



TLM Peripheral Modeling for Platform-Driven ESL Design

Using the SystemC Modeling Library

TLM Peripheral Modeling for Platform-Driven ESL Design Using the SystemC Modeling Library

by Tim Kogel, CoWare Inc., *Solution Specialist*

Abstract

This article provides an overview of a SystemC-based Transaction Level Modeling (TLM) methodology for the rapid creation of SoC platform models. First a brief overview of the ESL design tasks and the corresponding modeling requirements is given. The main topic is a methodology for the efficient creation of transaction-level peripheral models. Those are usually specific for a particular SoC platform and have to be created by the ESL user.

Platform-Driven ESL Design

Electronic System Level (ESL) design refers to a set of System-on-Chip (SoC) design tasks like embedded software development or architecture definition, which have to be addressed before the silicon or even the RTL implementation becomes available. Using ESL these tasks are performed by means of a transaction-level model of the SoC platform, which delivers the required simulation speed, visibility, and flexibility. This greatly improves the productivity to run and debug embedded software, investigate architectural alternatives, perform hardware/software integration, and validate the performance and efficiency of the resulting system.

However, every user has to create a transaction-level model of the SoC platform before reaping the benefits of ESL design. The significant investment in modeling can be decreased by IP and ESL tool vendors by providing model libraries. This remedial action is limited to typical common-off-the-shelf IP blocks like processors and buses. The majority of the platform-specific IP blocks need to be modeled by the ESL user. This proved to be very problematic for the following reasons:

- The current level of standardization is not sufficient to protect the investment in modeling.
- Only limited resources are available for the creation of transaction-level models.

- Each of the ESL design tasks has different modeling requirements in terms of accuracy and simulation speed. A standards-based methodology for the efficient creation of reusable transaction-level peripheral models is therefore one of the most important prerequisites for the urgently required adoption of ESL design.

TLM Use Cases and Abstraction Levels

The discussion about the right level of abstraction for transaction-level modeling is truly a difficult one. In general, a modeling approach can be divided into domains for communication, data, time, structure, and functionality. Each domain can be represented at different levels of abstraction, e.g. the time can be untimed, timed, or cycle accurate, the data can be modeled as abstract data types, bursts-of-words, or words. To simplify this discussion, we first classify TLM according to use cases rather than abstraction level. This way the purpose of the model is in the center of attention.

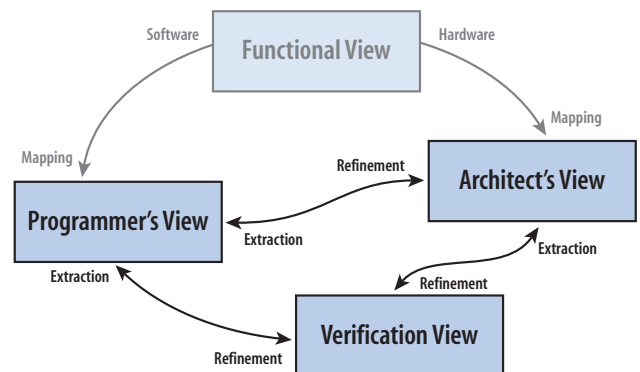


Figure 1: TLM Platform Models in the ESL Design Flow.

The Functional View (FV) is a TLM use case that represents an executable specification of the application, which is supposed to be executed on the SoC platform. The Functional View stands somewhat apart from the following three TLM use cases, because it is used for

modeling the application. Programmers, Architects, and Verification View on the other hand are used to model the platform architecture.

The Architects View (AV) is a TLM use case targeted to architectural exploration. Compared to an ad-hoc specification of the SoC architecture, the early exploration and quantitative assessment of architectural alternatives bears the potential of reducing the chip cost in terms of area and IP royalties. Additionally the AV use case mitigates the risk of late changes in the architecture due to missing the performance requirements.

A model deployed in this use case should have sufficient timing information to enable exploration of architectural choices and trade-off analysis. Usually the processor cores are abstracted to traffic generators and File Reader Bus Masters to mimic the on-chip communication load. This approach minimizes the initial modeling effort and yields an acceptable simulation speed.

