Chapter 2

TRANSACTION LEVEL MODELING

An Abstraction Beyond RTL

Laurent Maillet-Contoz and Frank Ghenassia
STMicroelectronics, France

Abstract: Transaction level modeling (TLM) is put forward as a promising solution above Register Transfer Level (RTL) in the SoC design flow. This chapter formalizes TLM abstractions to offer untimed and timed models to tackle SoC design activities ranging from early software development to architecture analysis and functional verification. The most rewarding benefit of TLM is the veritable hardware/software co-design founded on a unique reference, culminating in reduced time-to-market and comprehensive cross-team design methodology.

Key words: transaction; untimed model; timed model; initiator; target; channel; port; concurrent processes; timing accuracy; data granularity; model of computation; system synchronization; functional delay; annotated model; standalone timed model.

1. THE REVOLUTION

1.1 Call for Raising Abstraction Level

Squeezed by the ever-increasing SoC design complexity, cost, and time-to-market stress, the much-perturbed SoC industry is longing for a solution. The key to this solution is to improve the design productivity through a more reliable design methodology within a shorter design time-frame.

Forwarding critical software development earlier in the SoC design flow is unquestionably helpful to reduce the design cycle time. Such advance implies indeed a hardware/software co-design wherein the software is developed in parallel with the hardware for earlier system integration.

To cope with the rising SoC complexity, a much more rigorous methodology is sought after to assure the reliability of SoC performance at

F. Ghenassia (ed.), Transaction Level Modeling with SystemC, 23-55.

an earlier stage of the design cycle. A favorable approach is the architecture exploration that analyzes the potential effect of the realistic traffic performed by a system.

Pulling all these factors together, raising the level of abstraction above RTL in the overall SoC design and verification flow has appeared to be a promising solution for the SoC industry.

1.2 Attempts at Raising Abstraction Level

Bear in mind that any attempt made to raise the abstraction level is always a game of balancing the trade-off between the speed and accuracy of a potential simulation model. Our development effort has of course witnessed this game from tip to toe. Before tackling the subject of abstraction level, it is worth considering what the two extreme ends of the SoC design flow could offer.

First, consider the algorithmic model at the highest end of the flow. A complex design usually begins with the development of such a functional model. As an example, a digital signal processing oriented design will have a dataflow simulation engine as its algorithmic model. Since it only captures the algorithm regardless of the implementation details, an algorithmic model has a huge advantage in its high simulation speed. In spite of this, an algorithmic model has no notion of hardware or software component; it models neither registers nor system synchronizations related to SoC architecture. This model therefore cannot fulfill the need of executing the embedded software.

On the other end of the design flow, a pure logic simulation can take place at the register transfer level (RTL). In a conventional SoC logic simulation, RTL models written in hardware description language (HDL) such as VHDL and Verilog are employed as the system hardware. If a processor model is necessary, a design sign-off model (DSM) will typically be used. The advantage of the logic simulation is evidently its great fidelity to the real implementation, i.e. accurate SoC functional and performance analysis. This is nonetheless a price too expensive to pay in terms of the lengthy simulation time. The time consumption has actually further worsened lately due to the high SoC complexity that requires a longer RTL development phase. Moreover, a pure logic simulation cannot execute any software in a reasonable amount of time. A system can only integrate its associated software for observation and analysis rather late in the design flow. Since the breadboard is usually almost ready at this point, any system modification will certainly be too costly at this stage.

In brief, an in-between solution has to be resolved for which three fundamental criteria must always be respected as the doorway to early software development and architecture exploration:

1. *Speed*. The potential model must simulate millions of cycles within a reasonable time length. The target activities frequently involve a very large scale of simulation cycles. Some of them may entail user interactions that could probably slow down the process. It is unacceptable and unaffordable to wait for even just a day to complete a simulation run.
2. *Accuracy*. Although speed is an interesting advantage to enhance, the potential model should sustain a certain degree of accuracy to deliver reliable simulation results. Some of the analyses may require full-cycle accuracy to obtain adequate outcomes. As a rule of thumb, the potential model should at least be detailed enough to run the related embedded software.
3. *Lightweight Modeling*. Any other modeling effort in addition to the compulsory RTL modeling for hardware synthesis must be kept insubstantial to optimize the overall SoC project cost. The potential model should be, for this reason, a quick-to-develop model at a considerably low effort.

Collected here are some attempts to raising the abstraction level. Brief descriptions are provided for these attempts, including hardware/software co-verification, cycle-accurate model, and temporal model.

- **Hardware/Software Co-Verification**

The concept of hardware/software co-verification is suggested for reducing the critical SoC design time and cost to overcome the limitation of pure logic simulations. The underlying idea of this concept aims at leading hardware/software integration, verification, and debugging to an early phase of the design cycle before the real hardware is available.

RTL models remain the hardware models in a co-verification platform. An obvious difference from pure logic simulation is that co-verification uses a faster processor model, i.e. Instruction Set Simulator (ISS). This is an instruction-accurate model developed in C language at a higher level of abstraction.

The co-existence of hardware and software during the SoC verification process is the essence of co-verification. While the hardware platform is connected to a logic simulator, a symbolic debugger links the associated software program to the ISS for its execution on the platform. Such co-operation offers a simultaneous controllability and visibility over both hardware and software to analyze the system behavior or performance. The simulation speed is of orders of magnitude higher than the one of logic

simulation. Since the breadboard is not manufactured yet, any modification of the system hardware or software at this stage will be both time and cost-efficient.

Despite the numerous benefits yielded by the co-verification, it is still too long to wait for the development of RTL hardware models before the co-verification can be conducted. The time pressure has pushed us to tackle another approach: cycle-accurate model.

- **Cycle-Accurate Model**

This attempt tries to replace the non-processor hardware parts by a model residing at higher level of abstraction. The prospective model could be developed using high-level programming languages such as C. Compared to RTL models, this model is less precise. It is sensitive to whatever happens at the interval of each clock cycle, which is more than enough for software verification but not providing any synthesizable description.

With the emerging C-based dialects that support hardware concepts, it seems convincing that cycle-accurate models developed in a C-based environment could meet the three criteria mentioned earlier for raising the abstraction level. However, this hypothesis has stumbled upon a few obstacles [1-4]:

a) Most of the information captured by cycle-accurate models is unavailable in IP documentation but only in the designer's very mind and the RTL source code itself! Consequently, RTL designers have to invest much time to keep modeling engineers informed; otherwise modeling engineers must reverse-engineer the related RTL code. Either way ends up being a tedious and time-consuming process without actually solving the issue.
b) Cycle-accurate models can simulate merely an order of magnitude faster than the equivalent RTL models, which is really just too close to the speed of VHDL/Verilog models.

Not only is simulation speed too slow to run a significant amount of embedded software in a given time-frame, the development cost is also too dear to compensate for the negligible benefits of cycle-accurate models. In addition, architects and software engineers do not require cycle-accuracy for all of their activities; for instance, the software development may not involve any cycle-accuracy until engineers work on the optimization.

- **Temporal Model**

Instead of balancing speed and accuracy, the temporal model is attempted as quite a different approach to raise the abstraction level. This model is mainly opted for the performance analysis of a system. While timing analysis is the focus of temporal models, analytical accuracy is forgone.

Some efforts were given in the development of the temporal model. The resulted model provided extremely high simulation speed but with little or virtually no functional accuracy guaranteed. The temporal model is thus far from being the ideal solution to our need of raising the abstraction level.

1.3 Birth of Transaction Level Modeling

Through our different attempts for raising the abstraction level, we have concluded that the most compelling resolution is to adopt the famous "divide and conquer" approach. This approach counts on two complementary environments as the best bid to balance the trade-off between simulation speed and accuracy, i.e. transaction level modeling (TLM) platform and register transfer level (RTL) platform.

