FP6-2004-IST-4

Open SoC Design Platform for Reuse and Integration of IPs

TLM 2.0 Draft2 Feedback document

AUTHORS

Author	Company
Guillaume AUDEON	ARM
Jérôme CORNET/Laurent MAILLET-CONTOZ	STMicroelectronics
Victor REYES	NXP Semiconductors
Bart VANTHOURNOUT	Coware
Thomas WILDE	Infineon

DISTRIBUTION LIST

Date	Who
30-Jan-2008	SPRINT consortium
31-Jan-2008	OSCI TLM WG

Table of Contents

1 Introduction

1.1 Purpose

The goal of this document is to formulate the feedback from SPRINT partners on TLM 2.0 Draft 2 Standard. This feedback results from consensus on several issues raised by the members on the TLM 2.0 Draft2 Standard.

1.2 Scope

This document is intended to be provided as consolidated feedback from SPRINT partners to the OSCI TLM Working Group.

1.3 Structure

The document structure follows the structure of the TLM2.0 Draft2 User's Guide. Each section addresses one specific topic covered by TLM2 Draft2. It covers general feedback, problem statement if any, and when appropriate feedback on documentation

1.4 SPRINT overview

The global objectives of the EU funded SPRINT project are to *e*nable Europe to be the leader in design productivity and quality in Systems-on-Chip (SoC) design, by mastering the SoC design complexity with effective standards and design technology for reuse and integration of IP

Therefore, the approach is to develop a standards-based open design platform that supports the development of interoperable and reusable IPs and their efficient integration into high quality SoCs. More information can be found at http://www.sprint-project.net/

SPRINT partners are NXP, ST, Infineon, CoWare, ARM, Evatronix, Syosil, ECSI, C-Lab, KTH, Lauterbach, KeesDA, Magillem Design Services, and TIMA.

2 Standard Introduction

2.1.1 General feedback

The SPRINT partners notice that the draft 2 of the TLM standard introduces a new terminology for classifying modelling styles (UT, LT and AT) with respect to previous versions, which were using PV and PVT. The relation between these various terms should be clarified. Maybe a notion of "profile" should be identified, to classify the various modelling practices of the users. The partners remark at the same time that the standard seems to take very strong assumptions about these modelling practices and even pushes for particular choices whereas it is not its role.

Additionally, the arguments given in this section for choosing between blocking or non-blocking transport interfaces are unbalanced. The introduction seems to restrict the use of blocking transport interface for UT, whereas it can be used as well for LT. Actually, modelling using a one-phase protocol is a perfectly valid abstraction choice that covers both UT and LT. Also, the SPRINT partners believe that UT use case is somehow theoretical, as in most platforms there are components that are intrinsically "timed" (for instance timers). The benefits and drawbacks of each interface are not clearly explained, with respect to the level of details required in the models. For instance, blocking transport means simplified modelling but cannot be used for protocols that require multiple phases to be taken into account, whereas non-blocking transport can, but to the detriment of modelling effort and possibly

simulation speed. The documentation should be rephrased to better explain the various use cases and the related communication interfaces.

3 Core TLM2 interfaces

Overall, the SPRINT partners think there is an unbalance between the two communication interfaces that are proposed (blocking and non-blocking transport). Blocking transport is too restricted whereas non-blocking transport is overly complex. The rationale behind these decisions is not clear, as the relation with the use cases, that need somehow to be supported regardless of the interface chosen. For instance, a user choosing the blocking transport interface cannot add timings efficiently afterward, since the standard does not make any provision for transporting timings annotations with this interface, whereas non-blocking transport can. Conversely, a user wanting to transport timings annotations for a one-phase protocol in a LT model is forced to deal with the complexity of non-blocking transport and multiple phases management (the initiator has to support incoming nb_transport() calls whereas it should not). In the view of SPRINT, both interfaces should converge. This would extend the reach of blocking transport to concrete cases (such as LT) and ensure better interoperability between blocking and non-blocking interfaces.

3.1 Blocking transport interface

3.1.1 Problem statement

- This interface does not standardize how to transport timing annotations, in the goal of adding efficiently timings to a UT platform or supporting intrinsically timed components (timers, etc.) for UT/LT modelling. The standard should provide a way to transport timings with this interface.

Additionally, the SPRINT partners think there is a problem with the lack of proper synchronization guarantees in the TLM Standard, when interconnecting IPs using the blocking transport interface. Indeed, without any yielding instructions (wait() calls) executed on the transaction path, an initiator could monopolize the simulator with the rest of the simulation being frozen. The explicit synchronization points should be identified, that is, whether a transaction at a given address or in a given register will trigger a side-effect, and then will require a yielding instruction to be called.