The Programmers View (PV) is a TLM use case for embedded software design. Based on a “virtual prototype” of the SoC platform, companies can get a significant head-start in the embedded software development. The full visibility of the PV platform model greatly improves the debugging productivity of the embedded software developer compared to a development board or to an emulator-based solution. Additionally, the model of the new platform can be shipped to customers for early assessment of features.

Here functional correctness is important for the elements of the model which are visible by the software. Additionally, the memory map needs to be modeled correctly. The availability of fast Instruction-Set Simulators is a key prerequisite for this use case.

The Verification View (VV) is a TLM use case for cycle-accurate system validation and HW-SW co-verification. The performance can be optimized by fine-tuning the configuration of the interconnect IP using a functionally complete and cycle-accurate transaction level model of the SoC platform. This helps to reduce the chip cost, because you can optimize the resources to achieve the required performance. The expected performance of the final design can be fully qualified prior to the implementation phase. This further reduces the risk of late changes in the architecture to meet requirements. Additionally, the block-based verification of the RTL implementation against the golden TLM model in the realistic system context reduces the effort for the creation of testbenches and reference models at the RT level. This improves the productivity of the verification engineers.

Platform models for the VV use case are composed of cycle-accurate Instruction-Set Simulators and bus models. This of course impacts the simulation speed, but the accuracy cannot be compromised.

Starting from this use case-based classification it is much simpler to talk about the appropriate level of abstraction to achieve a particular design task. In general, it is not meaningful to position one use case at a “higher” or “lower” level of abstraction. Instead every use case has an optimal working point:

- AV requires a certain degree of timing information to capture the anticipated performance of the system. The functionality is usually not that important, so the application can be represented by a non-functional workload model.
- PV requires only very little timing but needs to be functionally complete.

Reuse-Driven Peripheral Modeling Methodology

In essence, CoWare’s Platform-Driven ESL Design paradigm combines the individual use cases into a consistent ESL design flow, where different design problems are solved using the appropriate TLM use case. The key enabler for this design paradigm is the reusability of design-specific peripheral models across multiple use cases. Otherwise the return of investment into modeling would not be sufficient. Additionally, the peripheral models must fulfill the requirements of the individual use cases in terms of simulation speed and accuracy.

The reuse is achieved on the basis of two assumptions:

The abstraction level of the SoC platform model is mostly influenced by the abstraction level of the deployed interconnect models as well as the abstraction level of the deployed Instruction-Set Simulator (ISS). More specifically, the accuracy and simulation speed inherent to the chosen models of the interconnect architectures and programmable architectures determine the aptitude of a TLM simulation model for a specific use case. For example:

- A platform model constructed from instruction-accurate ISSes and PV bus models is only usable for software development, as the platform does not contain any timing information.
- A platform model constructed from cycle-accurate ISSes and cycle-accurate bus models is only usable for verification purposes. For the software developer the simulation is way too slow. An architect would require higher flexibility to efficiently explore the design space.

- A platform model constructed with cycle-approximate bus models and without an ISS is only usable for architectural exploration.

The good news from this observation is that the accuracy of all the peripheral models is not really essential for the use case of the platform. The notable exception is the accuracy of the memory subsystem (like caches, memories, and memory controllers). We just have to make sure the peripheral models are fast enough to enable software development.

The second assumption states that the encapsulation mechanisms provided by C++ in general as well as SystemC and TLM in particular enable a mutual separation of communication, behavior, and timing. In other words, we can decompose the problem of modeling a platform element into several orthogonal aspects. What is more, each of these aspects can be nicely supported by a set of well-defined interfaces and modeling objects.

The good news from the second observation is that the communication interface and the timing model of a peripheral can be modified without too much impact on the behavior. As explained in more detail below, this enables a refinement strategy based on timing refinement and TLM transactors.

Figure 2 also shows the separation of the peripheral components into timing, behavior, and communication, which is discussed later in detail.

The construction of different platform models from a consistent model library is the essence of CoWare's Platform-Driven ESL Design paradigm.

