ESL Flow

- The fundamental principle of ESL design is managing abstraction refinement and complexity while preserving design intent.
- Throughout the design flow, the design itself will be represented at various abstraction levels:
  - Product market requirements
  - Functional specification
  - Architectural model
  - Hardware and software design specifications
  - Hardware and software functional and behavioral models
  - RTL and software models
  - Cell-level model and embedded production software
  - Layout database

ESL Flow

- The product market requirements dictate the end-user functional and physical requirements of the product.
- The functional specification describes the functional requirements of the design from an opaque box perspective.
- The architectural model captures the HW/SW partitioning of the design.
- The hardware and software design specifications describe the implementation requirements of the design from a clear box view.
- The behavioral hardware and software models implement the algorithms required of the subsequent models without timing accuracy or, for functional models, no timing whatsoever.

ESL flow

- The RTL and software models are timing-accurate implementations of the behavioral hardware and software models.
- The cell-level model is machine-synthesized from the RTL, whereas the embedded production software is either machine-translated from the software model or, as is more generally the case today, handwritten.
- Finally, the layout database captures the structural geometry of all hardware elements—and their interconnect—that are to be fabricated on each layer of the multilayer integrated circuit.

ESL Flow - Design Intent

- Design intent originates in the human mind, collectively from marketing, system architects, and designers.
- As it flows through the design process it may be likened to information transmitted through a communication channel because design intent is information.
- Unlike a communication channel that is supposed to preserve a transmitted message, the message that flows through a design channel is iteratively refined and augmented.
- Each refinement improves the fidelity of its representation—a model—until the final refinement is produced: layout database and production software.

ESL Flow - Design Intent

- Another way of viewing design intent transformations is that at each stage of the refinement process a few degrees of freedom are removed, until all we are left with is a particular implementation of the specification.
- The decisions that are made in this process are architectural decisions. Although we tend to think of architecture as being only at the highest abstraction levels, architecture is present at all stages in the design flow.

ESL Flow - Design Intent

- The earlier stages of refinement are manual processes: specifications down to RTL and design software. The later transformation stages are partially automated: cell-level netlist, layout, mask generation (for hardware), and C/C++ code generation (for software).
- Over time, upstream design transformations will be automated and become mainstream, such as behavioral synthesis to RTL, or will be based on reuse of IP blocks, such as the selection of embedded processors and software to implement a function.

ESL flow - Steps

- The ESL design flow may be divided into six steps that parallel the aforementioned abstraction refinement:
  - Specification and modeling
  - Pre-partitioning analysis
  - Partitioning
  - Post-partitioning analysis and debug
  - Post-partitioning verification
  - HW/SW implementation
- Although we describe the ESL flow as a top-down flow, starting at specifications and ending up with implementations in hardware and software, this is an idealized concept that is rarely used in its strict form in real designs of real systems.

ESL flow - Steps

- The partitioning step includes HW/SW partitioning— distributing design implementation components between hardware and software modules; software/software partitioning—distributing software modules among application-specific and general-purpose processors; and hardware/hardware partitioning—dividing the hardware content into several concurrent and commu- nicating blocks.
- Design teams use combinations of bottom-up, top-down, and middle-out design flows.

ESL flow - Steps

- Sometimes, a design may start with either an existing product or a "platform" that is used as a basis for creating a derivative system product. In addition, IP, both hardware and software, and components and whole subsystems, may be purchased or acquired to complement the existing platform(s) or existing completed design.
- This is a kind of "middle-out" design flow because some existing components and subsystems (both hardware and software) are used without change (especially IP components acquired from outside).
- Some components may be extensively modified, such as configurable and extensible IP.

ESL flow - Steps

- Other components may be created from scratch for the derivative product using both top-down and bottom-up flows.
- Sometimes, a new component is created from scratch as a direct hardware implementation in RTL or a software implementation in C/C++ or Java, without any system-level design steps at all. This is clearly a bottom-up kind of flow.

Specifications and Modeling

- Writing specifications, and modeling, is the process of developing documents that describe system or product intent and constraints, and their translation into a variety of executable and declarative models.
- The top-level specifications and requirements are usually intentionally written in a natural language, rather than in a synthetic language.
- Sufficient ambiguity must remain in each specification or model to allow the model at the next level of refinement to meet its functional (and non-functional) requirements without constraining it to a particular implementation.

Specifications and Modeling

- Once a specification having an appropriate balance of precision and ambiguity is written, it serves as the basis for pre-partitioning analysis.
- The specification cannot be written in the absence of implementation concerns because it is clearly possible to specify requirements that cannot be implemented.
- he ambiguity must be preserved at each stage in order to truly explore the space of design alternatives. Any "middle-out" design methodology here, in which there is a desire to reuse large amounts of existing IP—designs, platforms, libraries, and operating environment models—has a major impact on the specification of the system and its division into sub-specifications.

Specifications and Modeling

- Although hardware and software have traditionally been developed somewhat in isolation from one another, with software requiring a relatively stable hardware platform before software development proceeds, this is no longer possible in an ESL flow.
- The objective is to develop hardware and software concurrently to avoid late stage integration surprises.
- Platforms are used to manage the complexity of modern designs by employing pre-verified hardware blocks and software modules.
- In an ideal world, software models should be written first so that pre-partitioning analysis can be used to define the hardware requirements of the software.

