Chapter 16
OSCI TLM

In Chapter 2, we introduced the concept of transaction-level modeling-based (TLM) methodology. In the chapter, we talked about some of the reasons why one would use a TLM methodology, including design exploration, early hardware/software integration, and early verification development. Clearly, any one of the above is a good reason to develop a TLM model.

To implement a TLM methodology, there needs to be a precise definition of a TLM standard. This standard should be flexible enough to work at different levels of abstraction and be protocol-independent. Currently, there are various working groups defining a TLM specification. In this chapter, we will focus only on the Open SystemC Initiative (OSCI) TLM 1.0 standard, and we will provide a glimpse of objectives planned for the next release at the time of this writing.

16.1 Introduction

The OSCI TLM 1.0 specification was created in response to a need to standardize a TLM definition. The core component of the specification offers a set of TLM interfaces that define the APIs required for unidirectional blocking interface, unidirectional non-blocking interface, and bidirectional blocking interface. The specification also defines a set of channels that implement the above interfaces. While it is possible to define your own channels, the supplied example channels should be sufficient for the user to connect most system components.

In addition to a set of APIs, the OSCI TLM standard offers other benefits including abstraction, speed, and reuse. Using the TLM components, you can design a very high-level abstract model, refine portions of the design, or mix a variety of models at different abstraction levels. Of course, there are speed trade-offs to consider as you develop your model. Earlier in this book, we discussed the different abstraction levels including un-timed, approximate-timed, and cycle-timed. At the most abstract level, the un-timed models give the best performance in terms of speed. In many cases, this model can be refined further to an approximate-timed model to achieve better accuracy, though with some simulation performance

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_16, © Springer Science+Business Media, LLC 2010

degradation. Lastly, we need to mention the benefit of reuse in developing TLM models. As with any standardized API, one of the benefits is being able to reuse the components in different projects. In cases where there is a mixture of model abstractions, you can develop specific transactors or adaptors and still model the components together.

In the remainder of the chapter, we talk about the architecture of the TLM 1.0 standard, including the interface and channels, as well as offer an example of a TLM-based design.

16.2 Architecture

The TLM 1.0 specification provides a definition for the TLM architecture, including the transport layer, protocol layer, and user layer as shown below:

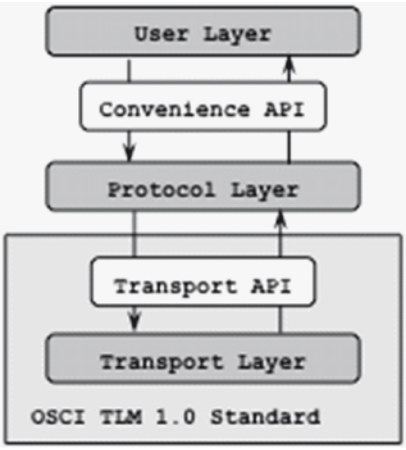

Fig. 16.1 TLM 1.0 Architecture

The transport layer is the heart of the TLM 1.0 specification. This layer focuses on the fundamental APIs that define the transfer of generic data, using either blocking or non-blocking low-level calls. The SystemC components here include the TLM interfaces that define the unidirectional `put()` and `get()` functions, as well as examples of the channels that implement them. As we mentioned previously, the user has the option to use the TLM channels supplied with the standard, or to create their own custom channels that implement the underlying TLM interfaces. Some of the components in the transport layer may include some generic system blocks such as routers and arbiters to assist in the transport of data.

The protocol layer consists of the classes that interface directly with the transport layer by calling the appropriate APIs. The OSCI standard provides simple initiator and target example classes that you may use, but in most cases, you will need to develop protocol-dependent code to interface with specific model components.

Lastly, the user layer contains convenience methods that let the user access data through simple function calls, such as reads, writes, block reads, and block writes. Since user class definitions inherit from the protocol class definitions, the user classes normally don't need to change when the underlying protocols change. The following diagram shows the interactions among the layers.