As an example, consider a DMA with several registers: “parameter” registers that will hold the characteristics of the transfer to be done, and a “command” register for triggering the transfer itself. Writing in the command register do is an explicit synchronization point, which will notify some event to a DMA process responsible for initiating the transfer’s transactions. For the transfer’s process to be given a chance to execute, the initiator programming the register should execute a yielding instruction on its transaction’s path.

To achieve this goal, several solutions can be used and possibly combined:

- The standard could mandate the execution of a yielding instruction in the target when a transaction corresponding to an explicit synchronization point is received (this would be the responsibility of the target to decide depending on the transaction address and command),
- The target could return a “synchronization flag” added to the transport interface, which would be set when an explicit synchronization point is reached. This latter solution would allow choosing the location for the yielding instruction (in the target, in the bus model or even in the initiator).

3.1.2 Documentation feedback

The explanations in the user manual for this interface are too limited when compared to the non-blocking interface, which is covered extensively.

3.2 Non blocking transport interface

3.2.1 General feedback

- A user that chooses this interface for one-phase models should not bear the cost of implementing the various possible behaviours related to the other phases. For this reason, there should be a default implementation of the nb_transport backward path, so that the initiators could ignore the backward path safely in this situation.

3.3 Direct memory interface

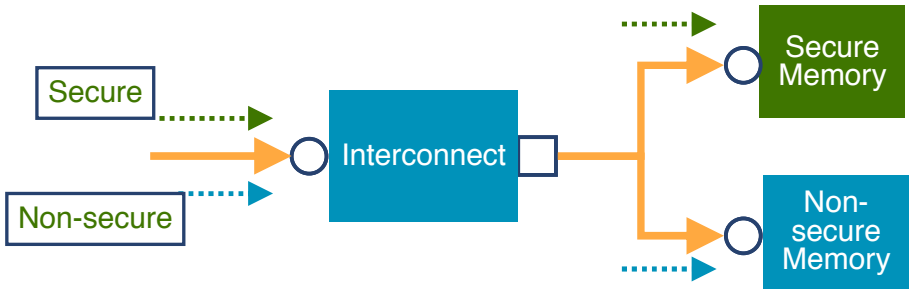
3.3.1 General feedback

The DMI has similar problems like the debug interface due to the fact that only raw memory can be accessed and the access does not consider specific features of the used protocol type.
In case multiple initiators have DMI access it is not clear how to prevent incoherencies.

3.3.2 Problem statement

- The access of raw host memory leaves tasks like type interpretation, range checking and endianness handling with the model writer. To be able to meaningfully access and interpret the granted memory region the initiator of a DMI access needs detailed knowledge of the target implementation (where is the physical storage location for a particular logical address?).
  To give some hint about how to interpret the memory content the **socket BUSWIDTH of the target** owning the memory region should be provided by the interface. At least for data that has been written to a word aligned address it is then possible for the initiator to reverse endianness manipulations. (Nevertheless this approach will fail for targets with multiple sockets of different size and data written using fancy unaligned streaming accesses in combination with byte enable ….)
  In addition also the **simulated endianness of the target** owning the memory region must be known to correctly interpret the contained data.
- When issuing a `get_direct_mem_ptr()`, address may not be the only generic payload attribute to be taken into account. Additional attributes, such as burst length, or protocol specific eventually, part of a mandatory extension, may be required.

  The AMBA AXI protocol provides the capability for a transaction to be routed differently according to its associated protection type. For instance, the interconnect will route a secure transaction to a secure memory; whereas, it will route a non-secure transaction to a non-secure memory. However the transaction address remains the same.



Therefore, in order to support DMI in such interconnect model, the information about the protection type of the transaction should be accessible, in order to route the DMI request to the corresponding memory model. The returned host pointer shall point to the corresponding host memory: secure or non-secure according to the protection type.

According to the standard recommendations, a specific protocol shall extend the generic payload by using the extension mechanism. Therefore, such AMBA AXI protection feature will be modelled as part of an AMBA AXI specific extension.

However, with the current DMI interface definition, i.e. `get_direct_mem_ptr()`, the only mean to carry the AMBA AXI specific extension, and thus to access the information required to implement DMI correctly, is via a new *dmi_mode* derived from `tlm_dmi_mode`. This is unfortunately not in favour of maximizing interoperability, as the DMI interface will then be protocol specific.