- **SoC TLM Platform**
  TLM platform is intended for early SoC exploration in the design flow at a relatively lightweight development effort. It is a transaction-based abstraction level residing between the bit-true cycle-accurate model and the untimed algorithmic model. Our development work has demonstrated that SoC TLM platform makes an excellent complement to RTL platform as an adequate trade-off between simulation speed and accuracy. On top of the untimed functional TLM, it is also possible to add timing annotations to TLM platforms for early performance analysis without paying the cost of cycle accurate models.
- **SoC RTL Platform**
  RTL platform aims for fine-grain SoC simulations at the expense of slower simulation speed and later availability. It applies cycle-accurate HDL models for a detailed timing analysis.

The idea of "divide and conquer" proves itself an extremely efficient modeling strategy. With the high modeling and simulation speed offered by TLM platforms, potential users could quickly accomplish a systematic analysis for a given SoC as the first approach. A comprehensive timing analysis based on RTL platforms will follow afterward to provide results that are more accurate. Hence, this complementary characteristic enables a system-under-design to go through rapid methodical study as well as in-depth exploration. Figure 2-1 gives the efficiency levels of the different modeling strategies, including RTL, cycle-accurate model (CA), and TLM. It shows clearly how TLM helps the concept of "divide and conquer" become a success through its high modeling and simulation speed.

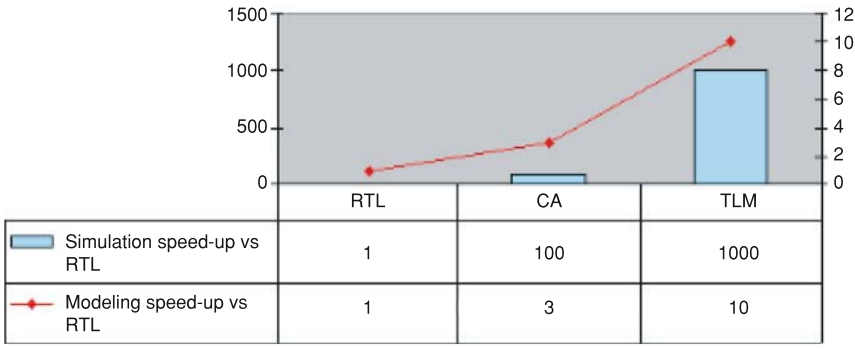

Figure 2-1. Efficiency of Modeling Strategies

A question wondering in your mind now could probably be "Why would TLM be so interesting compared to other rival propositions?" The answer is that we have successfully identified the appropriate level of abstraction, *TLM*, which has a description usable for embedded software development and early architecture analysis thanks to its adequate trade-off between simulation speed and accuracy.

Most of the propositions available in the field use proprietary C-based languages such as SpecC, Hpascal or HardwareC to implement cycle accurate models. High-level models, on the other hand, are either expensive solutions sold by CAD vendors or limited versions reserved for academic applications. Although these high-level models give temporal view of a system, they are not precise enough to develop any embedded software.

Before considering the advantages that TLM has to offer, its very distinct point from other propositions is the use of SystemC *-an open-source programming language-* that suggests a free of charge development environment for a tangible solution.

SystemC provides a foundation to model hardware and software of a system based on a single language. It is an object-oriented approach built on top of C++ as a set of classes. A system conceived by SystemC demonstrates particular characteristics in concurrency, reactivity, distributiveness, timing, and data types. Further details of TLM modeling techniques using System C will be discussed in Chapter 3.

The remainder of this chapter presents a zoom-in discussion on TLM ranging from its principles to its battle against the SoC design bottlenecks.

2. PRINCIPLES OF TLM

2.1 Terminology

TLM offers a new SoC design methodology at a higher abstraction level above RTL, i.e. a transaction-level modeling technique intended for digital electronic systems.

In a digital electronic system, every single component is composed of a finite set of states and a series of concurrent behavior. TLM models each of these components as a ***module***. The internal states of a component are represented by a set of variables defined within the scope of the corresponding TLM module, whereas the different behavior pieces of the component are modeled by a collection of ***concurrent processes*** or ***threads***, which can be executed in parallel.

Just like the components of a SoC, TLM modules are gathered to form a TLM system. Through a specific TLM communication structure, namely ***channel*** or ***interconnect***, communications are established between modules. Depending on the accuracy level required by the corresponding simulation, a channel could be a simple router, an abstract bus model, a network-on-chip model, or some other structures. This is essentially the very part that separates communication from computation in TLM modeling.

Modules and channels are bound to each other by means of communication ***ports***. Once they are bound together, data can be exchanged between them to perform the expected system behavior. Potentially, data can also be communicated between modules and test-benches.

The term ***transaction*** denotes the set of data being exchanged. A ***master*** or ***initiator*** is a module that initiates transactions in a system, while a ***slave*** or ***target*** is a module that receives and serves transactional requests. Any consecutive transactions may have various sizes of data transfer. This variable size corresponds to the amount of data being exchanged between two occurrences of system synchronization.

System synchronization is an explicit action between at least two modules (potentially test-benches) that need to coordinate or manage some behavior distributed over them. Such co-operation of different modules is vital to assure the predictable system behavior.

Since it is the only mechanism available for synchronizing the different processes in a system, the explicit system synchronization is compulsory to ensure a proper deterministic SoC TLM behavior. An example of system synchronization is the interrupt raised by a direct memory access (DMA) to notify a transfer completion within a system.

2.2 Modeling Approach

The terms of TLM defined in the last section can be attained through an appropriate electronic system level (ESL) modeling approach. The right candidate to do this job is a high-level programming language that is capable of developing not only a plain software program, but also of modeling electronic hardware at the conceptual level without describing the real implementation. The potential candidates include SystemC, SpecC, Hpascal, System Verilog, HardwareC, and the like. In our opinion, SystemC is the best candidate and we therefore rely on it for all of our TLM models.

As discussed earlier, a SoC component is modeled as a module in TLM. The primary modeling effort lies in the internal computation of the given hardware block at the functional or behavioral level. The input and output of the block as well as its synchronization are to be modeled. None of the micro-architectural implementation details should be included, i.e. neither internal pipelines nor structures are modeled. To sum up, TLM modules representing SoC hardware blocks or IPs must hold the three characteristics stated below:

1. bit-true behavior of the component;
2. register-accurate interface of the component;
3. system synchronizations managed by the component.

A complete SoC TLM platform is constructed by instantiating and binding different modules and channels together. Once the platform is integrated, SoC simulation is performed by executing the related embedded software either as native or cross compilation. The earlier is executed on a simulation workstation for fast simulation speed, while the latter is executed on the embedded processor architecture, i.e. ISS, for precise simulation accuracy.

To ensure a proper system functional behavior in TLM SoC simulation, there are two essential points that deserve attention in the modeling process. First, all the data transactions must be blocking i.e. the thread that initiates the transaction will resume its execution only if the current transaction is completed. Second, all the occurrences of the system synchronization must be potential re-scheduling points in a simulation environment in order to

guarantee an accurate simulation of the concurrency. The system synchronization could be modeled by specific means such as event, signal, and interrupt; or by data-exchanges such as polling. If any of these potential system synchronizations causes a call to the simulation kernel, it enables the scheduler to activate other modules. Hence, the simulated system will behave correctly in line with its functional concurrency.

The essence of working out an appropriate model at transactional level lies in the good sense of deciding where and when to implement system synchronization. If too many synchronized points are inserted, the model will tend to be too close to cycle-accurate or RTL models that will not help to gain much simulation speed. Contrarily, if too few synchronized points are implemented, the model may run the risk of having incorrect system execution.

