Center for Embedded Computer Systems
University of California, Irvine

System Level Modeling of an AMBA Bus

Gunar Schirner, Rainer Dömer

Technical Report CECS-05-03
April 1, 2005

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

hschirne@uci.edu, doemer@uci.edu
http://www.cecs.uci.edu/

System Level Modeling of an AMBA Bus

Gunar Schirner, Rainer Dömer

Technical Report CECS-05-03
April 1, 2005

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

hschirne@uci.edu, doemer@uci.edu
http://www.cecs.uci.edu

Abstract

The System-On-Chip (SoC) design faces a gap between the production capabilities and time to market pressures. The design space, to be explored during the SoC design, grows with the improvements in the production capabilities and it takes an increasing amount of time to design a system that utilizes those capabilities. On the other hand shorter product life cycles are forcing an aggressive reduction of the time-to-market. Addressing this gap has been the aim of recent research work. As one approach abstract models have been introduced and a design flow was devised that guides the designer in the process from a most abstract model down to a synthesizable model.

Throughout the design process computation and communication concerns are handled individually. The communication is mostly abstracted away from the designer, which allows the design focus to rest on the application specific computation. This separation requires the provider of an SoC design tool to supply fast and accurate communication models.

Fast simulation capabilities are required for coping with the immense design space that is to be explored; these are especially needed during early stages of the design. This need has pushed the development of transaction level models, which are abstract models that execute dramatically faster than synthesizable models. The pressure for fast executing models extends especially to the frequently used and reused communication libraries. This document describes the system level modeling of the Advanced High-performance Bus (AHB) part of the Advanced Microprocessor Bus Architecture (AMBA). Throughout this work the design of three bus models, at different levels of abstraction, is described; their simulation speed and accuracy is evaluated. As a result guidelines for the developer are derived that support selecting the most appropriate model for a given stage in the design process.

Contents

List of Figures

List of Acronyms

AHB Advanced High-performance Bus. System bus definition within the AMBA 2.0 specification. Defines a high-performance bus including pipelined access, bursts, split and retry operations.

AMBA Advanced Microprocessor Bus Architecture. Bus system defined by ARM Technologies for system-on-chip architectures.

APB Advanced Peripheral Bus. Peripheral bus definition within the AMBA 2.0 specification. The bus is used for low power peripheral devices, with a simple interface logic.

ASB Advanced System Bus. System bus definition within the AMBA 2.0 specification. Defines a high-performance bus including pipelined access and bursts.

ATLM Arbitrated Transaction Level Model. A model of a system in which communication is described as transactions, abstract of pins and wires. In addition to what is provided by the TLM, it models arbitration on a bus transaction level.

Behavior An encapsulating entity, which describes computation and functionality in the form of an algorithm.

Bus Functional Model A wire accurate and cycle accurate model of a bus.

Channel An encapsulating entity, which abstractly describes communication between two or more partners.

CLI Cycle Level Interface. Refers to ARMs definition of the AMBA bus, cycle level accurate for SystemC.

IP Intellectual Property. A pre-designed system component.

MAC Media Access Control. Layer within the OSI layering scheme.

NoC Network on Chip

OS Operating System. Software entity that manages and controls access to the hardware of a computer system. It usually provides scheduling, synchronization and communication primitives.

OSI Open Systems Interconnection. An communication architecture model, described in seven layers, developed by the ISO for the interconnection of data communication systems.

PE Processing Element. A system component that provides computation capabilities, e.g. a custom hardware or generic processor.

RTL Register Transfer Level. Description of hardware at the level of digital data paths, the data transfer and its storage.

RTOS Real-Time Operating System. An operating system that responds to an external event within a short, predictable time.

SCE SoC Environment. A set of tools for the automated, computer-aided design of SoC and computer systems.

SoC System-On-Chip. A highly integrated device implementing a complete computer system on a single chip.

TLM Transaction Level Model. A model of a system in which communication is described as transactions, abstract of pins and wires.

System Level Modeling of an AMBA Bus

G. Schirner, R. Dömer

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

hschirne@uci.edu, doemer@uci.edu
http://www.cecs.uci.edu

April 1, 2005

Abstract

The SoC design faces a gap between the production capabilities and time to market pressures. The design space, to be explored during the SoC design, grows with the improvements in the production capabilities and it takes an increasing amount of time to design a system that utilizes those capabilities. On the other hand shorter product life cycles are forcing an aggressive reduction of the time-to-market. Addressing this gap has been the aim of recent research work. As one approach abstract models have been introduced and a design flow was devised that guides the designer in the process from a most abstract model down to a synthesizable model.

Throughout the design process computation and communication concerns are handled individually. The communication is mostly abstracted away from the designer, which allows the design focus to rest on the application specific computation. This separation requires the provider of an SoC design tool to supply fast and accurate communication models.

Fast simulation capabilities are required for coping with the immense design space that is to be explored; these are especially needed during early stages of the design. This need has pushed the development of transaction level models, which are abstract models that execute dramatically faster than synthesizable models. The pressure for fast executing models extends especially to the frequently used and reused communication libraries. This document describes the system level modeling of the AHB part of the AMBA. Throughout this work the design of three bus models, at different levels of abstraction, is described; their simulation speed and accuracy is evaluated. As a result guidelines for the developer are derived that support selecting the most appropriate model for a given stage in the design process.

1 Introduction

1.1 Introduction to SoC Design

1.1.1 Overview

Improvements in manufacturing capabilities allow placing of a complete embedded system on a single chip. With that it becomes possible to design a system as a mix of software running on one or more generic processors and specialized hardware, which runs computation that is too costly for a generic processor (e.g. in terms of power or time). This design freedom leads ultimately to highly specialized chips and cost efficient production. However the newly gained freedom in design places a burden on the SoC designer. The next paragraphs will introduce the challenges of system level design, the specification of systems and the design space exploration.

1.1.2 Challenges

The design of embedded systems in general and an SoC in special will be done under functional and environmental constraints. Since the designed system will run under a well-specified operating environment, the strict functional requirements can be concretely defined. The environment restrictions on the other hand are more diverse: e.g. minimizing the cost, footprint, or power consumption. Due to the flexibility of a SoC design, achieving the set goals, involves analyzing a multi-dimensional design space. The degrees of freedom stem from the process element types and characteristics, their allocation, the mapping of functional elements to the process elements, their interconnection with busses and their scheduling.

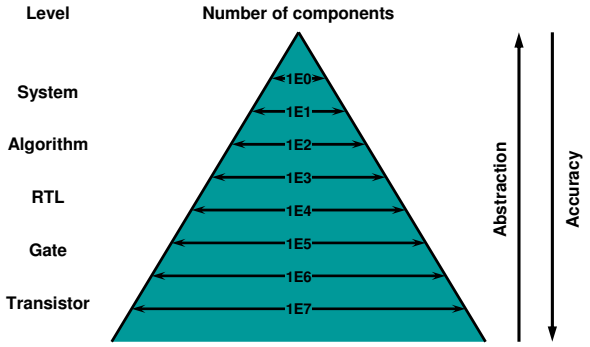


Figure 1: Abstraction levels in SoC design (source [13])

Looking at the levels of abstraction of the SoC design gives another perspective to the complexity of designing such systems. The process starts with a functional description on system level, where only the major function blocks are defined and timing information is not yet captured. During the SoC design process, the system description is refined step by step and additional details are captured. That process leads to a cycle accurate fully functional system description in RTL, which is the starting point of the production process. As Figure 1 shows, the amount of captured information increases by an oder of magnitude with each level of the design process. With each step within

the levels of abstraction a multi-dimensional design space has to be explored in order to make the necessary decisions.

The goal of SoC design paradigm is to guide the designer through the process, and aid the decision making. A well-defined flow of design steps makes the process manageable. The design steps and their associated models will be described in the next paragraphs.

1.1.3 SoC Specification

Hardware/Software co-design is an integral aspect of the SoC design. It requires a language that is capable of capturing the requirements of a hardware design from wire allocations to complex timing requirements, as well as the complexities of current software design. Some examples of such languages are SpecC [11], an ANSI-C based language extension and the C++ library extension SystemC [15].

Those languages allow grouping of functionality to behaviors, which later can be freely mapped to processing elements. In order to allow this free mapping the computation has to be separated from the communication. Therefore communication between the behaviors is abstractly defined as channels. The channel specific implementation (e.g. an AMBA protocol) will be filled in during later refinement stages. The specification model is free of such implementation detail (and their respective constraints). The SpecC language further introduces many concepts from hardware description languages like VHDL and Verilog. It introduces the concept of capturing scheduling information in the language, such as sequential, parallel and pipelined execution. The SpecC language very much supports the goals of specification capturing. It allows describing a fully functional model that incorporates design constraints and has a simulation environment for an integrated validation against a set of test vectors. The next section describes the exploration and refinement steps to transform the system specification into a manufacturable description.

1.1.4 SoC Design Space Exploration

In conjunction with the SpecC language a design paradigm was introduced, which formalizes the individual refinements steps. With that the designer has guidelines on how to efficiently handle the immense design space. Figure 2 shows an overview of the design flow. It also indicates the integration of the validation flow. The tool suite provided with the SpecC language closely follows the outlined design flow. The following paragraphs will describe each design step.

The SoC design starts with the specification model, which is a purely functional model - free of any implementation details. It focuses on capturing the algorithmic behavior and allows a functional validation of the description. The model is untimed and allows only for causal ordering. Once the specification model is finished, it will serve as a golden model, to compare simulation results during the design cycle.

Architecture information is added during the Computation design. During this step processing elements are inserted into the system and the previously defined functional behaviors are mapped to them. A processing element can be a predefined standard component such as generic processor core or a DSP, but a custom specific hardware component as well. Parameters, such as clock frequency, of the inserted elements can be adjusted to the application needs. Based on internal statistics, early

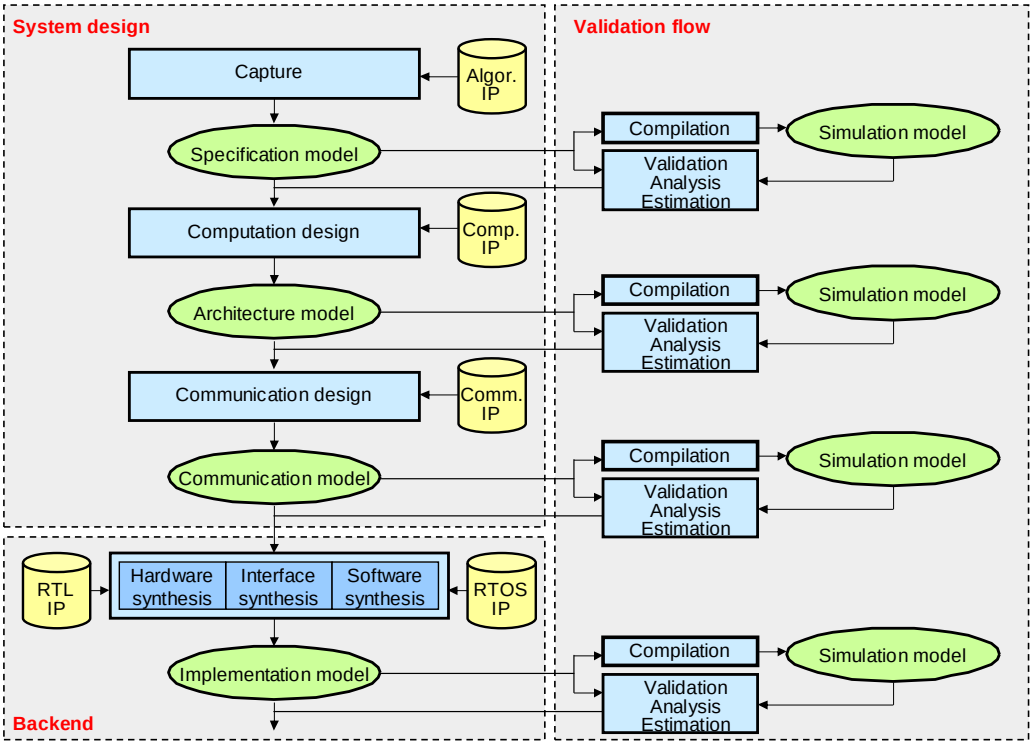


Figure 2: Design methodology for SoC design (Source [12])

estimations about the runtime performance can be made. This gives the designer the first feedback about the design decisions. Once the computation design is finished, the architecture model that captures the decisions is created. This model is the first timed model. It takes only computing time into account; all communication between the processing elements execute in zero time.

The next step in the refinement is the Scheduling Refinement (not shown in this graph). This refinement allows the designer to select suitable scheduling mechanisms to its processing elements. The scheduling capabilities range from an off-line static scheduling, which allows the most predictability, to a priority based dynamic scheduling.

The Communication design allows the user to select busses and protocols. Here the earlier defined abstract communication channels are mapped to physical busses and protocols. Detailed information about a utilized protocol is added. The resulting Communication model includes specific instructions for the particular bus implementation, like the access logic for a bus master or bus slave.

The synthesis step concludes the the design flow. Here the Register Transfer Level (RTL) code for the hardware will be generated with the prerequisite of RTL component allocation, their functional mapping and scheduling. As a result of the hardware synthesis a cycle accurate description of each hardware processing element is created. Similar activities take place for the software synthesis. Here specific code for the selected RTOS is inserted and target specific assembly code is compiled. The result is a cycle accurate model of each software-processing element, which can be simulated

using an instruction set simulator and executed on the target processor. The combination of both synthesis parts is captured in the Implementation model, which gives a cycle accurate description of the whole system.

1.2 Problem Definition

As it was described in the previous section the SoC design process is performed in several steps that formalize coping with the immense design space. Models of predefined standard components, such as basic communication elements, are needed for ease of design. Furthermore multiple models at different levels of abstraction are needed for each standard component, matching the stage within the design flow. An very abstract model can be used for fast high level exploration during early stages of the design, whereas a detailed model that yields most accurate results is needed for production validation.

The scope of this work is to model a library communication component as symbolically depicted in Figure 3. In particular, AMBA was chosen since it reached, especially after introducing revision 2.0 of the standard in 1999, a wide acceptance for interconnections within a system-on-chip. With ARM's strong support for design, development and testing it pushed "right-first-time" development and the bus AMBA specification became one de facto standard for on-chip bus [2]. The goal this work is to provide a bus functional model of an AMBA bus, that is synthesizable, and to model the bus as well at higher level of abstractions that allow a high simulation performance.

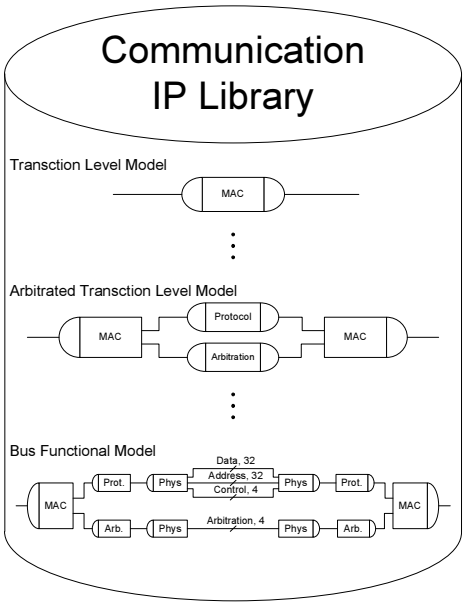


Figure 3: Scope of work: modeling of a communication IP (Symbolic Depiction)

Throughout the work appropriate levels of abstractions should be chosen for the abstract models. The implemented models should be validated against the standard with respect to functionality and timing accuracy. They should furthermore be compared to each other in terms of execution

performance and simulation speed. Based on the experimental results a guideline should be made on how to choose the right model for a particular goal.

1.3 Outline

In the remaining part of the documentation, first a general introduction to the AMBA bus gives the reader an overview of the specification. The overview is followed by the chapter on the actual design. The different models will be introduced. Their design will include a layered approach. Based on the design, accuracy expectations of each model will be described.

In the validation chapter (Section 4), the reader will find a functional and timing validation of the implemented models. Those validations will be made according to the specification [3].

The Section 5 shows measurements of the simulation speed and compares the accuracy of the individual bus models. It shows what trade offs the designer has to make for using a particular model. Finally Section 6 concludes and gives a summary.

1.4 Related Work

System level modeling has become a more important issue over the recent years, as a means to improve the SoC design process. Languages for capturing these models have been developed, such as SpecC [11] or SystemC [15]. Furthermore capturing and designing communication systems using transaction level models has received research attention.

Sgroi et al. [22] address the SoC communication with a Network on Chip (NoC) approach. They propose partitioning of the communication into separate layers that follow the Open Systems Interconnection (OSI) structure. Software reuse is promoted with an increase of abstraction from the underlying communication framework.

Siegmund and Müller [24] describe an extension to SystemC, and propose modeling of a SoC at different levels of abstraction. They describe three different levels of abstraction: the physical description at RTL level, then a more abstract model that covers individual messages, and a most abstract level that deals with transactions.

In application of transaction level models [15], the topic of capturing communications within a SoC has received attention. In particular the widely used bus specification AMBA was the goal of modeling support.

Most relevant to this work is ARMs definition of the Cycle Level Interface (CLI) of the AMBA bus [1]. This specification defines how to implement the AMBA bus architecture in SystemC [21]. It has the goal of defining an interfacing standard between SystemC design models of IP components. It is intended to be used for system simulation and transaction based verification.

In [6] Caldari et al. describe the results of capturing the AMBA rev. 2.0 bus standard in SystemC. The bus system has been modeled at two levels of abstraction, first a bus functional model on RTL level and second a model on TLM level. Their Transaction Level Model (TLM) model reached a speedup of 100 over the RTL level model.

Another modeling approach of the AMBA bus architecture is shown in [25], where a transaction-based modeling abstraction level was described. While maintaining the bus cycle accuracy, this approach achieved a 55% speedup over the bus functional model.

CoWare [7] provides with ConvergenSC a commercial AMBA Transactional Bus Simulator. It allows for a fast cycle accurate architectural optimization and verification of an SoC design. With that it provides a solution for designing system-on-chip products that make use of AMBA bus specification and are described in SystemC.

2 Introduction to the AMBA Bus

The Advanced Microprocessor Bus Architecture (AMBA) (see [3]) defined by ARM is a widely used open standard for an on-chip bus system. This standard aims to ease the component design, by allowing the combination of interchangeable components in the SoC design. It promotes the reuse of intellectual property components, so that at least a part of the SoC design can become a composition, rather than a complete rewrite every time.

The AMBA standard defines different groups of busses, which are typically used in a hierarchical fashion. The Figure 4 shows a schematic overview of a typical microprocessor design. The design usually consists of a system bus; either the older version the Advanced System Bus (ASB), or the more performant Advanced High-performance Bus (AHB). All high performance components are connected to the system bus. Low speed components are connected to the peripheral bus, the Advanced Peripheral Bus (APB).