The Importance of Standards

Every user is reluctant to tie his IP and system models to a vendor proprietary modeling style and tool environment. Only the standardization of model interfaces enables the interoperability of models from different origins. This lowers the barrier to invest in the creation of transaction-level models and therefore fosters the adoption of ESL design. The following list gives a brief overview of the relevant standards in the area of SystemC based TLM.

The *SystemC Language Reference Manual (LRM)* standardizes the basic SystemC language itself. This is the minimum requirement to build SystemC simulators, but is by far not sufficient to achieve interoperability among transaction-level models.

The *SystemC Transaction Level Modeling (TLM) 1.0 standard* defines the fundamental communication and synchronization constructs that can be used to create TLM interfaces. Usually a protocol layer is built on top of the basic TLM 1.0 interfaces, which provides a convenient API for a specific communication protocol or design task. The TLM 1.0 standard therefore does not enable the plug-and-play interoperability of TLM models, because different protocol APIs are used by the models. Still the definition of the basic TLM semantics facilitates the creation of transactors between different protocol APIs. Currently, the OSCI TLM working group is standardizing a set of generic protocol APIs for the Programmers View and Architects View use cases to further improve the interoperability.

The *OCP-IP SystemC channel library* is an example of a protocol layer for the Open Core Protocol. It defines TLM interfaces at multiple levels of abstraction. The highest abstraction level represents a generic interface for architectural modeling.

The *SystemC Verification (SCV) library* provides a standard interface for randomization and transaction recording.

The modeling style and all objects in the modeling library described in this article are compliant with SystemC and leverage the other SystemC-based standards mentioned above. This means that their functionality can be built on top of SystemC without any changes to SystemC IEEE 1666.

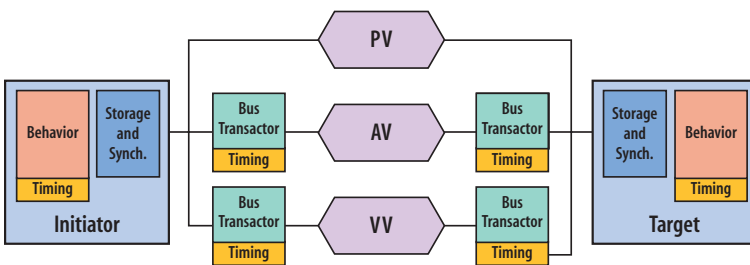


Figure 2: Reuse of Peripherals for Multiple Use Cases.

These two general observations lead to the modeling strategy as depicted in figure 2. The three different bus models in the middle represent the fact that the accuracy and the resulting simulation speed of the interconnect model determine the use case of platform simulation. Usually these different abstraction levels are not deployed in a single platform simulation. Instead, the three different bus models are part of the *platform model library*, from which the actual simulation for the respective use case is constructed. Most importantly, the peripheral models have only one representation in this platform library, i.e. they can be reused for the different use cases.

A TLM Pattern for SoC Peripherals

The *orthogonalization of concerns*, i.e. the separation of different aspects of the design process, is generally considered to be the key ingredient to tackle the complexity of SoC design in a divide-and-conquer kind of approach. In this context, the separation of *behavior* and **communication** is a well-known concept, but proves to be difficult to implement. In order to improve the reusability of the behavior we have developed a TLM pattern, which further pushes the concept of orthogonalization.

First, we propose to further decompose the communication aspect into a generic *storage and synchronization* layer and a protocol-specific *transactor* layer. The purpose of the storage and synchronization layer is to exhibit well-defined generic interfaces towards the behavior as well as towards the transactor. This effectively decouples the behavior from any particular bus interface. This way, a peripheral model can be hooked to a model of a different bus (or to a model of the same bus at a different abstraction level) by just replacing the bus transactor.

Additionally, we propose to separate as much as possible the timing of the model from the actual behavior. This way, the timing accuracy of a model can be increased by adding timing information. Augmenting purely functional models with timing information is essential in order to reuse them for the AV and VV use cases.