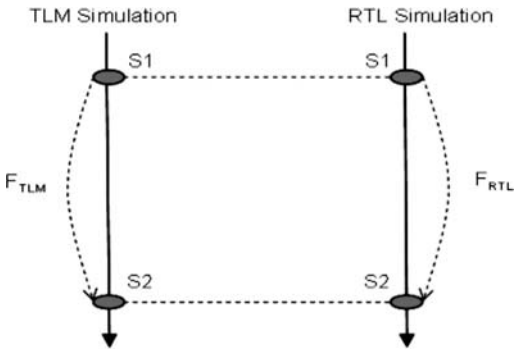

Figure 2-2. TLM vs RTL Simulation

Consider the two simulations depicted in Figure 2-2, which are correspondingly the RTL and TLM simulations for a given system. The evolution of the system from the first stable system state, S1, to the next stable system state, S2, is represented by F_{RTL} and F_{TLM} respectively. Indeed, S1 and S2 are two partial observation points in simulation, i.e. two synchronization points.

F_{RTL} is a collection of all necessary cycle-accurate computations to bring S1 to S2. These calculations are implemented by a set of clocked processes that represent the system micro-architecture. Upon each clock cycle, these processes are activated in the simulation kernel for execution; and that will consequently involve countless of context switches.

On the other hand, F_{TLM} is an equivalent function to bring S1 to S2 but without any clock implementation. Computations are defined by some high-level programming languages such as C or C++. There is principally

sequential execution of programming codes between S1 and S2. Compared to RTL simulation, it involves much fewer parallel executions of processes. As a result, there are relatively less context switches involved.

Recall the efficiency levels of different modeling techniques illustrated in Figure 2-1, the simulation speed-up achieved by TLM is vastly ahead of RTL up to a factor of 1000. Indeed, this speed-up correlates directly with the number of processes and context switches activated between two occurrences of system synchronization by RTL simulation *but not* by TLM simulation kernel.

2.3 Modeling Accuracy

The modeling accuracy of a given modeling approach indicates the precision or correctness of the model in replicating the intended behavior and activities of a system-under-design. For any modeling strategy in the SoC design flow, there are two decisive factors to determine the degree of modeling accuracy:

1. *Granularity of Communication Data*.
   This criterion reflects the fineness or coarseness of the data carried by the communication structure of a model. The data granularity can generally be categorized into three levels, i.e. application packet, bus packet, and bus size, in the order of increasing accurateness. The transfer of a video IP helps to illustrate the idea of data granularity. If the IP has a frame-based algorithm, a coarse granularity at application packet could be modeled as a frame-by-frame transfer. A finer granularity at bus packet level can be represented by a line- or column-based transfer, or a macro-block transfer consisting both lines and columns. The finest grain at bus size level will be the pixel-based transfer of the video.
2. *Timing Accuracy*.
   Timing accuracy determines the fidelity of a model to the intended timing behavior. It can be conceptually perceived as a scale of two extremes, i.e. untimed level and cycle-accurate level. Moving from the untimed end towards the cycle-accurate end will increase the timing accuracy of a model. Any level falling in between the two ends is considered as *approximately* timed level.

Just as any other modeling strategies in the SoC design flow do, the TLM approach naturally revolves around the two factors above to decide its modeling accuracy. Guided by these criteria, we have conceived two fundamental classes of TLM to date through our development effort:

- Untimed TLM.
- Timed TLM.

The untimed and timed TLM are models tailored for distinct purposes. The ultimate goal is to create a unique platform that simulates two different models according to user needs.

The untimed TLM is an *architectural* model targeted specifically at early functional software development and functional verification where timing annotations are not compulsory conditions. The high simulation speed is the objective of this model. Since the untimed TLM serves primarily programmers, it is hence given another name as *programmer's view* (PV).

On the other hand, the timed TLM is a *micro-architectural* model containing essential time annotations for behavioral and communication specifications. It is relatively a less abstract model located lower in the SoC design flow. The focus of timed TLM is the simulation accuracy required by real-time embedded software development and architecture analysis. Hence, the timed TLM is also known as *programmer's view plus timing* (PVT).

Figure 2-3 gives a glimpse at the modeling accuracy of the untimed and timed TLM with respect to other conventional models in the SoC design flow, including register transfer level (RTL), bus cycle accurate (BCA), and cycle accurate (CA) models.

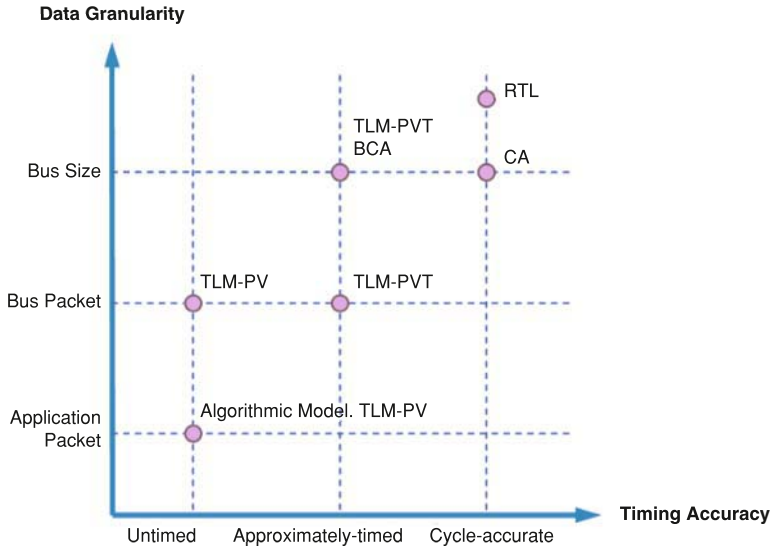

Figure 2-3. Modeling Accuracy of Various Approaches

3. UNTIMED TLM

3.1 Introduction

The untimed TLM is a level particularly conceived for serving software programmers and verification engineers in early functional software development and functional verification. Timing annotations are insignificant at this untimed level; thus, none of the information related to the micro-architecture of the component or IP-under-design should be included.

For the same reason, any information related to the interconnect topology and arbitration law will not be captured in the untimed TLM. The internal states of a component are modeled by using appropriate internal variables.

Certain information, for instance, the register bank or memory content of a given component, is made available and accessible to the outside world through a well-defined Application Programming Interface (API). The communication API is a blocking API that provides a particular interface to supervise full data transfer.

The granularity of the data transferred should correspond to the modeling level related to the target application. For example, data transfer of an image-processing block should be modeled at the frame level, i.e. one frame being transferred at a time rather than creating transfers of the bus width.

3.2 Model of Computation

The untimed TLM has absolutely no timing information related to the micro-architecture, i.e. there is *no* clock in an untimed TLM system. Since it has no clocked timing regulation, all processes are executed concurrently to access any of the system resources at the same time instant. Yet, the system must demonstrate a correct behavior during the parallel execution of concurrent processes. This implies that untimed TLM systems must respect a certain degree of process execution order to guarantee a proper system functional performance.

To fulfill this requirement, the untimed TLM employs a specific model of computation with the following characteristics:

1. concurrent execution of independent processes;
2. respect for causal dependencies between processes using system synchronization;
3. bit-true behavior;
4. bit-true communication.

3.2.1 System Synchronization

A system must clearly characterize the causal relation between its different processes in order to assure deterministic system behavior. The explicit system synchronization is therefore implemented within a system to respect such causal dependencies. The system synchronization only defines a partial execution order for SoC internal events, i.e. a partial execution order between the different processes in the whole system. In other words, any particular execution order among all of the processes is permitted as long as their causal dependencies are well respected.

To better illustrate this idea, consider three processes in a given system, P1, P2, and P3, as depicted in Figure 2-4. Assume that each process denotes a thread for a particular module in the system.