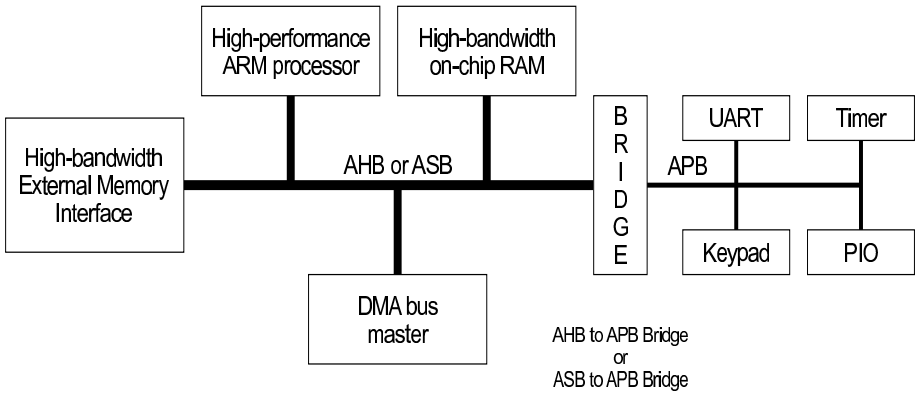


Figure 4: AMBA hierarchical bus architecture (Source [3]).

The system busses ASB and AHB are designed for high performance connection of processors, dedicated hardware and on chip memory. They allow:

- Multiple bus masters
- Pipelined operation
- Burst transfers

The peripheral bus APB on the other hand is designed for low power peripherals with a low complexity bus interface. The APB can be connected via a bridge to both system busses AHB and ASB. The APB bridge acts as a master on the APB bus and all peripheral devices are slaves.

The bridge appears as a single slave device on the system bus; it handles the APB control signals, performs retiming and buffering.

Between the two system busses the AHB delivers a higher performance than its older counterpart ASB. The AHB features:

- Retry and split transactions
- Single clock edge operation
- Non-tristate implementation
- Allows wider data bus configuration (e.g. 64 bits and 128 bits)

Retry and split transactions are introduced to reduce the bus utilization. Both can be used in case the slave does not have the requested data immediately available. In case of a retry transaction, the master retries the transaction after and own arbitrary delay. On the other hand in a split transaction the master waits for a signal from the slave that the split transaction can be completed.

One major factor for the high performance of the AMBA system busses is the pipelined access. For that, each bus access is executed in three separate stages, which can overlap between masters. The three phases for the pipelined bus access are:

Arbitration Phase. A master requests a bus access to the arbiter. The arbiter grants the access within an arbitrary number of bus cycles (at least one). Multiple masters may request the bus at the same time, however only a single master is granted at any given point in time.

Address Phase. The granted master applies the address and control signal to the bus. The address and control signals determine the activity for the next phase.

Data Phase. Depending on the control signals from the previous phase (e.g. write direction) either the granted master or the selected slave write the data to the data bus.

The AHB standard defines a non-tristate bus interface, which simplifies the design of the bus interfaces. It furthermore simplifies simulation of the bus system, since the costly three or four value logic - necessary for simulating a tristate interface - is not required. On the other hand, a non-tristate bus interface increases the number of connection for each bus interface; read and write bus have to be handled separately. This however is not a limiting factor, since the bus system is targeted for on-chip connections. It does, however, require an interconnection network, in which multiplexers select the bus access for each device. Figure 5 shows the AHB interconnection network.

Three separate virtual busses, implemented by multiplexers, compose the interconnection network. The address / control bus (represented with HADDR) and the write data bus (represented with HWRITE) are written by each master. A slave writes to the own portion of the read data bus; a multiplexer selects the bus portion of the active device and distributes the selected signals. Since the AHB performs operation in a pipelined fashion, two separate multiplexers are necessary for the address / control bus and the write data bus; their access happens in separate stages of the pipeline.

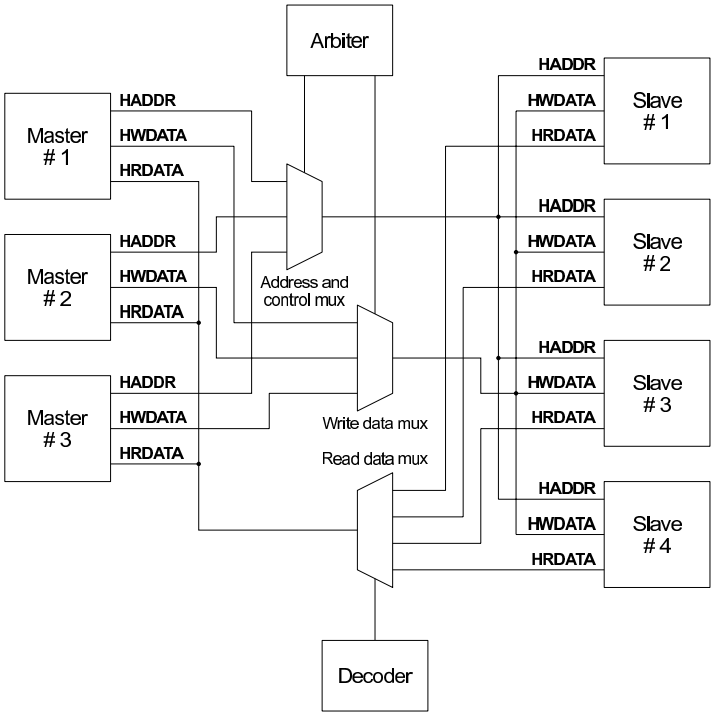


Figure 5: Interconnection network for the AMBA AHB (Source [3]).

3 Modeling

As the introduction has motivated, high simulation speeds are necessary for an efficient design space exploration. High simulation speeds allow the designer to explore more solutions, thus increasing the chance of arriving at solution that is closer to the optimum. One possibility for a fast exploration is modeling at higher levels of abstraction (i.e. TLM) and gradually filling in details until a detailed synthesizable model is reached. In order to effectively support different levels of abstraction throughout the design process, a matching set of abstraction levels for library component is needed. Due to their frequent use this is especially true for bus components.

The following sections describe the design of the bus models for the AMBA AHB. First a generic layering approach will be introduced, which helps coping with the complexity of a bus simulation. The OSI layering scheme [16] was used as a reference for deriving those layers. The sections following that will describe each bus model in detail and show how the layered approach is applied.

3.1 Layering

A layered architecture was chosen for the communication system modeling in order to cope with the complexity of communication, in that it is similar to a general network stack implementation. [12] has introduced the applied layering structure as shown in Table 1. The layering structure was derived from the ISO OSI reference model [16].

Layer	Interface semantics	Functionality	Impl.	OSI
Application	N/A	• Computation	Application	7
Presentation	PE-to-PE, typed, named messages • `v1.send(struct myData)`	• Data formatting	Application	6
Session	PE-to-PE, untyped, named messages • `v1.send(void*, unsigned len)`	• Synchronization • Multiplexing	OS kernel	5
Transport	PE-to-PE streams of untyped messages • `strm1.send(void*, unsigned len)`	• Packeting • Flow control • Error correction	OS kernel	4
Network	PE-to-PE streams of packets • `strm1.send(struct Packet)`	• Routing	OS kernel	3
Link	Station-to-station logical links • `link1.send(void*, unsigned len)`	• Station typing • Synchronization	Driver	2b
Stream	Station-to-station control and data streams • `ctrl1.receive()` • `data1.write(void*, unsigned len)`	• Multiplexing • Addressing	Driver	2b
Media Access	Shared medium byte streams • `bus.write(int addr, void*, unsigned len)`	• Data slicing • Arbitration	HAL	2a
Protocol	Unregulated word/frame media transmission • `bus.writeWord(bit[] addr, bit[] data)`	• Protocol timing	Hardware	2a
Physical	Pins, wires • `A.drive(0)` • `D.sample()`	• Driving, sampling	Interconnect	1

Table 1: Communication layers (source [12]).

Table 1 shows an overview of the layer separation, it also indicates where a particular layer is implemented and shows a representative code example for an invocation of each layer. The following list describes each layer in more detail. A full description can be found in [12, chapter 5].

Application Layer. The application layer implements the computational functionality of the system. The layers basic content is defined by the designer during the specification and gradually implemented during the development process. During the design process the initial application specification is mapped onto individual Processing Elements (PEs). This application layer defines the system behavior and describes how the user data is processed in the system.

Presentation Layer. The presentation layer provides named channels, over which structures can be repeatedly transferred. The data structures are converted by the presentation layer into blocks of ordered bytes. Transmissions using the presentation layer are reliable. They can be synchronous or asynchronous.

Session Layer. The session layer is the interface between the software application and the Operating System (OS). It provides synchronous and asynchronous transport of untyped blocks of bytes. In case the lower layers do not provide synchronous access, synchronization will be implemented in this layer and an end-to-end synchronized access is realized. The channels provided by the session layer are used for identification of individual software entities. The session layer multiplexes multiple message blocks into an untyped message stream within the transmitting stack. Within the receiving stack, the session layer demultiplexes the incoming message stream into message blocks.

Transport Layer. The transport layer provides a reliable transmission of untyped streams between PEs in the system. The channels between the PEs act as pipes that carry the streams of the layers above. The transmission characteristics are generally asynchronous. The transport layer implements end-to-end flow control as a part of the operating system. The transport layer implement segmentation and reassembly, to split up the streams into smaller packets.

Network Layer. The network layer provides services for establishment of end-to-end paths, which carry the packet streams from the layers above. It completes the operating system kernel implementation for high-level end-to-end communication. The layer routes individual packets over point-to-point links, separating different end-to-end paths going through the same station. For a particular SoC design this routing could be static, and may even involve dedicated logical links.

Link Layer. The link layer provides services for the link establishment between two directly connected stations. It allows the exchange of uninterpreted packets of bytes. The link layer is the highest layer for a peripheral driver inside the operating system kernel. It defines the type of station (e.g. master / slave) and supports synchronization primitives (i.e. splits each logical link into a separate data and control stream).

Stream Layer. The stream layer implements services for transporting control and data messages between stations. It provides merging of multiple separate data/control streams over a single shared medium. It therefore provides addressing by which it separates the individual streams. The data messages are uninterpreted blocks of bytes. The format of the control messages is heavily implementation dependent (e.g. interrupt handling, polling). The transportation services are generally asynchronous and unreliable. However the reliability may depend on synchronization on higher levels (e.g. flow control).

Media Access Layer (1). The media access layer provides services for the transmission of a contiguous block of bytes over the selected media. The layer hides the specific implementation of the transmission medium, it is the lowest layer that provides a medium independent access. The media access layer provides data slicing, for that the incoming data transfer request, called the user transaction, is split into individual bus transactions. The size of the bus transactions depends on the medium.

Protocol Layer (2). The protocol layer provides transmission capabilities for individual bus transactions - words, shorts, bytes and defined lengths of blocks. The layer also performs arbitra-

tion for each bus transaction.

Physical Layer (3). The physical layer implements a bus cycle access to the physical wires. It performs sampling and driving of individual bus wires. Separate facilities are provided for accessing the data, address and control portion of the bus. The physical layer also provides all implementation necessary for the bus connection scheme, i.e. in case of the AHB the interconnection network consisting of multiplexers. Furthermore the physical implementation of arbitration is included.

For the work described in this document, parts of the library structure of the existing modeling environment, SoC Environment (SCE), have been reused. It was therefore not necessary to implement all of the layers above. Instead only the media specific layers - Media Access Layer, Protocol Layer and Physical Layer - have been implemented. Additionally it has been shown, that the link layer and the stream layer, although technically media dependent, are identical to a previous existing master slave bus model of the Motorola Master Bus, hence these layers have been reused.

The following table lists the layers, that have been specifically implemented for the AMBA model. The table makes also a connection between the granularity of simulating the databus and the layering scheme, as an alternative explanation of the layering.

Number	Layer	Data Granularity
1	Media Access Layer	User Transaction
2	Protocol Layer	Bus Transaction
3	Physical Layer	Bus Cycle

Table 2: Implemented layers and their granularity of data handling

The previous layer description was based on functional concerns. In an alternative view of the same layering scheme, the implemented layers can be described by using the granularity of data handling.

User Transaction (1). A user transaction is a request for transferring a contiguous block of data to or from a particular bus base address. The size of that request is arbitrary - independent of the bus limitations. The base address of the transfer is arbitrary as well. User transactions are used as an interface to the media access layer. They are then divided into one or more bus transactions.

Bus Transaction (2). A bus transaction is bus primitive. It supports transmission of individual elements such as byte, word or long. A particular bus (like the AHB) may also support transporting a collection of those individual elements, which are then transferred as a burst. The possible values for the bus transaction size and the requirements for the base address depend on the bus implementation (e.g. a bus transaction may not have a size of 3 bytes, or bursts have to start on a long aligned address). Bus transactions are used as an interface to the protocol layer. They are then transferred using the physical layer within one or more bus cycles.

Bus Cycle (3). The timed access to a synchronous bus is performed with a bus cycle granularity. During a bus cycle the values of wires/signals composing a bus may be changed. Typically this access is grouped by functionality, e.g. writing of address lines / control lines or reading of the data lines. The physical layer provides a bus cycle access to the bus.

The above defined levels of data granularity can also be analyzed with respect to time. Figure 6 shows how a user transaction is successively decomposed in time into the smaller elements: bus transaction and finally bus cycles. The coarse grain description of a user transaction, as accepted by the media access layer, is divided into one or more bus transactions. An individual bus transaction is transferred by the protocol layer in one or more bus cycle using the facilities of the physical layer.

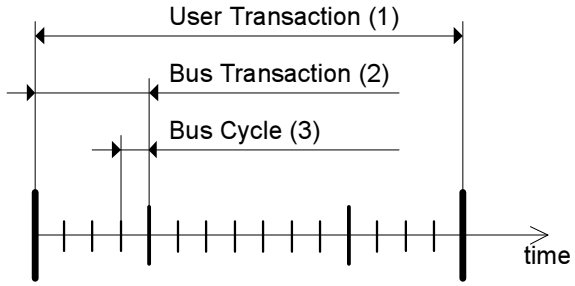
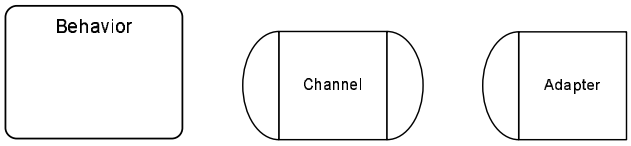


Figure 6: Decomposition of a user transaction in time into bus transactions and bus cycles.

Following the concepts of system level modeling, each of the described layers was implemented in form of an individual channel. Using the channel concept allows a convenient handling of the abstraction levels. As an example the bus functional model requires all channels (all layers) for its operation, a more abstract model may reuse a subset of the defined channels and implement only one channel for the abstract simulation.

3.2 Graphical Notation

The graphical notation for the model description follows the definitions used in [11]. Figure 7 shows the main items that come to use.



(a) Behavior (b) Channel (c) Adapter

Figure 7: Graphical notation for model description.

A behavior (Figure 7(a)) contains the computation part of the application. It has an own flow of execution. The system's functional behavior is captured in an hierarchy of behaviors.

A channel (Figure 7(b)) captures communication facilities. It does not have an own flow of execution. The services provided by a channel are described by an interface definition. Two behaviors may communicate through a channel, by mapping a port to an interface of the channel.

An adapter (Figure 7(c)), also called half channel, implements an interface to be mapped to another channel. The adapter does not have an own flow of execution.

3.3 Transaction Level Model - MAC

The Transaction Level Model (TLM) is the most abstract model; it is expected to yield the fastest simulation speed. This model implements only the media access layer, therefore it is sometimes referred as the MAC model. User data, regardless of its size, is transferred in one chunk as one user transaction. The bus access is checked only once for the whole user transaction. The fact that the user transaction would be split into many bus transactions is ignored in order to reach higher simulation speeds. The TLM is not wire accurate. The communication is performed on a more abstract level than pins and wires. The model is not cycle accurate in all cases.

Figure 8 shows the connection schema for two masters and two slaves for the TLM model. The bus is simulated by a single channel implementing the media access layer; all masters and slaves directly connect to it. There is no distinction made between the masters connected to the bus, hence no priority based access between the masters is observed. Instead concurrent access to the bus is avoided by use of a semaphore, hence the order of concurrency resolution relies on the simulation environment.

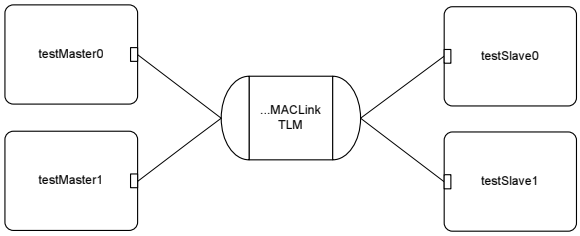


Figure 8: Transaction Level Model (MAC model) connection scheme

In the model implementation done for this work, the user data is transferred using a single *memcpy* between master and slave. The timing is simulated by a single *waitfor* statement covering the whole user transaction. The calculation of the wait time takes into account the way the transaction would be split into bus transaction. A high simulation speed is expected due to the fixed low number of operations per user transaction.

Two variances of this model were defined for evaluation purposes. The TLM variance A (TLM (a)) performs as described, concurrent access is sequentialized by the use of a semaphore. The TLM variance B on the other hand does not prohibit concurrent access. As a result two masters may access the bus at the same simulated time. One of the two variances will be selected during the evaluation process.

3.4 Arbitrated Transaction Level Model - Protocol

The Arbitrated Transaction Level Model (ATLM) simulates the bus access in the granularity of bus transactions, at the level of the protocol layer[1]. It is the first to perform arbitration, which is done as well at the level of bus transactions. To compose the ATLM, the medium access layer implementation is reused from the later described bus functional model. The medium access layer slices a user transaction into individual bus transactions, which are then transferred using the protocol layer implementation for this model.

Figure 9 shows the symbolic bus scheme. A hardware abstraction layer is created around each application behavior. The channel for the media access layer in inlined into the hardware abstraction layer and the application behavior is connected to this channel. The bus is simulated by the channel implementing the protocol layer. The slaves are directly connected to this channel. The masters on the other hand are connected through individual half channels (MasterProtocolTLM), which are required for defining the master's identity. The identity is necessary for accurately simulating arbitration. The scheme 'identity through connectivity' was chosen for modeling of the master's identity, since it closely resembles the physical implementation, where the master's identity is defined by its connection to the arbiter.

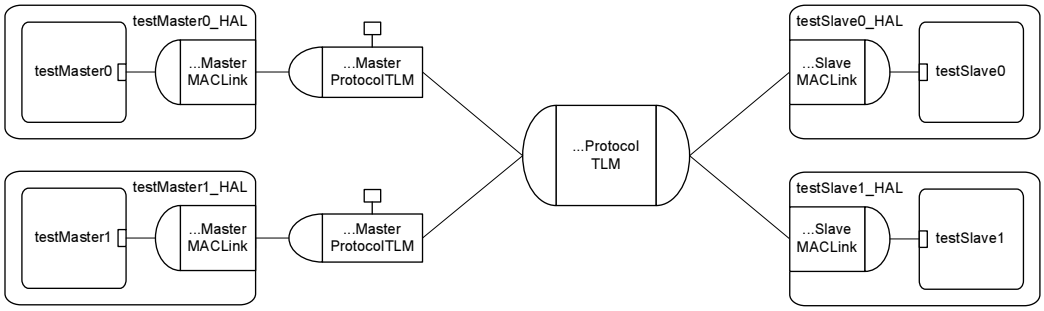


Figure 9: Arbitrated Transaction Level Model (protocol model) connection scheme.