In summary, the TLM pattern separates peripheral models into the following orthogonal parts: the bus-specific transactor, a generic synchronization and storage layer, the actual behavior, and the timing information. Ultimately this fine-granular separation of the different modeling aspects leads to a scalable accuracy, which enables a seamless reuse of models across multiple levels of abstraction.

In the following we will first present the TLM pattern for target and initiator peripherals and then talk about the modeling of timing.

Modeling TLM Targets

Targets are SoC platform elements like memories, timers, interrupt controllers, or passive hardware accelerators, which are situated on the receiving end of the communication architecture. The TLM pattern for target peripherals is depicted in figure 3.

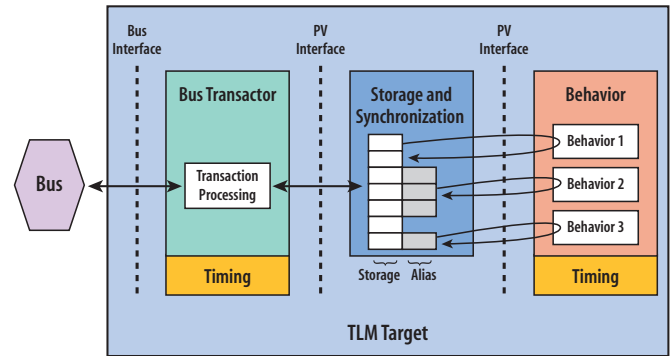


Figure 3: TLM Pattern for Target Peripherals.

The generic TLM component pattern comprises the following aspects:

- The communication part is modeled as a *bus transactor* that converts the specific communication protocol of the interconnect architecture into a generic communication TLM API. This generic API permits a simple and canonical processing of the transaction inside the component.
- The memory-mapped registers and memories of the peripheral correspond to the storage and synchronization layer.
- The behavior is modeled in terms of passive *callback functions*, which are activated when a particular region or field of the register interface is accessed. Of course, the behavior can also be registered directly to the generic interface without an intermediate register layer.
- The timing layer can be realized in the transactor or as part of the behavior. The latter is usually required to model data dependent timing. Modeling timing is discussed in more detail later.

This modeling pattern fits nicely to any memory mapped peripherals, as well as to blocks where functionality is triggered through a simple register write and read.

We have selected the Programmers View API [1] as the generic interface between both the bus transactor and the register interface as well as between the register interface and the behavior. The reason for this choice is twofold:

First, the PV API can be seen as the most simple communication interface for behavioral modeling. Yet the PV transaction data structures contain a sufficient number of attributes to capture the relevant aspects of any communication in a platform. A notable exception is of course the timing information of the communication, which is only included at the level of the Architects View or below. A PV-based peripheral modeling methodology

greatly increases the modeling productivity, because the effort of creating a TL model for the PV use case is approximately only 1/7 to 1/10 compared to the effort of creating a synthesizable RT-level implementation model of the same component.

Secondly, the PV API is used for the functional modeling of bus nodes. In case of a PV platform model we can omit the bus transactor and can directly hook the register interface to the PV bus node. This gives an extra boost to the simulation speed, which is required for the Programmers View use model.

Target Modeling Objects

The target modeling pattern can be nicely supported with a set of modeling objects for the storage and synchronization later. These memory objects provide the means to store data items. The data has to be readable and writable both from the bus transactor as well as from the behavior. Additionally, the scml objects are responsible for synchronizing behavior and communication. In case of a target peripheral this translates into the activation of behavior in response to communication events.

As shown in the center of figure 3 above, a memory object essentially acts as an array of data items. All kinds of accesses and arithmetical operators are defined, so the usage of the memory object is transparent for the behavior. From the communication side, the location of the array in the bus address map is defined at construction time. Hence, the memory object can autonomously decode incoming transactions and store or fetch the data item at or from the right location. This also covers the automatic processing of burst transactions, where a range of data items is written or read in one go. These features are usually sufficient to model plain storage elements like for example memories or passive register banks.

