with many code examples.

Students complete five main homework assignments throughout the semester, which may contain pencil-and-paper exercises from the principles track, programming assignments from the practice track, or a mix of both. We give roughly two weeks for each of these assignments and we expect students to spend the bulk of their time outside of class on the programming assignments. The last month of the course is reserved for the final project, which is discussed in depth in Sections 4 and 5. There are both a midterm and a final exam, which primarily focus on the principles track, but concepts from the practice track may appear (e.g., in multiple-choice questions). *Across all types of assignments and exams, students are generally expected to spend considerably more time reading (text, existing code) than writing (solutions, new code);* this pedagogical approach emphasizes the development of analytical skills to process large amounts of information and understand the broader context in depth before developing new contributions to be integrated in such a context.

The principles track starts from a survey on the economic and technological trends of modern SoC architectures. We then introduce the notion of platform-based design, where a *platform* is defined as the combination of a computer architecture and a companion computer-aided design methodology [8]. *Modularity* and *abstraction* are therefore presented as key mechanisms that allow architects to explore a large design space and reuse engineering effort. To reason about this design space, we introduce the notion of *Pareto optimality* in a multi-objective optimization space and discuss *Amdahl's Law* to highlight the diminishing returns of over-optimizing any individual component in a complex system.

We then shift our focus to modeling and analyzing complex computing systems. Formal guidance is provided by *Models of Computation (MoC)*. First, we present the Synchronous MoC that underpins conventional RTL design, and discuss the "crisis of the synchronous paradigm" that arose due to "the Wire Problem" [7, 31]. The discussion naturally leads to *latency-insensitive design (LID)* [9] and its Protocol-and-Shell paradigm [7]. With the help of the Tagged-Signal Model [28], students are taught how the LID shells and relay stations preserve the synchronous hypothesis while relaxing timing constraints during the early phases of the design process, when accurate path delays are still unknown [7, 9]. Marked graphs, a subclass of Petri nets, are introduced as a MoC for analyzing concurrency and throughput of complex systems [32].

In the second half of the semester, a series of lectures systematically explore the principal components of SoC platforms, illustrated through case studies of both state-of-the-art industrial products and recent academic research projects. Prominent instruction set architectures (ISAs), specifically ARM and RISC-V, are introduced [3]. Data-level parallelism is elucidated through a detailed examination of the NVIDIA Xavier GPU-based SoC [15]. Bus-based communication systems are analyzed in detail using the Advanced eXtensible Interface (AXI) protocol as a case study [2]. Subsequently, hardware accelerators are introduced as a critical response to the limitations imposed by the end of Dennard's scaling, illustrated through examples such as the open-source NVIDIA Deep Learning Accelerator (NVDLA) [33]. Finally, we address hardware-software integration by covering device-driver programming and discussing strategies for managing the shared-memory address space used by accelerators, along with various cache-coherence models.

The practice track consists of a series of lectures alternating with hands-on programming assignments. We begin by teaching the OOHLS design methodology, based on the combination of latency-insensitive communication channels [9] and Matchlib [26], a library of reusable hardware primitives. We illustrate the optimization knobs that this methodology exposes at the algorithm level, which enable students to rapidly generate entire families of accelerator micro-architectures that span different power, performance, and area (PPA) trade-offs. In the final project of the course, the students are taught how to integrate the accelerators into a complete SoC using *ESP* [30]. By coupling the OOHLS design methodology for accelerator generation with ESP for agile system prototyping, the course offers students the opportunity to experience a seamless path: from algorithms to complex FPGA-based SoC prototypes.

3 DESIGN ABSTRACTION FOR AGILE SOC DEVELOPMENT

The growing heterogeneity and complexity of contemporary SoCs—often featuring hundreds of IPs—render block-level RTL methodologies increasingly impractical. Validating the behavior of such systems requires verification environments that exercise

cross-IP interactions early. However, when bugs escape into sign-off, costly mask respins may be required. A recent industrial survey [38] reports that verification now consumes 55–70% of the entire project effort, while first-silicon success has fallen to 14%. Meanwhile, AI inference workloads evolve on a markedly shorter cadence: successive generations of generative models and emerging DNN architectures push computation and memory demands, pressuring architects to tape-out new micro-architectures each year, a lofty goal when exercising manual RTL design and gate-level verification.

These converging pressures have caused both industry and academia to pursue dramatically increased design productivity by favoring high-level, object-oriented design flows and transaction-level verification methodologies over conventional RTL-centric techniques. Domain-Specific Languages (DSLs), such as Spatial [27], Bluespec [4], and Chisel [1], and EDA technologies, such as HLS, exemplify the shift of hardware specifications above RTL. Spatial specifically addresses hardware accelerators for compute-intensive tasks like machine learning, providing high-level algorithmic abstractions. Bluespec uniquely offers rule-based concurrent semantics, simplifying synchronization and timing management. Chisel leverages Scala's functional and object-oriented programming paradigms, facilitating modular, parameterizable hardware definitions while maintaining explicit structural control. HLS tools automatically translate hardware specifications given in high-level programming languages (e.g. C, C++, or SystemC) into cycle-accurate synthesizable RTL implementations; HLS knobs (such as loop unrolling, loop pipelining, banking, and tiling choices) allow designers to perform complex DSE in hours, while the HLS tool guarantees structurally correct RTL [13, 19]. Reusable IP libraries such as Matchlib [26, 34] and BaseJump STL [39] provide common hardware primitives like FIFOs, crossbars, and memory macros, thus amortizing verification across projects and teams.