Since the ATLM implements the protocol layer as the lowest layer, it has to provide arbitration capabilities. With the previously described identity of each master, an accurate arbitration can be provided. The AHB definition does not require a specific arbitration scheme, so a priority based arbitration was implemented. In this model arbitration is performed on the granularity of a bus transaction. The arbitration scheme was implemented without an additional context switch (in addition to the executing masters), in order to ensure fast execution speed.

The ATLM with its arbitration per bus transaction is expected to be accurate already in case of locked transfers. In such transfers, a granted master may not be preempted during bus transaction, not even by a higher priority master. Hence all arbitration decision are done on a bus transaction boundary. However for unlocked transfers an inaccuracy is expected, here the bus owner ship may change even within a bus transaction (i.e. when a burst of a low priority master gets preempted by a high priority master).

[1]Outside of this work the Arbitrated Transaction Level Model may also be referred to as the protocol model. It may be even understood as a Transaction Level Model since the TLM carries only a broad definition.

As with the TLM, two variances have been created for the ATLM. The variances differ in the accuracy of the arbitration. The first variant of the ATLM, the ATLM (a), follows the concept of a delta cycle as it is used in hardware simulators. During a simulation two masters may attempt an bus access at the same simulated time. However due to the serialized execution of the simulation code, one master's code will be executed earlier. In order to handle this situation the ATLM (a) does first collect all bus requests during one delta cycle and then makes the decision based on the collected requests. The ATLM (b), on the other hand, does not collect the bus requests for a delta cycle; it makes the decision immediately at the arrival of the first request. As a result, in case that two masters request the bus within the same delta cycle, the master with the earlier executed simulation code will gain bus access regardless of the priority.

A lower execution speed over the TLM is expected for both variances of the ATLM. Each individual bus transaction is modeled in terms of timing and arbitration individually. In terms of execution speed, the ATLM is expected to outperform the bus functional model, which covers the bus in all detail.

3.5 Bus Functional Model - Physical

The bus functional model is a synthesizable model bus model that covers all timing and functional properties of the bus definition. Communication is performed at the level of pins and wires. It is a wire accurate and cycle accurate model of the bus.

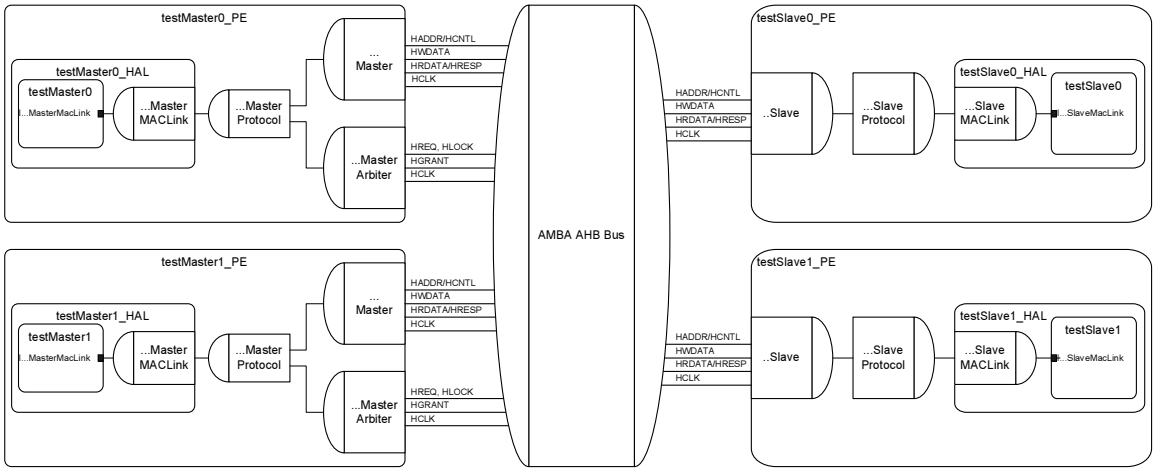


Figure 10: Bus functional model connection scheme.

Figure 10 shows how the application behaviors are wrapped for the bus functional access. As described for the ATLM, each application behavior is first wrapped in the hardware abstraction layer that inlines a half channel implementing the Media Access Control (MAC) layer. For the bus functional model each bus element is further wrapped into a processing element. The processing element inlines a channel instance that implements the protocol layer, where the MAC channel is connected to. Additionally a channel implementing the physical access is inlined. As a result each processing element is connected via wires to the actual bus.

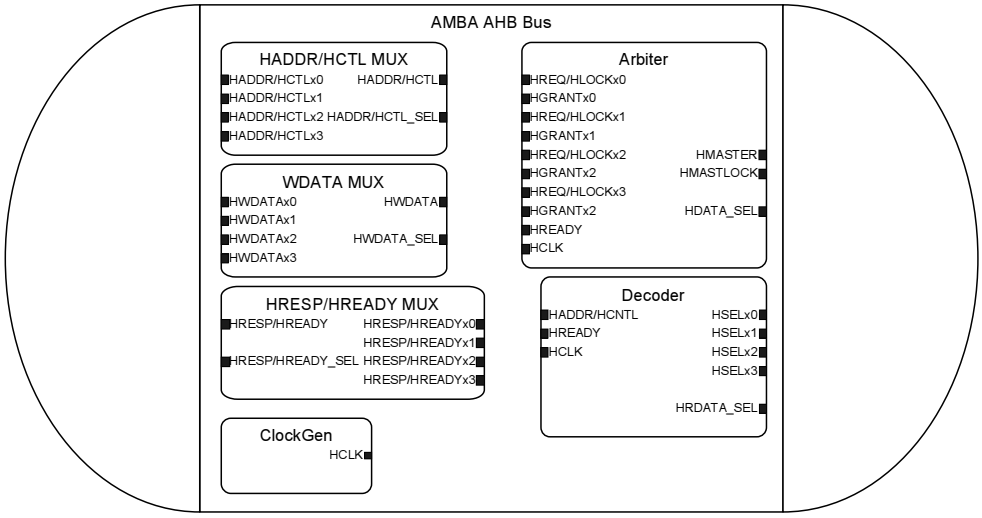


Figure 11: Content of the bus functional channel.

For ease of understanding, the bus in Figure 10 is graphed as a channel. However the bus consists of many individual elements as the Figure 11 shows. Since the AHB definition defines the bus access without tristate outputs, a set of multiplexers is required to select address, data and control signals from the active bus components. Additionally the bus functional implementation contains a clock generator, an arbiter and an address decoder. Please refer back to Figure 5 for an overview of the AHB interconnection scheme.

As it can be seen by the inlined channels, the bus functional model uses all described layers. Actual wires are used for the connection of the bus elements. The bus wires are driven and sampled according to the AMBA specification with the rising edge of the bus clock. The physical layer provides the access to the bus on a bus cycle basis. The services of the physical layer are used by the protocol layer, which implements arbitration and data transfer. The arbitration is done for each bus transaction, and for unlocked burst the bus grant state is verified additionally on each bus cycle. As in the ATLM the protocol model is invoked by the MAC layer, which slices the user transaction into bus transactions. Figure 12 and Figure 13 show an overview of the implemented channels for the master and slave side respectively.

3.6 Modes of Access

The utilized design environment SCE defines two distinct ways of accessing bus slaves, namely the memory style access and the rendezvous style access (also referred to as link style access). Both styles are depicted in Figure 14.

In a memory style access (Figure 14(a)), the slaves accessible memory is exposed to the bus over an address range. A master may access the provided address range at any point in time. This access style is applicable for memory and for memory mapped IO. This style of access allows burst accesses for improved performance. The abstract notation in Figure 14(a) indicates the memory as

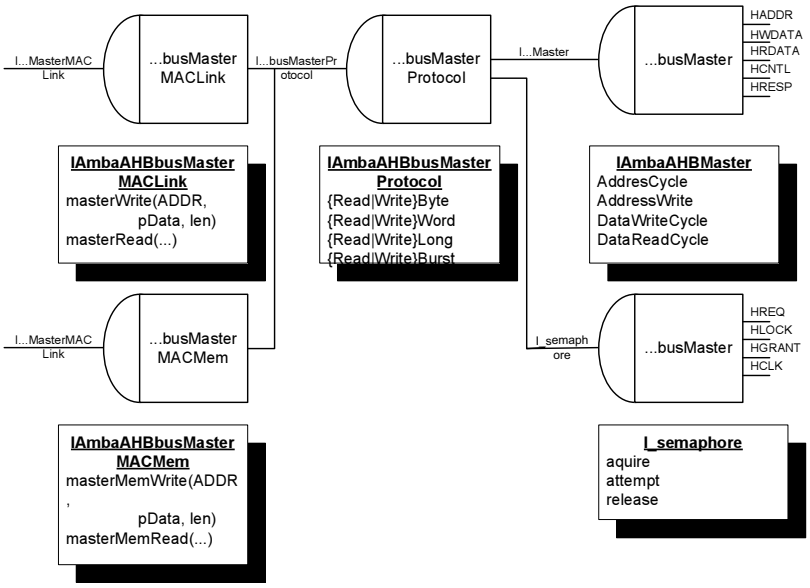


Figure 12: Channels for master bus functional model.

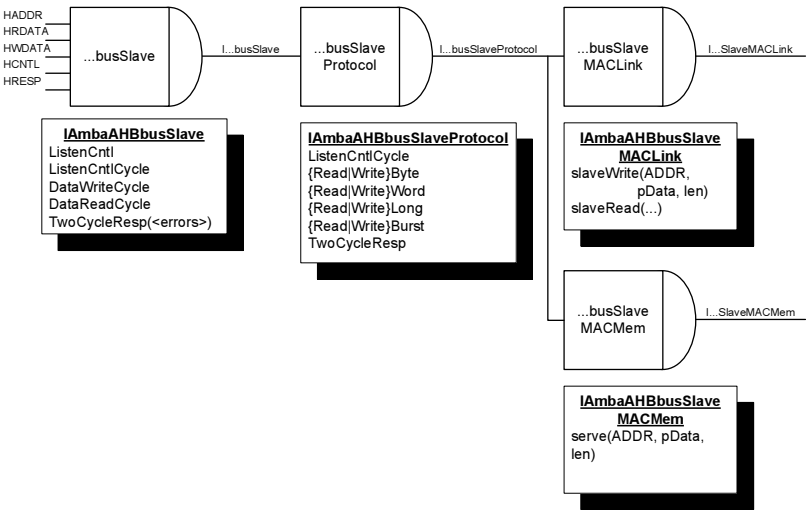


Figure 13: Channels for slave bus functional model.

(a) Memory Style (b) Randevouz Style

Figure 14: Modes of access

a half channel, which was made to show that the memory has no own flow of execution.

The rendezvous type access (Figure 14(b)), simulates a message passing interface. The slave only exposes a single address to the bus for each rendezvous type access. The content of a user transaction is written one-by-one to the same base address. With that a mailbox is simulated on the slave side. This is especially useful if the address space is limited, since the message length does does not influence the required address space. In a rendezvous style access a slave waits for an access on a particular address and further reacts to the request. Application level synchronization is needed for this model, since the access patterns have to be known on the slave side. The depiction of the rendezvous style access (Figure 14(b)), presents the slave (HW PE) as an own PE, thus it is shown to have an own flow of execution.

Since the rendezvous type access simulates a message passing interface, all words within a message are written to the same address. Due to this addressing pattern bursts can not be used, since the AHB specification requires to increase the address for each beat within a burst. Hence a user transaction in the rendezvous style access is transferred only with individual non sequential transfers.

In order to support both styles of access, two channel implementations of the MAC layer are provided. One channel per access type, the simulation environment generates code, that instantiates both channels and uses the appropriate channel for a particular transfer.

4 Validation

The previous chapters have presented the design and implementation of the AMBA AHB bus. In this chapter covers the validation results. Three aspects will be described in more detail. First, the functional validation is described in Section 4.1. Those tests aim to assert the correct functionality ignoring timing constraints. Following that, Section 4.2 describes the validation of the timing accuracy of the bus functional model. Finally, Section 4.3 will deal with the timing correctness of the abstract models, the ATLM and TLM. Throughout this chapter no differentiation is made between the two variations of each of the abstract models. Thus, using the generic model name refers to both variations.

4.1 Functional Validation

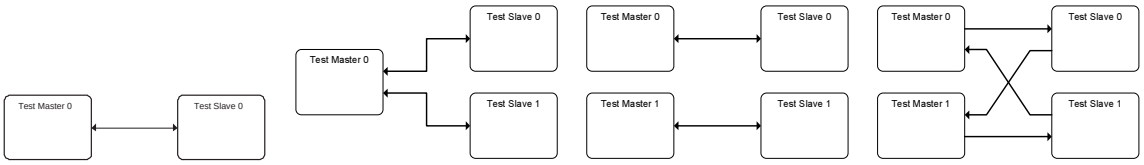
In an early part of the validation, the functional correctness of each AMBA AHB bus model is validated. Following a bottom up approach, a first set of tests will focus on individual bus transactions. Later more complex access patterns and corner cases are verified with the randomized tests utilizing the memory style MAC layer and the rendezvous style MAC layer.

4.1.1 Validation of Individual Bus Transfers – Fundamental Tests

The goal of the fundamental tests validating individual bus transfers is to ensure correct functionality of the bus primitives. The test provides the foundation for the construction of more complex tests. The following sequence of test was performed using the memory style MAC layer of each implemented model:

- Single Master Single Slave validates that each basic bus transaction yields the correct results. It validated read and write functionality for Byte, Word (16Bit), Long, fixed length burst (for 4, 8, and 16 beats).
- Single Master Dual Slave validates the connectivity and selection of multiple slaves addressed by a single master.
- Dual Master Single Slave introduces testing of the arbitration and validates that the bus is accessed exclusively by a single master as a result of arbitration.
- Dual Master Dual Slave validates the functional independent access to the bus for two master/slave pairs.

Figure 15 shows the logical connection scheme for each of the test groups. A range of predefined data was transferred to/from a set of predefined addresses for each individual test within a test group. A test was concluded successful if all data arrived correctly, in the predefined order, at the predefined addresses. Additionally *assert* statements have been manually introduced at critical places into the channel implementations, to detect invalid states within a channel. The results of the validation are shown in Table 3. All tests for all test groups have successfully passed for each implemented model. Hence a correct functional behavior is expected from each model.



(a) Single master single slave (b) Single master dual slave (c) Dual master dual slave parallel (d) Dual master dual slave interleaved

Figure 15: Logical connection for individual bus transfer validation.

Logical Connection under Test	Bus Functional Model	Arbitrated Transaction Level Model	Transaction Level Model
Single master single slave, Fig 15(a)	passed	passed	passed
Single master multi slave, Fig 15(b)	passed	passed	passed
Multi master multi slave (parallel), Fig 15(c)	passed	passed	passed
Multi master multi slave (interleaved), Fig 15(d)	passed	passed	passed

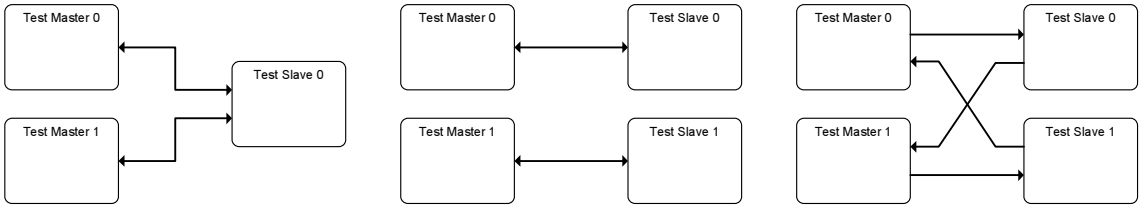
Table 3: Results of individual bus transfer validation

4.1.2 Validation of the Memory Interface

After having successfully validated individual bus transactions, now complex access patterns consisting of multiple bus transactions will be validated. This validation uses random access patterns, which statistically cover all access scenarios in accessing the components if executed long enough. The focus for this validation is the random interaction between two masters that access the same bus.

Two masters and two slaves are implemented for this test. The access is performed using the random access type. The memory exposed by the slaves present separate address regions for writing and reading. The following parameters are randomized for each transaction: read/write, the size of the transaction, the offset within the memory and the delay between transactions. The random selection algorithm ensures that each byte of the slave's memory is accessed exactly once during the test. Throughout the test the base address and the length of the user transaction, to be transfered, will vary. The way the MAC layer breaks down a user transaction into one or more bus transactions depends on exactly these two parameters. As a result the sequence of bus transactions per user transaction will vary throughout the test. This diversity is a good test for the slicing functionality of the MAC layer. The delay between the operations results in a random access pattern between the masters. This will test the arbitration implementation and validate the exclusive access to the bus in scenarios like concurrent bus request, back to back transmission, and handover between a high priority master and a low priority master. The correctness of each user transaction is validated directly after executing the user transaction; the master and slave memory area is compared for equality. Furthermore, after completing all user transactions, the complete memory area of master and slave are compared for equality as well.

In comparison to the earlier fundamental tests, not all of its configurations had to be retested. The utilized connection schemes are displayed in Figure 16. For a successful validation of a single connection scheme and bus model, two masters have to transfer 128KBytes each, using a random set of user transactions of up to 100 bytes each. The test has to fulfill the criteria in the previous paragraph and sustain the results for 1000 test repetitions. With an average user transaction size of 50 bytes, each bus model and connection scheme was validated with more than 2.5 million user transactions. Table 4 indicates the results of this test scenario, and shows that the test execution was successful for all configurations and all bus models.



(a) Dual master single slave (b) Dual master dual slave parallel (c) Dual master dual slave interleaved

Figure 16: User level logical connection for memory and rendezvous type access validation.

Logical Connection under Test	Bus Functional Model	Arbitrated Transaction Level Model	Transaction Level Model
Multi master single slave, Fig. 16(a)	passed	passed	passed
Multi master multi slave (parallel), Fig. 16(b)	passed	passed	passed
Multi master multi slave (interleaved), Fig. 16(c)	passed	passed	passed

Table 4: Results of validation for memory access

4.1.3 Validation of the Rendezvous Interface

In addition to the randomized test using the memory access style MAC layer, the rendezvous style MAC layer has to be verified as well. The two implementations differ in the way they slice the data. Here again random accesses have been utilized, varying the following parameters: read/write, size, offset, delay between accesses. In difference to the previous validation, only the independent access of two master slave pairs was tested (Fig. 16(b)). The other two connection schemes (Multi Master Single Slave and Multi Master Multi Slave (interleaved)) were not tested, since they are not applicable in the used simulation environment.