Specifications and Modeling

- In addition to the natural language specification, there is a class of executable specifications.
- An executable specification is a functional or behavioral description of a system that, when run as a computer program, exhibits the intended behavior of the design as observed from an opaque box perspective.
- The specifications that fall into this class are the executable architecture specification and executable design specification.
- The executable architecture specification serves as a vehicle for exploring design alternatives and demonstrating design concepts. It may be written at various abstraction levels.

Specifications and Modeling

- The executable design specification, on the other hand, reflects key microarchitecture structural decisions and exhibits high-level clear box behaviors. Transaction-level modeling may be used in the creation of an executable specification to gain substantial performance.
- To avoid reimplementing behaviors repeatedly during the modeling process, once behavior at a particular abstraction level is implemented, it may subsequently be refined but always reused, not re-implemented. We refer to this as “single sourcing”.

Specifications and Modeling

- There are a variety of ESL specification languages, each having its strengths and weaknesses.
- These include—in no particular order—MATLAB M-Code, SystemC, SystemVerilog, Specification and Description Language (SDL), UML, eXtensible Markup Language (XML), and BlueSpec.
- Model-based development replaces the traditional design flow of requirements -> analysis -> design -> implementation -> verify with one of successive declarative model refinement.
- Each model is written at a more refined (i.e., less abstract) level than the previous, and the additional feature level of detail verified. The model is written using a declarative language like UML. This approach postpones consideration of implementation details until quite late in the design process, thereby enabling a platform-independent design.

Pre-Partitioning Analysis

- Pre-partitioning analysis is the process of exploring the spectrum of algorithmic tradeoffs.
- These tradeoffs are in the time, space, power, complexity, and time-to-market domains.
- In the time domain, some algorithms suitable for meeting product requirements are highly parallel in nature, whereas others are serial.
- In the spatial domain, some algorithms require much more storage for data items, whereas others require substantial storage for control, such as algorithmic control parameters and user interface controls.

Pre-Partitioning Analysis

- In the power domain—and its integral cousin, energy— speculative operations gain performance while wasting power on every unused speculated operation. Caching data previously fetched from slower storage elements and results computed from earlier operations saves power if the caching algorithms use accurate predictors.
- Sometimes the choice of algorithm has a large impact on energy consumption—one that downstream compilation and optimization will rarely discover.
- In the time-to-market domain, the ideal solution in all other dimensions may result in the product arriving to market too late. Hence, compromises are often required to meet this all-important business need.
- During pre-partitioning, we need to examine costs and benefits of these options in preparation for the next step: partitioning.

Pre-Partitioning Analysis

- The following will be considered: static analysis, platform-based design, dynamic analysis, algorithmic analysis, and various analysis scenarios and models.
- In the absence of an executable specification, static analysis may be used to assess design tradeoffs.
- A number of static analysis techniques have been developed over the years, such as system complexity analysis and "ility" analysis (reliability, maintainability, usability, and criticality)
- System complexity analysis is based on software engineering function point analysis, a means of predicting characteris- tics of code during the early design stages. When complexity analysis is applied to a pre-partitioned design, system characteristics such as power, performance, and development cost may be forecast within reasonable error bounds.

Pre-Partitioning Analysis

- Platform-based design is a reuse-intensive design style wherein major elements of the system are incorporated from other sources, often substituting opaque box- equivalent components that deliver higher performance, consume less power, or otherwise improve upon the legacy elements without sacrificing functional requirements.
- A significant challenge of platform-based design is the inherent bias of the design team to model the new design architecture like the legacy architecture.

Pre-Partitioning Analysis

- Dynamic analysis affords us the opportunity to examine more accurate information only available from executable models, such as time-based performance (latency, throughput) and the influence of arbitration and scheduling policies.
- We can also derive computational, communications, and power burdens from the simulation of executable models.
- The purpose of algorithmic analysis is to estimate resource requirements and system operational parameters such as computation load (fixed- and floating-point), data transport pipe requirements, and bit error rates.

Pre-Partitioning Analysis

- It is important to remember the proper field of use of executable models at this stage. Because they are executable on a host or in some modeling environment, they by necessity contain some implementation artifacts. Therefore, it is important to use them to assess characteristics of the design that reflect only high-level specification characteristics that are relatively independent of the aforementioned artifacts.
- Pre-partitioning analysis is often either avoided or done in a cursory fashion so the "real job" of partitioning and detailed design and implementation can be started more rapidly. It is important, however, to take advantage of the known tools and methods capabilities in this space because time spent in this analysis can steer the subsequent stages to more likely parts of the design space and reduce the time spent working on unattractive alternatives.

Partitioning

- Partitioning is the process of choosing what algorithms (or parts thereof) defined in the specification to implement in software components running on processors, what to implement in hardware components, and the division of algorithms within the software and hardware components.
- Historically, complexity is greater in software implementations and less in hardware implementations.
- Beyond this, the types of processors (CISC, RISC, DSP, or configurable) on which the software is to run is determined, along with the numbers of processors, how they communicate, and how other hard- ware blocks communicate.
- This may be both a top-down and bottom-up process, in which a platform or aggregate of IP blocks is pre-chosen before partitioning to become the target for partitioning.

Partitioning

- In functional decomposition, we begin with a specification that is as free from implementation artifacts as possible.
- An implementation artifact is a behavior exhibited by a model that is purely a side effect of its implementation, yet imposes no requirement on the design to be implemented.
- Although a natural language specification may suffer from unintended ambiguities, it contains no implementation artifacts because it is not an implementation.
- On the other hand, implementation artifacts are inherent in executable specifications.