Obviously, the processing of more complex transactions cannot be done automatically, but requires the activation of user-defined behavior. In this case, the memory object is able to activate certain portions of the behavior. This kind of invocation of passive behavior is realized by means of simple call-back functions (represented by the curved arrows). These are regular functions carrying the signature of a PV transport call, which model the behavior and which can be associated with a memory object. Whenever the memory object is accessed by a transaction, the registered callback is activated.

However, it turns out that only one callback per memory object is not sufficient for practical usage. In case a memory object for example represents the control register

of a peripheral, a specific behavior can be associated with an individual bit of this register. For this purpose, we have conceived the concept of an alias to a certain region or bitfield in the memory. An alias does not represent additional storage, but enables the fine-granular registration of call-back functions to arbitrary regions of the memory object.

Target Modeling Example

The small example depicted in figure 4 below illustrates the value of the target modeling objects in terms of modeling efficiency.

Module Declaration	Module Construction
<pre> 1 class Target_PV : public sc_module 2 public : 3 PVTarget_port <uint, uint> p_PV; 4 5 private : 6 scml_memory <uint> m_RegBank; 7 8 scml_memory <uint> m_RWRegister; 9 scml_memory <uint> m_TRegister; 10 11 ReadRegCB(accessSize, offset); 12 WriteRegCB(d, accessSize, offset); 13 RSP TransportRegCB (const REQ &request); 14 }; </pre>	<pre> 1 Target_PV::Target_PV(sc_module_name & n) : 2 sc_module(n), 3 p_PV("p_PV"), 4 m_RegBank("RegBank", scml_memsize(4)), 5 m_RWRegister("RWReg", m_RegBank, 0, 1), 6 m_TRegister("TReg", m_RegBank, 2, 1) 7 { 8 9 m_RegisterBank.initialize(0); 10 11 REGISTER_WRITE(m_RWRegister, WriteRegCB); 12 REGISTER_READ(m_RWRegister, ReadRegCB); 13 14 REGISTER_TRANSPORT(m_TRegister, 15 TransportRegCB); 16 17 p_PV(m_RegisterBank); 18 } </pre>

Figure 4: Target Peripheral Example.

The module declaration in the left part of figure 4 shows the declaration of the memory objects (lines 6-9) and the callback functions (lines 11-13). The right part shows the construction of the register file memory map (lines 4-6) and the registration of the callback functions to the alias registers (lines 11-15). Note that `m_RWRegister` and `m_TRegister` are alias registers, because they are constructed with `m_RegBank` as a parent object.

The implementation of the callback functions representing the behavior is not shown. The behavior boils down to the actual functionality, since most of the modeling overhead in terms of address decoding, endianness conversion, error handling, and debug messages is implemented in the `scml_memory` objects. According to our experiences the usage of memory objects significantly reduces the code size and improves the modeling productivity by about 30% compared to a plain PV modeling style.

Modeling TLM Initiators

Initiators are SOC platform elements like dedicated processing elements, DMA controllers, or traffic generators, which actively insert transactions into the communication architecture. Programmable cores are modeled as Instruction Set Simulators (ISSes) and are not considered

in this article. The following figure 5 shows the proposed TLM modeling pattern for the modeling of non-programmable initiator.

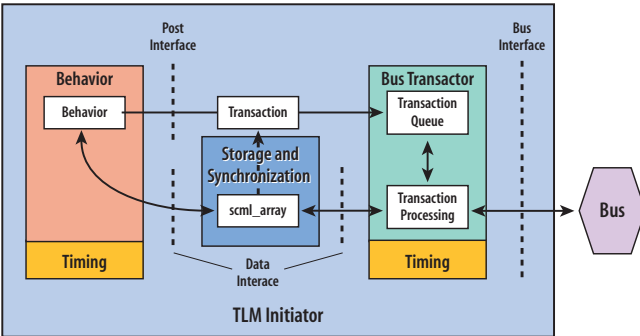


Figure 5: TLM Pattern for Initiator Peripherals.