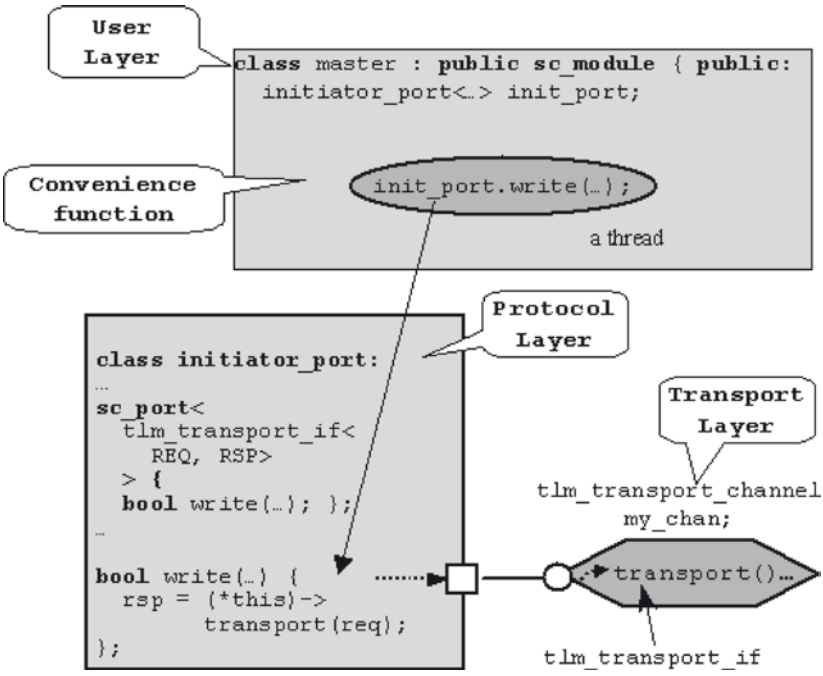

Fig. 16.2 OSCI TLM approach

In the preceding diagram, the user layer defined by master, instantiates a specialized port, the `init_port`, and calls the **`write`**`()` method in this class from an **`SC_THREAD`** process. The `initiator_port` class implements the protocol layer and calls the appropriate **`transport`**`()` function in the TLM channel, `my_chan`. This example illustrates one of many possibilities. In an alternative example, the **`transport`**`()` function could have been implemented in the target via an **`sc_export`**`<if>` connection, which we will explore later.

16.3 TLM Interfaces

The OSCI TLM 1.0 Standard is focused on specifying interfaces. There are three interface categories defined in the OSCI TLM standard:

- Unidirectional blocking interface
- Unidirectional non-blocking interface
- Bidirectional blocking interface

For abstract models, the **SC_THREAD** process style of coding is easier and more convenient to use when implementing. Thread processes are allowed to use the blocking calls, which simplify the use of the TLM channels. Method processes may use TLM channels, but are required to use the non-blocking functions. Using the non-blocking interface will tend to be more complex than using the blocking style. Though more tedious, some components may need to use this non-blocking style. The blocking and non-blocking interfaces are supplied to allow the user flexibility in using either **SC_THREAD** or **SC_METHOD** to their design modules. The following figure illustrates the class hierarchy that implements all of the TLM 1.0 interfaces. The interfaces are broken into **get** and **put** categories. As you can see, the standard divides the interfaces into quite a few fine-grained classes to provide maximum flexibility.

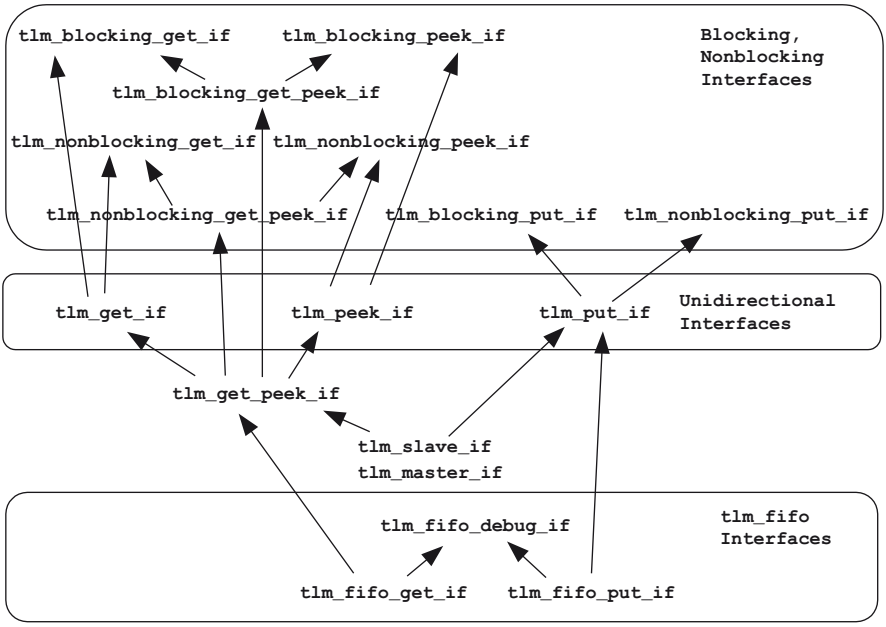

Fig. 16.3 TLM 1.0 class hierarchy

Let us look at each of the interface categories in more detail. Because all the methods are in the base classes, the methods are really not as daunting as the

