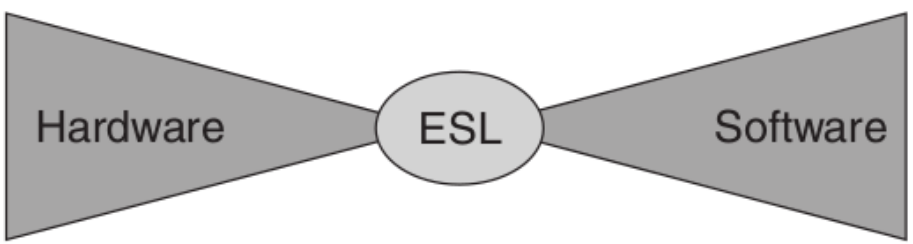
Enablers for ESL Design

- There are a number of values of ESL design, such as increasing quality and reliability and producing optimal designs, but the greatest perceived value proposition is to reduce the time to market for a design.
- The single biggest impact on time to market that ESL offers is to start software development before hardware design has been finalized.
- The expectation is that the users of ESL tools sit between and above the hardware and software teams, primarily looking at the system definition and defining the ways in which the two teams will interface with each other.



The Companies Using ESL

- The companies providing hardware solutions are now equally expected to provide the software components that complete an embedded system. It is no longer enough to provide chips, boards, or even device drivers. The qualification of operating systems for the embedded device is fast becoming a requirement for system manufacturers.
- IP is becoming configurable to the degree that it requires specialized tool support to be used, and these are typically developed by the IP vendor. The distinction between IP and tool is disappearing.
- In teh companies there are strict separation of hardware, software, and system-level design. That may be mistake, but for now it is like that.

System Designer Requirements

- The system design task is not one distinct activity. It ranges from careful analysis of specific system components to overall system demonstration.
- There are two ways to improve productivity of system designers.
- First, models and components must be reused, and second, creating the missing models and components must be easy.
- The result of this modeling process must satisfy a wide range of system designer needs, including:
  - System demonstration, sometimes in the form of a virtual system prototype and sometimes as a data collection and analysis exercise
  - Hardware/software partitioning and, increasingly, software/software partitioning
  - Selection of IP components, both hardware and software IP
  - Sizing, parameter selection, configuration, and extension of IP components
  - System performance analysis and verification
  - Power estimation
  - Algorithm selection

System Designer Requirements

- Possibly the most common tool used for system design in the past has been a spreadsheet. Crude statistics are used to estimate the performance requirements and potential performance characteristics of a system.
- The ESL flow brings a potentially more accurate means of modeling the system, but with the cost of having to build those models.
- The dynamic nature of today's complex systems is no longer suitable for the static types of computation that can be performed in a spreadsheet.
- The output from the system designer should be usable by both the hardware and software teams. Just as with software and hardware development, different system design activities require different types and levels of accuracy and performance.

System Designer - Accuracy

- From the perspective of choosing components, sizing and configuring them, and finally verifying the performance characteristics of the complete system, the expectation is that a model must be accurate. The typical requirement might be that performance measured on the model should be within a certain percentage of that measured on the real system.
- For the system designer, accuracy is about the confidence the designer can have in the information the model delivers.
- The key to modeling for system design is the realization that it is not necessarily the case that the model has to implement the functionality at all!
- Very often, measures of bus bandwidth, for instance, can be perfectly evaluated using nothing more than traffic generators or transaction-level models operating at an abstract packet level.

System Designer - Accuracy

- It is even possible to take into account the effect of some software algorithms that may not have been written, so long as their resource utilization can be budgeted.
- System designers may require a degree of timing accuracy, but they often do not require register accuracy.
- One of the key tasks in system design is the selection, sizing, and configuration of components. Often, components should be sized to accommodate the normal (mean) system load, in which case peak loads can cause poor performance, as might be the case for a user interface component. However, other components may have strict timing constraints that must be met no matter what the system load. These are called hard real-time constraints. A system designer needs to be able to choose between these measures.

System Designer - Accuracy

- Although time and performance measures remain the most measured factors, system designers increasingly require more information about power consumption.
- As mobile computing increases in importance, the issue of heat and power becomes ever more critical because battery development has not kept pace with the amount of electronics that can be packed into a small space.
- One of the key requirements of a model for system designers is that they should be able to instrument the model to extract whatever information they are interested in. This model flexibility is often a key value proposition for the model, but often overlooked.
- Model is tool in this case!