For the rendezvous style access, the simulation environment makes the assumption, that each access is predictable. As a result of the assumption, the slave code has to be implemented so that a particular user transaction is expected. Now, if two masters simultaneously request access to different portions of the slave's memory, the slave has to predict which user transaction is executed first. Since this depends on the arbitration, it is declared undecidable for a slave. In such situations, the memory style access should be used, hence the configurations are not applicable for this test.

Limiting the validated configurations does not limit the generality. The two access styles for the MAC layer differ in how a user transaction is sliced into bus transactions. This feature can be validated in any connection scheme. On the other hand the connection schemes differ in the way they create contention. The contention however is handled by the lower layers, which already have been successfully tested during earlier tests.

Table 5 summarizes the performed functional validations with the same set of connection schemes as before (Figure 16). The same execution criteria as for the memory interface validation were used here. Thus more than 2.5 million user transactions had to be transferred correctly for a successful validation of one bus model and connection scheme. The table shows successful

test execution for the tested configuration for the three abstraction levels: bus functional, arbitrated transaction level modeling and transaction level modeling.

Logical Connection under Test	Bus Functional Model	Arbitrated Transaction Level Model	Transaction Level Model
Multi master single slave, Fig. 16(a)	N/A	N/A	N/A
Multi master multi slave (parallel), Fig. 16(b)	passed	passed	passed
Multi master multi slave (interleaved), Fig. 16(c)	N/A	N/A	N/A

Table 5: Results of functional verification of rendezvous access

4.2 Timing Validation of the Bus Functional Model

Considering the results of the previous section, a correct functional behavior of all implemented models can be expected. Additionally important is a timing validation, which deals with the correct behavior of each signal in the temporal sense. This is particularly important for the synthesizable bus functional model, as a prerequisite for interoperability with other intellectual property components.

A validation of the timing behavior requires an independent reference. Since a physical implementation of the modeled bus structure was not available in the lab at the point of writing, the timing behavior of the model was compared against the specifications. The following sections will show the comparison of the implemented bus functional model against transfer scenarios selected from two sources: the AMBA specification [3] and the AMBA AHB Cycle Level Interface [1], which is an interpretation of the AMBA specification.

The selected scenarios have been be recreated with the implemented bus functional model, which in this setup simulates a bus with 50MHz bus clock. Additional probes have been inserted into the test bench for tracing of all important bus wires. The traces are displayed as waveforms, which have been generated using *gtkwave* (see [5]).

4.2.1 Basic Pipelined Bus Access

As described in Section 2, the AHB allows a pipelined access to the bus. The basic stages of the pipelined bus access are validated in the first pair of waveforms.

Figure 17 shows the reference waveform and Figure 18 displays the results of the actual implementation. As a general note, the specification [3] requires signals to be valid at the rising edge of HCLK, at this point the signals are sampled from participating bus elements (which are all implemented as sequential logic, see [4, question #4120]). The implemented model does not cover subcycle events, therefore each signal is applied immediately after the rising clock edge. Hence there will be an acceptable subcycle difference between the reference and the implemented model.

The following three points within the displayed transfer are of interest for deciding the timing correctness of the implementation:

1. In bus cycle T1, the master requests bus access. Within an arbitrary number of bus cycles (at least one) the arbiter grants access to the bus. In the particular reference waveform, the arbiter

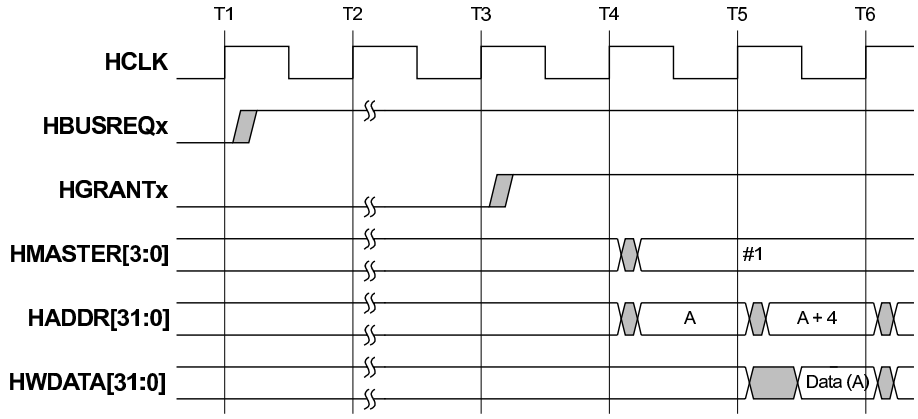


Figure 17: Reference sequence from [3] showing pipelined behavior

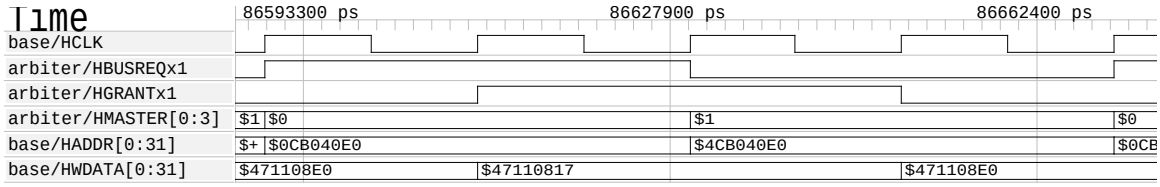


Figure 18: Waveform of implemented bus model, showing pipelined behavior

grants the access in T3. In the waveform of the implemented model, the bus is requested in the first clock cycle and granted in the second. Again, granting the bus within a single cycle is valid, an example of a one-cycle-grant can be found in the reference waveform in Figure 21.

2. In the bus cycle after granting the bus[2], the granted master applies the address and control signals to the bus. This happens in the reference in T4 and in the actual implementation in the third bus cycle, which is in both cases the cycle after the bus grant.

3. The data is written in the bus cycle after applying address and control information. The reference waveform shows this in T5, the actual implementation shows it in the fourth cycle. In both cases it happens in the cycle directly following the address and control signals. As it will be seen in later waveforms, the pipelined access allows concurrently applying the data for one cycle and the address and control lines for the next cycle.

4.2.2 Error Response

The previous subsection has shown that the basic pipeline stages are observed by the implemented model. This behavior was shown under the assumption that the selected slave always signals to proceed with the current transfer. In this subsection this restriction will be removed.

[2] A simplifying assumption is made for this subsection: the currently selected slave signals to proceed with the transfer, which is done by asserting HRESP == OKAY, and HREADY == HIGH.

The AHB standard defines that a slave has to reply back to the master for each bus operation. This reply indicates the success of the bus operation and is done on every bus cycle. Multiple slaves may be selected in different phases of the transfer due to the pipelined access nature of the AHB. However, only the selected slave that is in the data phase asserts the reply information. The reply information is provided by the following two signals:

HREADY is used by the slave to extend the the data portion of an AHB transfer. The slave inserts a wait state in the bus access by asserting LOW to HREADY. A transfer is finished regardless of the success once HREADY is HIGH.

HRESP is asserted by the slave and indicates the status of the current transfer. Possible values are OKAY, ERROR, SPLIT and RETRY. OKAY indicates a successful completion of the bus operation. The latter three result codes indicate additional handling for this operation and they require a two-cycle response. With a two-cycle response the pipeline of the bus access is flushed.

Figure 19 shows how a slave indicates a failed transfer. By setting HREADY to low, the slave inserts one additional wait state to make the decision about the transfer. The following timing points are of interest in order to validate implemented model as shown in Figure 20:

1. In the bus cycle following the address phase, the slave asserts HREADY to LOW and inserts a wait state. This happens in the second cycle in the reference and in the third cycle of the implementation waveform.

2. The slave has made the decision of failing the bus transfer in the third cycle of the reference waveform. At that point it starts the first cycle of the two-cycle error response. The slave applies the value of ERROR to HRESP. This happens in both waveforms in the cycle after the first wait state.

3. In the second cycle of the two-cycle error response the slave still applies ERROR to HRESP. In order to finish the bus transaction HREADY is set to HIGH. This behavior can be observed in both waveforms in the second cycle after the first wait state.

4.2.3 Unlocked Burst Handover

The previous timing validations were concerned with a single master. The scenarios in the following subsections will deal with the handover between two masters on the same bus. This subsections scenario describes the handover between unlocked burst transfers of two masters. In an unlocked transfer the granted master may lose bus grant during the transfer, if a higher priory master requested the bus.

In the scenario presented here, a high priority master performs a unlocked burst during which a low priority master requests the bus. Therefore the high priority master finishes the ongoing burst and the low priority master reaches the bus grant after that. This type of bus handover is most efficient, because it allows a single-cycle master change and the bus can be 100 % utilized. In the

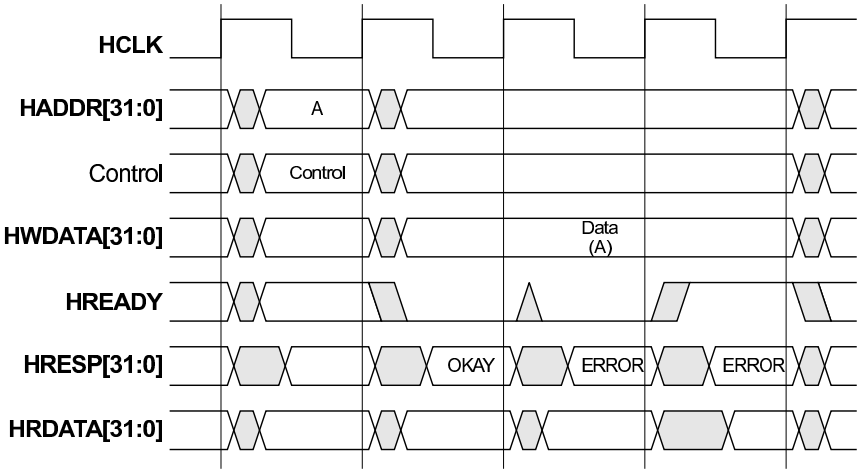


Figure 19: Reference sequence from [3] showing an error response

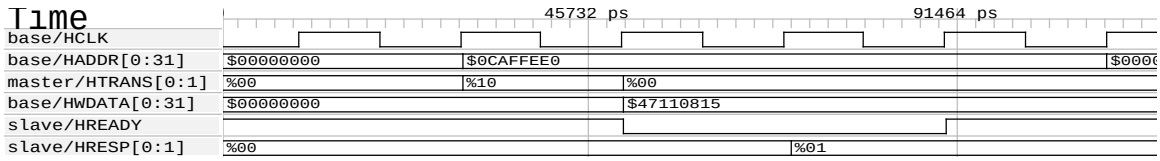


Figure 20: Waveform of implemented bus model, showing error response

presented scenario, however, the slave addressed in the ongoing burst of the high priority master inserts two wait cycles in the last burst cycle, which slows down the transfer.

The timing correctness of the waveform from the implemented model as shown in Figure 22 with respect to the reference waveform in Figure 21 can be checked by the following aspects:

1. After getting the bus granted (cycle T2 in the reference) the high priority master lowers the request line HREQ. The specification leaves it open, when exactly HREQ is lowered. The reference waveform shows that the granted master lowers HREQ directly after getting granted. In the implemented model however this happens one cycle later. This was done to ensure that the arbiter has sampled the control signals before lowering HREQ. After sampling the control signals the arbiter can predict the length of the current transfer.

2. The specification (see [3, section 3.6]) requires that in the first control cycle of a burst transfer HTRANS is set the NONSEQ, which can be seen in both waveforms (T3 of the reference waveform, third cycle of the implemented model).

3. The arbiter lowers the HGRANT signal of the granted master in the last control cycle of a burst, at the same time it may grant the bus to another master. The previously granted master still owns both the data and the address/control bus for the current cycle, and the data bus only for the next cycle. The change in HGRANT lines appears in both waveforms in the sixth cycle.

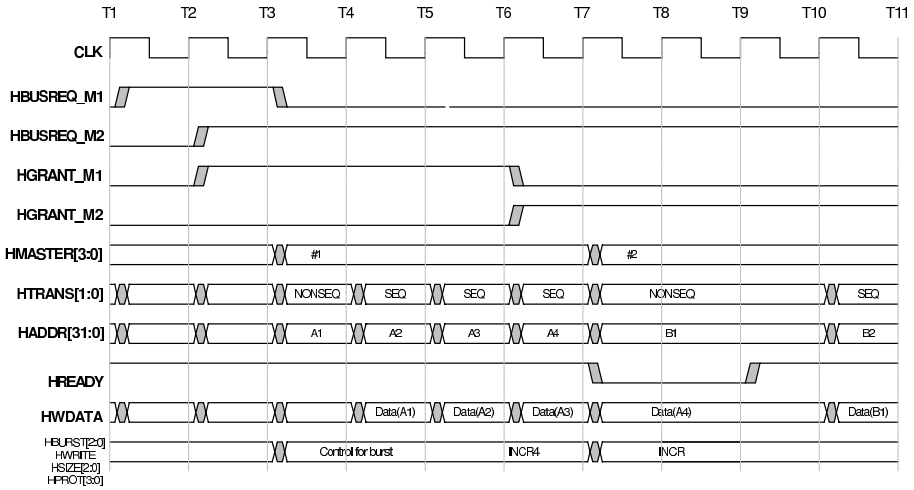


Figure 21: Reference sequence from [1] showing unlocked burst handover

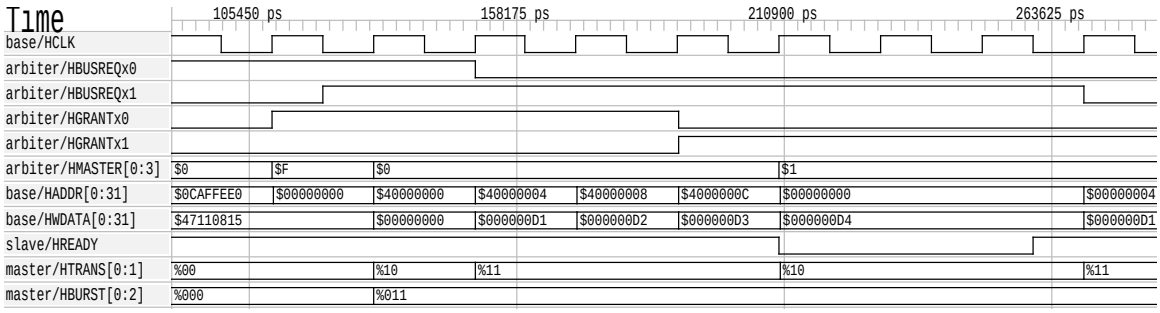


Figure 22: Waveform of implemented bus model, unlocked burst handover

4. In the cycle following the change in the HGRANT signals the selected master is updated. This can be seen by the HMASTER[3], which indicates the master owning the address/control bus. This change happens in both waveforms in cycle 7. At the same time the new granted master asserts the address/control lines for the new access. It has to be noted that there is no idle cycle between the end of the old burst and the beginning of the new burst, the bus is 100% utilized in this case. Also this fact can be seen both in the reference and in the implementation waveform.

5. As described earlier, the selected slave can insert wait states by lowering the HREADY signal. In this particular scenario the slave inserts two wait states in the last data cycle of the burst transfer. As a result, the address and control signals that have just been applied have to remain on the bus. Also the data on the write data bus has to remain constant in case of a bus write. In this case the master, which just got the bus grant, has to keep the applied address/control signals on the bus. The previous granted master has to keep the applied data word on the data write bus (HWDATA). In both waveforms the wait states take place in the cycles 7 and 8.

[3]The signal HASTER is driven by the arbiter and used by the selected slave for a split transfer, see [3, section 3.12].

6. In the cycle after HREADY is set to high, the pipelined bus access is resumed. As one indication the just granted lower priority master applies the second address of the just started burst. This appears in cycle 10 in both waveforms.

4.2.4 Locked Burst Handover

This scenario shows a locked burst handover. In a locked burst, a master may not be preempted during the transfer, even if a higher priority master requests the bus. Locked bursts are not as efficient as unlocked bursts, because one additional cycle is spend for the handover between masters. This additional cycle stems from the fact that the standard requires to apply HLOCK, the indication for a locked transfer, until the address phase of the last transfer. Figure 23 shows the reference waveform; the waveform of the implemented model is shown in Figure 24. For determining the correctness of the timing the following points should be evaluated:

1. For a locked transfer the standard [3, section 3.11.5] requires that the HREQ and HLOCK are asserted until the last address/control cycle of the burst. In both waveforms this change appears in the sixth cycle.

2. Since the arbiter samples the incoming signals on the rising edge of HCLK it takes until the next cycle to update the grant lines, which happens in cycle 7. As a result, the old granted master remains granted for one more bus cycle, although it has finished the burst transfer and has not requested the bus. Therefore the still granted master has to indicate that no actual transfer is performed in this bus cycle by setting HTRANS to IDLE. Both waveforms show this behavior in cycle 7.

3. As a particularity to this scenario the old selected slave inserts two additional wait states in the last cycle of the burst (cycle 7 and 8). The has the effect that the old granted master has to keep applying the data up to cycle 9.

4. Because of the wait states, the ownership of the bus does not change (even though the grant lines have changed in cycle 7) until the slave indicates it is done with the current bus operation in the data phase. Hence HMASTER and the address/control lines are not updated until the cycle after HREADY is raised. In both waveforms the slave sets HREADY to high in cycle 9 and the newly granted master gets the bus in cycle 10. Both waveforms show that in cycle 10 the arbiter updates HMASTER, and the newly granted master applies the address/control lines.

4.2.5 Locked Burst Handover with Master Busy

Following the specification [3, section 3.5] a master can, similar to a slave, insert wait states into a transfer. For doing so it inserts an HTRAN == BUSY cycle in the middle of a burst. This indicates that the master currently cannot perform the part of the bus operation. The slave has to respond with single cycle OKAY and otherwise ignore the transfer. The master has to keep the address/control lines constant during the BUSY cycle.

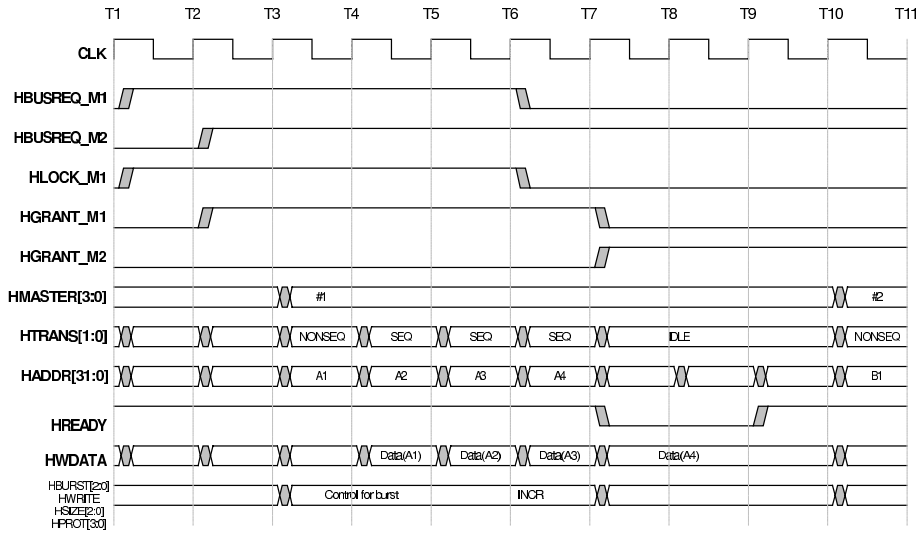


Figure 23: Reference sequence, source [1], locked burst handover.