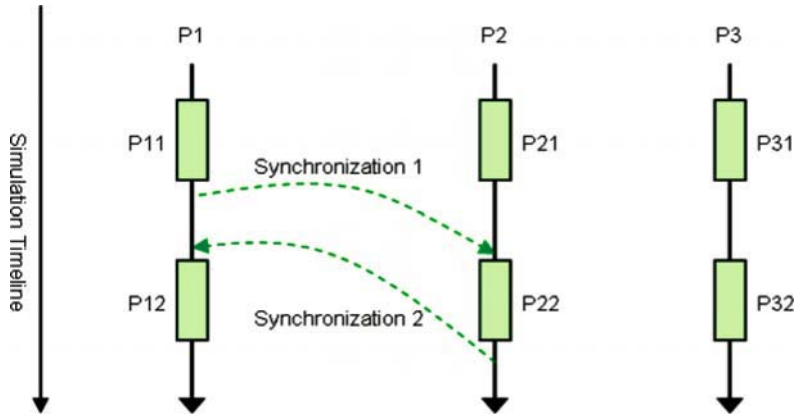

Figure 2-4. System Synchronization between Processes

The full execution order within each of these processes is represented by their own internal synchronized events:

a) P11 → P12 for process P1
b) P21 → P22 for process P2
c) P31 → P32 for process P3

Bear in mind that this "full" order is only a locally complete order within each process. It is indeed a "partial" execution order from the point view of the overall system execution. Besides, there are two occurrences of system synchronization between P1 and P2, which give additional constraints to the overall system execution order:

d) P11 → P22
e) P22 → P12

The constraints of execution order stated from (a) to (e) clearly describe the causal dependencies that must be respected within the system. The three processes can be executed with any particular order as long as these causal dependencies are followed. Here are some examples of the different *overall* system execution order (which are also known as process interleaves):

f) P21 → P11 → P22 → P12 → P31 → P32
g) P31 → P32 → P21 → P11 → P22 → P12
h) P11 → P21 → P22 → P31 → P32 → P12

The system synchronization is a mechanism to inform others or to get informed by others about some system state changes when these changes potentially influence the execution of some other parts of the system. In real hardware circuits, system synchronizations are modeled by means of interrupt signals, polling or mailbox. The TLM simulation will implement all of the system synchronizations as interrupts, mailbox or polling in line with the model of computation stated earlier. An abstract implementation of the various synchronization mechanisms, however, could be provided to better match with the considered level of abstraction.

According to its nature of informing or being informed, there are two kinds of synchronization. First, "emit-synchronization". This occurs when a process sends out a synchronization that may influence the behavior or state of other processes. Second, "receive-synchronization". This is a point where a process waits for an incoming event from the system that may influence its behavior or state.

Picture this: every synchronization point is a traffic light in a given system. Each of these "traffic lights" is associated to a certain condition; for instance, the occurrence of an event or the computation of a particular value. Once this condition is fulfilled, the green light will be on to allow the system to proceed to the next execution point. Otherwise, the red light is there to stop it. All these little "traffic lights" scattered in the system has a big mission: work hand-in-hand to guarantee a proper predicted system behavior.

An important employment of the system synchronization is the assurance of memory or data consistency. Here, the system synchronization prevents concurrent processes from reading data content at unknown state; it also prevents them from writing data at temporarily inaccessible memory area.

A direct beneficial impact of the system synchronization is the capability of executing any legal interleaves of processes without breaking the overall system synchronization. The system synchronization also serves as an efficient method to improve the validation of the system simulation model by allowing more process interleaves to be tested. The model of computation only requires the causal dependencies to be respected by the simulation. Thus, it is possible to randomize the process selection as long as the system

synchronization does not define a full order of process execution. This is particularly useful in the case where simulation kernels do not provide random process execution.

All of the system synchronization points in a system must be explicitly modeled for a correct system behavior. If an untimed TLM system ever generates any simulation deadlocks or failures, the explanation will be the system synchronization not being explicitly modeled to the fullest, or simply badly designed. By slight chances, a system with incomplete synchronization modeling may appear to function as normal at certain values of clock frequency. It will however fail to perform at other clock frequencies. Undoubtedly, such incomplete modeling will adversely jeopardize a safe chip execution.

3.2.2 Process Execution

The concurrent execution of independent processes is one of the major characteristics of the untimed TLM. Simulation kernels are usually implemented in such a way that they offer repeatable process executions to simplify debug activities. Note that simulation kernels cannot give a deterministic execution of concurrent processes (even the language reference manuals cannot guarantee a deterministic execution of concurrent processes). It means that we cannot predict which process that the simulation kernel is going to start executing; but once the simulation is executed, the kernel will repeat the same execution order.

Although the repetitive feature of simulation can facilitate the debugging procedure, a single system execution order may not provide satisfactory validation coverage. In our last example of system synchronization, the overall system execution can start with any of the three processes. If the simulation only covers a single execution order, we would probably miss catching the bugs hidden in other execution orders! As an example, imagine another synchronization that imposes a constraint of executing P21 before P11. If the repetitive simulation kernel picks the system execution order of (f) or (g), the simulation will pass without detecting any error. An error, however, would have occurred in the system performance by following the execution order of (h) where P21 is not executed before P11.

To tackle this limitation, we must make sure that any execution order will conform to the system functional specification. An appropriate solution to increase the coverage of system execution orders will be extending the standard simulation kernel with a random function that shuffles all of the legal process interleaves. With such mechanism, it is feasible to verify all of the possible micro-architectures of a given architecture specification.

This definition actually corresponds to the implementation of asynchronous processes that use synchronization points to ensure a correct execution of the system. If one expects to cover all of the possible process interleaves as in the real-life system, it will obviously produce a huge number of combinations with lots of them being meaningless. Hence, it is worth-noticed that it is possible to reduce the indeterminism of concurrent process execution by introducing successive constraints in the untimed models based on their partial system execution orders.

A typical example is the integration of timing constraints that make sense at the functional level. The objective is to reduce the number of potential process interleaves by adding constraints in the selection of the various processes for the simulation. Here, the timing information is only related to functional constraints (e.g. a video application imposes to decode 30 frames per second), but no information on the micro-architecture is incorporated yet. The result is a decreased indeterminism, which reduces the simulation variants to be considered for the system validation. This will be further discussed in Section 4, *Timed TLM*.

3.2.3 Time-Independent Deterministic Behavior

This section explains how the computational model of the untimed TLM handles the constraints of process execution order without implementing timing characteristics.

Consider a fixed set of input stimuli for a given SoC. The system synchronization points implemented among the different processes will induce a deterministic behavior that is independent of any timing behavior during the simulation. Each of these processes follows a particular sequence as described in Table 2-1.

Table 2-1. Untimed Process Sequence

Step	Action
1	Activate or resume a process.
2	Read input data for control flow and data processing.
3	Computation.
4	Write output data if there is any of them.
5	Return to step 2 if more computation is required.
6	Synchronization: (a) if it is “emit-synchronization”, return to step 2; (b) if it is “receive-synchronization”, the process will be suspended.

When a process reaches step 6 in the untimed sequence, the component state will have already been fully defined, and the memory state modified by the process should be fully defined as well. Only when a process reaches step 6(b) of "receive-synchronization", it will be suspended. This is the only situation where a process needs an update of the system state that might influence its own behavior. As a result, the simulation kernel could by no means suspend a process by itself, i.e. the simulation kernel is not pre-emptive. This will definitely assure predictable process states and process controls, which are independent of any specific implementation of the simulation kernel.

Most of the time, a process could include many synchronization points and that will produce a very complex control flow graph with many possible activation-synchronization paths. Note that reducing the number of the descheduling points in a system model to the "receive-synchronization" can be very beneficial. While assuring a correct simulation of the SoC architecture, such reduction can greatly minimize the number of context switches compared to other computational models. Therefore, the kernel overhead is minimal, leading to the simulation speed close to the one of pure algorithm.

3.3 Modeling of Interrupts

Literally, interrupts mean disruptions that could result in certain consequences. For electronic systems, an interrupt is considered as a system event with side effects such as triggering a delayed management of processes or updating registers of interrupt-status.

