Modeling techniques maximize value of virtual platforms

By: Andy Ladd, Carbon Design Systems

Abstract

Over the past several years, development teams and methodology groups have placed greater emphasis on platform-driven design techniques. Shorter product life cycles and heightened time-to-market pressures have forced companies to invest in system-level platforms available earlier in the design cycle. In addition, the transition to System-on-Chip (SoC) design techniques leveraging legacy and third-party IP has provided better structure and methodology for modeling at the system level. Meanwhile, the increasing amount of embedded software and firmware content has repositioned the majority of resources required to produce a product into the software domain. In a traditional design flow, this means a larger portion of the development process is shifted later in the design flow, increasing schedule risk.

Virtual platforms help address these ever-increasing complexity issues, market pressures, and changes in content. Although some engineers have described the benefits of these platforms in detail[1], identifying appropriate modeling methods is usually left as an exercise for the reader. To shed some light on this facet of virtual platforms, the following discussion will analyze different aspects of proper modeling techniques.

Ironically, while models provide the backbone for any system-level platform, difficulties and expenses in developing these models have limited virtual platform adoption. In addition, modeling and support efforts consume the lion's share of the costs for developing and supporting these platforms.

Model abstraction levels

For this discussion, models can be partitioned into four distinct parts: functionality, addressable state, timing, and interfaces. Each of the model's four parts can vary at different abstraction levels, from high-level behavioral descriptions to the actual design implementation. Models created directly from design descriptions that reflect true design behavior are referred to as *implementation-accurate* models. The modeling stack in Table 1 shows the continuum of abstractions between the highest and lowest extremes.

	Abstraction	Common Names
Highest	Behavioral	Programmer's View (PV), untimed (UT)
↕	Timed	PVT, loosely timed (LT), approximately timed (AT), cycle approximate
	Cycle Accurate	CA, clock accurate
Lowest	Implementation Accurate	Register Transfer level (RTL), Design Simulation Model (DSM)

Table 1

As with most applications related to computing, increasing accuracy has a direct impact on reducing execution speed. This is no different with modeling; boosting a model's accuracy requires more processing and a reduction in execution speed.

In addition, increasing model accuracy directly correlates to the amount of effort and time required to create and support models. Finding the proper speed versus accuracy trade-off is paramount to achieving a modeling paradigm that will meet virtual platform users' needs and limit the amount of effort required to develop and maintain the platform.

At one end of the spectrum, hardware engineers need implementation-accurate models to validate their designs. At the other end, application software developers can get by with high-level behavioral models. Between these two extremes lie lower levels of software, including the Operating System (OS), driver, firmware, and architectural and performance analyses.

Application software engineers are most concerned about developing their applications and having a productive debug environment. They don't need the accuracy of a detailed model; their code rarely touches the actual hardware because it's layered on other software. However, application software engineers in some cases might need to understand simple performance metrics that require more accuracy.

Unlike application code, OS and driver development touches the hardware; thus, those who develop these components need a higher degree of accuracy to understand how their software and the underlying hardware interact. They can exchange speed for higher accuracy because their code base is smaller than application software engineers' code base. Untimed behavioral models might be useful for early development, but ultimately, OS and driver developers must understand how their software works using more accurate models to ensure that the whole system (hardware and software) will work together.

Firmware engineers develop code – boot code, self-test, diagnostics, and console – that interacts with hardware. Given this high level of interaction with and dependency on hardware, these engineers have little use for inaccurate models. They can swap model speed for higher accuracy because their software is at the lowest level and is usually small compared to that of higher levels. Tuning low-level firmware and driver software performance also requires cycle-accurate models to understand timing dependencies on hardware as well as resource bottlenecks[2].

Architects need to know how their hardware/software partitioning, IP selection, bus architecture, memory architecture, and overall architectural decisions impact the system as they relate to performance, area, and power. They also must understand pipeline effects, latencies, throughput, bandwidth, and activity. A final design that doesn't perform as architects planned could dramatically affect the product's cost, performance, and schedule. Therefore, architects must validate their designs using highly accurate models that build confidence in their decisions. Hardware engineers must have implementation-accurate models; any other level of accuracy is unsuitable for validating designs.