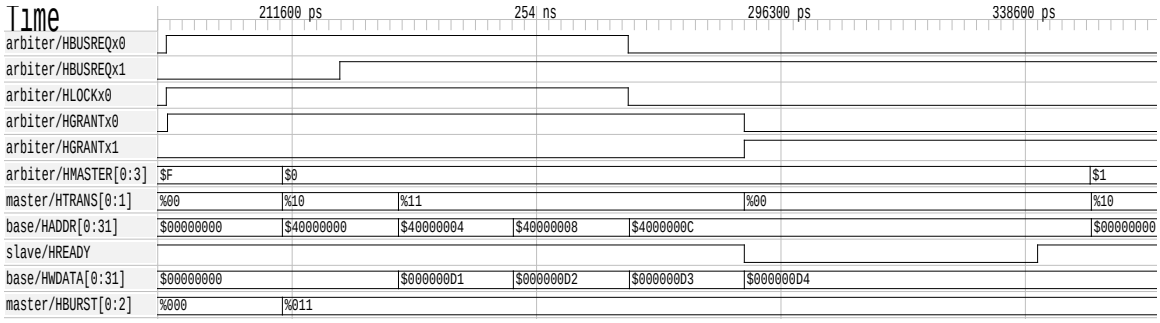


Figure 24: Waveform of implemented bus model, locked burst handover

The master in the scenario shown in the reference waveform Figure 25 inserts a BUSY cycle in the second address/control cycle of a locked 4 beat burst, and does so again in the last address/-control cycle of the same burst. The correctness of the implemented model (see the waveform Figure 26) can be assessed using the following points:

1. In both waveforms the granted master inserts a BUSY cycle in the bus cycle 4. As a result the master keeps the address/control lines constant over cycle 4 and 5. Note that the content of the data bus in cycle 5 is not defined.

2. In cycle 7 the master decides not to use the last cycle of the burst. It applies HTRANS == BUSY and lowers HREQUEST. As a result, the arbiter changes bus grant in cycle 8, which becomes effective in cycle 9. As in previous locked bursts the old granted master sets HTRANS == IDLE during the, now ignored, data phase of the last burst cycle (cycle 8). The new granted master owns the address/control bus in the 9^{th} bus cycle. This behavior can be seen in both waveforms.

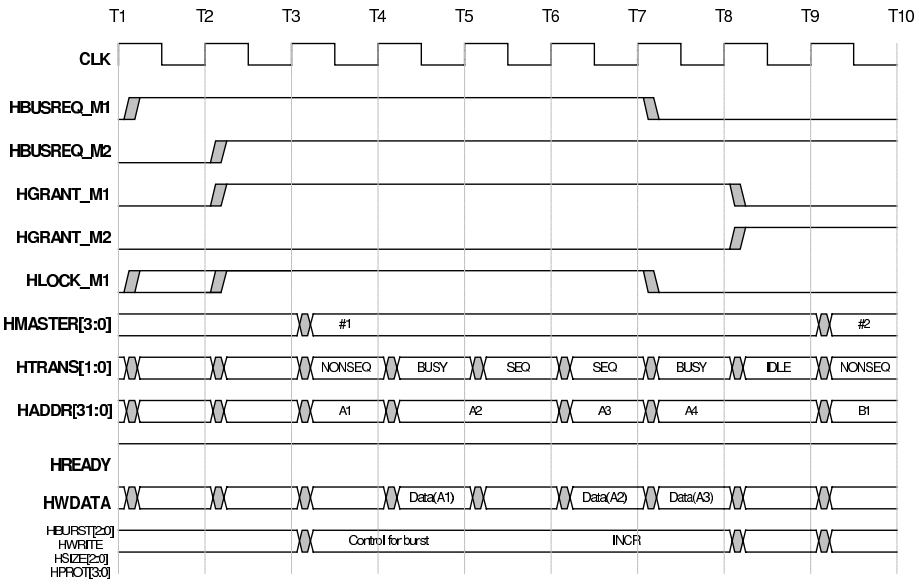


Figure 25: Reference sequence from [1] shows a locked burst with the master inserting a busy cycle.

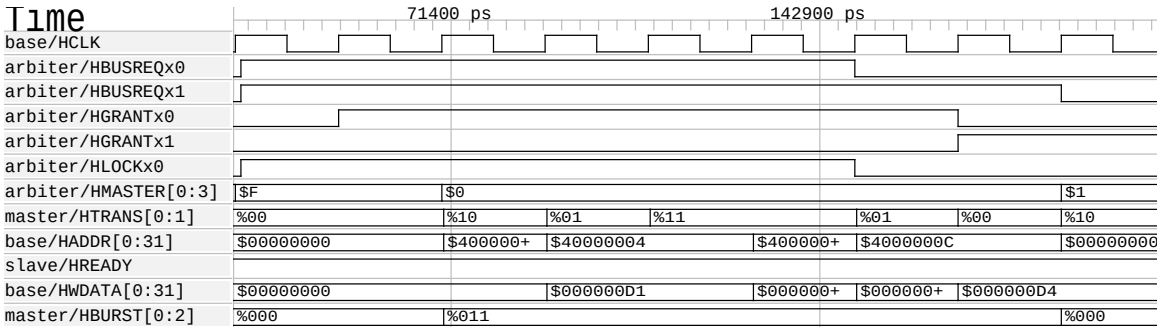


Figure 26: Waveform of implemented bus model, showing locked transfer with busy master.

4.2.6 Retry

In the following presented scenario, the slave indicates that it is not able to complete the current requested transaction. For that the slave replies with a response code of RETRY. This indicates to the master to abort the transaction and retry at a later time. The time after which the master may retry the operation is not specified. In the presented scenario the master attempts the retry immediately after the aborted bus transaction. In general the RETRY as well as the SPLIT operation allow the slave to finish the operation even though the slave is not able to supply the requested data. With that, excessive wait cycles can be avoided and the bus is available for other transactions.

Figure 27 shows the reference waveform and Figure 28 displays the results for the implemented model. The following points are of interest for comparing both waveforms [4]:

1. In bus cycle 6, while the master applies the data for the second burst beat, the slave inserts a wait state. In the following two cycles (7 and 8) the slave sets HRESP to RETRY signaling that the transfer cannot be completed right now and that the master has to retry.

2. As a result of the RETRY response in bus cycle 7, the arbiter removes the bus grant from the first master and grants the bus to the second master in bus cycle 8.

3. The first master reacts to the retry response and re-requests the bus in cycle 9. Meanwhile the second master performs a non-sequential single beat transfer.

4. During the data phase of the second master's individual transfer (it now applies HTRANS == IDLE), the arbiter changes the bus grant back to the first master in bus cycle 10.

5. The first master starts a retry of the previously aborted operation in bus cycle 11. Note that in the reference waveform, the retried transfer is performed in a burst, while it is done with individual transfers in the implemented model.

4.2.7 Preemption of an Unlocked Burst

The scenario presented in this subsection shows a preempted unlocked burst. A low priority master performs an unlocked burst. While this burst is in progress a higher priority master requests the bus, and consequently the lower priority master loses the bus grant. After the high priority master has finished its own transfer, the bus grant changes back and the low priority master may resume the interrupted transfer.

Figure 29 shows the reference waveform, the measured waveform of the implemented model is shown in Figure 30. In the implemented model master 0 is the higher priory master, and master 1 is the lower priority master. The reference waveform M2 is the high priory master and M1 has lower priority. As a difference from the previously shown waveforms of the implemented model,

[4] Note that both waveforms differ in the first three bus cycles. The reference waveform shows that the previously selected slave inserts a wait state in the last transfer. As a result the bus ownership for the first master is delayed by one cycle. Since the delayed bus handover was already tested in Section 4.2.3, the additional wait state was not inserted for a simpler test bench implementation.

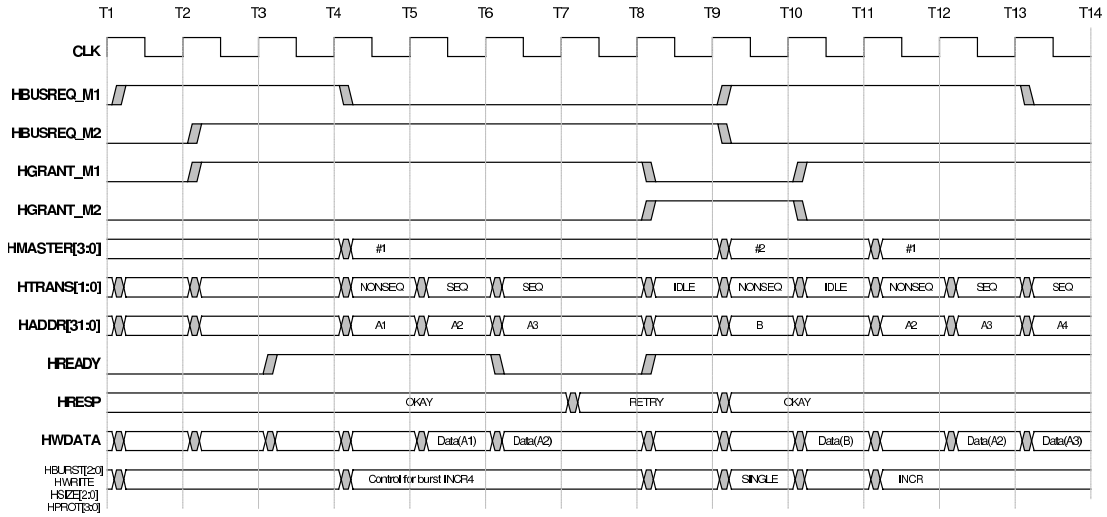


Figure 27: Reference sequence from [1] showing an aborted burst due to the slave sending a retry

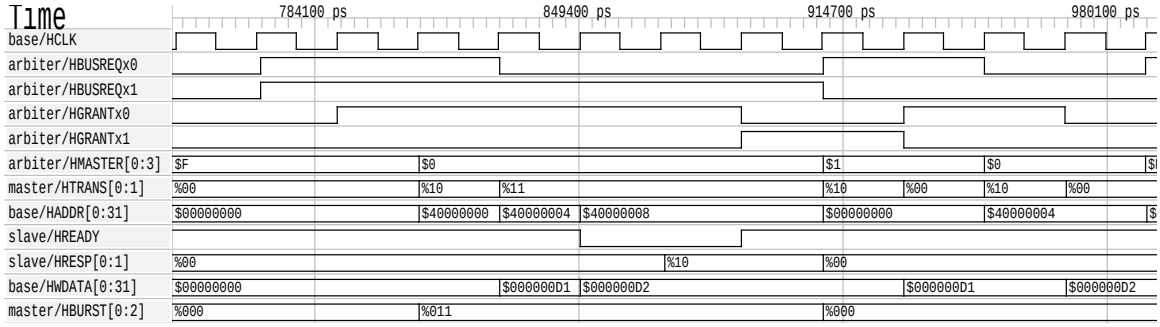


Figure 28: Waveform of implemented bus model, showing handling of a retry operation. Note that in this model the recovery from a retry happens with individual transfers, instead of an undefined length burst.

the request and grant lines are swapped between the two masters to match the reference waveform. The following points are of interest for comparing the waveforms:

1. In bus cycle 2 the low priority master gets the bus granted and subsequently starts an unlocked burst transfer. While the burst is in progress the high priority master requests the bus as well.

2. The arbiter observes the request of the higher priority master and changes the bus grant from the low priority master to the high priority master in bus cycle 5 (the third beat of the unlocked burst).

3. The now granted high priority master starts its transfer in bus cycle 6. At the same time the low priority master requests again the bus in order to complete the interrupted transfer. These two facts can be seen in both waveforms.

However the two waveforms disagree in cycle 6 for the HGRANT lines: while the reference waveform shows a change of bus grant from the high priority master to the low priority master in this cycle, the same change is delayed by one cycle in the implemented model. It is the author's understanding that the bus grant change is a result of lowering the request line of the high priority master, which happens at the very same time in the cycle 6. Such immediate change (without a rising HCLK) would require combinatorial logic in the arbiter. This however is not allowed according to ARM's FAQ [4, question #4120].

Following the argumentation presented in the FAQ the implemented model shows the grant change one cycle later, after the arbiter has sampled the input lines at the rising edge of HCLK. Although this difference exists between the two waveforms it has no bearing on the further timing, since the slave of the preempted burst inserts a wait state in the last beat (also cycle 6). This extends the transfer of the high priority master by one cycle and masks the difference in bus grant state between the two waveforms.

4. In bus cycle 8 the low priority master owns again the address bus and resumes the interrupted burst. The remaining last beat is performed as an individual transfer; its data is visible during the bus cycle 9.

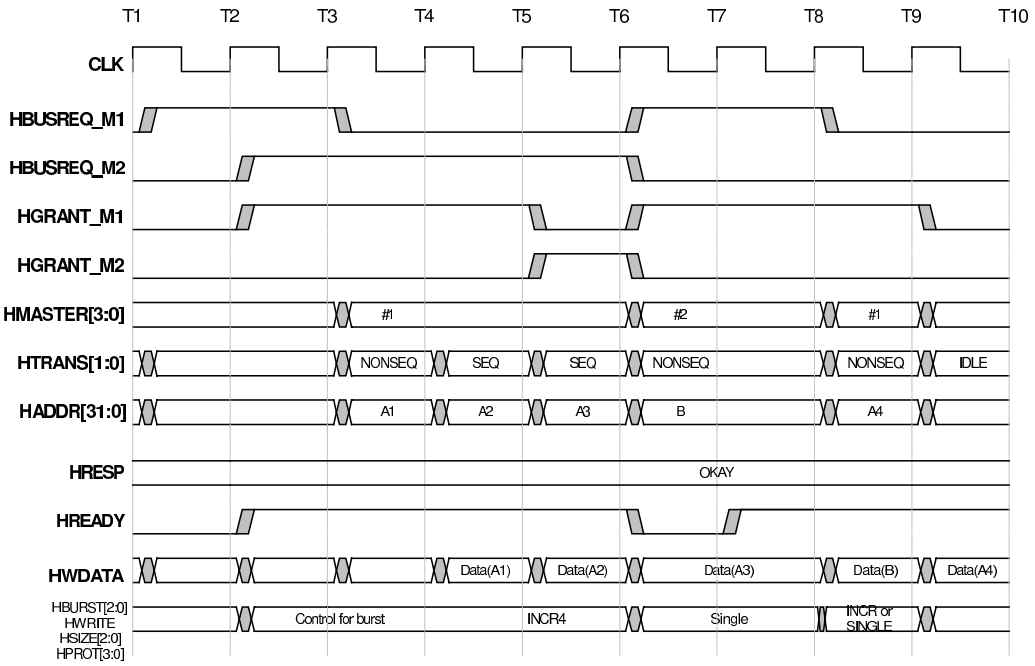


Figure 29: Reference sequence [1] showing loss of bus grant during burst

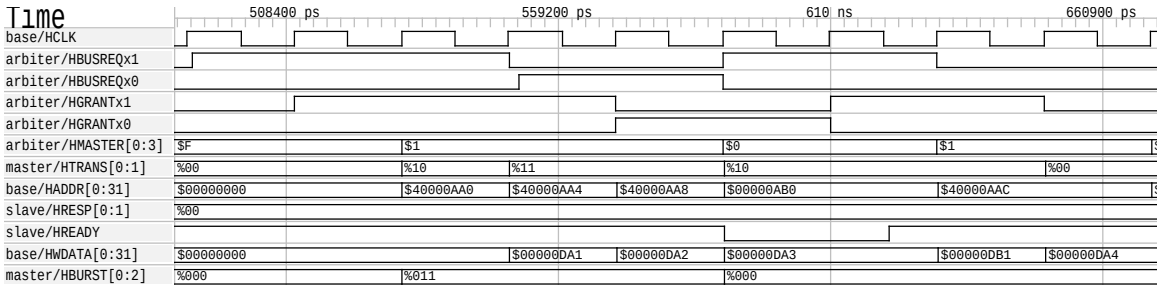


Figure 30: Waveform of implemented bus model, showing loss of the buss grant while in a burst. NOTE master 0 and master 1 are exchanged to match the reference waveform.

4.3 Timing Validation of the Transaction Level Models

The previous section has evaluated the timing accuracy of the bus functional model. Hence this model can now be used as a reference model for comparing the abstract models (the Transaction Level Model and the Arbitrated Transaction). This section will evaluate the timing accuracy of the abstract models in comparison to the bus functional model.

The more abstract models do not capture all properties of the modeled bus architecture. Therefore they do not have the same accuracy in all situations. It is however possible to define a set of restrictions, so that the properties, which are not implemented in the abstract models, are not exercised. When these restrictions apply, all models have to agree on the exact same timing. For this modeling of the AHB the restrictions are:

- Usage of only a single master and a single slave. As described in the design chapter each model handles the handover between masters at different level of accuracy. By eliminating such handovers the differences will not be perceivable.

- Utilization of locked burst transfer only; the more abstract models have been implemented making this assumption.

- A transfer initiating master should not be the default master, hence it has to request the bus for each access. Again the abstract models use this assumption for the timing, because they cannot distinguish between accessing from the default master and from a non-default master. The arbitration is not modeled this detail.

For comparing the models, some example transfers have to be defined. However given the flexibility of the AHB bus, it is not practical to exhaustively test all possible transfers. Therefore some representative examples with an increasing complexity have been chosen. The transfers are defined at a level of a user transaction. Such a transaction is broken down by the MAC layer into smaller bus transactions depending on the user transaction size and the start address. The later is important since the AHB standard makes the restriction, that all bursts have to be word aligned. As an example the MAC layer has to transfer first an alignment byte, if the user transfer starts at an odd address.

Table 6 lists the selected transfers, with the transfer size, the base address offset to the word boundary and an enumeration of the resulting bus transactions. All user transactions were performed with each implemented model and the number of cycles to complete each individual user transaction (which may be composed of multiple bus transactions) was measured. In addition to the measurements, the number of cycles for each transfer has also been manually calculated based on the standard [3]. The results can be found in Table 7.

Testcase	**Size**	**Offset to Word Alignment**	**Resulting Bus Transactions**
1	4	0	long
2	16	0	4 beat burst
3	17	3	alignment byte, 4 beat burst
4	50	0	8 beat burst, 4 beat burst, short
5	107	2	alignment short, 16 beat burst, 8 beat burst, 4 beat burst, byte

Table 6: Testcase definition for individual bus transfer timing validation.

Testcase	**AMBA Spec.** [specified cycles]	**Bus Functional Model** [measured cycles]	**Arbitrated Transaction Level Model** [measured cycles]	**Transaction Level Model** [measured cycles]
1	4	4	4	4
2	7	7	7	7
3	11	11	11	11
4	22	22	22	22
5	46	46	46	46

Table 7: Results of individual bus transfer timing validation in number of bus cycles.