Partitioning

- The specification must also expose substantial application-level parallelism for mapping. These specification requirements are met by two approaches:
  - Use of a functional concurrent executable specification language (a representative language is Simulink)
  - Use of a sequential language and tools that can automatically extract parallelism.
- Extracting parallelism from a sequential language is both promising and difficult.

Partitioning

- Extracting parallelism is attractive because legacy sequential descriptions, such as C models, that were originally written for sequential execution may be realized as performance-capable implementations through automatic decomposition, parallelism extraction, and HW/SW mapping.
- It is quite difficult because the dependency analysis on which it rests is restricted to parallelizing array references within loops under many restrictions today. These restrictions naturally lead to hardware-dependent solutions that must be considered during design space exploration.

Partitioning

- An architecture description makes use of elemental building blocks, each of which implements a kind of behavior: general-purpose or algorithm-specific computation, data storage, and communication.
- Some of these building blocks are fixed CPU, configurable CPU, custom hardware, standard hardware, buses, memories, operating systems and services, APIs, and protocol stacks.
- We make use of these building blocks in mapping a functional description to an overall architecture, often a platform.
- The initial mapping of functions to architectural elements often mirrors the final selected hardware partitioning. However, in more complex designs, functions will not naturally map to hardware components but rather to, for example, objects of an object-oriented software implementation.

Partitioning

- To amortize the cost of designing these more complex architectures, they are realized as platforms-parameterized HW/SW topologies, configurable for a range of applications. The use of platforms in the consumer electronics market is quite mature.
- Another consideration for architectural description is the modeling abstraction level. The least abstract hardware model in ESL design and verification is RTL.
- Next up in abstraction is the transaction-level model, itself further divided into PV, PV + T, and cycle-accurate.

Partitioning

- A concurrent application may be partitioned through either successive refinement or use of an explicit mapping notation understood by models and synthesis.
- Each step of the successive refinement process adds sufficient detail at each step to transform the functional model into a mapped model with a single multilevel modeling mechanism.
- Timing fidelity of two orthogonal aspects of the model—communication and computation—is improved through each refinement.
- There are three timing resolutions: untimed, approximate-timed, and cycle-timed. The SystemC library facilitates this refinement quite naturally with its inherent timing abstractions

Partitioning

- The hardware partition may be realized with one or more processors and behave as a distributed system.
- In an embedded system, it may be event- or time-driven (aperiodic or periodic points in time) and control- or data-dominated.
- More and more, system-level hardware design is becoming nothing more than platform configuration, while the bulk of bottom-up design moves into software. Any time we are looking at fresh design, the opportunity to insert novel bugs presents itself.
- This means that software verification becomes much more important for system-level design.
- Another aspect of hardware partitioning is the refinement of each model to meet system-level scheduling requirements. These requirements originate in the control and data dependencies of concurrent processes.

Partitioning

- The software partition is composed not only of the operating system, libraries, and middleware, but of the applications that often implement the majority of the system functionality.
- It may itself be partitioned in a number of different ways: across multiple processors, across multiple tasks, and across various memories. When distributing functionality across multiple processors, they may be symmetric or heterogeneous CPUs.
- The choice of CPUs and their mix is determined by the metrics to be optimized; power and performance are the most common driving metrics.
- A typical configuration is composed of a general-purpose CPU for handling all user interface and networking functions and a DSP for processing radio-frequency and media data streams.
- On any particular processor, software functions may be further partitioned across tasks, processes, and threads.

Partitioning

- Once so partitioned, a scheduling algorithm must be chosen.
- Both CPU partitioning and task partitioning impose communication burdens that must be considered when optimizing for overall system performance.
- Finally, in distributing code and data across multiple memory subsystems, we need to consider the virtualization of large, slow, inexpensive storage technologies using small, fast, relatively expensive memories.
- Such caching techniques are well studied, but the impact of assigning data structures by their spatial and temporal access patterns becomes a significant concern for system-level software design.

Partitioning

- Reconfigurable computing is becoming a strong contender to bridge the yawning chasm between software and hardware as determined by cost, speed, and power.
- The hardware computing elements may be reconfigured once when the system is manufactured, yearly for upgrades, enhancement, and bug fixes, or every few thousand clocks to adapt to the running task.

Partitioning

- A substantial cost of reconfigurable computing elements is the difficulty of programming them. Neither the traditional hardware implementation flow: synthesis, place and route, nor the traditional software implementation flow: compile, execute, debug, support reconfigurable computing elements, such as FPGAs.
- The two main categories of reconfigurable computing architectures are
  - a reconfigurable array operating as a functional unit of a control processor
  - a reconfigurable array operating as a coprocessor attached to a main processor.

Partitioning

- Communication can often be the performance bottleneck in bus-based systems.
- Two implementation approaches are available to us for communication architectures:
  - Template instantiation.
  - Interface synthesis.
- Template instantiation matches one or more templates against functional and performance requirements, selects those that best match the system needs and instantiates each with carefully chosen parameters.
- For example, if a SIMD processor needs an interface to a general-purpose CPU, a processor-to-processor template may be instantiated with the required number and kind of buses (e.g., pipelined, serial, burst, 32-bit).

Partitioning

- The second approach, interface synthesis, has several requirements before it can be applied. First, the average throughput of the transmitting and receiving interfaces must be the same (i.e., no deep buffers are used).
- Second, the signal-level transmission protocol must have FSM semantics in order to use automata-based or language-based synthesis. If these two requirements are satisfied, a number of strategies are available to synthesize a communication interface.