A single abstraction level for all models is not always appropriate in every case. For example, an architect considering memory architecture trade-offs might try to analyze each prospective memory subsystem's memory latency and throughput. In this case, architects might need highly accurate models for memory controllers and memory interfaces to ensure that they fully understand the performance. The rest of the system can be modeled at more abstract levels because it isn't critical to the analysis. Using a model methodology that supports mixing abstraction levels and enables plug-and-play for models of different abstraction levels can help optimize execution speed and analysis accuracy.

Finally, when considering all possible use cases, engineers should note that a platform rarely targets only one type of user. It is more common that a virtual platform will be created to address the needs of many types of users, ranging from software developers to architects and, in some cases, hardware designers. Therefore, different abstraction levels must be supported within the platform.

Interoperability and compatibility

When creating a modeling methodology, developers should make sure models are interoperable with each other, spread across abstraction layers, and compatible with various platforms and third-party tools. Consistency is also important to guarantee that models created by different model developers are compatible with each other.

Though not perfect, standards help add consistency, compatibility, and interoperability among models by supporting various model abstractions and providing compatibility with different platforms and third-party tools.

Modeling languages such as SystemC provide a base platform to connect and execute different models. SystemC provides the flexibility to support multiple abstraction levels and communication interfaces. Combining SystemC with interface standards, such as the proposed Transaction-Level Modeling (TLM) 2.0 specification in development by the Open SystemC Initiative (OSCI), provides an environment that maintains compatibility with various modeling elements and abstractions and makes them interoperable with each other and other platforms.

In addition, when refining models from different abstraction levels, developers should reuse as much information as possible from one model to another and reuse modeling information from one revision of an IP block to another. Consistent methodology and standards provide the mechanisms to accomplish these tasks. Developers also can reuse interfaces, state access mechanisms, and timing from one model to another. Standards such as Spirit IP-XACT, the IP metadata specification from the SPIRIT Consortium, can help developers import and export configuration information and check differences between model revisions and abstractions.

A modeling methodology that doesn't guarantee interoperability among different models and abstractions or provide compatibility with other platforms and third-party tools is ill suited for most projects. In fact, lack of interoperability and compatibility has slowed virtual platform adoption within the embedded industry.

Meeting supply chain needs

The growing need for providing models across the supply chain reinforces the importance of interoperability, compatibility, and standards. IP providers must supply early models of their IP because customers need the ability to select the appropriate IP for their products. Without proper models, customers have no way of knowing what IP will work best in their systems. Customers also need a platform for their own development (architecture, hardware, and software) to hit their market window with the correct product.

Any modeling methodology that supports IP delivery across the supply chain must take this into consideration. IP must be compatible with end customers' platforms and models, yet at the same time provide security and be impervious to reverse engineering.

Breaking down modeling barriers

A well thought-out modeling methodology can overcome obstacles to virtual platform adoption as well as ensure that users attain all the value that a virtual platform can provide. Virtual platform modeling should support models of various abstraction levels, model plug-and-play, and standards for interoperability, compatibility, and reuse.

The industry needs to provide tools that automatically support model generation in a consistent manner across the various abstraction levels. These tools must incorporate standards and preserve the investment that developers put into their models. Requirements should:

- Make the model developer more productive
- Reuse information from legacy models or models of different abstractions levels
- Support standards to ensure interoperability and a migration path for models to other virtual platforms
- Provide consistency checks to validate models across abstraction levels
- Offer configuration management and revision control aid for model support and distribution

Andy Ladd *is VP of applications and methodology at Acton, Massachusetts-based Carbon Design Systems, Inc., where he is responsible for providing and supporting automatic system-level modeling and validation solutions.*

Carbon Design Systems
978-264-7300
aladd@carbondesignsystems.com
www.carbondesignsystems.com

References

[1] Serughetti, Marc, "Platform-driven ESL design: Enterprise enabler for platform customization," *Embedded Computing Design*. August 2007. www.embedded-computing.com/articles/id/?2212

[2] Hong, Sungpack, et al., "Creation and utilization of a virtual platform for embedded software optimization: An industrial case study," Proceeding of the 4th International Conference on Hardware/Software Codesign and System Synthesis. October 2006.