As the Table 7 indicates all models take the exact same number of bus cycles for each of the test transfers. They also agree with the interpretation of the standard. Since the accuracy of the bus functional model has been shown already (see Section 4.2), it can be concluded that TLM and ATLM are 100% accurate for the given conditions.

An automated test has been implemented to extend the validation above to more user transactions. During this test, a random set of user transactions, varying in size and base address, is transferred and the transfer time for each individual user transaction is compared among the models. After transferring 100000 user transactions per bus model, this test has confirmed as well the timing equivalence of all models. Although the models are timing equivalent for the given set of restrictions, their accuracy differs if these restrictions are taken away. Section 5 contains a discussion of the accuracy results of the different models in such a case.

4.4 Validation Summary

In summary, all the functional and timing validations were successful. All implemented models have passed the functional validation. The timing accuracy of the bus functional model has been successfully shown with the use of example transfers from the AMBA standard. Furthermore, the timing accuracy of the abstracted models was successfully validated for a restricted setup.

5 Model Analysis

The previous chapter has asserted that the implemented models are functionally correct and that the bus functional model is implemented according to the AMBA specification. It was further shown that the more abstract models, the TLM and the ATLM yield correct timing under certain restrictions. With those results in mind, this chapter will explore how the implemented models affect the designer when modeling a system. For that two main aspects will be examined. First simulation performance will be evaluated, since the main premise of developing abstract models was to speed up simulation. This will show the benefit of each model. Secondly the accuracy of the more abstract models will be examined in a generic environment (outside of the restrictions posed in Section 4.3), which will explore the drawbacks for using the faster models. Combining both, the designer will be able to decide for the applicable trade-off between simulation speed and accuracy for a particular design stage.

5.1 Performance Analysis

The main goal of simulating with higher levels of abstraction is to increase the simulation speed. This section will give quantitative results about the performance and assert whether this goal was achieved in this implementation.

5.1.1 Test Setup

A test was devised, in order to measure the performance of each model, in which a single master is connected through the simulated bus to a single slave. No other masters or slaves are connected to the bus. A user transaction of a certain size is performed repeatedly a constant number of times without any delay in between, with that the simulation speed is limited only by the simulation environment. The simulation time for executing all repetitions of the user transaction was measured (simulation time can also be named as real time or wall clock time). The measured time for all repetitions was then divided by the number of repetitions, to yield the average execution time for a single user transaction. This process was repeated with a varying size of user transactions, to gain insight to the scalability of the implementation.

All tests have been performed on a Pentium 4, 2.8 GHz with HyperThreading under RedHat Enterprise 2 with the kernel version 2.4.21-20.ELsmp. The simulation environment was SCC version 2.2.0 (using QuickThreads).

5.1.2 Simulation Time

The performance measurements in terms of simulation time are shown in Figure 31. The y-axis denotes the execution time for an individual user transaction, the simulation time. The x-axis denotes the size of a user transaction in bytes. The start address for all user transactions remains constant on a word boundary, which avoids performance penalties due to alignment transfers.

As described in the design section, the two abstract models have been split into two variations each. The two ATLM variations differ in the way they handle the arbitration. ATLM (b) makes the arbitration decision immediately when a bus request arrives, whereas the ATLM (a) delays the decision for one delta cycle. As a result the second variation handles the case where two masters request the bus at the very same simulated time and grants the bus to the higher priority master. In contrary the variation (b) makes the grant decision immediately, hence the bus is granted the master with the earlier executed simulation code.

The both variations of the most abstract model, the Transaction Level Model, differ as well in the way arbitration is handled. The TLM (b) does not restrict the bus access. As a result two masters can access the bus concurrently at the same time. Hence each master performs as if it were the only master on the bus. The TLM (a) does limit concurrent access and simulates arbitration on the level of a user transaction. Once a master got the bus granted, it remains granted until the user transaction finishes regardless of other masters' requests to the same bus.

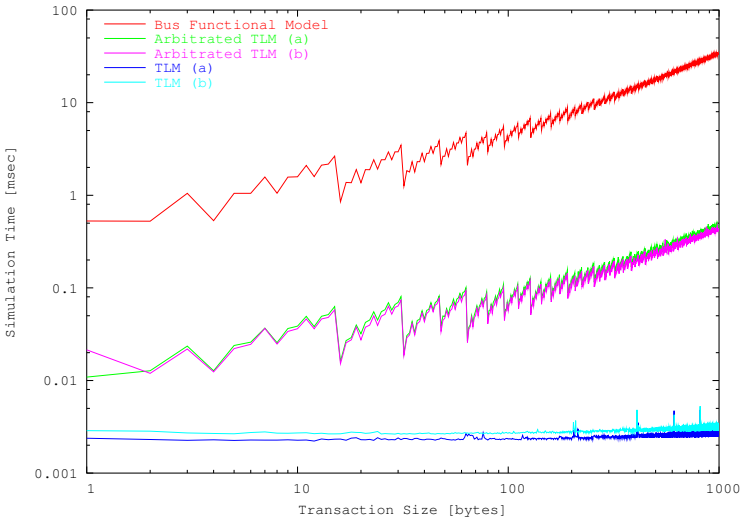


Figure 31: Execution time for completing a user transaction of varying size.

As Figure 31 indicates there is no significant difference between the variations of the two abstract models (ATLM and TLM). There is a significant difference between the major models; they are two orders of magnitude apart.

As expected, the most accurate model, the bus functional model, is the slowest in simulation speed. It requires 34ms for transferring 1000bytes. The Arbitrated Transaction Level Model model is faster. Here the 1000 byte transfer takes 0.48ms. The Transaction Level Model model executes

fastest, with mere 0.0025ms. It is noticeable that the simulation time for the TLM does not increase much with an increase in size. The simulation code is almost independent of the transaction size, only a single *memcpy* is used for the data transfer and the performance of the *memcpy* does not change much in the measured range. The performance graphs for the ATLM and the bus functional model are sawtooth shaped. A user transaction in this models is sliced into smaller bus transactions. However, the number of bus transactions does not increase linearly with the user transaction size, since fixed length bursts are used. To give an example: a user transaction of 3 words requires 3 individual bus transactions, a 4-word user transaction on the other hand can be transferred in a single burst of 4 beats, resulting in a single bus transaction.

5.1.3 Simulated Bandwidth

For a better understanding, the same measurements have been graphed again in Figure 32. Here the performance is expressed in simulated transfer bandwidth ($\frac{MBytes}{seconds\ simulation\ time}$). In terms of simulation performance the TLM reaches the highest bandwidth: 382MByte/sec, followed by the ATLM with 2MBytes/sec. The bus functional model only reaches a bandwidth of 0.028 MBytes/sec.

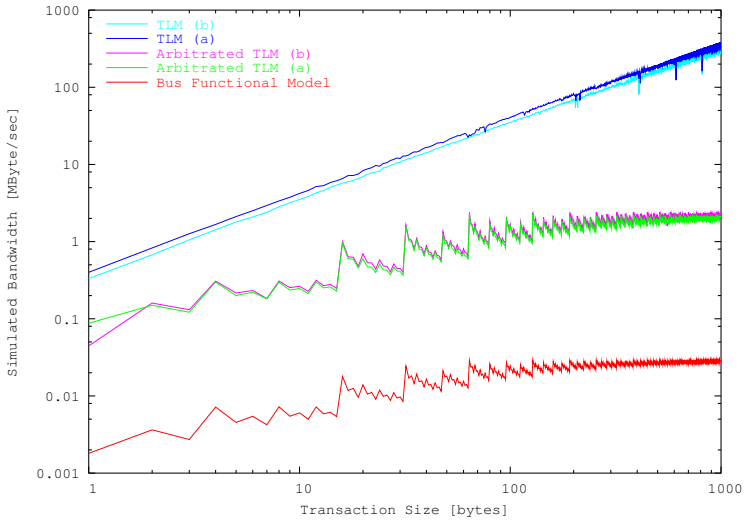


Figure 32: Simulation performance expressed as simulated transfer bandwidth.

The significant difference between the simulation models can be explained with their respective simulation detail. The TLM model handles data transfers on the level of a user transaction. Only C primitives are used for this communication. The ATLM model on the other hand does break a user transaction into many bus transactions, thus the effort is multiplied. Furthermore the ATLM model uses the standard implementation of the MAC layer, hence it has to adhere to the interface between the MAC and the protocol layer. Since this interface uses already bit vectors, the simulation requires more effort. The bus functional model is by far the slowest, since it simulates the transfer on a bus cycle level. Since this model is a synthesizable model all wires of the AHB are modeled and additional bus elements such as the multiplexers are covered as well (see Section 2).

In summary, the expected performance gains where actually achieved. With each more abstract model the simulation speed increases by two orders of magnitude. Judging from the performance results, no selection can be neither made for a variation of the ATLM nor a for variation of the TLM. The accuracy analysis that will be done in the next section is needed for a decision between the variations.

5.2 Accuracy Analysis

In the previous section, the gain of speedup by using models at higher level of abstraction was quantified. The drawbacks of abstract modeling in terms of accuracy limitations will be evaluated in this section. Unlike the performance measurements before, it is hard to define a single expressive number that allows comparing the accuracy of the different models. The actual accuracy depends too much on the environment and the actual application properties. Therefore, a generic test setup and procedure was defined that covers a range of application specifics, so that the designer can derive the expected accuracy for a particular setup. The next section describes the test setup and test methodology, followed by the presentation of the results.

5.2.1 Test Setup

For the test setup a generic scheme, with two masters and two slaves connected to the same bus, was used. Each master accesses one slave exclusively[5]. Figure 33 shows the logical connection scheme for the test setup.

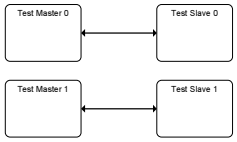


Figure 33: Logical connection scheme for accuracy tests

During the test, each master transfers a predefined set of user transactions. The user transactions vary in the base address, length of the transaction and in the delay between two transactions. The delay between transactions simulates local computation time. During the test execution, the start time and the duration (each in simulated time) of each individual user transaction is recorded for further analysis. The test is repeated once for each implemented bus model. Since the same set of user transactions is transferred by each of the models, their results are comparable.

It is expected that the accuracy changes significantly with the amount of concurrent bus access between two masters. As an example, all transfers may be executed without any concurrent bus access between the masters. That means, there is no overlap between any two user transactions. In such a case the timing for each master is as if it were the only master connected to the bus. Therefore the logical connection is as if each master had its own bus and the equivalence of the models, as asserted in Section 4.3 applies; then all models perform with 100% accuracy. However, with an

[5]Other configurations have been measured as well, but their results do not add additional insight. Hence, their presentation is omitted.

increasing amount of overlap between the user transactions, the masters will influence each other more in the bus access and the timing accuracy will differ with each model.

The amount of bus utilization, and subsequently the expected amount of transfer overlap, depend on the type of application. A communication-bound application will have a high bus utilization, a computation-bound application, on the other hand, results in a low bus utilization. In order to cover the range between those two extremes, the previously described tests have been repeated with a step-by-step increasing amount of overlap between transactions of two masters. By adjusting the average delay time between two user transactions of the same master, the bus utilization of each master was varied. As a result the overlap between the transfers of the two masters differs too. This overlap was measured during the experiment using the bus functional model. The overlap (in percent) has been defined as:

$$overlap = 100 * \frac{number\ of\ bus\ cycles\ with\ two\ active\ user\ transactions}{number\ of\ bus\ cycles\ with\ at\ least\ one\ active\ user\ transaction} \tag{1}$$

A user transaction is seen as active during the time the application is blocked executing a user transaction. Note that this definition is independent of the stage within the pipelined bus access.

In the following subsections, the achieved accuracy of the implemented models for locked transfers and unlocked transfers will be shown. The first subsection will also introduce the analysis methods when they are first used. The results for each master will be displayed separately; due to their difference in priority, the accuracy results may vary. As one effect of the prioritized bus grant, especially with higher amounts of overlap, the higher priority master may finish transferring the test sequence earlier than the lower priority master. After that point, the low priority master operates undisturbed on the bus, which will affect the accuracy measurements. In order to exclude this effect, only those measurements are taken into account, when both masters have not yet finished their test sequence, hence a chance of a concurrent access exists.

5.2.2 Accuracy of Locked Transfers

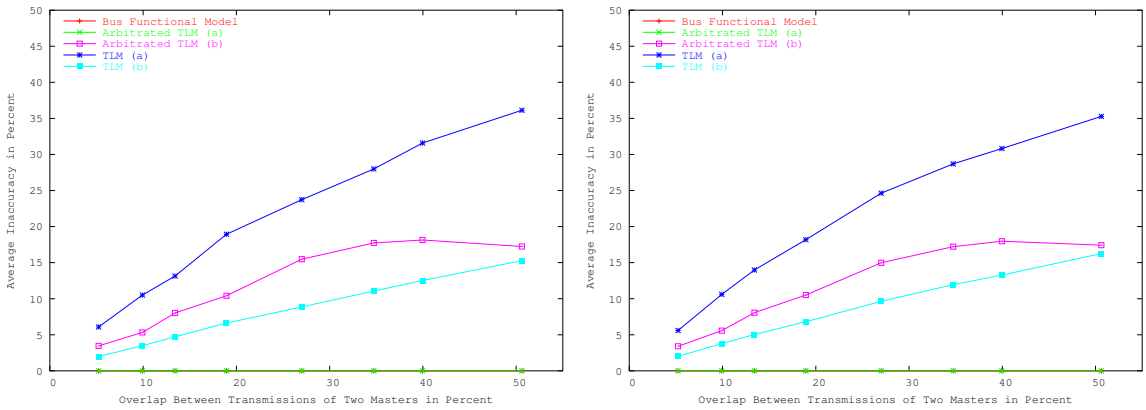
As previously described in the test setup, a test run yields an execution record of each individual user transaction. This subsection will describe how this data is analyzed for the locked transfers. As a reminder, a burst in a locked transfer cannot be interrupted, not even by a higher priority master.

The transfer duration of an individual user transaction is an important measure for predicting the application latency due to bus access. Therefore, in a first step, the accuracy of the models has been evaluated with respect to the transfer duration. For this purpose, the percentage inaccuracy of an individual user transaction is defined as:

$$\begin{aligned} duration_{bf}: &\quad \text{transfer duration in bus functional model} \\ duration_{test}: &\quad \text{transfer duration in model under test} \\ inaccuracy_i &= 100 * \frac{\left| duration_{test} - duration_{bf} \right|}{duration_{bf}} \end{aligned} \tag{2}$$

Given this inaccuracy definition, a timing accurate model exhibits 0% inaccuracy. It was avoided to directly express the accuracy in percent, since a particular model may have an inaccuracy of more

than 100% (i.e. the model under test predicts more than twice the simulated time), which would incorrecly lead to a negative accuracy. The average inaccuracy over all user transactions of a test sequence is displayed in the first set of graphs. Figure 34 shows the average inaccuracy of each model over an increasing amount of transfer overlap.



(a) High priority master

(b) Low priority master

Figure 34: Locked transfer accuracy based on duration

As Figure 34 indicates, the ATLM (a), which collects bus requests for one delta cycle before making a decision, exhibits no inaccuracy over the whole range of evaluated overlap. Since all transfers are executed locked, the arbitration decisions are done even in the bus functional model only at the bus transactions boundary. With this limitation the bus functional model and the ATLM (a) do the arbitration decision at the same time points and yield the same simulated timing.

The ATLM (b) yields more imprecise results. It may mispredict the arbitration in the situation when two masters access the bus at the exact same simulation time. Then the master, with the earlier executed simulation code, will gain bus access, even though it may be the lower priority master. For both the high priority and low priority master the inaccuracy rises with an increase of overlap; it plateaus at 40%. Due to the higher bus utilization, fewer simultanious arbitration requests happen. With the shorter delays between the transactions, it becomes likely that the bus is occupied by the other master when requesting the bus.

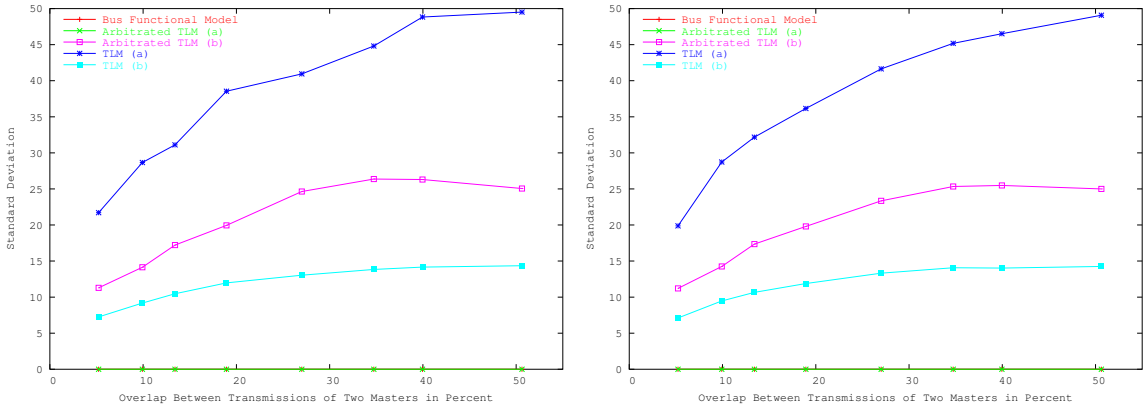
It is interesting to note, that the ATLM (b) performs also worse than the TLM (b). The latter yields a timing as if the bus were used exclusively by each master, thus it always predicts the optimal transfer time. As expected, its results get linearly worse with an increase in overlap, since the individual transfers will increasingly take longer than the optimum.

The TLM (a) shows the highest inaccuracies. Its arbitration decision are made on the level of user transactions, whereas the real decisions are way more fine grained. Furthermore, the bus access decision in this model is independent of the master's priority. The inaccuracy produced by this model increases linearly with the amount of overlap and tops with 35% at 50% overlap.

Summarizing the first measurements, all models show only little difference in accuracy between the high and low priority master. Between the two variations of the ATLM, the version with delta cycle delay is preferable, since it reaches optimal results. Between the two variation of the TLM

model, surprisingly the version without any arbitration yields better results. The expectation was that the lack of arbitration would yield worse results.

Additionally to the average inaccuracy (computed from the absolute inacurracies), the deviations of the inaccuracies are displayed. The standard deviation indicates an inaccuracy range, so that 68% of the individual transfers exhibit an inaccuracy within this range. As one example from the measurements, 68% of the user transactions have an inaccuracy of 25% or less, when transferred using the ATLM (b) with an overlap of 40%.