preceding figures might imply. The derived interface classes represent convenience groupings.

16.3.1 Unidirectional Blocking Interfaces

The following code shows the APIs defined in the OSCI TLM standard for a unidirectional blocking interface, which loosely has behavior matching a FIFO.

```
tlm_uni_channel<T> instance("instance");
instance.get(result_ptr);
result = get();
instance.put(value);
```

Fig. 16.4 Unidirectional blocking `get` interface syntax

The functions `get()` and `put()` correspond to the `sc_fifo<T>` `read()` and `write()`. It should be no surprize that the `get()` and `put()` functions can contain waits and only return when data is transferred. Thus, these functions can only be called from an `SC_THREAD` process because `SC_METHOD` processes are not allowed to wait.

The `get` interface contains an additional API that uses the `tlm_tag<T>`, which allows a target to implement multiple versions of an interface.

```
tlm_uni_channel<T> instance("instance");
instance.peek(result_ptr);
instance.peek(result_ptr, offset);
result = peek();
result = peek(offset);
instance.poke(value);
instance.poke(value, offset);
```

Fig. 16.5 Unidirectional blocking interface syntax

The blocking interfaces also include classes used to check for data availability without actually consuming the data. This is useful when designing a distributed decode and each target needs to determine if the value presented is for the interogating target before consuming the data.

16.3.2 Unidirectional Non-Blocking Interfaces

Methods in the non-blocking category all have the prefix `nb_`. The following code shows the syntax for the basic unidirectional non-blocking interfaces:

```
tlm_uni_channel<T> instance("instance");
if (not instance.nb_get(variable) {
  next_trigger(instance.ok_to_get());
}
if (instance.nb_can_get()) {
  cout << "instance available for get" << endl;
}
if (not instance.nb_put(value) {
  next_trigger(instance.ok_to_put());
}
if (instance.nb_can_put()) {
  cout << "instance available for put" << endl;
}
```

Fig. 16.6 Unidirectional non-blocking syntax

As with the blocking counterparts discussed above, these APIs include **nb_get**() and **nb_put**() functions.

These methods let users check whether data transfer will succeed before actually calling the transfer functions. This set includes functions with return values or events that indicate it is ok to proceed with the data transfer. Depending on the FIFOs and number of requestors, the user is not guaranteed the success of the data transfer even after waiting on the **ok_to_put**() and **ok_to_get**() events. You can verify the success of your transfer by the fail/success return value from the transfer functions.

Note that the non-blocking functions return a **bool** type. This return data type indicates whether the data transfer actually occurred. These non-blocking interfaces can be called from within both **SC_METHOD** and **SC_THREAD** processes.

There are also non-blocking debug interface methods that provide additional features used primarily for verification.

```
tlm_uni_channel<T> instance("instance");
if (!instance.nb_peek(variable, offset) {
  next_trigger(instance.ok_to_peek())&
}
if (instance.nb_can_peek(offset)) {
  cout << "instance available for get" << endl;
}
if (!instance.nb_poke(value) {
  next_trigger(instance.ok_to_poke())&
}
if (instance.nb_can_poke()) {
  cout << "instance available for poke" << endl;
}
```

Fig. 16.7 TLM non-blocking debug interfaces

Debug interfaces never consume time, because they are intended for debug. The idea of the `offset` in the peek and poke routines is that the interface has some depth (e.g., a FIFO). An offset of zero would indicate the topmost piece of

information (i.e., what is retrieved with **get**( )). Other values would probe deeper into the interface. For a FIFO, the `offset` is somewhat obvious. For other constructs, it is not obvious and would be documented by the channel creators.