The four parts of the initiator TLM pattern are defined as follows:

- The user-defined behavior is modeled in terms of autonomous SystemC processes, which actively initiate transactions.
- The storage and synchronization layer is composed of a post port and an initiator storage element called `scml_array`. The posting of transaction is non-blocking and merely specifies the necessary transaction attributes. The real synchronization of the behavior is based on the availability of data and space in the `scml_array` objects.
- The communication part is modeled in the bus transactor. Here posted transactions are queued and converted into actual bus transactions according to the respective bus protocol. The bus transactor operates on the same storage elements as the behavior to avoid needless copying of data.
- The timing layer can be realized by means of clock modeling objects or any mechanism for implicit and explicit timing annotation provided by the TLM communication API.

The interaction between the behavior on the one hand and the synchronization and storage layer on the other hand is not obvious. Unfortunately, the initiator side of the PV initiator API is too simplistic to enable reusability at different abstraction levels. The blocking semantics of the PV transport call rule out the possibility to refine the communication to a more realistic bus protocol.

In order to enable reuse of initiator peripherals we have developed a generic TLM initiator API, which combines the non-blocking posting of transaction with the synchro-

nization on data and space availability in the `scml_array` objects. The transaction data structure is a subtype of the regular PV data structure with a few additional attributes.

Initiator Modeling Objects

In analogy to the `scml_memory` on the target side, `scml_array` is the key modeling element on the initiator side. In general terms, the major purpose of this object is to synchronize the data exchange between a producer and a consumer. Hence, `scml_array` can be seen as a data manager for data items which are stored in an array. Similar concepts can be found in the Task Transaction Level (TTL) interface defined by Philips [2]. The hardware analogy of `scml_array` would be a data buffer, where the incoming and outgoing data is stored.

This paragraph briefly introduces the concept of synchronization on data and space availability using `scml_array`. As shown in figure 6, a producer first needs to claim the required space in the array before it can write the data. This is a blocking operation, which does not return before the space is available. On the other side, a consumer has to claim the data before it can read the data items. This `claim_data` operation is also blocking, which only returns when the producer releases the region in the `scml_array` (see the upper dotted arrow in figure 6). Now the consumer owns the valid data, so a subsequent `claim_space` on the producer side only returns when the consumer releases the array region (see the lower dotted arrow in figure 6).

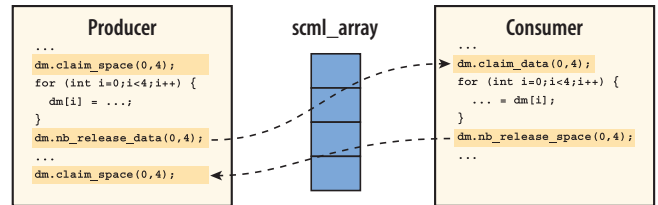


Figure 6: Synchronization of Data Access Using `scml_array`.

By means of `scml_array`, the producer and the consumer can communicate through a shared memory region without additional copy operations. During the period between the `claim_space` and `release_data` operations the producer module can use the same memory just like any regular array container for the actual processing.

In the context of modeling TLM initiators, `scml_array` separates the user-defined behavior from the bus transactor. During a write operation, the behavior acts as producer and the bus transactor represents the consumer. In case of a read transaction, on the other hand, the bus transactor produces the data requested by the behavior.

Note that the actual data is shared between producer and consumer to improve the simulation speed. Note also that the claim operations never actually block in case a `scml_array`-based initiator is attached to a pure PV system, because all communication is happening without delay. Hence in a pure PV system no SystemC event notification occurs, which is again important to maintain high simulation speed. Regrettably, the discussion of an example for the initiator TLM pattern is beyond the space constraints of this article.

Modeling Timing

The entry point for the creation of SoC peripheral models is the Programmer's View API. These purely functional models need to be augmented with timing information in order to reuse them for the AV and VV use cases. Hence the upgrading of functional models with timing information is a cornerstone of the overall Platform-Driven ESL Design modeling methodology.

In general we distinguish 3 different ways of adding timing to a functional peripheral:

1. Explicit Timing using SystemC objects like clocks and events.
2. Static Timing Annotation using annotation in the transactors.
3. Dynamic Timing Annotation using annotation in the behavior.