Recall that system synchronization is very often implemented by an interrupt signal. In the untimed view, an interrupt is however an impulsive system event without any persistence. It is therefore inappropriate to model it using a signal. Instead, a dedicated TLM synchronization protocol with the following features is employed:

a) immediate propagation of interrupts from an initiator to a target;
b) notice of potential IP internal state change, i.e. status register update.

While developing untimed interrupt models, the first-in-first-out (FIFO) mechanism must not be implemented in the reception structure as it may cause serialization of concurrent events undesirable at that level. Upon the generation of an interrupt, the target IP may invoke a consequent effect out of its own scope. In that case, meticulous care must be taken so that another process *but* not the one generating the interrupt will handle the consequent effect. This will avoid changes in the system state caused by the process generating the interrupt in the Remote Procedure Call (RPC) coding style.

3.4 Insertion of Functional Delay

At the architectural level, it is still necessary to introduce some functional timing information, i.e. *functional delay*, when these delays are part of the system specification (e.g. a video decoder decodes 30 frames per second).

Sometimes, an untimed TLM IP is inserted with functional delay to implement implicit synchronization points related to specific timing information. As an example, a Liquid Crystal Display (LCD) controller with a screen-refresh frequency of every 1/30-second can be modeled without any explicit synchronization. It means that the untimed LCD controller can be created with implicit timings by adding some delay information and wait statements of specified time length into the model.

From the angle of computational model, such implicit timings bring additional constraints to the execution order of processes in the simulation, and thus reduce the set of possible process interleaves. As a result, the untimed model inserted with functional delay is created as an intermediate level between the purely untimed TLM and the timed TLM. Model developers should guarantee a flexible manipulation of this intermediate model by allowing users to easily enable or disable the annotated delay information. It must leave users enough room to switch back to a purely untimed model for validation purposes. Furthermore, this intermediate model should never cover any functional information related to the micro-architecture such as FIFO, Finite State Machine (FSM) related to cycle-accurate behavior, or any other implementation-dependent features.

Figure 2-5 illustrates the typical timelines of a process execution occurring in the untimed TLM. Two cases are demonstrated:

a) Simulation without functional delays based on a functional specification that only defines sequences of actions.
b) Simulation with functional delays based on a functional specification that defines some timing attributes such as UART baud rates.

Adding functional delays to an untimed model does not particularly influence the model of computation. Processes will still have activation, emit-synchronization, and receive-synchronization points. The execution order of various processes will be more constrained because the inserted functional delays restrict the set of potential process interleaves eligible for simulation. In other words, there are fewer choices of process interleaves for the simulation kernel at a given time instant.

Functional delays can suspend a process to induce the simulation kernel to choose other eligible processes for execution. This cause-and-effect phenomenon can influence the system state, but should *never* cause any system inconsistency from the perspective of computational model. The

reason is that the system synchronization must *fully and explicitly* model all causal relations of a system. An error will otherwise arise in the system synchronization scheme, and that is considered as a serious bug in the SoC specification.

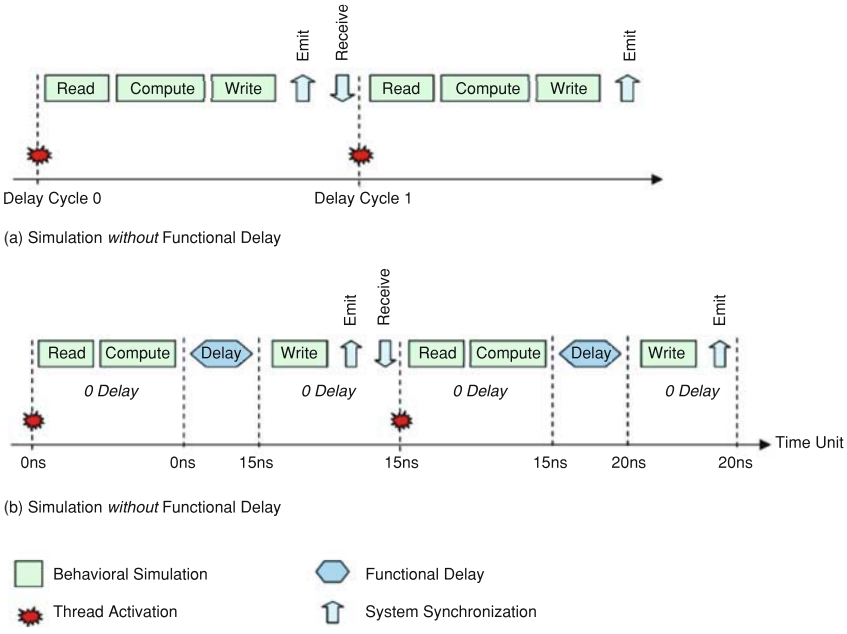

Figure 2-5. Simulation Timelines of Untimed TLM

Let us look into this statement more carefully through an example. Consider a system that is modeled by a group of processes denoted from P1 to Pn. Assume that a functional delay is inserted into the codes of P1, and that induces the simulation kernel to select another process, say P2, for execution. The system state could potentially be affected by the execution of P2. If that is the case, the global system state will have already changed when P1 resumes its execution.

Such global change of the system state should not influence the remaining execution of P1. This process should be able to continue its activities until it reaches the next functional delay or receive-synchronization point. If this interleaved execution of P1 and P2 happens to affect the remaining execution of P1, there is certainly a missing part of system synchronization somewhere between P1 and P2.

The adverse consequence of such incomplete modeling in the system synchronization is the dreadful inconsistent simulation result. This is the

reason why the computational model of the untimed TLM obliges explicit modeling for every single system synchronization point in a given system.

Modeling engineers must insert functional delays into untimed models in such a way that the system synchronization can still manage to capture all the causal dependencies in a given system. This is a good modeling practice to assure the system stability, despite the variations of the clock frequency and the indeterminism of the micro-architecture (e.g. transaction latency on a bus depends on the bus load) in the sub-systems of a SoC.

3.5 Recommendation for Modeling Practices

Collected hereafter are our general recommendations for the untimed TLM modeling practices based on our experience in TLM development. Advices on implementation concerns are provided in Chapter 3.

1. Consider the intended uses of IPs on the final platform to efficiently determine how the corresponding TLM models should be written up.
2. To increase reusability, organize models in such a way that the algorithm can easily be updated, and reuse readily available standalone C models as much as possible. For the reason of code portability and management, these C models should never be replicated as "hardwired" copies in the TLM environment. Rather, they should be reused by means of wrappers or external function calls.
3. Determine the data granularity of models according to the algorithmic accuracy and the expected precision in terms of transfers. For example, the model of a video IP expecting frame-level input should be modeled with data granularity at frame level but not pixel level, despite the actual capability of the interconnect in the silicon. However, if there is a mismatch between the data granularity of the algorithm and the data layout in the memory according to the memory map, it will be the job of the TLM wrapper to generate the correct addresses so that the data is stored and retrieved from the correct memory locations.
4. Model all sorts of communication interfaces at bit-accurate level, particularly for register modeling.
5. Model all sorts of behavior at bit-accurate level.
6. Focus modeling with respect to the functional specification only, i.e. including *no* micro-architectural and clock-based information, resources, or details.
7. Model explicitly the system synchronization that affects the IP behavior.
8. Employ events within a model whenever that is appropriate for modeling the inter-process synchronization.

9. Utilize specific synchronization means such as synchronization protocols to model the inter-module synchronization.
10. Avoid implementing the process-activation based on a regular basis; the process-activation based on system activity is compulsory.
11. Ban uses of global variables.
12. Adopt good software implementation style to facilitate code debugging and maintenance, e.g. add comments in codes.

4. TIMED TLM

4.1 Objective

As far as we have discussed for the untimed TLM, the system synchronization only defines a partial order of the overall SoC internal events. The identification of the full order of SoC events is hampered by an indeterminism because the untimed TLM does not capture micro-architectural details, i.e. the timing behavior of the implementation.