16.3.3 Bidirectional Blocking Interface

The bidirectional blocking interfaces are provided for cases where there is a one-to-one relationship between the request and response. This interface is provided as a convenience, since the underlying implementation can call the unidirectional blocking **put**( ) and **get**( ) functions. In fact, the OSCI TLM standard implements the transport channel example in this manner.

```
tlm_transport<REQ,RSP> instance;
result = instance.transport(request);
```

Fig. 16.8 Bidirectional blocking syntax

The request and response must be different classes to avoid ambiguities within the class. This structure can be accomplished by deriving one function from the other. This structure avoids a C++ ambiguity to differentiate the signatures of two **get**( ) functions and **put**( ) functions within the class implementation.

Keep in mind that the unidirectional and bidirectional interfaces we just covered are OSCI TLM's core interfaces. The specification provides additional interfaces that bundle some of the core interface APIs in addition to providing a debug interface.

16.4 TLM Channels

As we mentioned before, channels implement the APIs of the interfaces they inherit. The OSCI TLM standard provides three different example channels:

- **tlm_fifo**<*T*> - implements unidirectional interfaces
- **tlm_req_rsp_channel**<*Req,Rsp*> - implements two unidirectional interfaces
- **tlm_transport_channel**<*Req,Rsp*> - implements bidirectional interface

Note that these channels are not part of the TLM specification. These channels are provided as supplementary communication components. In some cases, you may need to develop protocol-specific channels as required by your design.

The **tlm_fifo**<*T*> channel is modeled after the SystemC FIFO class, and this channel implements the unidirectional interfaces, including both blocking

and non-blocking interfaces. The FIFO depth can be defined as any size from zero to infinite depth.

The **tlm_req_rsp_channel**<*Req,Rsp*> implements two unidirectional interfaces, using TLM FIFOS. One FIFO is used for request and the other is used for response. From the connectivity point of view, the **tlm_req_rsp_channel**<*Req,Rsp*> exports the **put** request FIFO interface and **get** response FIFO interface to the master. Likewise, the slave is connected to the **get** request FIFO interface and the **put** response interface.

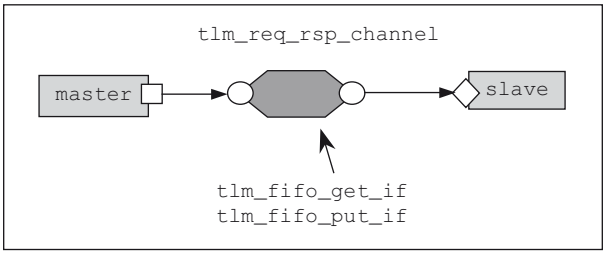

Fig. 16.9 **tlm_req_resp_channel**

The last example channel provided by TLM 1.0 is the **tlm_transport_channel**<*Req,Rsp*>. Internally, this channel is implemented by a **tlm_req_rsp_channel**<*Req,Rsp*> with a FIFO size of 1. The master is connected via a **tlm_transport_if**<*Req,Rsp*> **sc_export**<*T*>, while the slave is connected via the unidirectional interfaces.

16.5 Auxiliary Components

Thus far in this chapter, we have focused on the core TLM 1.0 APIs that define the communication between design components. We also covered some basic OSCI TLM channels that are provided as optional components to the TLM standard. In this section, we focus on the components you will need to model your designs.

In many systems, designs typically contain multiple components including masters, slaves, routers, and arbiters. Regardless of function, each component is connected via a port/export pair, or through an intermediate channel.

The OSCI TLM 1.0 kit provides documentation, working examples, and example components masters, slaves, an arbiter, and a router. These components and example configurations provide a base for developing your own TLM components. Developers new to SystemC should study these examples closely; some features are subtle such as model connectivity and the use of specialized ports for detail abstraction. Developers would be well served, if they created their own block diagrams to trace model connectivity, and review the previous chapter on specialized ports.

16.5.1 TLM Master

The figure below shows a master instantiating a port using the **tlm_transport_if**<*Req*,*Rsp*> interface.