System Designer - Time and Speed

- Although the principal endeavor of ESL is to deliver a parallel software and hardware design flow to enable system design, the models may be required before the HW/SW partition is completely decided. It is quite possible that aspects of the design will not exist, and others will exist as legacy.
- This often leads to a collection of models, ideas, drawings, and paper specifications, and severely limited time in which to bring them all into a coherent form.
- At this point in the flow, tools that can help combine old and new models and tools that assist in creating new models are invaluable.
- Unfortunately, they are not always perfect for the task at hand, but may be adequate when the alternative is paying remodeling costs.

System Designer - Time and Speed - Traffic Generator Models

- Not all tasks require a complete model of the system, and in those cases the system designer will model only those aspects of the system under investigation that are important to the measurements required. Traffic generators can replace aspects of the system that are not under investigation with a footprint of the traffic that they are expected to generate. Such models can execute quickly because they have little to do and are extremely flexible.
- There are possible pitfalls
  - Traffic generators must truly represent the footprint of the system blocks they are representing. This is often hard to achieve.
  - Not only must a traffic generator model its own behavior, but, crucially, it must model the interaction with other system blocks.
  - Traffic generators can become messy as more and more detail is added to them. In the end, their complexity can grow to resemble that of the blocks they are representing.

System Designer - Time and Speed - Tool Cost and Value Proposition

- Cost for system designers should not, in principle, be an issue.
- The question is, what value does a tool provide, over and a spreadsheet? There are many costs to consider, and the value is often intangible, or not directly visible.
- On the cost side there is the initial tool evaluation cost, and this can be a significant hurdle for tools companies because it requires the time of engineers on both sides.
- Second, the costs of integrating a tool into a design flow are often greater than the cost of the tool. This integration stretches from training costs to development of the "scripts" (really shell programs) that users will need to adapt for their existing design databases to be usable by the tool.
- Organizations such as the Structure for Packaging, Integrating and Re-using IP within Tool-flows consortium (SPIRIT) are trying to standardize some of this, but there are still large areas of incompatibility between languages and models.

System Designer - Time and Speed - Tool Cost and Value Proposition

- The variety of companies providing tools for this market shows that a single business model has not yet become the standard because tool costs can range from open source, to just a few thousand U.S. dollars, to several hundred thousand dollars per year.
- The reward is most apparent if the value comes in terms of faster development of models that serve useful purposes in both the hardware and software design, development, and verification flows. At the same time, these models enable the system designer to reach a decision point more quickly.
- Many ESL tools focus on other areas, such as better downstream validation, more consistency in the design database, or speedier model execution that may be helpful to downstream software developers. The difficulty with this approach is that the tools must be used up front by the system designer to have a positive impact later in the design flow. This is a harder sell to a system design team who will suffer all the aforementioned costs, with little or no direct reward to their job.

Software Team Requirements

- As already stated, one of the biggest impacts of ESL is the potential to start software development before hardware has been finalized. This parallelizing of the software and hardware flows has a large positive effect on time to market and hence the entire ESL value proposition.
- In addition, it enables the interface between the hard- ware and software teams to be properly designed, taking into account the needs of both teams.
- This aspect of ESL will continue to grow in the future, especially when multiple heterogeneous processors are available in the hardware platform.
- Software teams will use ESL models and their associated tools for a number of activities. Principally, this includes the design and debug of complete hardware and software systems.

Software Team Requirements

- We should not forget that probably the most popular development environment today is to connect a piece of real hardware, whether target hardware or a prototyping hardware engine, to the software engineer's computer.
- However, this cannot be done when the hardware does not yet exist, and thus we must find alternatives to this methodology.
- The software task can itself generally be split into distinct activities:
  - Software for hardware verification and debug
  - Low level (device drivers), often called Hardware-dependent Software (HdS)
  - Medium level (middleware, protocols, and operating system code)
  - High level (application code)

Software Team Requirements

- Software represents a very large investment, larger in many cases than for hardware.
- The life span of the software and the hardware often overlap. In other words, just as in previous generations of CPU architectures, designers have paid considerable attention to maintaining backward compatibility such that code designed for a previous CPU will run on a new version; now this is true in reverse as well. Software designed to run on one device must be implemented such that if the hardware is upgraded, the software does not need to change.
- Software designed to run on one device must be implemented such that if the hardware is upgraded, the software does not need to change.