However, we emphasize that these higher-level specifications complement, but do not replace, hand-coded RTL. Critical datapaths, ultra-fine-grained control logic, and select mixed-signal interfaces are still best crafted and simulated in Verilog or VHDL. Our argument is that, for the hundreds of heterogeneous components integrated into a billion-transistor SoC, investing the bulk of effort at an abstraction level higher than RTL yields the best overall return. Together, these methodologies let architects keep pace with fast-moving product cycles—often measured in months rather than years—without rewriting thousands of lines of RTL, and they push verification effort to fast transaction-level models where property checks and software testbenches expose protocol errors long before gate-level simulation. For these reasons, we adopt OOHLS as the methodological backbone of the SoCP course, equipping students to collaboratively design SoCs where rapid iteration, system-level integration, and agile verification are essential.

3.1 Object-Oriented High-Level Synthesis

Presented in [26], the Object-Oriented High-Level Synthesis (OOHLS) methodology leverages the familiarity of C++ alongside the maturity of state-of-the-art HLS tools to address existing challenges in hardware design flows. Architectural models are coded in untimed C++ or loosely timed SystemC [6, 25], while test-benches incorporate flexible C++ code, including standard libraries like STL and Boost. A distinctive feature of the OOHLS flow is its use of *latency-insensitive* channels [7, 9], providing an abstract, synthesizable communication mechanism compatible with SystemC performance modeling. The OOHLS framework supports two distinct simulation modes: signal-accurate, which is used for accurate modeling and HLS, and fast-TLM for enhanced simulation speed. It also offers features such as random-stall injection, to stress-test potential RTL-level corner cases early in the design process, and RTL co-simulation, which reuses the SystemC test-bench to verify the correct functionality of the HLS-generated RTL. Another essential element of the OOHLS-based methodology is the integration of *MatchLib*, a modular object-oriented library originally developed at NVIDIA and then integrated into Catapult HLS [16], a popular commercial HLS tool. MatchLib contains parameterized hardware components, such as FIFOs, arbiters, crossbars, banked memories, and complex mathematical functions, explicitly designed to be synthesized with HLS into high-performance RTL implementations.

Table 1: Matchlib Connections API

Notation / Description
`Connections::In/Out< T >`: For handling blocking (Pop/Push) and non-blocking (PopNB/PushNB) transaction inputs/outputs.
`Connections::Combinational< T >`: A combinational channel synthesizing to ready, valid, and data signals (rdy/vld/dat).
`Connections::Fifo< T >`: A parameterized FIFO for buffering.
`Connections::SyncIn/Out/Channel< T >`: Two-wire handshake synchronization primitives analogous to software barriers.

By combining latency-insensitive channels and reusable Matchlib components, OOHLS ensures rapid and accurate performance modeling and simulation and significantly enhances design productivity through improved code reuse and the automatic encapsulation of IPs with latency-insensitive logic [7, 9]. By decoupling the functionality of the target logic model from hardware-related design constraints specified in dedicated HLS synthesis scripts, OOHLS enables efficient DSE without altering the source code, while achieving a quality of results (QoR) that is competitive with manually optimized RTL [26]. Ultimately, this methodology enables design teams to focus more on architectural innovation, substantially improving the productivity, flexibility, and effectiveness of the hardware design process.

3.2 Programming Assignments

The course's practice track gradually introduces students to a complete OOHLS-based design flow, while reducing the implementation effort and promoting design reuse in their weekly assignments. To complete the programming assignments, each student is given an account on a course server, which contains all the necessary tools, including MatchLib, Catapult HLS, QuestaSim, Vivado, and ESP. The assignments must be completed individually, and students are expected to write their own code, although we encourage discussion and provide an online forum for them to interact with each other and the course staff.

The students are first introduced to the working principles of the event-driven SystemC simulation kernel, which reproduces the execution of multiple concurrent threads on a single processor [6, 25]. They are then taught how to express system functionality in untimed or loosely timed C++/SystemC. This involves specifying clocks and resets, and organizing the logic into `SC_THREAD` and `SC_METHOD` — the

Table 2: Matchlib Memory Modeling Components

Memory Model / Description
ac_array_1D: Non-shared 1D array mapped directly to RAM or ROM with built-in index-violation checks.
ac_shared_bank_array_2D/3D/vary: Shared banked memories ensuring throughput optimization without bank conflicts.
Scratchpad: Banked memory ideal for conflict-free addressing using low-order address bits.
ArbitratedScratchpad: Advanced Scratchpad variant with arbitration and queuing for complex bank-conflict scenarios.