Post-partitioning Analysis and Debug

- Post-partitioning analysis and debug is the second exploration stage wherein the effects of hardware and software partitioning are examined.
- During this step, archi- tectural models developed during pre-partitioning analysis are refined to reflect the partitioning choices.
- Algorithms that will run on a processor are implemented in a programming language (C, C ++ , and Java) and compiled for the target processor, or compiled to mimic the effects of the software running on the target processor.
- Algorithms that are to be implemented in hardware are modeled at the behavioral level in a hardware description language (SystemVerilog, Verilog, VHDL, and SystemC).
- During this phase, iteration may occur back to the partitioning stage to optimize the resulting system. This exploration is often called "design space exploration."
- Following closely in the footsteps of post-partitioning analysis is verification.

Post-partitioning Analysis and Debug

- Modeling the hardware and software of a full system is constrained by modeling objectives, and if those objectives are too broad they may not be met.
- These objectives may include design space exploration, design, validation, and verification. However, if we attempt to completely meet the needs of any one of these objectives, we must inevitably compromise the requirements of one or more of the others.
- The models developed for hardware and software must be able to execute cooperatively, possessing either compatible interfaces or using shims or wrappers to adapt each one to the others.
- Three options exist for modeling a software element:
  - Unified HW/SW model
  - Software-only model with hardware communication adapter
  - Discrete concurrent hardware and software models.

Post-partitioning Analysis and Debug

- Although the unified HW/SW model has the advantage that the fidelity of the mixed model may be restricted to what is absolutely required for the abstraction level of the model's objective, it suffers from a lack of separation of HW/SW concerns.
- The software-only model benefits from a clear separation of concerns but requires careful identification of all elements that dictate hardware requirements.
- The concurrent hardware and software model paradigm is quite versatile, running target machine code on a CPU model or running machine code on a partially timed instruction set simulator. However, this model is usually only employed at low abstraction levels.

Post-partitioning Analysis and Debug

- Once we have our initial partitioned system, we may start making more detailed performance, power, and cost estimates.
- This early partitioning often uses a network of communicating processes, where we vary the network itself for what-if analysis.
- Unbounded queues will be used to buffer differing traffic rates between elements so that we can observe high- and low-water marks while considering any given network configuration.
- For more detailed analysis, weights and time stamps may be applied to traffic data to gain early insight into performance and power envelopes.
- Once a given partition is chosen, we need to be sure the component interfaces can meet the traffic demands of the components, so we may actually implement the interfaces themselves to ensure they may indeed be implemented.
- We revert to the more abstract interfaces for continued analysis once we satisfy any implementation concerns.

Post-partitioning Analysis and Debug

- The modeling style of pre-partitioned model components must be compatible with that of the allocated components to effect a smooth transition to a partitioned set of models.
- The purpose of partitioning is not only to divide an algorithm into hardware and software components but to tackle the design challenge using the age-old divide-and-conquer approach.
- Because detail is subsequently added to each model through abstraction refinement, increasing its complexity (i.e., information content), each model must be subsequently partitioned.
- Partitioning a model and allocating each partition to hardware or software are independent tasks, allowing a mix of elements during analysis.

Post-partitioning Analysis and Debug

- Even pre-partitioned and post-partitioned models may be used together if their functionality is clearly separated from their communication interfaces.
- This allows us to explore the feasibility of implementation directions without having to pull the whole system down to a lower abstraction level.

Post-partitioning Analysis and Debug

- We use abstraction to limit the amount of information and complexity to be managed at any one time.
- However, without conventions in place defining abstraction levels, each engineer tends to choose their own boundaries.
- These conventions lead to agreed-upon standard abstraction levels that quantify, along each abstraction axis, what defines a given abstraction level. These axes were introduced in Chapter 2: temporal, data, functionality, and structural.
- OSCI also defined a standard set of useful abstraction levels: PV, PV + T, cycle callable, and RTL.

Post-partitioning Analysis and Debug

- In addition to standard abstraction levels, standard interfaces (APIs) are needed to facilitate communication among the models.
- These interfaces are also responsible for bridging the abstraction gap of the information transferred between models.
- Translating along the data axis is relatively straightforward, whereas translating along the temporal axis is more difficult.
- Information must be added when translating down in abstraction (greater fidelity) and removed when translating up in abstraction (less fidelity).
- When shifting up in abstraction, care must be taken not to generalize beyond the design space of the lower-level model.

Post-partitioning Analysis and Debug

- In addition to managing the abstraction level and structure of the models, we need to specify the communication requirements of each model.
- The communication interface of a model dictates the abstractions of information to be translated to and from the model and also its timing, concurrency, and model configurations.
- Separately specifying the functional and communication requirements of a model leads to many downstream benefits for design, verification, configuration, and reuse.
- Without orthogonal specifications for each, the cost of adapting models in a mixed abstraction environment is substantial.

Post-partitioning Analysis and Debug

- The interface specification should be the first part of a model that is documented because it is required both for new models as well as external IP. It should be written by those partitioning the design because it captures the initial requirements for inter-model communication.
- Traditional interfaces specified in hardware and software description languages are limited to static elements such as ports, methods, and data types.
- However, interface specifications should be extended with a declarative temporal language that allows legal sequences and scenarios to be described, facilitating both data and temporal checking of models during integration.