Software Team Requirements - Accuracy

- The common perception is that the model of the hardware must be accurate.
- For the ESL design flow, often the requirement is that if the software works on the model, it should work on the real hardware.
- This would be true of a highly accurate model, but may be equally true of less accurate models.
- The model is likely to be used before the hardware has been completed and thus the actual timing will be unknown.

Software Team Requirements - Accuracy

- Modern SoCs can alter clock frequencies in response to system load, operating temperature, or the state of the battery. In this case, the requirement of the software is to work on a hardware platform for which specific performance metrics cannot be guaranteed.
- Because the ESL models and tools are not physical hardware they may have features that real hardware does not have.
- Because a model is provided instead of real hardware, software engineers can be empowered with features that help them do their job. The requirement becomes a useful configurable, but potentially inaccurate model.

Software Team Requirements - Model Features Examples

- A model that caches differently (in a selectable way) than the hardware. Such a model can, for instance, be used to find cases where the software engineer has forgotten to flush the cache, or mark a page as I/O. In addition, it is possible to simulate multiple cache configurations concurrently, providing another way to detect cache problems.
- A model that completes all memory reads and writes in reverse order to a degree permitted by the architecture of the system. Memory ordering can be a significant cause of software bugs.
- A model that speeds up element X compared with element Y. In a system with multiple independent masters (e.g., X and Y), problems with semaphores and race conditions can be detected with such a model.

Software Team Requirements - Register Accuracy

- The most common form of accuracy quoted for a model suitable for software use is register accuracy, sometimes called the programmer's view (PV). This refers to the state of all of the registers in the system, both in I/O elements and programmable devices.
- Additional information, such as an approximate cycle count, is often provided to software engineers. It can help them optimize their code. Good models might also provide cache statistics, which can be extremely helpful.
- For software engineers, chasing the last cycle seems a worthless preoccupation on modern SoCs, where the reality may be non-deterministic anyway.

Software Team Requirements - Model Execution Performance

- For the model to be useful, it must assist the software engineer in providing solutions earlier than would otherwise be the case. The common assumption is that a model should ideally run at near real-time speeds.
- What is often overlooked is the time that it takes to get the model to the software engineer. If this can be reduced, the software engineer has more time with the model.
- There are several types of models:
  - Interpreted, Stand-Alone Models
  - Interpreted Slave Models
  - Cache Line Just-In-Time Model
  - Cache Page JIT Models
  - Host Compiled Models

Software Team Requirements - Interpreted, Stand-Alone Models

- The most common form of model used by software engineers is an interpreted model. This means that each instruction is read into the model and the binary encoding of the instruction is decoded and interpreted sequentially. The effect of each instruction causes changes to the state of a model in a similar manner to that which would happen in the actual device.
- Models of CPUs are often constructed as stand-alone entities and called ISSs. They may even have their own debugger. Software engineers write and compile their programs using the same suite of tools they will deploy on the real hardware.

Software Team Requirements - Interpreted, Stand-Alone Models

- The interpreter typically performs in sequence the fetching of an instruction from a memory hierarchy, the loading of any necessary data, the execution of the instruction, and the final commitment or writeback of any results to a register file, or back to memory. These kinds of models can become very complex, especially in the way in which the memory hierarchy is modeled, or if exact timing is required. They have been the backbone of the processor industry for many years, and almost every processor has been modeled this way at some stage.
- The advantage of this approach is that it is relatively simple to understand and to construct models of programmable elements such as processors. The software writer also has the security of using the same tool chain that will be used for the real hardware.

Software Team Requirements - Interpreted, Stand-Alone Models

- The model typically does not reflect the hardware architecture because notions of pipelines are generally absent. This means that the models are hard to correlate with hardware implementations, and it is not always possible to extract cycle count information from them. If the pipeline is modeled, performance degrades considerably, with an inverse correlation between accuracy and model performance.
- The models are typically not as fast as the software engineer would like. In a typical scenario, such a model might consume on the order of 1,000 host instructions to perform a single modeled target instruction.