SystemC counterparts of `process` and `always` blocks in VHDL and Verilog — for sequential and combinational processes, respectively. Students are then introduced to the Connections Library (Table 1), an open-source API of latency-insensitive ports and channels for HLS. Ports and channels are templated for an arbitrary datatype `T` and support both blocking and non-blocking message-passing functions, relying on the data/ready/valid latency-insensitive protocol. Blocking functions (`Push()` and `Pop()`) complete when the ready and valid signals are high, thereby forcing both the software simulation and the generated hardware to stall until this condition is met. In contrast, non-blocking functions (`PushNB()` and `PopNB()`), return a flag indicating whether the operation was successful and never stall; they can be useful for arbitration and flushing.

The programming assignments further expose students to representative MatchLib IPs (e.g., parameterized AXI interfaces and configurable scratchpad memories) implemented as templated SystemC modules. Since on-chip memory largely determines the SoC area, energy, and throughput, the course progressively introduces more sophisticated MatchLib memory models (Table 2). It begins with single-port SRAMs (`ac_array_1D`), advances to parameterizable multi-bank arrays (`ac_shared_bank_array`), and culminates with scratchpad memories (`Scratchpad` and `ArbitratedScratchpad`), which integrate internal crossbars and arbitration logic. Leveraging these Matchlib components improves design reuse and enables rapid functional validation and early performance- and power-estimation studies at speeds orders of magnitude faster than RTL simulation.

As students progress, they are introduced to the Catapult HLS tool [16] to automatically derive RTL implementations from behavioral specifications. They learn to use optimization directives to perform loop unrolling, pipelining, precision tuning, memory banking, and communication bandwidth tuning to complete an extensive DSE. To leverage compiler pragmas for effective hardware optimization, students learn to tune synthesizable Algorithmic-C data types to balance accuracy and resource constraints. Two coding styles are introduced to specify the target hardware model for Catapult HLS. The "RTL in SystemC" style resembles traditional RTL code for fixed-protocol implementations with valid data without backpressure [7]. This coding style does not use the Connections API, but instead instantiates SystemC native `sc_in<>` and `sc_out<>` ports and, therefore, cannot benefit from the fast TLM simulation mode for SystemC prototyping or random-stall injection for debugging. Additionally, the control logic of each module needs to be specified with an explicit finite-state machine (FSM). In contrast, the higher-level "MatchLib SystemC" coding style uses `Connections::In` and `Connections::Out` ports, as well as components from the growing library of Matchlib IPs. This allows designers to specify more concise control structures,

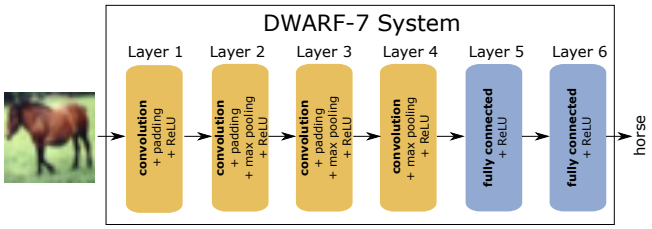

Figure 2: The DWARF7 6-layer CNN for the CIFAR-10 dataset.

with the detailed FSM state encoding delegated to Catapult HLS optimization steps. This flow offers "throughput-accurate" performance modeling in pre-HLS simulation for complex models, as well as fast TLM simulation, thus significantly enhancing the verification flow. Students are also given examples of how the two coding styles can be combined, depending on the needs of the target design.

By mastering different HLS directives and coding practices, students generate families of implementations across various cost-performance points. Embodying an "engineer-in-the-loop" approach, the programming assignments lead students to engage interactively with Catapult HLS, compiling code, inspecting schedules, adjusting directives, and recompiling. By semester's end, the students possess a robust library of verified IP blocks and a reproducible methodology for translating algorithmic code into timing-clean RTL, effectively minimizing design effort and maximizing reuse. This comprehensive and structured approach underpins the extensive DSE that is the main goal of the SoCP final project.

4 FINAL PROJECT - COMPONENT-LEVEL DSE

During the final month of the semester, students shift their focus towards a comprehensive final project, which may be completed in teams of two students. The main goal of the project is the design, optimization, and system-level integration of hardware accelerators into a fully functional SoC. This project builds upon the foundational skills developed throughout the course, including synthesizable hardware specification with SystemC, HLS-driven DSE, and hardware/software co-design. The project is structured as a competitive, yet collaborative DSE among teams and is split into two parts: a component-level DSE and a SoC-level DSE.

In the first part of the project, each team individually develops a synthesizable design for one of the four convolutional layers of DWARF-7, a lightweight CNN for image recognition developed by our research group (Figure 2). The teams start from a provided baseline implementation of the accelerator and conduct extensive DSE to optimize for latency and FPGA resource utilization, with the ultimate goal of deriving three distinct Pareto-optimal designs (labeled FAST, MEDIUM, and SMALL). To validate their designs, teams submit their implementations daily for automated testing. Each valid implementation must satisfy three correctness requirements: (1) functional correctness by matching floating-point results at the behavioral level, (2) HLS-generated RTL correctness verified through co-simulation, and (3) accuracy, measured by achieving at least 50% classification accuracy over six input images. Each of these verification steps is automated thanks to the built-in features of the OOHLS flow described in Section 3: (1) leverages the fast-TLM simulation mode, while (2) and (3) utilize the signal-accurate simulation combined with RTL co-simulation.