The standard should provide a mean to carry additional information, protocol specific eventually, in an interoperable manner, along side the address, in the DMI interface.

- In a multi-core system a memory region may be used with DMI by multiple initiators. At the moment the standard does not provide rules/mechanisms like write snooping to keep them updated. Either a rule should disallow DMI accesses to the same memory region by more than one initiator at a time or a notification mechanism is needed to trigger re-read of an updated region.

3.3.3 Documentation feedback

3.4 Debug transaction interface

3.4.1 General feedback

The SPRINT partners appreciate OSCIs attempt to standardize a way to perform a non-intrusive and side effect free debug accesses. Unfortunately the debug interface offers only very basic functionality and leaves a lot of responsibility with the creator of a TLM model.

3.4.2 Problem statement

- The use model covered by the proposed approach is more or less restricted to hardware debugger support. It is only possible to access the raw memory location of peripheral resources that are mapped to addresses accessible through a bus, i.e. registers and memory blocks. For that purpose a distinct API method and payload type has to be used. Convenience and safety features already available with the GP are not used and the model designer is instead forced to implement a similar, parallel behaviour for the debug payload with the added burden to be responsible for the interpretation of the raw array of bytes, endianness handling etc. Complex data types cannot be debugged at all as their byte representation is compiler and/or platform specific.
- In our opinion it would make more sense to **use the same payload type** used for regular transaction also for debug accesses. This could be achieved either by
  1. passing a regular payload object (reference) as argument to dbg_transport or
  2. by using the normal xxx_transport API along with a command type indicating the debug access, e.g. DEBUG_READ/ DEBUG_WRITE

  This needs to be combined with a set of rules (which attributes of the payload objects are to be ignored, phase and timing behaviour.)

- Re-using the payload type used for regular transactions had the added benefit that **extensions** can be considered for protocol specific debugging. Assume a protocol that features a protection mode that influences which physical resource is represented by a particular address: with active protection mode it's memory region A and region B otherwise. A debug access needs to consider that kind of mode as well - which is not possible with a plain vanilla debug payload object. The same example that is given for DMI (Modelling Amba AXI protocol with the standard) is also relevant here.
- Another shortcoming of the debug API is its focus on memory mapped resources only. What does not have an address or what for some reason is not accessible through the bus is not visible. SoC level debugging makes it desirable to have a **standardised way for more abstract functional debugging** (above the pure HW debugger view). Besides register- and memory-content it should be possible to observe also **modelling artefacts** of arbitrary type that do not have a hardware representation and exist just in the implementation.
  A simple example is an expression that arithmetically combines the content of some registers and is just used internally in the model. To observe this expression (**that may functionally make more sense** than the individual register values) it is now necessary to duplicate the expression in the initiator/debugger, which may diverge from the original after modifications. It would be safer and easier to directly observe the expression.
- If this requirement goes beyond the scope of a debug API then an additional monitoring API is required. This is especially important considering that TLM2 is expected to be adopted by SoC companies as well as by EDA tool vendors. So a standardized **abstract** debug/monitoring API (beyond HW debugging capabilities) would introduce a lot of interoperability potential between TLM2 compliant IP and TLM2 compliant tools.
- The debug transaction interface is non-blocking by definition and is expected to return immediately. This will prevent its usage in a co-simulation context where e.g. an RTL block is interfaced with a transactor. The RTL block will most likely not be able to deliver the result in zero time. Even if the debug call spans a certain time interval it can be still side effect free. Is the non-blocking requirement really necessary?

4 Sockets and combined interfaces

4.1 General feedback

Sockets and combined interfaces are a good addition to simplify the binding of forward and backward paths of an interconnect. A number of things should be considered:

- In order to enable compile time errors when binding sockets of different size, and to enable endianness calculations it is important that buswidth is a template for the socket or the combined interface.
- As sockets are a new way of binding modules together there should be ways to use them with regular ports and exports.
- Since a module can have multiple sockets with the same set of interfaces it is important to provide with a means to determine the origin of an incoming interface call.

4.2 Problem statement

- When developing complex blocks it is useful to be able to decompose the code into hierarchical elements each responsible for part of the function of the block. In order to support such a coding style and to integrate sockets with the other SystemC constructs it might be useful to

provide with a tlm_port and tlm_export, each with a buswidth template to enable separate binding of the forward and backwards paths within a socket.