Software Team Requirements - Interpreted, Stand-Alone Models

- Putting stand-alone models within a system context where there may be multiple CPUs can be challenging.
  - The model must yield control to the rest of the system. Often synchronization points are required where the software can rely on the hardware reaching a certain state.
  - The debugger connection must be able to handle multiple instances.
  - There must be a means by which multiple debuggers are synchronized.
- Connections to the model are normally made though a memory interface. This interface is typically implemented as a single API call that a slave model must implement. The API call reads or writes one or more memory locations. This can be seen as a request or transaction that would normally be transmitted over a bus connected to the processor.

Software Team Requirements - Interpreted Slave Models

- To perform better in a system context, programmable element models may be available as slave models. These are still interpreted models, but with different interfaces built around them.
- The model in this case provides a procedural entry point that typically executes a single instruction. This can then be integrated into a language-based simulation environment such as SystemC.
- In the general case, the model still has no real notion of time, so it is still very important that synchronization points be established.

Software Team Requirements - Cache Line Just-In-Time Model

- To achieve higher speeds, fewer host instructions must be executed. There are several places where host instructions are used: fetching, decoding and executing the instruction.
- A cache line Just-In-Time (JIT) model takes advantage of the target processor instruction cache often present in the programmable element itself.
- Rather than storing the data fetched from memory, a cache line JIT typically builds a function that will perform the instructions that have been loaded all together. This function is built "just-in-time," hence its name. In other words, both the fetch and decode phases of the instructions in the cache line are evaluated when the cache line is filled, and can subsequently be executed multiple times, thus avoiding fetch and decode overheads.

Software Team Requirements - Cache Line Just-In-Time Model

- Care has to be taken if there are branches in the code. It is often called a basic block compiled model because everything between branch statements can be compiled ahead of time without a change in functionality.
- The compelling advantage of such a model is that with care, the model can also evaluate the number of cycles taken to execute the cache line or basic block, and thus the model can deliver cycle count accuracy at very high model performance.
- There is of course no reason to limit the cache size of the model to the exact cache size of the programmable element, so cache hit rates can be much higher.
- This form of model can be expected to execute an order of magnitude faster than a standard interpreted model. Furthermore, the corresponding bus interface is no more or less complicated than a normal interpreted model.

Software Team Requirements - Cache Page JIT Models

- In an extension to cache line JIT, a full cache page can be compiled. This has the advantage of potentially allowing more optimization when mapping the programmable element onto the host that will execute the model.
- Some of these optimizations are dynamic (e.g., branch prediction) and an optimizing JIT compiler may be able to make use of these at runtime.
- This form of model can be extremely fast (an order of magnitude again faster than a cache line JIT).
- However, there are added complications:
  - Keeping track of memory pages.
  - There is no longer an instruction boundary.
  - It is no longer acceptable to execute an entire cache page because it may continue indefinitely.

Software Team Requirements - Host Compiled Models

- The ultimate option for a model is to remove the model altogether. In this case, the software that would have been compiled and executed on the programmable component is instead compiled in its entirety for the host platform.
- Considerable care must be taken to identify the elements in the code that would be expected to interact with volatile elements in the system, such as I/O elements.
- The disadvantage of such a model is that the tool chain used to create it is not the same as the tool chain used to create the actual executable that will run on the device, unless a binary-level compilation process is used. This limits the usefulness of such a model primarily to application code development.
- The resulting model runs at the speed of the host processor, which could be faster than the target hardware.

Software Team Requirements - Tool Chain Cost

- Software tools are typically 10 to 100 times cheaper than their counterpart hardware design tools. This causes some difficulty for tools that could be used by both hardware and software engineers.
- The discrepancy between the cost of software tools and EDA tools is normally justified by the size of the market into which the tools are deployed.
- As simulation tools are increasingly deployed into software environments, the expectation must be that their price will fall accordingly. However, this erodes the price of the same tools in the original market. Naturally, simulator companies are reluctant to do this, although there should be a large growth in the total number of available customers.

Hardware Team Requirements

- Hardware teams typically need a wide range of models with various degrees of accuracy. The ESL flow promises to rectify this, but to do so it must take into account the wide range of requirements that the hardware team has over and above the other teams in the design flow.
- With the emergence of platform-based design, it could be assumed that hardware design would become increasingly a matter of configuration. However, this does not seem to be the case.
- The same verification problems exist. The responsibility of the design team to create, acquire, or configure IP is still there.
- The requirements for system-level verification are perhaps the most interesting. They range from performance and throughput verification to ensuring that semaphore and synchronization points are adhered to.