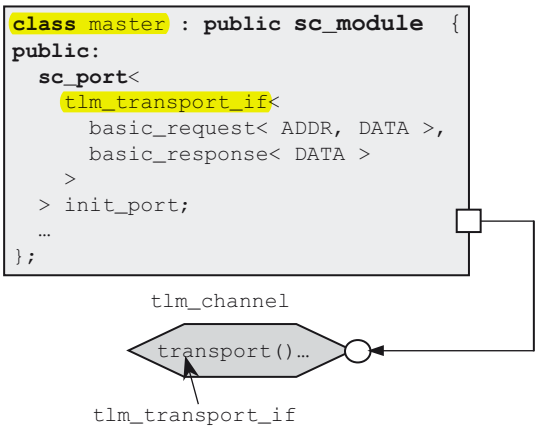

Fig. 16.10 Master connected to TLM channel

Notice that the master can be easily modified to use a different interface by changing its port specification. Of course, the channel will need to match the corresponding interface. In this example, we have defined a single port master, but you may find it necessary to instantiate multiple ports, each of which may be connected via different interfaces.

Next, note that the request and response types are user-defined and can be customized to your design requirements. Lastly, the example shows a master connected to a channel. In some cases, you may want to directly connect the master to a memory target, bypassing channels entirely. This structure is possible if the target also defines the same port TLM interface and implements the associated APIs, as described in the next section.

16.5.2 TLM Slave

A TLM slave contains similar components. The following figure shows a slave target connected to a master via an **sc_export**<*T*> and **sc_port**<*I*> connection, respectively. In the example, both ports are declared with the bidirectional interface, **tlm_transport_if**<*Req*,*Rsp*>. Subsequently, the slave must implement the interface API, **transport**() function.

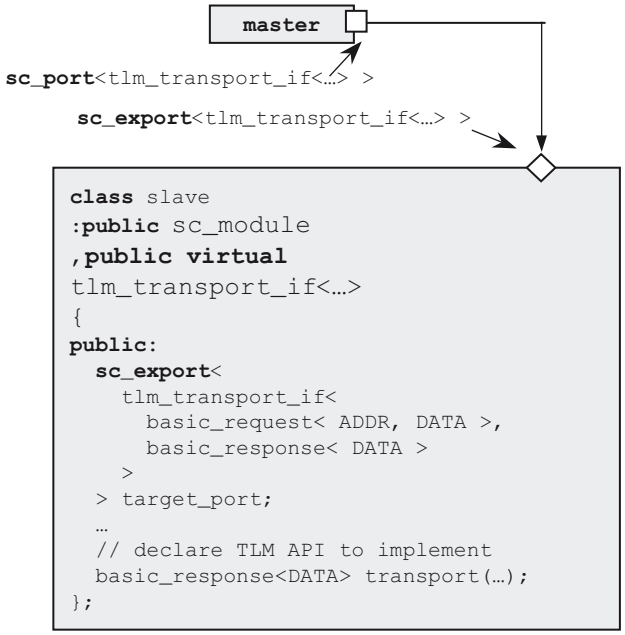

Fig. 16.11 TLM slave and TLM master connection

The designer is free to choose any of the TLM interface, transfer data type, and number of ports for the module. Just make sure the interface is the same between connections and the data type is consistent between modules.

16.5.3 Router and Arbiter

Routers and arbiters are often used in designs to model real systems. As components in TLM modeling, they transfer data in the same manner as master and slave components we just discussed. The router uses the **sc_export**<*I*> construct for connection, and the router master interface uses the **sc_port**<*I*> construct for connection as shown below. Note that the TLM 1.0 kit examples have some interesting names.

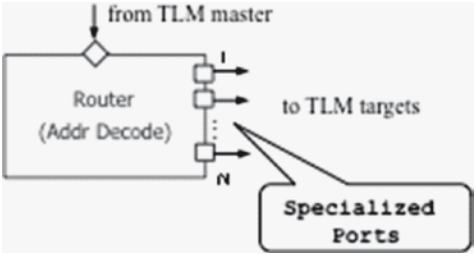

Fig. 16.12 TLM router

The OSCI TLM provides an example of a basic router utility that uses a specialized port to communicate to TLM targets.

The OSCI TLM arbiter uses polling to get requests from its master interface ports as shown below. When the arbiter finds a request on the master ports, it forwards the request onto the TLM target. The OSCI arbiter uses a starving priority algorithm, which you may want to consider modifying based on your design.

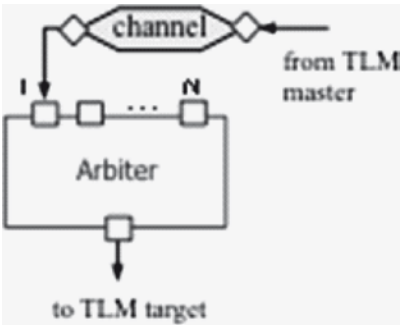

Fig. 16.13 TLM arbiter

As with the TLM channels, the master, slave, router, and arbiter components are not part of the OSCI TLM 1.0 specification. However, these utilities can be used to start off a high-level abstract design with very little effort.