Post-partitioning Analysis and Debug

- The final subject we discuss in this section is dynamic and static analysis.
- a wide variety of models are required for post-partitioned analysis and debug.
- These are further diversified by the need to simulate with models that precede partitioning as well as with those that follow.
- The analyses that are commonly performed with these models are:
  - Functional analysis
  - Performance analysis
  - Interface analysis
  - Power analysis
  - Area analysis
  - Cost analysis
  - Debug capability analysis

Post-partitioning Analysis and Debug

- Functional analysis, most commonly the concern of functional verification, is also required at this stage to understand the size requirements of various storage elements.
- Initial performance analysis is performed to validate the partitioning and sizing of hardware and software elements.
- Interface analysis aims to discover early, long before implementation, whether or not particular module communication choices can be implemented.
- Power analysis takes advantage of this stage of design where the model first resembles a physical implementation, apportioning power budgets to each of the functional units and interfaces based on the results of simulating both representative as well as corner-case scenarios.

Post-partitioning Analysis and Debug

- Area analysis employs complexity metrics to estimate the die area requirements of the hardware elements.
- Cost analysis uses heuristics to estimate design costs, product costs, support costs, and complete lifetime costs.
- Finally, debug capability analysis examines the complexity and risk of functional errors of the chip design to determine what controllability and observability features are required from external pin interfaces of the physical chip implementation.

Post-partitioning Verification

- Post-partitioning verification is the process of demonstrating that the intended behavior of the hardware and software components of the design is preserved in their post-partitioned models.
- This is the first of two verification steps, the second being implementation verification. In both cases, the behavior of a model at a higher abstraction level must be compared with a model at a lower abstraction level.
- However, this raises an interesting question. Because only the behavior common to the two models may be compared, how is the refinement introduced in the lower abstraction model verified?
- This is best answered by examining the flows that comprise the verification flow: verification planning, verification environment implementation, and verification results analysis.

Post-partitioning Verification

- The purpose of verification planning is to capture in a document, possibly in a tool-readable format, the scope of the verification problem and its solution.
- The scope of the problem is described in a hierarchical fashion, beginning with specification analysis, followed by feature identification of both opaque box and clear box features, and ultimately quantified in a set of coverage models.
- This hierarchy nicely fits the ESL concept of abstraction refinement wherein soft design requirements are iteratively constrained toward an implementation.

Post-partitioning Verification

- The parallel within the verification flow is iteratively constraining the valid behavioral space of the design in order quantitatively to describe its operational space.
- Once described, this space may be explored both statically, using formal analysis, and dynamically, using simulation.
- The solution to the verification problem using static and dynamic techniques is written in the latter half of the verification plan, serving as the functional specification for components to be written for each.
- Property specifications for assertions and design requirements for the coverage, checking, and stimulus generation aspects of the simulation environment are recorded.

Post-partitioning Verification

- Once the scope of the verification problem has been quantified in the verification plan and the functional specification for the verification environment has been written, the verification environment must be implemented.
- The implementation burden is eased by modern High-Level Verification Languages (HLVL) such as e, SystemVerilog.
- Although these languages share a common set of verification constructs that address the three aspects of a dynamic verification environment—generation, checking, and coverage, such as constraint specification, constrained random generation, functional coverage, and assertions, only e at this time is an aspect-oriented language.
- Aspect-Oriented Programming (AOP) has proved useful for distributing the implementation of verification environment functionality across the objects that compose the environment, one of several opportunities for the separation of concerns that characterize AOP.

Post-partitioning Verification

- Once the verification environment has been implemented, it will be put to use, initially running bring-up simulations and throughout the rest of the project running regressions.
- A bring-up simulation or test is intended to exercise a small set of basic operations in order to expose those bugs that prevent the Design Under Verification [DUV] from operating at all. It is sometimes known as a "pipe cleaner."
- These runs need to be analyzed from both correctness and completeness perspectives. The correctness perspective, failure analysis, concerns itself with determining a common source of failure for a subset of runs.

Post-partitioning Verification

- This root-cause analysis is the second of three steps of debug: bug discovery, bug diagnosis, and bug repair.
- Bug discovery is the job of the stimulus generation and checking aspects of a dynamic verification environment or the model checker of a static verification environment.
- The completeness perspective, coverage analysis, concerns itself with understanding why particular coverage holes remain.
- The last topic of interest for post-partitioning verification is abstract coverage.
- Functional coverage measurement and analysis have been widely applied to implementation models, both hardware and software, for a number of years.
- Because we recommend starting verification much earlier in an ESL flow, we need to measure verification progress at this stage.

Hardware Implementation

- Hardware design implementation is the process of creating models that may be synthesized into gate-level models.
- The hardware models are usually written at the RT level, but behavioral synthesis technology will soon allow those models to be written at the behavioral level.
- There are still many choices that are being made at this stage, such as resource sharing and pipeline insertion, that can affect the performance of the system, so a certain amount of analysis will have to be performed to validate the choice.
- If synthesis is being used, it is also important to ensure that the connection back to the original description is preserved.
- There are five general hardware implementation options available to the design at this stage: extensible processors, DSP coprocessors, customized VLIW coprocessors, application-specific coprocessors, and ASIC and FPGA.