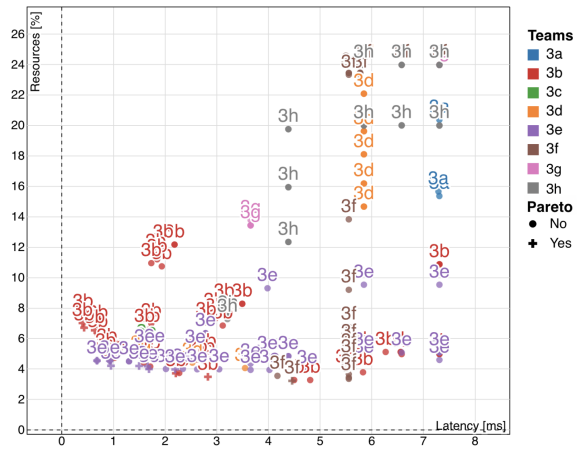

Figure 3: Component-Level DSE: the teams' progress is updated daily in the bi-objective optimization space on the project website.

To help the teams progress towards the Pareto frontier, the course revisits the same optimization knobs explored in the laboratory sessions on HLS, but now in a live competitive setting:

• **Source-code refactoring**. Splitting the convolution into tiled kernels, rearranging loops or re-ordering operators can exploit data locality. Alternative versions coexist under #ifdef SMALL / MEDIUM / FAST, making A/B comparisons painless.

• **HLS scheduling directives**. Adjusting the target clock, selectively unrolling inner loops, or pipelining long dependency chains changes the compute/memory ratio and enables balancing throughput with area.

• **Precision tuning**. Narrowing the arithmetic precision of activations or weights trims the sizes of multipliers and local memories, but must be verified to meet accuracy requirements.

• **Private Buffer and DMA**. Re-banking the private buffers and widening the Direct Memory Access (DMA) interface improves utilization of the on-chip BRAMs and processing throughput.

Teams are encouraged to refine their designs continuously throughout the duration of the project. A live Pareto chart hosted on the course website is updated daily to provide anonymized feedback on the performance of all designs in terms of two performance metrics: effective latency (from RTL simulation) and FPGA resources (aggregated BRAM, LUT, FF and DSP use). These metrics define the position of each design point in the bi-objective optimization space, as shown in Figure 3 for teams assigned to Layer 3. The quality of each design is evaluated in the context of the work done by the entire class: Pareto-optimal implementations receive the highest score, while the penalty for Pareto-dominated implementations is proportional to the distance from the Pareto frontier. This system fosters both competition and self-improvement, as teams aim to maintain their designs on the Pareto frontier of the class while privately benchmarking their progress against peers. Because the evaluation loop is automated, students experience an authentic engineer-in-the-loop workflow: code in the afternoon, push changes before midnight, wake up to a refreshed Pareto plot, and decide the next changes to make. We incentivize this type of continuous self-improvement by awarding a small amount of extra credit for each day that a team improves one of its design by at least 5% in one objective direction.

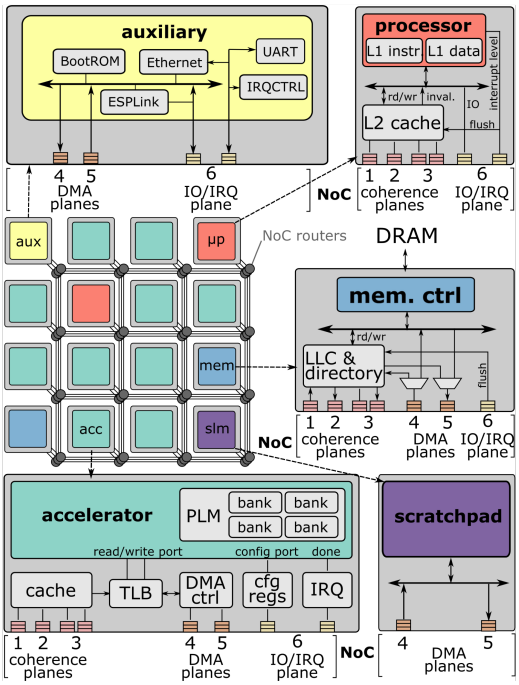

Figure 4: The ESP architecture and the main types of tiles.

5 FINAL PROJECT - SYSTEM-LEVEL DSE

After the completion of the first part of the final project, the class has effectively developed a large library of accelerator implementations for the DWARF7 CNN. In the second part, the student teams are tasked with using this library of accelerators to conduct an SoC-level DSE for the full DWARF7 application. While we have used this structure for the final project for many years, recent developments have shifted the DSE from a merely conceptual approach to leveraging real prototyping.