The timing behavior of a component specifies the delay between each activation and synchronization-suspension. If this timing behavior is incorporated into TLM, the resulted timed model will be able to determine a full order of SoC events; hence leading to a complete specification of the implementation.

The main objectives for developing the timed TLM are:

- benchmarking of the performance of a given micro-architecture;
- fine tuning the micro-architecture;
- optimizing the software for a given micro-architecture to meet real time constraints.

Other objectives for implementing timed TLM models include:

- flexible modeling and refinement of timing accuracy according to customized user needs;
- reuse of untimed models to reduce time-to-market of SoC products;
- ability to plug different timing models into the same untimed model;
- dynamic switch to turn timing on/off in a given model;
- legacy management of reusing cycle-accurate models;
- independent, concurrent yet integrated developments between untimed-oriented verification team and timed-oriented architecture team.

4.2 Modeling Approach

To develop a timed model at the transactional level, considerations must be given to the time consumption of two aspects: computation and communication.

The computational delay is the time amount required to perform specific calculations in characterizing a given system behavior or function; whereas the communication delay is the total time consumed in accessing and transferring data or information. The various physical constraints that could bring a significant impact on the system timing behavior such as bus size, bus throughput, or memory size, must also be taken into account during the timed TLM development.

We model the time consumption of a given component in timed TLM through two different tactics:

1. Annotated model
2. Standalone timed model.

4.2.1 Annotated Model

The annotated model is a modeling approach where timing delays are annotated, i.e. inserted, into an untimed model. These annotated delays are the timing information of the *micro-architecture* level, which make the annotated model distinct from the untimed TLM model inserted with functional delay at *architecture* level (as described in Section 3.4).

Here, the delay of each possible set of activation-synchronization in a process is defined based on the control flow of the concerning component. This delay can be modeled with the values of the best, mean, or worst cases. A process could sometimes include very complex control graphs that will consequently entail a large set of timing attributes. If the modeling task becomes too large to handle, a “lazy” approach could probably be adopted by providing only the default conservative values for the unresolved activation-synchronization path. These conservative values constitute the minimum acceptable set of timing constraints that an implementation must comply to.

In general, the annotation approach is well suited if the structure of the untimed model already matches the structure of a micro-architectural model, where annotations will be simple wait statements related to the computation time of a specific functionality. We try to reuse untimed TLM models without any alterations through this approach, although some adaptations could be necessary in certain cases. It is essential to protect the timing

annotations with preprocessing directives (e.g. `#ifdef ANNOTATED_MODEL`) in order to select the appropriate execution mode (untimed or annotated) according to user needs.

4.2.2 Standalone Timed Model

The standalone timed model is a different approach where the actual timing behavior is modeled in such a way that delays are computed during the execution of a *standalone* timing model. Our development results have shown that this is applicable on hardware IPs and processor models.

A standalone timed model denotes a detached model incorporated with the timing information. This model is suitable when the structure of the algorithm is very different from the structure of the micro-architecture. Indeed, annotations cannot lead to an accurate timing in such cases. Consider the example of modeling a video application. If modeled at the frame level, only those delays associated with decoding a frame can be annotated. The micro-architecture of the application, on the other hand, allows both the communication and computation to be interleaved.

Conceptually speaking, standalone timed models are high-level analytical timing models *without* functional information. They can be built as traffic generators, which model the channel or interconnect traffic with some timing information.

If the timing behavior of a component depends on its functional behavior, the corresponding standalone timed model can be controlled externally, for instance, by an untimed TLM model. In that case, all the functional events occur during the functional execution of the untimed TLM model must be traced and provided to the standalone timed model. A timing control unit is used to manipulate this information between the untimed TLM and the standalone timed model.

Figure 2-6 gives a better idea about the concept and structure of a standalone timed TLM model combined with an untimed TLM model.

There are two general guidelines to realize the mixed model described above for a given IP. First, develop a purely untimed TLM model describing the functional behavior of the IP regardless of its timing characteristics. Second, develop a timed module in charge of all timing and micro-architecture related information of the IP, without duplicating the functional codes already done in the untimed model. The overall mission of the mixed model is characterized hereafter.

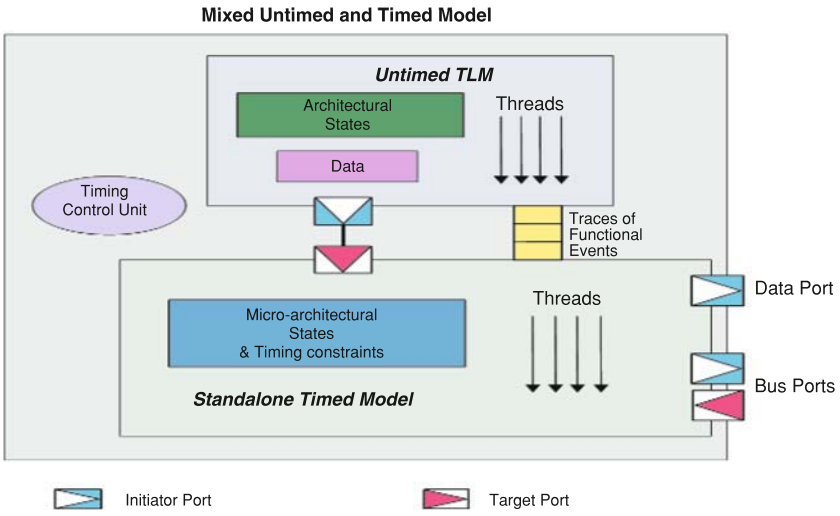

Figure 2-6. Combination of Untimed TLM and Standalone Timed Models

Untimed TLM Model

a) The untimed TLM model executes the pure untimed behavior that will consequently generate or receive transactions through its communication ports. This model must be instrumented for generating traces of functional events, which will trigger certain activities in the timed model.

Standalone Timed Model

a) The standalone timed model implements the mechanism to represent the timing behavior. If the design schedule is too tight to allow developing a very detailed and accurate model, the standalone timed model can be modeled with coarse grains. For a precise implementation, it can be modeled at the micro-architectural level with approximate cycle-accuracy. Standalone timed models are normally controlled by using functional traces generated in the untimed TLM model.

b) The standalone timed model declares communication ports to capture transactions from the untimed model and to insert time delays according to the traces of functional events. Transactions are exchanged through both untimed and timed ports of the timed model. Untimed ports are connected to an untimed communication channel/interconnect while timed ports are connected to a timed channel/interconnect. Details on the model of computation and rules to issue transactions on untimed/timed interconnects are provided in Section 4.3.

The mixed model offers numerous advantages as follows:

- Concurrent development of functional and timing models facilitated by the clear distinction between their modeling strategies.
- Multiple timing scenarios ranging from high-level to very accurate low timing level can be defined, and they can coexist for a unique functional model.
- Untimed models are reusable as the golden reference for functional verification without modifications.
- Optimized speed granted by the dynamic switching between untimed and timed models at the simulation run time.
- Mixed simulations involving timed and RTL models are feasible.
- Architecture and micro-architecture teams can work concurrently on different but complementary models

4.3 Model of Computation

4.3.1 Inter-Execution of Untimed and Timed Models

The working concept of the timed TLM can be pictured as an inter-execution of untimed TLM and standalone timed TLM models. Figure 2-7 illustrates the simulation timelines representing the activities of a process execution in the timed TLM.