(a) High priority master

(b) Low priority master

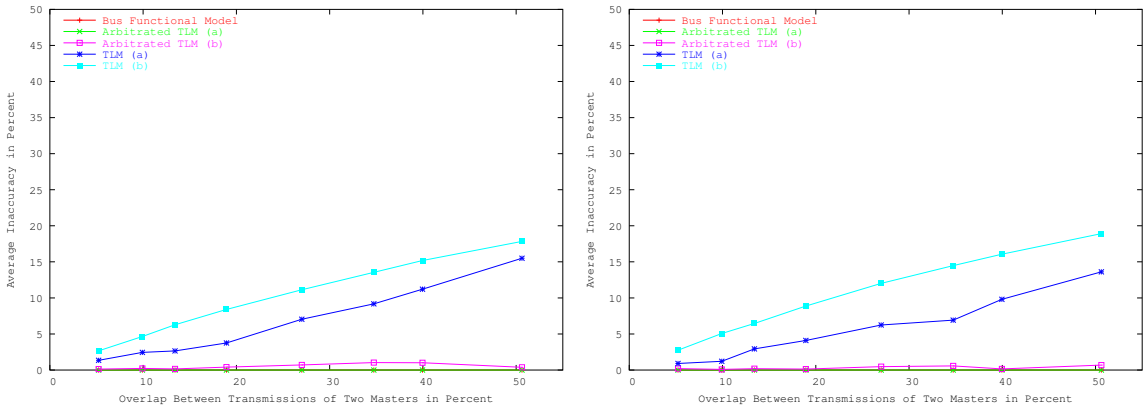
Figure 35: Locked transfer deviation based on duration

Figure 35 shows the graphs of the standard deviation based analysis. Since the graphs stem from the same set of measurements, the same conclusions as shown above hold true. The bus functional model and the ATLM (a) deliver 100% accurate results. The TLM (b) is more accurate than the ATLM (b), since the latter may mispredict the bus grant, whereas the former assumes an always avaiable bus. As with the previous results, the TLM (a) shows the most inaccurate results, due to the coarse granularity of arbtiration decision.

The accuracy analysis based on the transfer duration is the measure to predict the application latency due to bus traffic. Additionally, the overall timing (e.g. when does the application finish?) may be of interest for design decisions. For this, the cumulative transfer time, that is the sum of the user transactions durations, was evaluated. The cumulative transfer time is preferred over the actual finish time, since the latter includes the constant computation time between transactions (simulated by a delay), which is independent of the utilized bus model.

Figure 36 shows the results of the accuracy based on the cumulative transfer time. Here the differences between the two variations of the ATLM are significantly smaller than in the duration based analysis. The mispredictions made in the ATLM (b) model seem to cancel out over time. The remaining inaccuracies between the two variations are within 2% and are independent of the amount of overlap.

It is noticeable that the inaccuracies of the two TLM variations have reversed in comparison to the duration based evaluation. Now the TLM (b) exhibits the highest inaccuracies, since this model assumes an always available bus. With that, the predictions are almost always too optimistic, hence



(a) High priority master

(b) Low priority master

Figure 36: Locked transfer accuracy based on cumulative transfer time

the error accumulates over many transactions and creates an increasing difference in application timing. The TLM (a) does perform better in that respect, even though it produces larger error on an individual user transaction level, the mispredictions fall on both sides (too short and too long) so that they average out. The TLM (a) predicts more correctly the timing than the TLM (b).

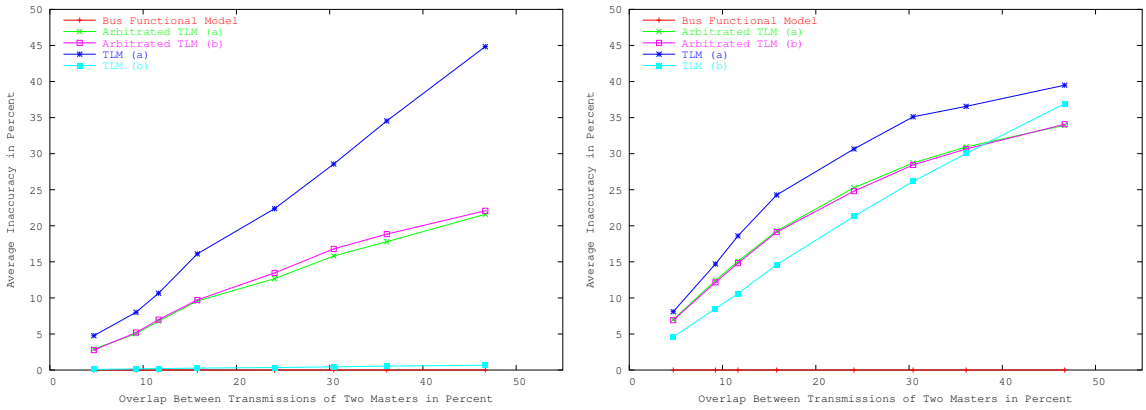
In summary, the models with more detail perform better than the more abstract models, if the main focus rests on the cumulative transfer time. Between the variations of the TLM models, the version with arbitration is selected, because it predicts more accurately the overall application timing. Among the ATLM variations, the version with the delta cycle delay - ATLM (a) is chosen, since it reaches 100% accuracy. It has to be noted that for locked transfers the bus functional model does not have an accuracy advantage. Due to the locked transfers the arbitration decision is done at the same level of granularity for the ATLM and the busfunctional model.

5.2.3 Accuracy of Unlocked Transfers

The analysis of the locked transfers may suggest questioning the added value of the bus functional model. This will be revisited in this section, where the unlocked transfers are evaluated. In case of an unlocked transfer an arbitration decision is done on each individual bus cycle. A transaction initiated by a low priority master may be preempted by a higher priority master. Since only the bus functional model deals with arbitration on each bus cycle, it is expected to be the only accurate model.

Figure 37 shows the graphs for the accuracy evaluation based on the transfer duration. Here unlike for the locked transfers the graphs between the high and the low priority master differ now. With the preemption possibility of the unlocked transfers, there are higher chances that the low priority master has to yield access for the high priority master.

The differences previously observed between the two variations of the ATLM model are no longer significant for the unlocked transfers. Both ATLM models exhibit significant inaccuracies over the bus functional model, caused by the lower granularity of the arbitration decision. The



(a) High Priority Master

(b) Low Priority Master

Figure 37: Unlocked transfer accuracy based on duration

ATLM model decides per bus transaction, whereas the bus functional model revisits the arbitration on each bus cycle.

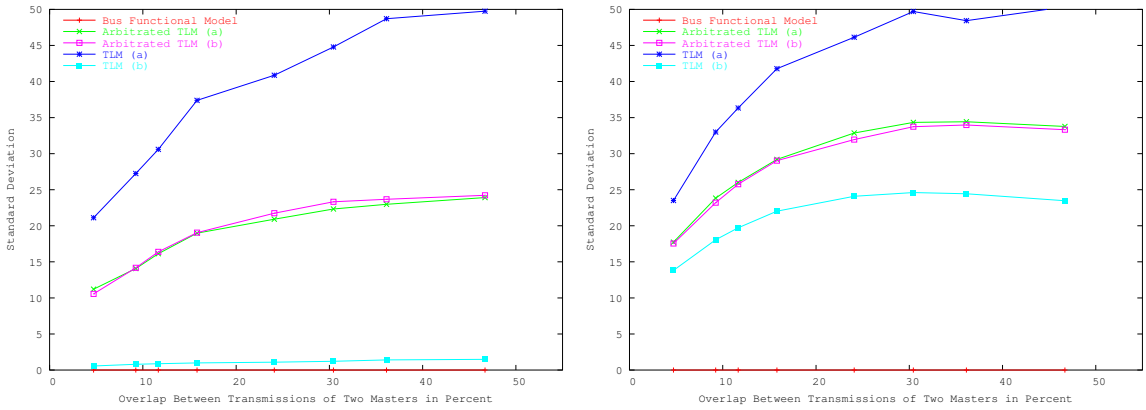
The results for the two TLM variations differ even more dramatically between the masters. For the high priority master the TLM (b) reaches almost 100% accruacy. Again, this model assumes an alway available bus. This is actually close to reality for the high priority master, since it may preempt a tranfer of the low priority master. On the other hand, the opposite is true for the low priority master. The predicion of the TLM (b) are always too optimistic. With an increase of the transfer overlap, the prediction becomes linearly less.

The TLM (a) performs most inaccurately for both the high and the low priority master. The inaccuracy increases linearly for the high priority master, with the increase of transfer overlap. Since the TLM (a) performs the arbitration decision only once for each user transaction, the error increases with the transfer overlap. For the low priority master it is interesting to note that the inaccuracy does not increase linearly, but it rather tails off.

As done before the duration based accuracy data is displayed again using the standard deviation, see Figure 38. The graphs do confirm what has been already described for the previous set.

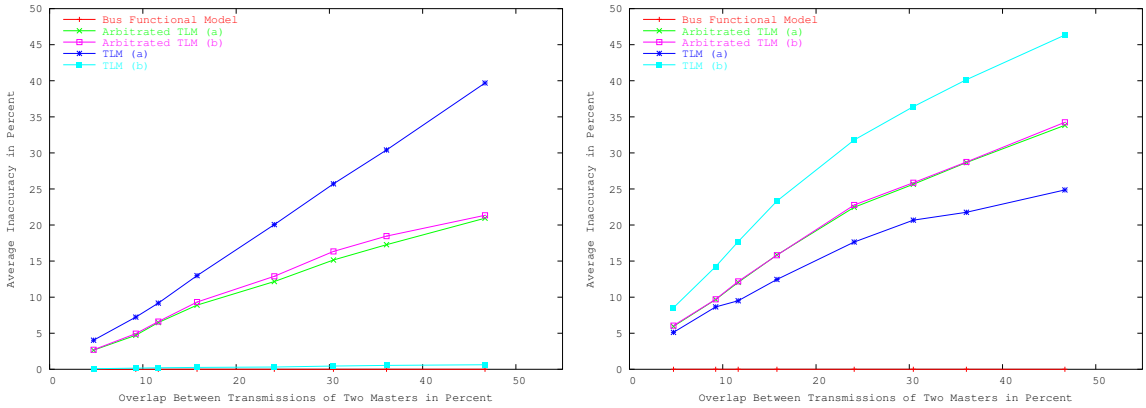
The graphs for the cumulative transfer time are shown in Figure 39. The results for the high priority master are similar to what has been observed during the duration based analysis. For the low priority master, only the variations of the ATLM model perform comparable. However the TLM models behave opposite to what has been seen in the duration based accuracy evaluation. Now the TLM (b) shows the highest inaccuracies. Its constantly overoptimistic predictions accumulate and result in almost 50% inaccuracy at 50% overlap. The TLM (a) performs better than the TLM (b). Although it exhibits a high error amout for the duration based analysis, the errors on the individual transfers average out, yielding a lower inaccuracy for the cumulative transfer time. In general comparing back to the locked transfers, all abstract models exibit higher inaccuracies simulating unlocked transfers.

Considering the accuracy of unlocked transfers, there is no clear choice between the variations of the TLM model. The differences between the accuracy for the high priority and the low priority



(a) High Priority Master (b) Low Priority Master

Figure 38: Unlocked transfer deviation based on duration



(a) High Priority Master (b) Low Priority Master

Figure 39: Unlocked transfer accuracy based on cumulative transfer time

master are too significant. However, since the accuracy of TLM (a) is more predictable, preference is given to this model. Both variations of the ATLM model perform very similarly. In general, for unlocked transfers a designer should use at least a variation of the ATLM in order to gain predictable results. However it becomes clear that only the bus functional model yields accurate results.

5.3 Analysis Summary

The performance analysis of the different models did show a speed up of two orders of magnitude with each additional abstraction level (i.e. among the major models). Therefore, the goal of drastically speeding up simulation, by means of abstract models, has been fulfilled. The performance analysis alone does not yield a decision for choosing among the variations of the ATLM and TLM; no significant performance difference was measured between each variation pair.

Combining the accuracy analysis of both the locked and the unlocked transfers provides a better ground for making a decision in this aspect. Between the TLM variations, the TLM (a) is selected. It performed more accurately in the cumulative tests for the locked transfers, and was more consistent in its predictions for the unlocked transfers. The ATLM (a) is chosen among the ATLM variations, since it was accurate in the locked transfer tests and both variations performed similarly for the unlocked transfers.

In general the accuracy analysis has shown that the advantage of faster simulation speeds has to be weighted against the loss in simulation accuracy. The more abstract models did deliver overall more inaccurate results. However, the results are strongly correlated with the application characteristics. The guidelines of model use, extracted from this correlation, are described in the next chapter.

6 Summary and Conclusions

This work has reported on the modeling of the AMBA AHB bus architecture. Three major models have been implemented: the bus functional mode, arbitrated transaction level model (ATLM) and the transaction level model (TLM). Additionally, two variations have been created for each of the ATLM and the TLM. The correctness of each model in terms of functionality and in terms of timing has been validated. The AMBA models have been integrated with the SCE design environment.

The usability of the models has been evaluated. With respect to the simulation performance, a speedup of two orders of magnitudes was measured with each abstraction step. A detailed analysis of the simulation accuracy of each model has been done. As a result of the analysis the TLM (a) – which models concurrency – and the ATLM (a) – which implements the delta cycle delay for arbitration requests – have been chosen for continued use. Based on the analysis results, the summary as shown in Table 8 can be made for the user of the implemented models.

Environment Condition	**Applicable Model**
• single master bus • no overlap between masters bus access	TLM
• only locked transfers • unlocked transfers and low overlap	ATLM
• unlocked transfers and high overlap	bus functional

Table 8: Conclusion summary

For computation bound applications, or when almost no overlap between transactions of two masters on the same bus is expected, all models have almost accurate results. In this case the most abstract model – the TLM – delivers acceptable results the fastest.

In a system, where only locked transfers are used, already the ATLM model gives accurate results, hence in such a case a simulation with the bus functional model is not needed for accuracy reasons. Also the TLM model delivers usable results. It peaks with only 15% inaccuracy at 50% tansfer overlap.

Should the system use unlocked transfers, the importance of the bus functional model comes into play. It is the only one that delivers accurate results. The ATLM model may be used for estimation, in case the transmissions between masters do not overlap too much (25% inaccuracy at 25% overlap). The TLM model should be avoided since it gives inconsistent results between a high and a low priority master.

In future work, the AHB model will be extended to support more complex bus transactions like split transfers, which will expand the usability of the model. For a further performance increase, multi-threaded master / slave bus interfaces will be included in the model. This will allow the modeling environment to take full advantage of the pipelined access even within bus accesses of the same master. Furthermore, it is planned to model the peripheral bus APB for an efficient connection to peripheral devices.

References

[1] Advanced RISC Machines Ltd (ARM). AMBA AHB Cycle Level Interface (AHB CLI) Specification. http://www.arm.com/products/solutions/ahbcli.html. ARM IHI 0011A.

[2] Advanced RISC Machines Ltd (ARM). AMBA Home Page. http://www.arm.com/products/solutions/AMBAHomePage.html.

[3] Advanced RISC Machines Ltd (ARM). AMBA Specification (Rev. 2.0), ARM IHI 0011A. http://www.arm.com/products/solutions/AMBA_Spec.html.

[4] Advanced RISC Machines Ltd (ARM). Technical Support FAQs - AMBA. http://www.arm.com/support/AMBA.html.

[5] Advanced Processor Technologies Group at University of Manchester. GTKWave Electronic Waveform Viewer. http://www.cs.man.ac.uk/apt/tools/gtkwave.

[6] Marco Caldari, Massimo Conti, Marcello Coppola, Stephane Curaba, Lorenzo Pieralisi, and Claudio Turchetti. Transaction-level models for AMBA bus architecture using SystemC 2.0. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2003.

[7] CoWare. CoWare Launches Fast AMBA Transactional Bus Simulator for SystemC. http://www.coware.com/portal/page?_pageid=166,105427&_dad=cust_portal&_schema=STAGE.

[8] Rainer Dömer. *System-Level Modeling and Design with the SpecC Language*. PhD thesis, University of Dortmund, Germany, April 2000.

[9] John W. Eaton. Octave Home Page. http://www.octave.org/, 1998.

[10] A. Gerstlauer; D. Shin; R. Doemer; D. Gajski. System-Level Communication Modeling for Network-on-Chip Synthesis. In *Asia and South Pacific Design Automation Conference*, Shanghai, China, January 2005.

[11] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[12] Andreas Gerstlauer. *Modeling Flow for Automated System Design and Exploration*. PhD thesis, University of California Irvine, U.S.A., April 2004.

[13] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[14] Andreas Gerstlauer and Daniel D. Gajski. System-level abstraction semantics. In *Proceedings of the International Symposium on System Synthesis*, Kyoto, Japan, October 2002.

[15] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[16] Internation Organization for Standardization (ISO). *Reference Model of Open System Interconnection (OSI)*, second edition, 1994. ISO/IEC 7498 Standard.

[17] W. Rosenstiel J. Gerlach. A methodology and tool for automated transformational high-level design space exploration. In *Computer Design, 2000. Proceedings. 2000 International Conference on, Vol., Iss.*, 2000.

[18] A.A. Jerraya K. Svarstad, G. Nicolescu. A model for describing communication between aggregate objects in the specification and design of embedded systems. In *Design, Automation and Test in Europe*, 2001.

[19] Peter Marwedel. *Embedded Systems Design*. Kluwer Academic Publishers, 2003.

[20] Wolfgang Mueller, Rainer Dömer, and Andreas Gerstlauer. The formal execution semantics of SpecC. In *Proceedings of the International Symposium on System Synthesis*, Kyoto, Japan, October 2002.

[21] Open SystemC Initiative, http://www.systemc.org. *Functional Specification for SystemC 2.0*, 2000.

[22] M. Sgroi; M. Sheets; M. Mihal; K. Keutzer; S. Malik; J. Rabaey; and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication based design. In *Proceedings of the Design Automation Conference*, June 2001.

[23] Gunar Schirner. System Level Modeling of an AMBA Bus. Master's thesis, University of California, Irvine, March 2005.

[24] Robert Siegmund and Dietmar Müller. SystemC^{SV}: An extension of SystemC for mixed multi-level communication modeling and interface-based system design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2001.

[25] Mohamed Ben-Romdhane Sudeep Pasricha, Nikil Dutt. Fast exploration of bus-based on-chip communication architectures. In *CODES and ISSS*, Stockholm, Sweden, September 2004.

[26] Karl van Rompaey, Diederick Verkest Ivo Bolsens, and Hugo De Man. CoWare: A design environment for heterogeneous hardware/software systems. In *Proceedings of the European Design Automation Conference (Euro-DAC)*, Geneva, Switzerland, September 1996.

[27] Wayne Wolf. Hardware/software co-synthesis algorithms. In Ahmed A. Jerraya and Jean Mermet, editors, *System-Level Synthesis*. Kluwer Academic Publishers, 1998.

[28] Jianwen Zhu, Rainer Dömer, and Daniel D. Gajski. Syntax and semantics of the SpecC language. In *Proceedings of the International Symposium on System Synthesis*, Osaka, Japan, December 1997.