5.1 Prior Approach

Composing many accelerator designs into a single SoC is a difficult task, especially for students who are just gaining exposure to SoC design, with limited time to complete a project. For this reason, for many years, the second part of the project conducted the SoC-level DSE using simple mathematical models to evaluate the performance of a chosen SoC architecture. Specifically, we computed the area of the full system as the sum of the area of the selected accelerators and the throughput of the system as the minimum throughput supported by any chosen accelerator.

Although this approach allowed students to reason about component selection at a conceptual level, it does not expose them to the complex interactions that occur when combining many different components into one system. Moreover, it does not equip them with any additional practical skills for system design and programming. Hence, we decided to create a new version of this part of the project that allows students to actually prototype their target systems.

Figure 5: The ESP GUI, labeled with options students can change during their SoC-level DSE.

5.2 Background on ESP

Our attempt to bring real SoC prototyping into the classroom is based on the use of ESP, an open-source platform for SoC design developed by our research group over the last 12 years [8, 30] that has been used to design complex heterogeneous SoCs [10]. ESP combines a scalable SoC architecture with a flexible system-level design methodology to enable truly agile SoC design. This agility is key to enabling SoC prototyping in a classroom setting.

As shown in Figure 4, the ESP SoC architecture is a heterogeneous tile grid built on top of a 2D-mesh network-on-chip (NoC). There are 5 main types of tiles. The **Processor Tile** instantiates an open-source CPU core from one of three available options: 1) the 32-bit SPARC Leon3 core [11], 2) the 32-bit RISC-V Ibex core [29], and 3) the 64-bit RISC-V Ariane core [41]. The **Memory Tile** contains a channel to external memory – through a DDR controller on FPGA-based designs – and a partition of the ESP last-level cache, which orchestrates the system's coherence protocol. The **Auxiliary Tile** contains various peripherals (e.g., Ethernet, UART) and miscellaneous components (e.g., interrupt controller, bootROM). The **Scratchpad Tile** contains additional software-managed memory that can be used to expand the on-chip storage capacity. Finally, the **Accelerator Tile** instantiates a loosely coupled accelerator [17], which executes coarse-grained tasks on large datasets. The Accelerator Tile provides several pre-designed *platform services* [8] (e.g., DMA, interrupts, coherence, configuration registers, debug units [40]) that free designers from the need to worry about complex system-integration aspects.

The ESP methodology consists of two main pieces: the accelerator design flows and the SoC prototyping flow. The accelerator design flows greatly simplify the creation of new IPs that automatically integrate into the ESP architecture. They include flows based on RTL design, HLS with SystemC or C++, as well as flows based on some domain-specific languages [22]. They provide an interactive script that generates a customized skeleton for the accelerator's implementation. So long as users work within the confines of this skeleton, the accelerator automatically integrates into the ESP architecture [21]. The SoC design flow enables users to seamlessly compose their mix of tiles and configure various other parameters with the assistance of the ESP graphical user interface (GUI). All RTL for the SoC is generated by the back-end of the GUI, and a single `make` target is sufficient to generate an FPGA bit-stream for the target SoC architecture.

5.3 Collaborative Engineering with ESP

Fall 2023. In this iteration of the course, we used ESP for the first time. Because of the complexity involved in introducing such a tool to a class of over 50 people, we restricted the scope of SoC prototyping for the first year. Specifically, we tasked students with creating an SoC prototype that contains one instance of one of their accelerators – alongside a single CPU, memory, and auxiliary tile – within a small 2×2 ESP system. While the student teams were limited to this 2×2 SoC architecture, there were still many parameters to use in the DSE. As shown in Figure 5, with the help of the ESP GUI, teams could change: 1) the selected accelerator implementation, 2) the position of each tile in the SoC, 3) the selected CPU core, 4) the presence and size of the cache hierarchy, 5) the presence of a private cache for the accelerator, 6) the NoC bitwidth for coherence traffic, and 7) the NoC bit-width for DMA traffic. The teams created FPGA prototypes of their target SoC, which they evaluated by running a bare-metal application that we provided, which offloads part of the CNN inference to their accelerator. The teams could also optimize the application to improve performance, enabling a holistic HW/SW DSE.

Creating this first iteration of an ESP-based final project had several challenges:

• **Accelerator interface.** We needed to modify the accelerator design provided in Part A to be compatible with ESP. This involved changing the memory interface from AXI, which we have historically