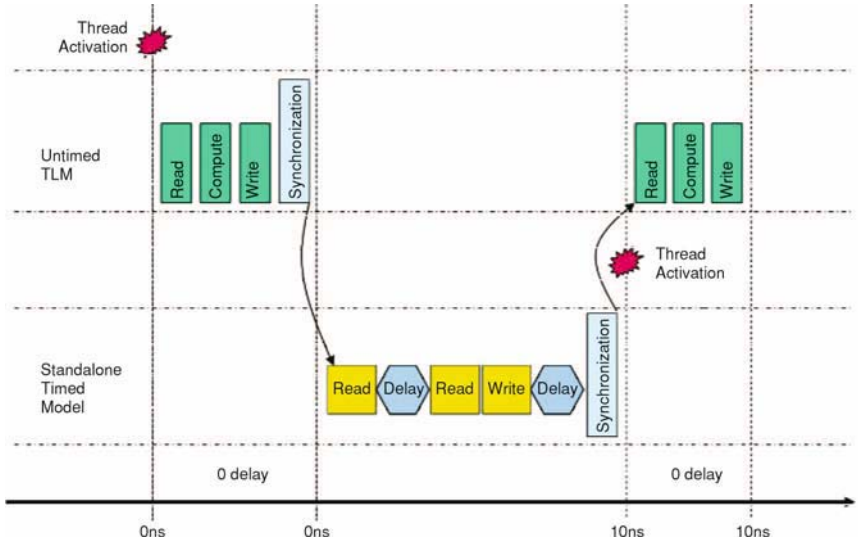

Figure 2-7 Inter-Execution of Untimed and Timed Models

Note that the functional behavior of the untimed model is executed until it reaches a synchronization point. The execution is then passed to the standalone timed model. The timing model will start simulating the delays associated to the functional parts that have just been executed earlier. Meanwhile, time delays of communications and computations are simulated in the timing model as well. Once all of the relevant delays are simulated, the untimed model will resume its execution until the end of its simulation.

The "inter-execution" of untimed and timed models is permissible as long as the untimed model is *fully* modeled using explicit system synchronizations. In this condition, read/write operations are generated only when the data is ready within a stable system. Let us zoom in on the details of such inter-executing mechanism by considering the platform depicted in Figure 2-8. The initiator IP is the master while the target IP is the slave.

The untimed platform is composed of:

- the untimed model of the initiator (I);
- the untimed model of the target (S);
- an untimed communication channel (C).

The bindings for the untimed platform are as follows:

- the initiator port of I is connected to the target port of C;
- the initiator port of C is connected to the target port of S.

In addition, the following modules are instantiated in the platform to support the "inter-execution":

- the standalone timed model of the initiator (TI);
- the standalone timed model of the target (TS);
- a timed communication channel (TC)[1].

[1] Timed channel can be hierarchical to represent the internal topology of the interconnect.

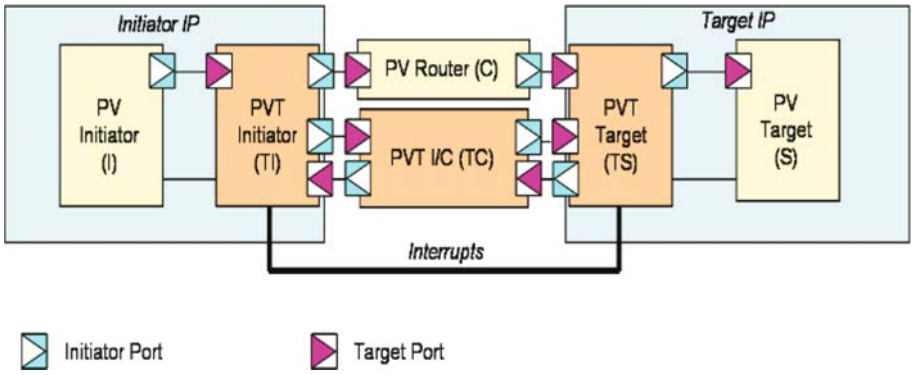

Figure 2-8. Mechanical Structure of Inter-Execution

The bindings related to the inter-execution are as follows:

- the initiator port of I is connected to the untimed target port of TI;
- the untimed initiator port of TI is connected to the target port of C;
- the timed initiator port of TI is connected to the target port of TC;
- the initiator port of TC is connected to the timed target port of TS;
- the initiator port of C is connected to the untimed target port of TS;
- the untimed initiator port of TS is connected to the target port of S.

All sorts of transactional accesses are set off from the initiator to the target through the initiator port; and the functional information is passed to the standalone timed model through the appropriate data structure.

Referring to Figure 2-8, *TI* traps transactions issued by *I*. When *I* meets a synchronization point, the standalone timed model *TI* will start its execution. It computes all of the necessary delays as modeled in the timing model of the micro-architecture, and it issues transactions. As *I* may have generated transactions at a high level of abstraction (e.g. frame), *TI* will generate the appropriate number of transactions from the micro-architectural point of view (e.g. pixel). *TI* may also reorder the transactions to represent read and write interleaves in cases like pipeline.

The overall communication mechanism is as follows:

1. Transactions are issued by *TI* on *C*.
2. Transactions are received by *TS* from *C*. A careful analysis is diagnosed on the transactional access to identify its nature. Depending on the nature of the access, *TS* will handle the transaction accordingly. There are two kinds of accesses:
   a) *insensitive access* - no impact on the IP synchronization scheme.
   b) *sensitive access* - leave impact on the IP synchronization scheme.

For an insensitive access, the simulation continues directly in *TS* for any potential computational time delays associated with the transaction. Indeed,

the TLM transaction is propagated "in advance" compared to the actual event occurrence in the silicon. Such advance is permissible on condition that the synchronization scheme can prevent the system consistency from being corrupted by the access. For analysis purposes, the corresponding communication delay from the initiator to the target is passed through the timed channel *TC*, although they will be ignored by the target for the simulation.

For a sensitive access, on the other hand, the transaction emitted from *TI* is rejected by *TS*. Early accesses are not granted in this case because certain behavior could be triggered earlier than what it should be. The adverse consequence will be the undesirable system inconsistency. To prevent this from occurring, *TI* must re-generate the transaction by transferring it through the timed channel, *TC*, in order to include the related communication time delay. The transaction will now be received and accepted by *TS* with the correct time granularity at the right timing. Then, the access will be re-generated by TI on C to actually read/write the data.

Any computational time delay closely related to the initiator or target IP is managed locally by the timed models of the respective IP. Asynchronous events such as interrupts are handled at every single activation boundary. Fine-tuned behavior can be obtained in using pseudo synchronization points as described in Section 4.3.3.

4.3.2 Discussion on Standalone Timed Model Techniques

The standalone timed model is a technique implying a strict compliance with the modeling rules discussed earlier to ensure *no* micro-architectural timing information is implemented in the untimed model. The key advantage is the very neat separation of functional untimed models from micro-architectural timing representations. Thus, it is straightforward to develop several standalone timed models for a given functional model, which allow investigating several micro-architecture scenarios.

With such techniques, the sequence of communication and computation delays may not correspond to the associated functional sequence (while they usually do). For example, an untimed model may grab a full image to process it in one-shot while a timed model would process the data accesses and computations as interleaves. In addition, communication and computation delays can be interleaved in various manners, which could probably be different from its sequence of functional behavior too, e.g. pipeline characteristics. Compared to the functional model, validating the standalone timed model should be handled more carefully to ensure that no error is inserted.

Since the functional and timing information are clearly separated between untimed and standalone timed models, it is possible to couple untimed models with traffic generators. Traffic generators connected to the timed interconnect can act as standalone timed models. The untimed model drives the traffic generator, which is not aware of the functionality but able to generate meaningful bus sequences on the interconnect. This method is particularly useful when traffic generators are developed before transactional models, with the intention of reusing both of them in the future.

4.3.3 Pseudo Synchronization Points

Based on the principles of the system synchronization described so far for the untimed TLM, asynchronous events such as interrupts are perceived only at the activation "boundaries" of the untimed TLM. This is due to the synchronization mechanism coupled to a non-preemptive simulation kernel.