The techniques 2 and 3 are definitely the preferred ones, because they do not compromise the simulation speed. This way, the timing annotated models are still applicable for the Programmers View use case. The details of the timing annotation techniques for PV and AV use cases are already discussed in [3], so we will restrict this article to an overview of the major concepts.

Timing Annotation Principles

In general, *timing annotation* refers to the specification of delays as part of the TLM API instead of using `wait` or other forms of delayed event notification in SystemC. The benefit of this approach is, that the realization of the timing is deferred from the component to the interconnect models. Now these interconnect models can decide to ignore the annotation (in the PV use case) or to translate the annotated timing into delays (AV use case) or cycles (VV use case).

Timed PV Target Model	Annotated PV Target Model
<pre>PVResp& transport(const PVReq& req) { PVResp resp = req.obtainResp(); // do processing // ... unsigned int latency = ...; wait(latency * clk_period); return resp; }</pre>	<pre>PVResp& transport(const PVReq& req) { PVResp resp = req.obtainResp(); // do processing // ... unsigned int latency = ...; resp.latency = latency; return resp; }</pre>

Figure 7: Timing Annotation.

The difference between explicit and annotated timing is illustrated in figure 7 above. In both cases the processing latency in the target peripheral is in one way or the other dynamically computed depending on the actual data and the internal state of the peripheral.

- In the explicit case (left), the peripheral itself calls `wait`. Here the peripheral also needs to know about the clock period to translate the number of cycles into an actual delay. The delayed event notification could be avoided by means of a run-time or compile-time configurable conditional statement. However all of this does not help the usability and maintainability of the peripheral models.
- In the annotated case (right) the peripheral annotates the delay to the latency attribute in the PV response data structure. This way, the respective caller of the transport method can decide to use or ignore this information. A PV bus node for example would ignore the annotation whereas an OCP transactor would consider it (see below). Also the clock-period information can be maintained in a more central place.

Timing Annotation Parameters

In principle we could define an arbitrary number of timing parameters. However, an excessive number of parameters would be complex for the peripheral models to compute as well as complex for the interconnect models to interpret. In practice, we made good experience to restrict the granularity of the timing annotation to the boundaries of transactions. This way, two timing parameters characterize the performance of any peripheral:

The *accept delay* specifies the minimum time between two consecutive activations. In essence, the accept delay constraints the bandwidth of a block, that is, during this period a module is busy with the processing of an activation.

The *latency* specifies the time between the process activation and the sending of the result.

In this way, the timing requirements of arbitrary platform building blocks can be roughly specified. For example, a pipelined ASIC block will exhibit an accept delay smaller than the latency, whereas for a task executed on a programmable core it will be the other way around. According to our experiments, sophisticated timing behavior can also be modeled using these two parameters. The timing of a memory controller, for example, is pretty complex and depends on numerous state variables and timing parameters. Still the timing of an individual memory access can be quite accurately characterized by computing the respective accept delay and latency.

Naturally, the concept of a timing annotation scheme based on only two parameters has certain limits in terms of accuracy. In some cases it might be unavoidable to model the timing on a cycle-by-cycle level to reach the required accuracy. However, these models become hard to maintain and are not reusable any more.

Transactors

As already indicated in the figure 2, transactors play an important role to facilitate the reuse of peripherals for multiple use cases. Apart from the plain translation between two TLM APIs, the transactor acts as a *performance overlay model* for the untimed PV peripherals.

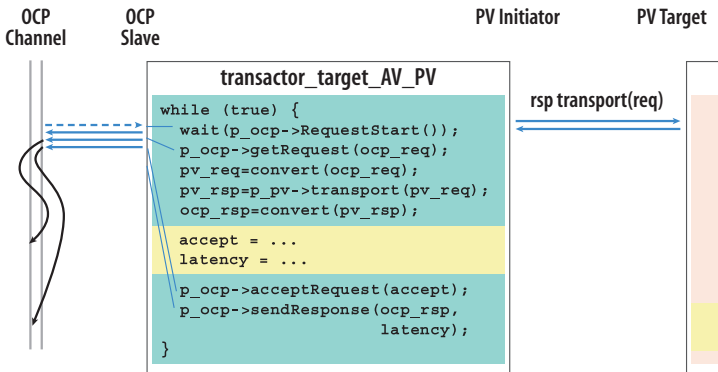


Figure 8: Transactor as a Performance Overlay Model for PV Peripherals.