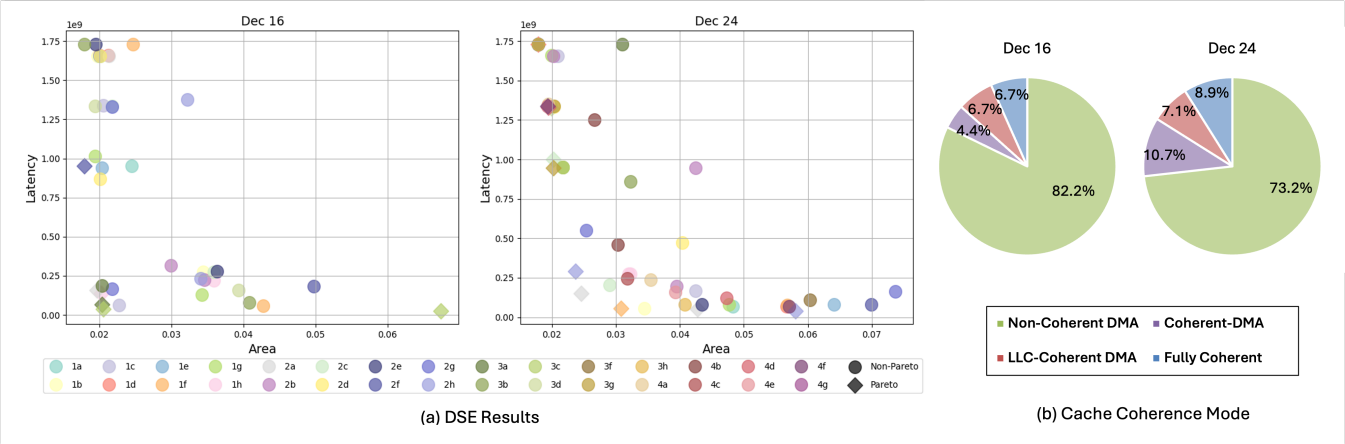

Figure 6: SoC-Level DSE across two submission days: (a) Latency-area results. (b) Usage of different accelerator cache-coherence modes.

used, to the native ESP interface. We also changed the accelerator's configuration interface.

• **Multi-User Setup.** With all its dependencies, ESP is relatively large, and it would not be feasible for each team to maintain a copy on our course server. We made modifications to ESP to allow multiple users to work from a single installation; this also ensured that all teams were working with the same version of ESP.

• **Compute Resources.** Since bit-stream generation is relatively intensive, we restricted each team to running a single bit-stream generation process at any given time.

• **FPGA Deployment.** With over 50 students in the class, allowing students to test their designs on FPGAs was a challenge. Our lab has 5 copies of one of the FPGAs supported by ESP; these FPGAs are connected to a router to allow remote access. We partitioned the teams into 5 groups; each group was assigned to one of the FPGAs, and teams in that group could access a shared spreadsheet to reserve up to 4 hours of board time per day (with a maximum of 1 hour per each 6-hour block of the day).

Overall, this iteration of the project was successful. A vast majority of teams were able to generate a functional SoC and run a software application on top of it. This provided us with the foundation for scaling up the complexity of the project in the following year.

Fall 2024. In the last fall semester, we scaled up the complexity of the project and emphasized *collaborative SoC engineering*. In particular, we allowed each team to use accelerators developed by other teams and to prototype ESP SoCs larger than 2×2 tiles. On the last day of the component-level DSE, we evaluated the three final accelerator designs of each team to create a class-wide *accelerator library*. We installed the 93 synthesized accelerator designs (three each from 31 teams) within the global instance of ESP on each course server. From the ESP GUI, then, each team could select any accelerator from the entire library with one caveat: to accelerate the layer of the CNN assigned to a team in Part A, the team must use one of their own accelerator designs. This iteration of the project also came with new challenges:

• **Providing Design Visibility:** The post-HLS implementations are not human-readable and therefore do not provide any information to the SoC designer about implementation details of the accelerator. This is important because certain optimizations of the accelerator design affect the data type and layout, which in turn requires modifications to the software that invokes the accelerator. To enable teams to understand the proper way to invoke an accelerator designed by another team, we also made the test-bench of each installed accelerator available. Future iterations of the project could further constrain the accelerator design to simplify the software integration.

• **Uniquifying Module Names:** Because teams started from the same baseline design, the HLS tool creates modules with the same names, even for different designs. To avoid conflicts when synthesizing the full SoC, we modified the HLS scripts to prepend a unique name (based on the team's ID) to every module.

5.4 Fall 2024 Project Results

Overall, this iteration of the project was also very successful. We allowed the 31 teams in the class to submit two SoC designs (FAST and SMALL) on two dates: December 16 for an intermediate evaluation and December 24 for the final submission. Figure 6(a) shows the latency-area Pareto frontier from the designs submitted on these two days. The color of each point denotes the ID of the team that submitted that design; the number in the ID indicates the layer to which they were assigned in the first part of the project. In addition to different hardware configurations, the teams could also explore different optimizations from software. For example, it was possible to change the *cache coherence mode* [42] used by the accelerator, which controls the level of the memory hierarchy from which the accelerator fetches its data; ESP supports dynamic selection of this mode at runtime. As shown in Figure 6(b), more teams explored different coherence modes for the second submission.

Of the 62 designs submitted on December 24, 53 (85%) were fully functional. Figure 7(a) shows the SoC designs that accelerated a given number of layers, ranging from zero (a pure software implementation) to four (all convolutional layers accelerated in hardware). We can see that from the first submission to the second one, more teams were able to successfully accelerate multiple layers; furthermore, more teams chose to accelerate zero layers, as they realized they can achieve minimal area by not using any accelerators. In the final submission, 25 designs (40%) accelerated more than one layer; eight designs (13%) accelerated all four layers.