- There is no standard mean to discriminate/distinguish between several incoming paths in a module (both for forward and backwards path).
    - For instance, with a model of a dual port memory, or an interconnect arbiter, one needs to discriminate between incoming ports in order to model, e.g. the arbitration algorithm.
    - Similarly in case the target wants to send a delayed response on the BW path there is no standardized way to identify the receiving socket and route the BW transaction accordingly
    - This is also one of the TLM requirements, *13 Multi-port target support (2.0)*.
        - There was the tlm_export_id attribute in TLM 2.0 draft 1.
        - There is the tagged sockets and APIs in Doulos tutorial on “Multiple Initiators and Multiple Targets” as a possible way of dealing with this requirement, which as the drawback of implying to maintain a map of transaction, tag association pairs.
        - There are the derived sockets with callbacks in TLM 2.0 draft 2 examples/tlm/common/include/SimpleSocket/, which has the drawback of introducing an additional indirection and thus a possible negative impact on the simulation speed, maybe a better implementation should be investigated.

      One of these options (or another) should be standardized, i.e. defined under tlm_h/, in order to provide maximum interoperability.
- The SPRINT members point out that with the current definition and implementation of sockets, it is not possible to create sockets that accept multiple connections. The standard does not currently provision the template parameter indicating the maximum number of connections, which is then "one" by default when inheriting from sc_port/sc_export. However, the use cases for multiple connections do exists. As an example, it is desirable to simplify the netlisting of platforms at high level of abstraction by using a bus model that contains only one initiator socket and one target socket, with all initiators connected to the single target socket and all targets connected to the initiator socket. This aspect is independent of the actual implementation of the bus model.

5 Analysis interface and analysis ports

5.1 General feedback

The SPRINT partners consider the proposed analysis concept of TLM2D2 not to be mature enough.

The use model should be better explained, use cases should to be presented and capabilities need to be extended to allow a convenient, flexible and robust use.

5.2 Problem statement

- A single analysis port can more or less just take transaction snapshots, potentially annotated with duration information. The timing granularity introduced with the flexible phase/refinement concept of nb_transport cannot be represented by this approach appropriately or would at least require multiple port instances.
- It should be possible to check - in addition to pure transaction data - also other functional **system** details (not present in the transaction object). For this to work at least one start-of-transaction and

one end-of-transaction callback to the subscriber/observer is expected. When the callback is made the observer can sample additional system state data at this point in time and combine this information with transaction data to complex property expressions. Thus transaction data details as well as internal system states and timing constraints can be checked. The increased flexibility and additional analysis capabilities would justify an extension of the analysis interface with a start and end callback. **The analysis triple is not sufficient here** and having multiple analysis port to achieve this adds unacceptable complexity.

- The suggested usage of the analysis port leaves it to the user if and when to trigger observer notifications. An additional specialized "T-branch initiator port" could be added to encapsulate this. Each transaction going over that port also triggers an observer notification or -in other words - automatically branches off an additional analysis interface call. This would allow for convenient bus traffic monitoring. The new port could add convenience interfaces to control analysis notification behaviour (notify on/off, masked notification, …). A general availability of such a standardized port in the TLM2 kit would add out-of-the-box analysis capabilities to a model without the need to touch the model.
- Current analysis' paradigm is "push-mode". Basically the component owning the analysis port decides when to notify its observers. It should also be possible to "pull" or monitor data from the outside world without model modifications. In that case the environment can (dynamically) decide when and which data to observe. Otherwise this information needs to be hard-coded (statically) in the observed model. (This requirement may overlap with those for an extended debug/monitoring interface mentioned in another section of this document.)
- Again the interoperability potential should be mentioned if TLM2 compliant EDA tools and IP easily match. It is very likely that the analysis capabilities of a decent EDA tool will go beyond just recording transactions. So a standardized capable analysis interface would ease linking IP and such tools.

6 Generic payload

The implementation of the generic payload is well-aligned with the requirements specification document in the following aspects:

- Minimum set of attributes existing on typical memory-map bus protocols
- Same structure for LT and AT modelling styles
- Non-template attributes
- Supports transaction of run-time selectable size (and burst)

However with the current proposal there is a very important requirement that it is not fulfil: **simplicity**.

6.1 Generic payload attributes

6.1.1 General feedback

With the current set of attributes and semantics, GP support the following functionality:

- Unaligned addresses
- Arbitrary transaction size (smaller, equal or bigger that BusWidth and non-multiple of BusWidht). This implies sub-word, word and burst support (being the word-size defined for the BusWidth parameter).
- Arbitrary byte enable configuration. This implies support for lacy bursts as well.
- Incremental and streaming addressing modes.