A Header Files

This chapter will give an overview of the implemented software structure. In general the following file separation was used for the AMBA model:

i_ambaAHBbus.sh contains interface definitions which are used by all models. These interfaces cover the MAC layer only.

ambaAHBbusMaster.sc defines the interfaces for the protocol layer and the physical layer for the master side. It also contains the bus functional implementation of all layers for the master side.

ambaAHBbusSlave.sc is symmetric to the previous file. It contains all the interface definitions for the slave side of the protocol and physical layer. The file also contains the slaves bus functional implementation.

ambaAHBbusTLM.sc contains the implementations of the abstract models for master and slave side. In particular it contains both variances of the ATLM and both variances of the TLM. The file contains as well the arbitration implementation for the abstract models.

ambaAHBarbiter.sc implements the arbitration for the bus functional model. The main cvs branch contains the arbiter for locked transfers, the branch *unlockedTransfers* implements arbitration for unlocked transfers.

ambaAHBMuxes.sc implements additional logic necessary for the bus functional model of the AHB; these are in particular the multiplexers (read bus, write bus, address and control bus) and the address decoder.

The following sections will show the interface definitions of the implemented models. The channel declarations are listed as well, which allows insight into how the different layers (implemented in channels) are composed to a bus model.

A.1 *i_ambaAHBbus.sh*: MAC Layer Interface Definitions for Master and Slave

```
/* ---- interfaces visible to the upper layers ---- */

/* all MAC layer interface definitions.
    two access types memory and link
    twice once for the master side and once for the slave side
*/

interface IAmbaAHBbusMasterMACLink
{
  void masterWrite(unsigned long addr, const void* data, unsigned long len);
  void masterRead(unsigned long addr, void* data, unsigned long len);
};
```

```
   interface IAmbaAHBbusMasterMACMem
15 {
     void masterMemWrite(unsigned long addr, const void* data, unsigned long len);
     void masterMemRead(unsigned long addr, void* data, unsigned long len);
   };

20 interface IAmbaAHBbusSlaveMACLink
   {
     void slaveWrite(unsigned long addr, const void* data, unsigned long len);
     void slaveRead(unsigned long addr, void* data, unsigned long len);
   };
25
   interface IAmbaAHBbusSlaveMACMem
   {
     void serve(unsigned long addr, void* data, unsigned long len);
   };
```

A.2 *ambaAHBbusMaster.sc*: Bus Functional Interfaces and Channel Definition for Master

```
 1 /* ——— Physical layer, bus protocol handling ——— */

   // regular bus primitives
   interface IAmbaAHBbusMaster
   {
 6   /* GS Access methods for the Address Cycle */
     /* Writes out address and control signals, waits for
        completion of previous Slave.
        NOTE: Has to be called at beginning of clock cycle. */
     void AddressCycle (
11                      tAHBAddr   addr,
                        tAHBWrite  write,
                        tAHBSize   size,
                        tAHBProt   prot,
                        tAHBBurst  burst,
16                      tAHBTrans  trans
                        );

    /* writes out the address and control signals, without waiting for timing
        NOTE: Has to be called after rising edge. */
21   void AddressWrite (
                        tAHBAddr   addr,
                        tAHBWrite  write,
                        tAHBSize   size,
                        tAHBProt   prot,
26                      tAHBBurst  burst,
                        tAHBTrans  trans
                        );

     /* write the given data on the bus and wait until slave
31      has accepted the data,
        RETURNS: Status code from slave */
```

```
    tAHBResp DataWriteCycle(tAHBData data);


36  /* write the given data on the bus and wait until slave
       has accepted the data,
       RETURNS: Status code from slave */
    tAHBResp DataReadCycle(tAHBData *data);

41 };

  /* physical layer for master side */
  channel AmbaAHBbusMaster(
    in  signal bit[1]      HCLK,     // from external clk, all on rising edge
46  in  signal bit[1]      HRESETn,  // low active reset signal for bus component
    out signal bit[31:0]   HADDR,    // 32 bit system address bus
    out signal bit[1:0]    HTRANS,   // transfer type (IDLE, ...)
    out signal bit[1]      HWRITE,   // write on high
    out signal bit[2:0]    HSIZE,    // size of transfer
51  out signal bit[2:0]    HBURST,   // burst mode selection
    out signal bit[3:0]    HPROT,    // protection bits
    out signal bit[HDATA_BUS_HIGH_BIT:0]
                       HWDATA,   // write data bus (master -> slave)
    in  signal bit[HDATA_BUS_HIGH_BIT:0]
56                     HRDATA,   // read data bus (slave -> master)
    in  signal bit[1]      HREADY,   // slave indicates operation complete
    in  signal bit[1:0]    HRESP     // slave indicates return code for op.


61  ) implements IAmbaAHBbusMaster
  {
  };

  /* ----- Protocol Layer, includes arbitration ----- */
66 interface IAmbaAHBbusMasterProtocol
  {
    /* convention, all functions have to be called on a rising clock edge,
       this has to be guaranteed by the calling mac layer */

71  bit[7:0] ReadByte(bit[31:0] addr);
    bit[15:0] ReadWord(bit[31:1] addr);
    bit[31:0] ReadLong(bit[31:2] addr);
    // multi burst size 4, 8, 16 longs
    tAHBResp ReadBurst(bit[31:2] addr, tAHBData data[], unsigned char size);
76
    void WriteByte(bit[31:0] addr, bit[7:0] val);
    void WriteWord(bit[31:1] addr, bit[15:0] val);
    void WriteLong(bit[31:2] addr, bit[31:0] val);
    // multi burst size 4, 8, 16 longs
81  tAHBResp WriteBurst(bit[31:2] addr, tAHBData data[], unsigned char size);
    // NOTE not implemented undefined bursts, burst for words or bytes
  };
```

```
   /* protocol layer master side */
 86 channel AmbaAHBbusMasterProtocol(IAmbaAHBbusMaster bus,
                                     i_semaphore       access)
      implements IAmbaAHBbusMasterProtocol
   {
   };
 91

   /* ------  Media access layer, links ------ */
   /* This is a simplified version of the memory access,
      - no address increase
 96    - no bursts
      - no alignment for addresses, off aligned access
        gives bus error simulated by core dump

        Compatible with AmbaAHBbusSlaveMacLinkNoAdddrInc
101 */

   channel AmbaAHBbusMasterMACLinkNoAddrInc(IAmbaAHBbusMasterProtocol mac)
     implements IAmbaAHBbusMasterMACLink
   {
106 };
```

A.3 *ambaAHBbusSlave.sc*: Bus Functional Interfaces and Channel Definition for Slave

```
   /* ------ Physical layer, bus protocol handling ------ */

  3 interface IAmbaAHBbusSlave
   {

     /* listen to specified set of control signals without waiting for clock */
     tAHBSize ListenCntl(tAHBAddr *addr,
  8                      tAHBAddr addrMask,
                         tAHBBurst *burst,    // burst mode
                         tAHBBurst burstMask,
                         tAHBProt *prot,      // protection type
                         tAHBProt protMask,   // ~ mask
 13                      tAHBWrite    *write,    // write mode ?
                         tAHBWrite writeMask );


     /* listen to specified set of control signals with waiting for clock */
 18  tAHBSize ListenCntlCycle(tAHBAddr *addr,
                          tAHBAddr addrMask,
                          tAHBBurst *burst,    // burst mode
                          tAHBBurst burstMask,
                          tAHBProt *prot,      // protection type
 23                       tAHBProt protMask,   // ~ mask
                          tAHBWrite    *write,    // write mode ?
                          tAHBWrite writeMask );
```

```
28    /* write data to bus (master read), and consume a cycle */
      void WriteCycle(tAHBData val);


      /* read data from bus (masters write) and concume a cycle */
33    tAHBData ReadCycle(void);

      /* signal an error or other condition to master, called
         instead of WriteCycle or ReadCycle */
      void TwoCycleResp(bit[1:0] resp);
38  };

    channel AmbaAHBbusSlave (

      in  signal bit[1]      HCLK,     // from external clk, all on rising edge
43    in  signal bit[1]      HRESETn,  // low active reset signal for bus component
      in  signal bit[31:0]   HADDR,    // 32 bit system address bus
      in  signal bit[1:0]    HTRANS,   // transfer type (IDLE, ...)
      in  signal bit[1]      HWRITE,   // write on high
      in  signal bit[2:0]    HSIZE,    // size of transfer
48    in  signal bit[2:0]    HBURST,   // burst mode selection
      in  signal bit[3:0]    HPROT,    // protection bits
      in  signal bit[HDATA_BUS_HIGH_BIT:0]
                        HWDATA,  // write data bus (master -> slave)
      out signal bit[HDATA_BUS_HIGH_BIT:0]
53                      HRDATA,  // read data bus (slave -> master)
      in  signal bit[1]      HSELx,    // select signal for slave
          signal bit[1]      HREADY,   // slave indicates operation complete
      out signal bit[1:0]    HRESP     // slave indicates return code for op.
      )
58       implements IAmbaAHBbusSlave
    {
    };


63
    /* ------ Protocol layer, arbitration ------ */

    interface IAmbaAHBbusSlaveProtocol {

68    /* listen to specified set of control signals with waiting for clock */
      tAHBSize ListenCntlCycle(tAHBAddr *addr,
                               tAHBAddr addrMask,
                               tAHBBurst *burst,    // burst mode
                               tAHBBurst burstMask,
73                             tAHBProt *prot,      // protection type
                               tAHBProt protMask,   // ~ mask
                               tAHBWrite    *write,
                               tAHBWrite writeMask);     // write mode ?

78    /* bus data cycle operations, each one consumes a cycle  */
```

```
    bit[7:0] ReadByte(bit[1:0] addr);
    bit[15:0] ReadWord(bit[1:1] addr);
    bit[31:0] ReadLong(void);
    void ReadBurst(tAHBData data[], unsigned char numBeats);

    void WriteByte(bit[7:0] val);
    void WriteWord(bit[15:0] val);
    void WriteLong(bit[31:0] val);
    void WriteBurst(tAHBData data[], unsigned char numBeats);

    /* signal an error or other condition to master */
    void TwoCycleResp(bit[1:0] resp);

};

/* ------ MAC layer, segmentation, reassmbly ------------- */


/* MAC layer slave, randezvouz access (link access) */
/* Reduced version of MACLink with the following simplifying assumptions
   - no address increase during transmission
   - no bursts
   - no alignment transfers, off alignment access results in
   bus access violation */
channel AmbaAHBbusSlaveMACLink(IAmbaAHBbusSlaveProtocol protocol)
  implements IAmbaAHBbusSlaveMACLink
{
};

/* MAC layer, slave, memory access */
channel AmbaAHBbusSlaveMACMem(IAmbaAHBbusSlaveProtocol protocol)
      implements IAmbaAHBbusSlaveMACMem
{
};
```

A.4 *ambaAHBbusTLM.sc*: Interfaces and Channel Definitions for Abstract Models

```
/* Transaction Level Modelling
   on level of MAC.link,
   Can be used instead of AmbaAHBbusMasterMACLink

   Allows access of multiple multiple masters and multiple slaves at the
   same time.
*/
channel AmbaAHBbusMasterMACLinkTLM(void)
      implements IAmbaAHBbusMasterMACLink,
                 IAmbaAHBbusMasterMACMem,
                 IAmbaAHBbusSlaveMACLink,
                 IAmbaAHBbusSlaveMACMem
{
}
```

```
/* Transaction Level Modelling
   on level of MAC.link,
   Can be used instead of AmbaAHBbusMasterMACLink

   Allows access of multiple multiple masters and multiple slaves at the
   same time.

   ATTENTION without arbitration ! */
channel AmbaAHBbusMasterMACLinkTLMNoArbit(void)
      implements IAmbaAHBbusMasterMACLink,
                 IAmbaAHBbusMasterMACMem,
                 IAmbaAHBbusSlaveMACLink,
                 IAmbaAHBbusSlaveMACMem
{
}

/* give each master an identity for arbitration */
channel AmbaAHBbusMasterProtocolTLM(
       unsigned int masterNr,  // identity of the master
       // tlm model containing the bus and the arbitration modelling
       IAmbaAHBbusProtocolTLM_Arbitration busAndArb
                                         )
      implements IAmbaAHBbusMasterProtocol
{
}

/* protocol layer implementation master and slave for ATLM */
channel AmbaAHBbusProtocolTLM()
      implements IAmbaAHBbusSlaveProtocol,
                 IAmbaAHBbusProtocolTLM_Arbitration
{
}

/* same as above but no delta cycle collection of requests */
channel AmbaAHBbusProtocolTLMNoDelta()
      implements IAmbaAHBbusSlaveProtocol,
                 IAmbaAHBbusProtocolTLM_Arbitration
{
}
```

B Testing Environment

B.1 Source Code Structure

In addition to the previously described files, which contain the AMBA models, a set of files is required for the testing environment. For ease of debugging and controlling, it was decided that each test group is captured in an own executable. Since for each test group up to 5 different models had to be validated, a large number of test executables is created during compilation process.

In order to minimize code duplication, as a means of reducing the maintenance effort in the ongoing project, a single test bench file *testbench.sc* was developed. This test bench file conditionally

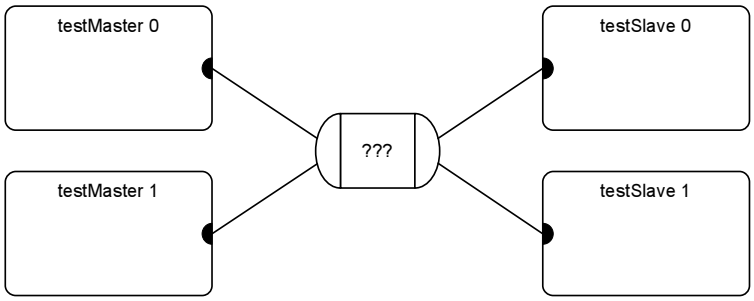


Figure 40: Generic connection scheme

includes a particular test group and a particular bus model. All necessary components are instantiated and connected in the test bench. This includes the test master behaviors and the test slave behaviors, which are connected to selected bus model (see Figure 40). Additionally supporting elements are handled, e.g. in the bus functional model: multiplexers, arbiter, clock driver, and address decoder. Since the test bench file contains all wiring information, having a single version for all test cases significantly simplified changes during the development time.

For each group of tests a separate master and slave behavior were implemented. Precompiler directives were used to conditionally include the selected master and slave code. For ease of identification the naming convention below was followed:

testMaster_*testName*.sh contains common definitions used for both the behavior running in the master and the behavior running in the slave.

testMaster_*testName*.sc implements the behavior for the master side of the bus access.

testSlave_*testName*.sc implements the behavior for the slave side of the bus access.

The test behaviors use an interface to the according MAC layer (either memory or rendezvous style) as an input. They are connected by the test bench through the bus model under test. The the part *testName* in the file name above is replaced by the short name as defined in Table 9 of the according test. The set of files that have to be included for a particular test setup are selected using preprocessor directives within the test bench. Table 9 lists the testcases with their short names and the macro definitions for test selection.

B.2 Test Executables

As indicated earlier, the test bench will not only select a test group to be executed, but also a model for the actual transmission. As for the testcases the according model (or the stack of channels) is selected with the precompiler directives as shown in Table 10.

With the short names defined for the test group and the bus model, the name of the executables can be constructed. All test executables obey the following naming convention: **test_*channelName*_*testName***. Where the *channelName* is replaced with the short name of the bus model (3^{rd} column of Table 10) and the *testName* is replaced with the short name of the test group

Test Name	Section	Short Name	Macro Definition
Individual Transfers	4.1.1	indiv	TEST_INDIV
Random Access using Memory Style Access	4.1.2	randMem	TEST_RAND
Random Access using Rendezvous Style Access	4.1.3	randMsg	TEST_RAND_MSG
Timing Validation for Bus Functional Model	4.2	print	TEST_PRINT
TLM Timing Validation versus Bus Functional Model	4.3	tlmTiming	TEST_TLM_TIMING
Explicit Timing Measurements for Example Transfers	4.3	memTiming	TEST_MEM_TIMING
Transfer Performance for Memory Style Access	5.1	perfMem	TEST_PERF_MEM
Transfer Performance for Rendezvous Style Access	5.1	perfRand	TEST_PERF_RAND
Timing Accuracy of TLM Models	5.2	perfTiming	TEST_PERF_TIMING

Table 9: List of implemented tests, with the section where the results are discussed, a short name that is used for test file naming, and the define statement used in the test bench for the test selection.

(3^{rd} column of Table 9). As an example the executable for testing individual transfers with the bus functional model is named: test_bf_indiv.

With the large amount of test executables an automatic test execution becomes necessary. As described in the results section, the test execution is categorized into three parts. The functional tests have a build in failure detection and terminate with an error. The timing validation of the abstract models with respect to the bus functional model includes an error detection. A makefile rule can be used to iterate through all bus models and the tests in these two categories and the test will stop on the first detected error:

```
make test
```

A large number of test executions is required for the performance tests, hence this has been

Model Name	Section	Short Name	Macro Definition
Transaction Level Model (A)	3.3	tlm	USE_CHANNEL_TLM
Transaction Level Model (B)	3.3	tlmb	USE_CHANNEL_TLM_B
Arbitrated Transaction Level Model (A)	3.4	prot	USE_CHANNEL_PROT
Arbitrated Transaction Level Model (B)	3.4	protb	USE_CHANNEL_PROT_B
Bus Functional Model	3.5	bf	USE_CHANNEL_BF

Table 10: List of implemented bus models, with a reference to the chapter explaining the design, a short name for file naming convention, and the macro name for the channel selection in the test bench.

automated with wrapping shell scripts. Measuring of the execution performance of the memory and rendezvous style access over all implemented channels can initiated with the following commands:

```
run_perfMem
run_perfRand
```

Octave [9], a Matlab-like numerical evaluation environment, is used for automatically graphing the results of the performance tests. Two scripts (*gen_transferTime.m* and *gen_transferTimeRand.m*) generate graphs for the performance in terms of execution speed (*transferTime.eps* and *transferTimeRand.eps*, see Figure 31) and transfer bandwidth (*transferBandwidth.eps* and *transferBandwidthRand.eps*, see Figure 32).

The measurements for the timing accuracy of the implemented models have been wrapped into:

```
run_perfTiming
```

Again the results are automatically graphed by Octave scripts. *gen_perfTiming* generates the graphic files as listed in table Table 11. In addition to the files in the table, which are specific to the first master, a same set of files is created for the second master. Their names can be distinguished by an *_M*1 instead of *_M*0 in the end of the file name.

File Name	**Description**
accuray.duration.2*M*2*SP_M*0.*eps*	accuracy based on transfer duration (Fig. 34)
accuracy.finish.2*M*2*SP_M*0.*eps*	accuracy based on finish time of each transfer
accuracy.comulative.2*M*2*SP_M*0.*eps*	accuracy based on cumulative transfer time (Fig. 36)
deviation.duration.2*M*2*SP_M*0.*eps*	deviation based on transfer duration (Fig. 35)
deviation.finish.2*M*2*SP_M*0.*eps*	deviation based on finish time of each transfer
deviation.comulative.2*M*2*SP_M*0.*eps*	deviation based on cumulative transfer time

Table 11: Generated graphics for timing accuracy