Hardware Implementation

- An extensible processor is a general-purpose CPU core that may be tailored to a specific application or set of applications with the addition of SIMD instructions, multiply–accumulate units, zero-overhead looping, dual load–stores, DSP hardware units, and multioperation instructions.
- An extensible processor does not usually require hardware design because the vendor's tool flow—from instruction definition to RTL implementation—performs the necessary translation steps.
- The production flow is usually parameterized with optimization targets to guide implementation decisions.

Hardware Implementation

- A DSP is a processor optimized to execute signal-processing algorithms, such as those applied to audio and video data streams.
- These optimizations include single-cycle multiply–accumulate, parallel memory access data paths and dedicated hardware support for zero-overhead loops.
- A DSP coprocessor is attached to a general-purpose CPU through a bus and mapped to the CPU's memory or I/O address space.

Hardware Implementation

- A customized VLIW coprocessor is optimized in the number and kind of functional units for executing a particular algorithm.
- The number of functional units is chosen to match the number of algorithm operations that may be performed in parallel.
- The kind of functional units are chosen based on the data types and operations to be performed.
- Several EDA vendors offer tools that generate a VLIW coprocessor from an algorithm specification, with a bus attachment to the primary CPU as with the DSP coprocessor.

Hardware Implementation

- An application-specific coprocessor makes use of a custom data path and FSM to implement a particular algorithm.
- A commercial tool flow will typically analyze an algorithm written in C or C ++ to generate the data path and FSM.
- Unlike coprocessors that are based on a DSP or VLIW processor, the form of an application-specific coprocessor is essentially unconstrained, although it has to be integrated back into the defined system architecture.

Hardware Implementation

- An ESL design flow is built on top of the traditional RTL flow: create RTL -> verify RTL -> synthesize RTL to gates -> verify timing -> place and route gates -> design rule check -> generate GDSII.
- The RTL typically is written with data path and control flow components, where the control flow is composed of interacting FSMs that multiplex data through the data path.
- The RTL becomes the data interchange between the existing RTL flow and the new ESL stages: system specification -> HW/SW partitioning -> virtual prototype -> transaction-level design -> transaction-level verification -> ESL synthesis to RTL.

Hardware Implementation - Behavioral Synthesis

- Before introducing ESL synthesis, we first discuss behavioral synthesis to distinguish the two. Behavioral synthesis takes an untimed (or partially timed) procedural algorithm description and schedules it using an FSM, hardware elements, and a clock.
- The description may be represented in the behavioral constructs of Verilog or VHDL, among other languages.
- Dependency analysis and resource constraints dictate the schedule selected for the hardware realization. There are several operations generally prohibited in RTL that are allowed in behavioral synthesis.
- Multicycle functions are allowed because FSMs and the requisite hardware are generated. Although loops with fixed indices are sometimes allowed by an RTL synthesis tool, much more freedom in loop structures is allowed by a behavioral synthesis engine.
- Last, because memory accesses require complex operations involving sequential row and column strobing, they are not allowed in RTL synthesis but are in behavioral synthesis.

Hardware Implementation - Behavioral Synthesis

- When behavioral synthesis was introduced commercially in the early 1990s, it ultimately failed because:
  - The input language was not appropriate.
  - Timing was too difficult to manage.
  - Verification was difficult.
- The input languages chosen were VHDL and Verilog, both suitable in some respects for procedural descriptions, but they are not the languages in which algorithm design is generally performed.

Hardware Implementation - Behavioral Synthesis

- Timing was unpredictable because the synthesis program had to estimate the number of logic levels that would fit within a clock cycle and construct an FSM whose state count accommodated the mapping of the procedure into the predicted clock cycles.
- However, the model responsible for logic-level estimation was often optimistic, resulting in combinatorial logic that was too deep for the cycle time in one or more states.
- It was extremely difficult to correct this problem without completely rewriting the algorithm.
- The third contributor to the demise of behavioral synthesis was that the verification environment written to verify the behavioral model reflected timing that was not preserved in the synthesized RTL.
- This required extensive changes to the verification environment to verify the RTL, preventing its direct reuse.

Hardware Implementation - ESL synthesis

- ESL synthesis may be seen as an evolutionary step beyond behavioral synthesis that addresses each of its aforementioned limitations.
- First, the input language is C or some variant of C.
- This allows the architectural exploration models to be migrated to implementation with refinements in structure, concurrency, data types, and operation.
- The structure of the model may reflect the hierarchical structure of the resultant hardware. Concurrency is introduced using synchronous interprocessor communication. Data types must include a means of specifying explicit data widths that map to wires and buses.

Hardware Implementation - ESL synthesis

- The second behavioral synthesis issue addressed by ESL synthesis is management of timing.
- Timing is composed of logic delays and routing delays. Estimation of logic delays is aided by information from the technology library for the target process technology.
- Routing delays are just a guess, but may be refined after first-pass synthesis using timing back-annotation.
- The third behavioral synthesis issue - verification environment reuse - is addressed by using transaction-level modeling.
- The verification environment written to verify the behavioral model may be reused, unchanged, to verify the synthesized RTL model using transaction abstraction.

Hardware Implementation - ESL synthesis

- ESL synthesis enables more productive hardware design by facilitating the translation of high-level models into RTL.
- Just as a good RTL synthesis engineer understands the power, space, and timing tradeoffs available and uses them to meet their objectives, so will an ESL synthesis engineer use constraints to guide generation of the resultant RTL.
- These constraints must be used because the synthesis problem is generally NP-complete and multidimensional.
- Local constraints on latency, structure, functional units, and area are applied, whereas global constraints on clock frequency and technology library are used.