As a process will suspend only on explicit synchronization points, no other processes can execute in the background. While this is not an issue for purely untimed models, it becomes a concern when mixing untimed and standalone timed models. Indeed, asynchronous events may occur too late during the suspended phase of a thread under certain circumstances. Consequently, they may not be caught at the appropriate time.

To handle this problem, finer-grain *pseudo* synchronization points are defined in untimed TLM models. These false synchronization points behave as if many pre-emption points appear more frequently to check for asynchronous event occurrences. They enable timed TLM threads to manage incoming asynchronous events such as those for memory accesses in between synchronization points.

4.3.4 Absolute Micro-Architectural Features

Most of the features for a given system can be modeled as a pure functional model, and can be further refined as a timing model. Certain features, however, are not represented in a pure functional model because they are not relevant at that level of abstraction.

Modeling engineers should be aware of some complex micro-architecture blocks that might be added at the micro-architectural level to optimize the (timing) performance. While such blocks have no relevance to the functional level, it becomes compulsory to model them in a standalone timed model. The reasons are that these blocks definitely related to the micro-architectural information of the system, and they have known impact on the system performance.

TLM can manage such features by integrating the micro-architectural information as well as the related behavior into the timed module. An excellent example to illustrate this idea is the modeling of memory cache.

By definition, a cache is an implementation to improve the performance of the real system. It is not required to be included in the simulation to verify the functional correctness of the design. For this reason, a cache should not be conceived as an architectural model. What we wish to observe in the simulation is the actual traffic of cache activities on the channel for collecting its actual timing figures.

Therefore, the cache needs to be modeled accurately for its traffic and timing changes in the timed simulation as the micro-architectural model. The timed model of the memory cache includes not only timing information, but also some code pieces that reflect the cache effect on the data amount generated onto the channel.

The same approach applies to the reuse of an instruction-accurate ISS in a timed platform. The modeling of the pipeline and cache features as micro-architectural timed models is compulsory to obtain accurate timing figures.

5. ADVANTAGES OF TLM

Amongst the abundant endeavors proposing modeling techniques at higher abstraction level, TLM has managed to sail its way through to offer a promising solution to SoC industry. As a reliable methodology that can rapidly improve the design productivity, TLM confronts the SoC design bottlenecks in complexity and time pressure through three axes:

1. Early software development.
2. Architecture analysis.
3. Functional verification.

- **Early Software Development**

Software development activities, especially debugging and validation, will have effect only if the software could be executed on its target platform.

Conventionally, a physical prototype such as emulator or FPGA board prototype is considered as the starting point of software development. The downside of this approach is obviously the late availability of such starting point too close to the end of the hardware development. Not only is the time a hindrance, any hardware issues revealed by the software execution at this stage will be too costly to fix as well.

The hardware/software co-verification could of course start executing the software earlier on the target hardware platform. But then again, it still needs to wait quite long for RTL hardware models before running anything.

Rather different from the two approaches mentioned above, TLM SoC platform can be developed right after the delivery of system specifications. The target platform is therefore available for the software development much earlier in the SoC design cycle. In other words, the software development is now conducted in parallel with the lengthy hardware development, i.e. a veritable concurrent hardware/software design is attained.

With the "contract" of TLM platform signed between them, both software and hardware teams cooperate in an independent but converging manner. Software developers regard TLM platform as the reference to run their codes while hardware designers consider it as the golden reference for their RTL design.

In general, software developers employ TLM platform for two kinds of software development:

a) functional software development using untimed TLM;
b) optimized software development using timed TLM.

The greatest advantage of having early software development based on TLM platform is the reduced time-to-market of SoC products through concurrent hardware/software design.

- **Architecture Analysis**

To increase the chances of first-time silicon success, a system must be thoroughly controlled at each step of the design flow against the real-time constraints stated in the initial system definition. An architecture exploration allowing system performance analysis and verification will fulfill this requirement. The timing information is often essential in such analysis.

System architects and RTL designers seek constantly a better solution for the architecture exploration at an earlier SoC design phase. For this, TLM offers a favorable approach by providing the possibility to explore a system architecture shortly after the system specification is completed. Depending on the user needs, either the untimed TLM inserted with functional delay or the timed TLM can be used for this purpose.

Through an earlier architecture analysis, any system optimization or modification could be handled in time- and cost-efficient way. Besides, it helps to improve the design consistency between hardware and software teams since they are both founded on the same TLM architectural model.

- **Functional Verification**

Functional verification is intended for assuring the compliance of a given component or system implementation with its corresponding functional specification. RTL models of the design-under-test are analyzed in a functional verification environment by various test scenarios. These test scenarios are developed by verification engineers referring to the paper

specification. Most of the time, the engineers need to "manually" determine the expected results of each scenario.

In fact, TLM is the actual functional specification of a component or system. More precisely, TLM is the executable specification of a given design that captures the intended behavior perceived by end-users, i.e. architectural view; but not the implementation details of micro-architectural view. Thus, TLM can replace the manual process undertaken by verification engineers to generate the expected results of test scenarios as the golden reference for functional verification.

Not only is TLM platform used for developing the reference output of test scenarios, it is also reused to conduct functional verification of RTL models with the same test scenarios. The outcomes of the RTL functional verification will be compared to the reference output generated by TLM for analyzing and verifying the design behavior.

As a result, TLM can really save the verification team a huge amount of working time. In addition, it aligns their job constancy with those of software and hardware design teams through referring to the same TLM platform.

6. CONCLUSION

Concisely, TLM plays the role as the *unique* reference for different teams all the way through the SoC design cycle. Such idea of centralized reference is depicted in Figure 2-9.

Not only is TLM a reliable methodology to face SoC design bottlenecks, it is essentially the single reference that puts into effect a "contract" among the different teams to achieve three durable objectives:

- Work consistency across various teams.
- Rationalization of modeling efforts.
- Cross-team communication and interaction.

In conclusion, the ultimate goal of TLM is leading the SoC industry to a cost- and time-efficient SoC project management in the long run.

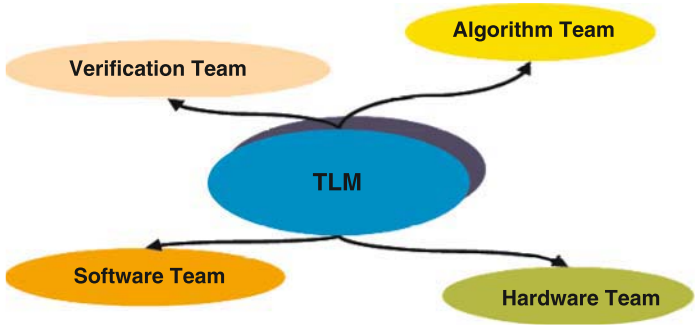

Figure 2-9. TLM as Unique Reference Model

REFERENCES

[1] A. Haverinen, M. Leclercq, N. Weyrich, and D. Wingard, "SystemC-based SoC Communication Modeling for the OCP Protocol," [Online document] 2002 Oct 14 (V 1.0), [cited 2004 Nov 5], Available at HTTP: http://www.ocpip.org/socket/whitepapers/

[2] J. Gerlach and W. Rosenstiel, "System Level Design Using the SystemC Modeling Platform," in Proc. of the Forum on Specification & Design Languages (FDL'00), 2000.

[3] L. Semeria and A. Ghosh, "Methodology for Hardware/Software Co-verification in C/C++," in Proc. of the High-Level Design Validation and Test Workshop (HLDVT'99), 1999.

[4] A. Fin, F. Fummi, M. Martignano, and M. Signoretto, "SystemC: A Homogenous Environment to Test Embedded Systems," in Proc. of the IEEE International Symposium on Hardware/Software Co-design (CODES'01), 2001.

http://www.springer.com/978-0-387-26232-1

Transaction-Level Modeling with SystemC
TLM Concepts and Applications for Embedded Systems
Ghenassia, F. (Ed.)
2005, XVI, 272 p., Hardcover
ISBN: 978-0-387-26232-1