16.6 A TLM Example

In this section, we will discuss using the TLM methodology to design a realistic system. When creating a system-level design, it is important to ask yourself what the purpose of your model is. As we mentioned earlier in the chapter, a TLM model can be used to aid architectural exploration, verify system performance, assist in early software development, or aid in functional verification.

In some cases, your model may be used for multiple purposes, which means your system-level model architecture should also facilitate modeling at different levels of abstraction. The example in this section will focus on a high-level abstract system model that can be used for architectural exploration and software development. In addition, we will present some simulation performance statistics that show significant incentives for using TLM modeling.

The figure below shows the system block diagram for a Voice-Over-IP (VoIP) design. This system uses many typical SoC components, including a processor, flash memory, RAM, and IO devices. All system components are connected using TLM.

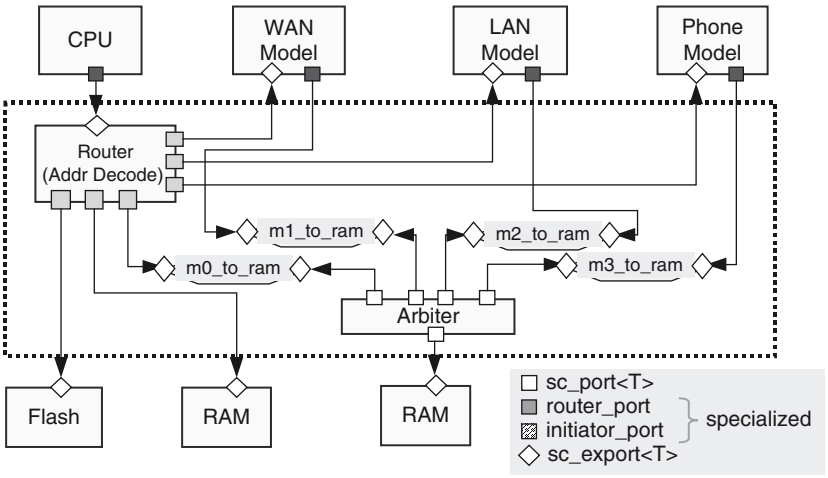

Fig. 16.14 VoIP system

This system model simulates a home network with multiple PCs and a phone. The WAN model generates incoming Ethernet traffic, the LAN model generates output Ethernet traffic, and the phone model generates in/out phone traffic. Specifically, we would like to simulate a 20 second phone call during simultaneous network traffic.

Each IO device uses the same initiator class, configured to generate different types of traffic. In addition, each IO device also instantiates a target port to support programmer's view, which allows bit accurate control and status register access. The target models are simple memories with storage capabilities and read/write timing delays. The router is an address decoder, and the arbiter uses a simple round-robin algorithm. Our CPU model simulates random traffic to memory and also loads device drivers for each IO device. The device drivers manage transmit and receive queues on the IO devices, transfer received packets to destination NICs, and manage buffer allocation.

The following figure shows how the target finally processes an initiator request using the **`tlm_transport_if`**<*`Req, Rsp`*> interface. First, a SystemC process (e.g., testbench, stimulus generator, etc.) calls the initiator to issue a memory write request. The initiator calls the **`transport`**() function via the TLM interface port, **`init_port`**. At this point, the request may travel through other components, including channels, routers, and arbiters, before finally arriving at the memory target.

The target "receives" the write request in the form of a function call to its **`transport`**() function and processes the request accordingly. Note that this interface, **`tlm_transport_if`**<*`Req, Rsp`*>, is a blocking interface. This means that the originating SystemC thread will block until it receives a response from the slave.