Hardware Implementation - ESL Synthesis

- One of the most important benefits of ESL synthesis is enabling early design exploration.
- By using variations of constraints on a fixed behavioral description, the engineer may become familiar with the solution landscape and its sensitivity to certain parameters.
- This leads to opportunities to examine the cost/benefit tradeoffs of performance, power, area, and complexity.

Software Implementation

- The software component of an implementation may serve one or more of several roles, such as glue logic or core algorithm implementation.
- In either role, the software development process has traditionally followed the classic waterfall model.
- The hardware is designed and implemented and, afterward, the software is written to meet all remaining product requirements. This usually includes "coding over" hardware bugs and attempting to mitigate performance shortfalls.

Software Implementation

- The alternative approach we is using ESL models to prototype the software components of a system with a spiral development process.
- In a spiral development process, a series of implementations is created, each evaluated with its associated hardware in a system context, and subsequently refined to address the limitations discovered in the latest incarnation.
- This flow allows the hardware and software to incrementally morph into a solution that meets the end-product requirements.

Software Implementation

- Another consideration of software implementation is performance estimation.
- In the early days of embedded system development, measuring instruction execution rates - such as in Millions of Instructions Per Second (MIPS) - was quite popular. Not reliable.
- In addition to determining the computational footprint of an algorithm on a given processor, equally important is its memory footprint and memory bandwidth requirements.
- instruction encoding like RISC and CISC has a strong influence on instruction fetch bandwidth as well as memory footprint.
- The programming languages and development tools used in an ESL-driven software development flow are close cousins to classical environments, with a few twists.

Software Implementation

- To meet the product development schedule of a modern SoC, containing two or more processors and millions of lines of software, the software must be developed concurrently with the SoC hardware.
- ESL models may be used to craft a run-time environment that facilitates both running the embedded code and interacting with system-level components.
- Each model must trade off fidelity for execution speed.
- We classify the models used for software development along the following dimensions: system scope, time domain accuracy, execution platform, and real-time performance.

Software Implementation

- A system scope model represents a system-level component visible from the software, such as a real-time clock or encryption engine.
- Time domain accuracy refers to the size of the interval between behavioral events.
- For example, instruction execution, register write-back, and register file precharge complete, illustrate behavioral events in order of increasing time domain accuracy and decreasing performance.
- Execution platform refers to the engine responsible for running the software, be it a workstation, FPGA emulator, or hardware accelerator.
- Real-time performance is a measure of the ability of a model to reproduce the latency between a system-level event and software response.

Software Implementation

- An ESL model offers another benefit to the software developer as system integration buries buses and other interconnects previously accessible from the chip I/O. The ESL models enable visibility into the designs.
- Although on-chip debug hardware, partially mitigates this loss of visibility, a high-speed ESL model such as a VSP allows full visibility into interacting software.

Implementation Verification

- The last step in an ESL design flow is implementation verification.
- During this step we are again demonstrating that design intent has been preserved. However, at this stage, the intent in the implementation - the RTL model and embedded software - is supposed to be fully refined, with no remaining ambiguities.
- The reference intent for comparison is captured in the post-partitioning models.
- Other inputs to this verification step may come from the constraints defined in the specification that apply to the environment.

Implementation Verification

- It is likely that much of the verification performed at this step will be at the subsystem level because of the expected levels of performance of the execution environment.
- Additional tools such as emulators or prototyping systems may be used to raise the performance level.
- Alternatively, mixed abstraction execution can be employed where parts of the system use the post-partitioning verification models mixed with the implementation models.
- A few tests will be run on the complete system at the RT level.
- These need to be selected carefully to ensure good use of the available time and resources.

Implementation Verification

- Two fundamental approaches to demonstrating the preservation of design intent are positive and negative verification.
- Positive verification is demonstrating or proving that the design satisfies a requirement.
- Negative verification is demonstrating or proving that no flaws exist in the implementation.
- The former concerns itself with system-level requirements, whereas the latter ensures that a low-level design component - software or hardware module - was properly implemented.

Implementation Verification

- An ESL design flow is driven by a high-level specification that captures all system-level requirements.
- Positive verification is used during post-partitioning verification to ensure that the abstract hardware and software models still meet the system-level requirements.
- Negative verification is used during implementation verification to ensure that each concrete hardware and software model (RTL and embedded code) is behaviorally equivalent to its corresponding abstract model.
- At the implementation abstraction level, we need to accommodate the execution performance degradation that accompanies the size and fidelity of these models.

Implementation Verification

- For hardware models, to maintain reasonable simulation performance (thousands to hundreds of thousands of cycles per second), we typically verify blocks and subsystems using a conventional logic simulator.
- Static formal analysis, as discussed later, is also widely used at the block level to avoid simulation altogether.
- Larger integrations of hardware logic should use a hardware accelerator or emulator to maintain productive execution rates for verification and debug.
- An alternative approach that allows verifying an RTL block in a system context is to use mixed abstraction modeling with virtual system proto- types, transaction-level models that offer megahertz performance with RTL signal interfaces.

Implementation Verification

- Verification software written with reuse in mind is referred to as Verification IP (VIP).
- We distinguish "reuse" from the more common "salvage" operation in that reuse simply requires configuration and instantiation.
- "Salvage" is the copy-and-paste of old code into a new environment with the hope that time and effort are saved, although they generally are not.
- There are two kinds of VIP, corresponding to two kinds of verification: dynamic VIP and static VIP.
- Dynamic VIP is a verification environment addressing the three aspects of dynamic verification: stimulus generation, response checking, and coverage measurement.
- A static VIP is usually an assertion library that captures specific requirements at the RT level.
- Static VIP can also be used in a dynamic execution environment.