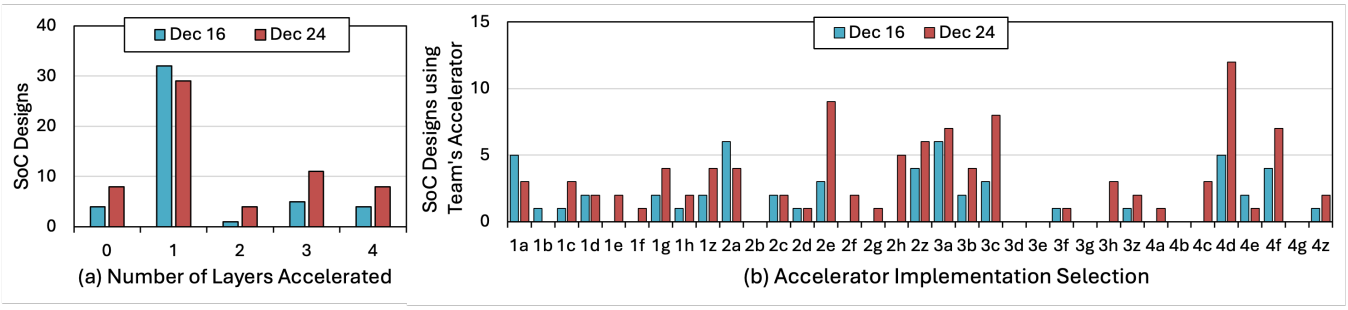

Figure 7: Key metrics from submitted SoC designs: (a) number of layers accelerated per design and (b) usage of accelerators in the course "library".

Figure 7(b) shows the usage of each team's accelerators across all SoC designs. We see a nice variation of usage of designs across the whole library. The teams were awarded a small amount of extra credit for each SoC design that used one of their accelerators. This incentivized collaboration; we saw teams posting on the course forum offering assistance to anyone using their design. One team's accelerators were used in as many as 12 SoC designs. Overall, 31 (50%) of the SoCs from the final submission used an accelerator designed by a different team. This exceeded our expectations for collaboration in the first edition of this project.

6 CONCLUSION

We presented recent updates to the System-on-Chip Platforms course at Columbia University that focuses on enabling collaborative SoC engineering in the classroom. In particular, transitioning the course to OOHLS-based design with Matchlib and Catapult HLS greatly raises the level of abstraction in hardware design. In the final project, students leverage this methodology to design accelerators for a CNN application. Thanks to ESP, students could combine their accelerators with those designed by other students to create complete SoCs, prototype them on FPGA, and deploy complex software applications, all within just a few weeks. We encourage instructors from other universities to adopt our final project infrastructure and to reach out to us for any assistance in doing so.

Acknowledgments. The authors thank present and past members of the SLD Group at Columbia. They gratefully acknowledge the support received by the group from DOD, DOE, NSF, Cadence, Intel, Qualcomm, Siemens, and Xilinx over the years. They also thank the students of SoCP over the years for their eager engagement with the course material and assignments.