Although this indeed means a lot of flexibility, it has implications regarding the amount of cases that has to be considered when a user writes code for an initiator and target module. This is even worst when models have to be host-endianness independent. Moreover some of the rules regarding the attributes are not robust enough in all cases. These problems are stated in the next sub-section.

6.1.2 Problem statement

a) Data array

There are not clear rules on how the data array has to be interpreted in all the conditions described above. The lack of clear rules and an ambiguous understanding of the data array format will hamper interoperability, since users will not know how to put (or get) data in the array such as it will be understood for other modules in the system regardless their simulated endianness. Available helpers and serializers functions in the TLM kit are not implemented according to the rules described in the TLM user manual. Moreover with the current unclear scenario and rules the probability to do a out-of-range access to the data array is high. This is unacceptable from any point view.

Another unacceptable fact derived from the current proposal is that users can not create easily host-endianness independent code. That is, a Little- or Big-endian model of an initiator module will require two branches of code depending on whether it is being executed on a little-endian or a big-endian host machine. Moreover there is no possibility to encapsulate this in helper functions since depending on the host machine the size of the data array and the usage (or not) of byte enable array can vary.

With the current approach the rules to perform sub-word access are not clear, some scenarios will need byte-enables other will not. A more consistent way of dealing with sub-word accessed in the GP is required. In order to simplify sub-word accesses an extra attribute that indicates the size of the sub-word (data_width) can be helpful. With such an attribute normal sub-word accesses will not require byte enables, which may improves efficiency as well.

b) Byte enables

Implementing byte enables as a bool pointer has issues since different compilers have different implementation for the bool data type. Replacing the bool pointer for a char pointer would allow creating compiler independent optimized implementations of the functions that handle byte enables.

c) Streaming

The rules that accompany the streaming attribute and its relationship with the data array and endianness are not clear enough. Moreover the differences in terms of how to interpret the data when doing incremental or streaming transactions are not justified. When sending an incremental transaction the size of a data word adapts to the BusWidth of the port, however when doing a streaming transaction the size of a data word is dependent on the streaming_width attribute. This decision is not justify enough on the user manual and it implies again more different ways to look to the data array format.

d) Lock (atomic transactions)

Although atomic transactions and a lock attribute to indicate it can be an interest feature, there is not clarity on how this attribute has to be interpreted and used for the diverse components of a system. Moreover the lock attribute could lead to deadlock situations when the LT modelling style and/or temporal decoupling is used. Adding such attribute without a clear view on how should be used and its semantics can have a very negative impact and hamper interoperability.

6.2 Extension mechanism

6.2.1 Problem statement

When modelling a specific protocol, based on the generic payload, the extension mechanism shall be used in order to provide for attributes/features not provided with the generic payload, as per the standard recommendations.

Moreover, as it is a specific protocol, the extension(s) is(are) (static) mandatory, i.e. the specific protocol payload instance created by the initiator and passed to the target, via a possible interconnect, is assumed to hold the specific protocol extension(s) throughout its lifetime.

A specific template argument, `TYPES`, shall be defined and used for sockets, as per the standard recommendations.

The current extension mechanism has the following advantages:

- Converting to and from generic payload-specific protocol payload, is direct and costless; there is no need for a possibly costly cast

However this has the following drawbacks:

- There is no provision in the standard extension mechanism for refining behaviours, i.e. for overloading a generic payload method. One possible use case would be to override the copy constructor, copy assignment or clone methods, in order to implement a custom memory management for the specific protocol payload and/or extension(s).
- There is no provision in the standard extension mechanism for the possibility to add helper methods, especially in relation with the specific protocol extension(s). One possible use case would be to define a computed attribute for the specific protocol payload, which would be computed based on generic payload and specific extension attributes.
- There is no provision in the standard extension mechanism for ensuring a specific protocol and thus mandatory extension is present, apart from relying on the socket template parameter and the modelling engineer willingness to strictly follow the standard recommendations and usage intent. Therefore, interconnect and target models implementing such specific protocol will have to check for the presence of such extension(s).

The standard should thus provide with the ability to override methods of the generic payload, to define helper methods in relation with specific protocol extension(s), as well as to ensure such extension(s) is(are) really mandatory, i.e. present in the payload.

NOTE: An envisaged solution would be to allow for the use of inheritance, when modelling a specific protocol based on the generic payload. This would be specified in the second and third alternatives of section 9.2: “Extensions and interoperability” in the User’s Manual.