Figure 8 above illustrates this concept by means of a simplified transactor for hooking PV target peripherals to OCP TL3 channels (please refer to [3] for more detailed explanations). The three dots represent the configurability in the calculation of the timing model.

On the one hand, the delay could be based on static or stochastic values configured in the constructor of the model. In this case the transactor serves as a rough, first-order approximation for any untimed peripheral. On the

other hand, the PV target could already be annotated with timing information. In this case the transactor should evaluate the timing annotation in the PV response data structure.

Creating a transactor is usually not a trivial task. The TL3/PV target transactor is relatively simple, because the TL3 API also supports timing annotation. In other cases, the transactor needs to deal with timing in a more explicit way.

Summary: An ESL Design Flow

In this article we have presented a re-use driven methodology for the transaction-level modeling of SoC peripherals. Under the assumption that Instruction-Set Simulators and bus models at different levels of abstractions are available from IP and ESL tool vendors, peripheral modeling is an essential ingredient to ease the adoption of ESL Design.

The investment in peripheral modeling can be justified much more easily when multiple platform models for different purposes can be built from the same model base. In other words, the individual ESL design tasks like architecture exploration, software development, and verification should be combined into a seamless ESL design flow.

An ideal ESL design flow is depicted in figure 1 on page 1. The Functional View transaction-level model of the algorithm represents the optional starting point. In case an executable specification of the application is available (e.g. from an algorithm development environment with SystemC export capabilities) it can be used by architects to define the right HW/SW partitioning and to explore the system architecture.

Alternatively, the Architect’s View model of the platform can be constructed from non-functional models of the application workload. The focus of the AV platform model is usually on the interconnect architecture and the memory subsystems. These peripherals should already deploy the modeling methodology outlined in this article to be reusable for further ESL design tasks.

A platform model for software development can be extracted from the AV platform model by integrating instruction-accurate processor simulators and replacing the AV-level bus with a simple PV bus. The peripheral models are reused by removing the PV/AV transactors. This effectively eliminates any timing information that is not important for the embedded software designer in order to achieve the highest possible simulation speed.

On the other hand, additional peripherals need to be added to the platform in order to complete the programmer's model of the platform.

Once the complete set of peripherals is available, a cycle-accurate model of the platform can be constructed. Here a cycle-accurate interconnect model needs to be integrated and the instruction-accurate ISSes need to be replaced by their cycle-accurate counterpart. The peripherals can again be re-used by introducing the corresponding transactors and refining the timing annotation to an appropriate level. The resulting VV platform model can be used by verification engineers to perform HW/SW as well as TLM/RTL co-verification.

Obviously, this flow assumes the availability of transaction-level models for the processor and interconnect architectures on different levels of abstraction. Fueled by the progress of the standardization in IEEE, OSCI and OCP-IP, the amount of models available from IP providers and ESL tool vendors is constantly increasing. The methodology described in this article leverages currently available TLM API standards as much as possible to benefit from this SystemC ecosystem.

References

- [1] Frank Ghenassia (Ed.), *Transaction-Level Modeling with SystemC*, Springer, 2005
- [2] Pieter van der Wolf et al., *Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach*, CODES+ISSS, 2004
- [3] Tim Kogel, Anssi Haverinen, James Aldis, *OCP TLM for Architectural Modeling*, OCP-IP whitepaper, 2005, available from www.ocpip.org



The ESL Design Leader

CoWare, Inc.
Corporate Headquarters
1732 N. First Street
San Jose, CA 95112

Main: 408-436-4720
Fax: 408-436-4740
www.CoWare.com