REFERENCES

[1] Chips Alliance. 2025. Constructing hardware in a Scala embedded language (Chisel). https://github.com/chipsalliance/chisel. (2025).
[2] ARM. 2020. AMBA AXI and ACE Protocol Specification. https://developer.arm.com/documentation/ihi0022/h. (2020).
[3] K. Asanović and D. A. Patterson. 2014. Instruction Sets Should Be Free: The Case For RISC-V. Tech. rep. UCB/EECS-2014-146. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.
[4] B-lang-org. 2025. Bluespec compiler. https://github.com/B-Lang-org/bsc. (2025).
[5] P. Bannon et al. 2019. Compute and redundancy solutions for the full self-driving computer. In *Hot Chips: A Symp. on High Performance Chips*.
[6] D. C. Black et al. 2009. *SystemC: From the Ground Up, Second Edition*. Springer.
[7] L. P. Carloni. 2015. From latency-insensitive design to communication-based system-level design. *Proc. of the IEEE*, 103, 11, 2133–2151.
[8] L. P. Carloni. 2016. The case for Embedded Scalable Platforms. In *Proc. of the Design Automation Conf. (DAC)*. 17:1–17:6.
[9] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20, 9, 1059–1076.
[10] M. Cassel dos Santos et al. 2024. A 12nm Linux-SMP-capable RISC-V SoC with 14 accelerator types, distributed hardware power managment, and flexible NoC-based data orchestration. In *Intl. Solid-State Circuits Conf. (ISSCC)*, 262–264.
[11] Cobham Gaisler. 2025. Leon3 processor. www.gaisler.com/products/leon3. (2025).
[12] F. Conti et al. 2023. A 12.4TOPS/W @ 136GOPS AI-IoT system-on-chip with 16 RISC-V, 2-to-8b precision-scalable DNN acceleration and 30%-boost adaptive body biasing. In *Intl. Solid-State Circuits Conf. (ISSCC)*, 21–23.
[13] Philippe Coussy and Adam Morawiec. 2008. *High-level synthesis: from algorithm to digital circuit*. Springer.
[14] W. J. Dally, Y. Turakhia, and S. Han. 2020. Domain-specific hardware accelerators. *Communications of the ACM*, 63, 7, 48–57.
[15] M. Ditty, A. Karandikar, and D. Reed. 2018. Nvidia's Xavier SoC. In *Hot Chips: A Symp. on High Performance Chips*.
[16] Siemens EDA. 2022. Catapult High-Level Synthesis and Verification. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/. (2022).
[17] E. G. Cota et al. 2015. An analysis of accelerator coupling in heterogeneous architectures. In *Proc. of the Design Automation Conf. (DAC)*. 1–6.
[18] L. P. Carloni et al. 2019. Teaching heterogeneous computing with system-level design methods. In *Proc. of the Workshop on Computer Architecture Education*.
[19] Michael Fingeroff. 2010. *High-Level Synthesis Blue Book*. Mentor Graphics Corp.
[20] F. Gao et al. 2023. DECADES: a 67mm2, 1.46TOPS, 55 Giga cache-coherent 64-bit RISC-V instructions per second, heterogeneous manycore SoC with 109 tiles including accelerators, intelligent storage, and eFPGA in 12nm FinFET. In *Custom Integrated Circuits Conf. (CICC)*.
[21] D. Giri et al. 2021. Accelerator integration for open-source SoC design. *IEEE Micro (Special Issue: FPGAs in Computing)*, 41, 4, 8–14.
[22] D. Giri et al. 2020. ESP4ML: platform-based design of systems-on-chip for embedded machine learning. In *Proc. of the Design, Automation and Test in Europe Conf. (DATE)*, 1049–1054.
[23] G. Gupta et al. 2017. Kickstarting semiconductor innovation with open source hardware. *IEEE Computer*, 50, 6, 50–59.
[24] P. Hübner et al. 2025. Apple vs. oranges: evaluating the apple silicon M-Series SoCs for HPC performance and efficiency. (2025). arXiv: 2502.05317 [cs.AR].
[25] IEEE. 2012. SystemC Standardization Working Group. 1666-2011 - IEEE standard for standard SystemC reference manual. (2012).
[26] B. Khailany et al. 2018. A modular digital VLSI flow for high-productivity SoC design. In *Proc. of the Design Automation Conf. (DAC)*. 1–6.
[27] D. Koeplinger et al. 2018. Spatial: a language and compiler for application accelerators. *SIGPLAN Not.*, 53, 4, 296–311.
[28] E.A. Lee and A. Sangiovanni-Vincentelli. 1998. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17, 12, 1217–1229.
[29] lowRISC. 2025. Ibex RISC-V Core. https://github.com/lowRISC/ibex. (2025).
[30] P. Mantovani et al. 2020. Agile SoC development with open ESP. In *Proc. of Intl. Conf. on Computer Aided Design (ICCAD)*, 1–9.
[31] D. Matzke. 1997. Will physical scalability sabotage performance gains? *Computer*, 30, 9, 37–39.
[32] T. Murata. 1989. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77, 4, 541–580.
[33] NVIDIA. 2017. NVIDIA Deep Learning Accelerator (NVDLA). In *www.nvdla.org*.
[34] NVLabs. 2025. MatchLib. https://github.com/NVlabs/matchlib. (2025).
[35] P. Scheffler et al. 2025. Occamy: a 432-core dual-chiplet dual-HBM2E 768-DP-GFLOP/s RISC-V system for 8-to-64-bit dense and sparse computing in 12-nm FinFET. *IEEE Journal of Solid-State Circuits*, 60, 4, 1324–1338.
[36] C. Schmidt et al. 2022. An eight-core 1.44-GHz RISC-V vector processor in 16-nm FinFET. *IEEE Journal of Solid-State Circuits*, 57, 140–152.
[37] Y. S. Shao et al. 2015. The Aladdin approach to accelerator design and modeling. *IEEE Micro*, 35, 3, 58–70.

[38] SIEMENS_EDA. 2024. Siemens EDA and Wilson Research Group, 2024 Functional Verification Study. https://resources.sw.siemens.com/en-US/white-paper-2024-wilson-research-group-ic-asic-functional-verification-trend-report/. (2024).

[39] M. B. Taylor. 2018. Basejump STL: SystemVerilog needs a standard template library for hardware design. In *Proc. of the Design Automation Conf. (DAC)*. 1–6.

[40] G. Tombesi et al. 2023. SoCProbe: compositional post-silicon validation of heterogeneous NoC-based SoCs. *IEEE Design Test*, 40, 6, 64–75.

[41] F. Zaruba and L. Benini. 2019. The cost of application-class processing: energy and performance analysis of a Linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 27, 11, 2629–2640.

[42] J. Zuckerman et al. 2021. Cohmeleon: learning-based orchestration of accelerator coherence in heterogeneous SoCs. In *Proc. of the IEEE/ACM Symp. on Microarchitecture (MICRO)*, 350–365.