Implementation Verification

- If not for the limitations in standard assertion languages, properties and assertions could be used during post-partitioning verification. However, both Property Specification Language (PSL) and SystemVerilog Assertion (SVA) require a sampling signal that is commonly a clock, a notion not introduced until RTL implementation.
- Hence, assertions are first used for block-level implementation today.
- An assertion is an implemented property, where the property is generally specified in natural language in the verification plan and implemented in PSL, SVA, Open Verification Library (OVL) or e.
- A property is a statement of an expected liveness, safety or fairness behavior.

Implementation Verification

- A liveness behavior states that something should eventually happen, whereas a safety behavior states that something should never happen.
- Fairness behavior specifies an equitable access to shared resources.
- The same assertions may be used for static analysis as well as simulation.
- This is convenient because, although tool capacity constraints typically limit the use of static analysis to the block level, these same implemented properties continue to detect requirement violations in subsystem and full-system simulation.

Implementation Verification

- One measure of verification progress at both the post-partition and implementation stages is coverage.
- Other measures include bug discovery rate and RTL and software change rates.
- Although most functional verification will be performed at the post-partition level in an ESL flow, behavior introduced during implementation and visible only using clear box probes must also be verified.
- Three kinds of coverage are used to assess verification progress: functional coverage, structural coverage, and assertion coverage.

Implementation Verification

- Functional coverage quantifies implementation requirements at a chosen level of fidelity using implementation-level attributes, such as register fields and interrupt latencies.
- Structural coverage quantifies how extensively the RTL and embedded software has been exercised.
- Assertion coverage is a measure of the distribution of assertions throughout the implementation, a means of implementing functional coverage, and a measure of how frequently and exhaustively each assertion has been evaluated or executed.
- The relative levels of coverage and rates of convergence give us insight into our verification progress and where our effort should be focused.

Implementation Verification

- Although we need to fully verify the implementation components, we must also verify the full system during implementation verification.
- Simulating the RTL of a modern SoC can be excruciatingly slow, so alternatives are available, each with their pros and cons.
- For example, mixed abstraction models may be used, composed of virtual system prototypes with one or two RTL blocks inserted in each simulation.
- Another approach is to use hardware acceleration or emulation to run the DUV at a substantial fraction of real-time speeds.
- Yet another is to use FPGA prototyping, which is becoming more and more popular with the increasing capacities of FPGAs and mature tool flows that ease RTL mapping.

Implementation Verification

- On-chip debug hardware also makes these attractive from the bug diagnosis and repair perspective.
- Although we have used all the best-known methods for discovering and excising bugs before chip fabrication, unfortunately, now and then, some bugs remain undetected until silicon: bug escapes.
- Post-silicon debug is becoming a growing concern as design complexity continues to increase while development schedules shrink.
- More often than not, a conscious decision is made to tape-out a chip with known functional risks in order to meet time- to-market demands.

Implementation Verification

- To address post-silicon debugging requirements, a number of techniques are now commonly used.
- First, we have the traditional scan chain and associated Joint Test Action Group (JTAG) port that allows serial access to most, if not all, storage elements on the chip.
- Second, trace logic and associated control and multiplexers route data to external pins for off-chip capture and analysis.
- Third, processors now employ on-board In-Circuit Emulators (ICEs) that provide access to most ISA storage and control for a source-language debugger to operate a running program on target hardware as though it were running on its development platform.
- Finally, internal logic analyzers are optionally available on some processors that go beyond the ICE to provide access to microarchitectural elements.

Implementation Verification

- When a bug is found using one or more of these techniques, spare gates commonly sprinkled throughout the various silicon layers may be used by respinning the chip with only a metal layer change to correct the faulty logic.
- Alternatively, further risk reduction can be achieved at the cost of additional chip resources by adding reconfigurable blocks into the design at strategic points so that limited changes can be made dynamically.

Problems

- In the past, when a move to a higher level of abstraction took place, the details of the previous level were left behind.
- But recent developments at the silicon level are beginning to percolate back up the design hierarchy, with companies who ignore them having unexpected problems later.
- At the RT level, tools are now necessary to dig down quickly in an attempt to locate hidden problems.
- What implementation details must be retained as we move up to the system level: floor planning, globally connected buses affecting wire length, and multiple power domains?
- Could the design derived from ESL flow really be "correct by synthesis"

Conclusion

- ESL flow is composed of:
  1. Specification and modeling,
  2. Pre-partitioning analysis,
  3. Partitioning,
  4. Post-partitioning analysis and debug,
  5. Post-partitioning verification,
  6. Hardware implementation,
  7. Software implementation, and
  8. Implementation verification
- The most important thing about a design flow is that it is a recommended method not mandatory.

Conclusion

- After a successful design process is complete, the best design teams review the process of the design in order to develop lessons learned from the process and apply them in subsequent designs.
- Some of these will be technical lessons, but others will be process lessons, and these should be used to improve the design flow for future use.
- Although it almost seems like a cliché, the time spent in the early planning and specification stages has more impact on the rest of the flow than at any other point in the process.
- Mistakes made here are almost impossible to correct, or will be extremely difficult and time-consuming to fix at a downstream stage.