Hardware Team Requirements - Model Refinement

- The promise of ESL has always been the notion that the system designers could create an executable specification. The hardware team would then be provided with a golden reference model from which they can derive their hardware.
- When such a model is available there are two ways: model refinement and model synthesis.
- In these cases, either by use of a tool or by modifying the model, the model itself is progressively refined. Strict synthesis routes might offer the benefit of correct-by-construction refinement, but they have the disadvantage of demanding that the initial model already be sufficiently well specified to enable them to do their job.
- If synthesis is not used (and sometimes, even if it is used), the designer is still faced with the task of verifying the refined model's correctness. The remaining use cases focus on this scenario.

Hardware Team Requirements - Verification Environment Provision

- The ESL design flow provides a productive path for IP blocks and, from the hardware verification engineer's point of view, the most useful component may well be the verification environment itself.
- It should be noted that one effect of ESL languages such as SystemC is to cause the verification environment to become essentially a programmable element, in that the code used to program it may be just as complicated as a device driver or similar software component.
- The verification environment will invariably have to interact with the software code used for verification to cause the IP block to move between the required states. The verification environment becomes part of the verification code.

Hardware Team Requirements - Verification

- It has already been stated that one of the primary value propositions of ESL design is to enable the creation and verification of software earlier in the design flow. Thus, it should come as no surprise that the most common use, and benefit, of ESL models is to enable verification and software engineers to create software use-case patterns that will exercise a hardware block from the software engineer's perspective.
- A hardware block will have other requirements that are not visible from the programmer's perspective, but software test generation remains one of the most powerful tools.
- A programmer may not be able to see the output of an LCD controller from within a program, but coding a program to cause the LCD controller to display a known pattern on the screen is most definitely a useful task.
- One of the key advantages proposed by ESL is to enable the automatic generation, recording, and control of complex system-level tests.

Hardware Team Requirements - Verification Simulation

- Technique 1: Parts of a modeled system can be replaced with their respective implementations. The rest of the system then becomes the verification environment. This has the significant advantage of reducing the work of the hardware engineer in constructing the verification environment, although in reality it cannot completely replace it. Such system-level verification environments are more likely to be used to find system-wide problems, such as bandwidth or communication problems, rather than be especially effective at isolating internal block issues. (Co-simulation)
- Technique 2: The programmable blocks of a hardware simulation are replaced with models that will run the use cases. This is done to increase the speed of the simulation and also to allow the verification engineer access to the programmable element through a debugger interface that will enable them to see what is happening in a more natural way, rather than looking directly at the hardware. (Co-verification)

Who Will Service These Diverse Requirements

- There are two difficulties for tools companies trying to service the ESL market. First, the various users have radically different price expectations for their tools.
- Second, IP creators are increasingly being required to package their models as tools.
- The solution is to develop standards. One example is SystemC.
- Now a layer of infrastructure and commonality needs to be grown above SystemC, the language, in order to enable IP creators to construct IP that not only matches their own needs but can be deployed by a variety of users.

Free or Open Source Software

- Using the open source business model to provide the ESL infrastructure that both traditional IP and tools companies require has some positive benefits for both the industry and the wider user community.
- The term Free or Open Source Software (F/OSS) scares people in the silicon industry.
- One of the key advantages that F/OSS gives is the community effect.
- Mostly used license terms
  - BSD—Named after the Berkeley Software Distribution of Unix that carries this license. With the BSD license, you may distribute both source and binaries freely.
  - GPL—The General Public License states that you must publicly distribute source, and explicit derivative works must also publicly distribute source, with the same license.

Summary

- Models with many different and often competing attributes are needed by all the groups that are fed from the ESL flow, namely hardware, software, and verification.
- Many of the models will be created by the system designers who will use them for a variety of sizing and selection functions. Today's complex systems have made it necessary to develop dynamic analysis tools because traditional spreadsheet analysis is no longer capable of providing the required accuracy.
- Models do not always have to be faithful to the eventual design. Providing the means to inject errors or exhibit alternative behaviors can help in tracking down problems.

The Prescription

- Create a clear plan for the models that will be required by all of the groups in a team. It should identify where each of the models will come from, when they should be available, and their exact capabilities.
- Identify the application of each model.
- Define clear paths for model maintenance and proliferation.
- When choosing tool and model vendors, pay close attention to the standards for interoperability.