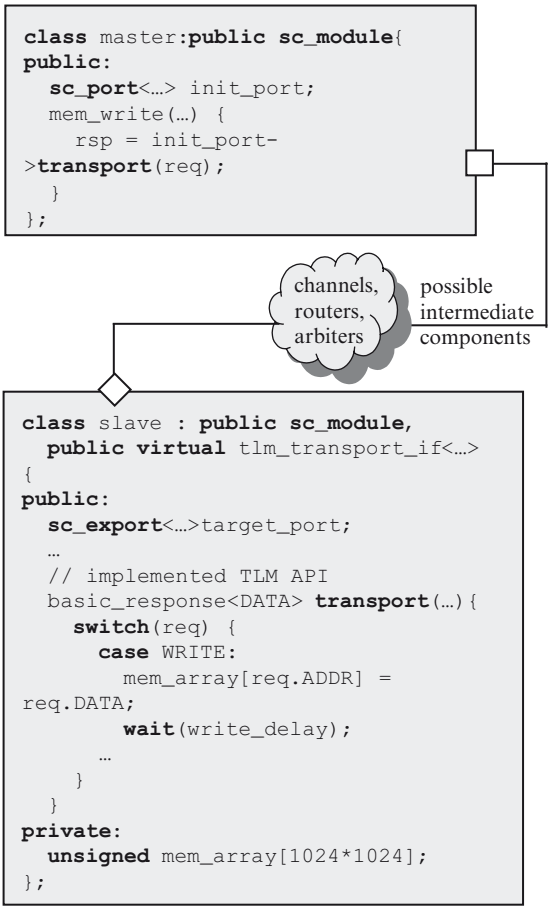

Fig. 16.15 Transaction sequence via **`tlm_transport_if`** interface

By using this system model, we are able to validate and analyze several characteristics about our design, including:

- Bus sizing
- Single cycle vs. block transfer vs. bus widths
- Block diagram—identify all required blocks and IO
- HW/SW partitioning
- Memory partitioning and performance—separating instruction and data memory.
- Memory access contention using the arbiter
- Memory transmit/receive processing queue size
- HW/SW interface

One advantage to designing a model at this level of abstraction is the simulation performance. We simulated our model for 20 seconds, which only required 8.43

seconds of CPU time; 42% of simulated time. Of course, your simulation performance is heavily dependent on your designs and your host computer. We used timed approximations for all read/write transactions, used an event-driven vs. polling where possible, and used burst transactions for data transactions.

To summarize, TLM modeling is great for system design feedback. Doing so requires a reasonable amount of effort, and the simulation performance lets you change your design and re-simulate very quickly. Some important thoughts to remember are to focus on what you are trying to achieve with your model, and leverage the existing IP where possible.

16.7 Summary

In this chapter, we presented the OSCI TLM 1.0 standard, which defines a set of TLM API interfaces. We also presented some of the main components for TLM designs, including channels, initiator, target, router, and arbiter. With this knowledge, you are ready to design a TLM system-level model.

Before you start your design you may want to review the TLM methodology chapter earlier in the book. While we have focused on the "how" of TLM-based design in this chapter, the TLM methodology discussion earlier in the book discusses the "why" of TLM design and how it fits with project flow.

It will become apparent, that while one design group may use this methodology for architectural exploration, another group may extend the design for another purpose, such as early software development. It is important to understand the usage model and create the system-level design to be flexible and to allow for different levels of abstraction.

Finally, keep in mind that the material in this chapter applies to the OSCI TLM 1.0 standard. The OSCI TLM Working Group is currently developing the TLM 2.0 standard. The updated standard is targeting improvements in the TLM APIs, adding generic data structures, and adding timing annotation capabilities. The API improvements and generic data structures will help greatly in model interoperability, and the timing annotation capabilities will allow faster simulations. We strongly encourage the reader to find out more on both the TLM 1.0 and planned features for TLM 2.0 on the OSCI SystemC site, www.systemc.org.

16.8 Exercises

For the following exercises, use the samples provided at www.scftgu.com

Exercise 16.1: Using the TLM 1.0 example channels, create a simple design that uses both the **`sc_fifo`**<*T*> and the **`tlm_fifo`**<*T*>. Discuss how these differ from one another. In what situations might you prefer the TLM FIFO over the core FIFO?

Exercise 16.2: Write your own TLM FIFO derived from the **`tlm_fifo`**<*T*> that adds the following features:

- Add a monitor to efficiently count the number of times the FIFO becomes full or empty, and report the count at the end of the simulation.
- Add a monitor to determine the average FIFO depth.
- Add a functional coverage monitor to determine how many different values were placed in the FIFO and report the coverage at the end of simulation.
- Create a testbench to exercise your FIFO.

Exercise 16.3: Discuss the advantages and disadvantages of passing arguments by-value vs. by-reference. Explain data ownership and data lifetime. How might the boost.org shared pointer class help this situation?