An Innovative AMBA Compliant Crossbar Scheme and its Fault Tolerance Analysis

BY

ANDREA GIANARRO
Laurea, Politecnico di Torino, Turin, Italy, 2012

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2013

Chicago, Illinois

Defense Committee:

Zhichun Zhu, Chair and Advisor
Mariagrazia Graziano, Politecnico di Torino
Wenjing Rao

In loving memory of my grandmother Ada

ACKNOWLEDGMENTS

The preparation of this thesis would not have been possible without the support of my parents, that believed in my personal abilities and skills through all of these years.

I also would like to thank my home university, Politecnico di Torino, and my adoptive city, Torino, which deeply motivated me in the past five-and-a-half years with their vitality and interest.

Finally my work would not have been possible without the Internet, which provided the means to access the essential resources and infrastructure I needed, including the fundamental entertainment during breaks between endless testing and debugging sessions.

AG

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF ABBREVIATIONS

ARM	Advanced RISC Machines or Acorn RISC Machine
AMBA	Advanced Microcontroller Bus Architecture
AHB	AMBA High-performance Bus
SEU	Single Event Upset
IP core	Intellectual Property core
SMP	Synchronous Multi-Processing
SoC	System on Chip
VHDL	VHSIC Hardware Description Language
GCC	GNU Compiler Collection
SPARC	Scalable Processor Architecture
UIC	University of Illinois at Chicago

SUMMARY

The recent trend in the semiconductor industry to increase the number of computing cores on a single microprocessor has encountered a bottleneck in processor to memory communication bandwidth. To ensure that higher parallelism could be achieved between processing elements and memory banks, the old fashioned shared bus topology for on-chip intercommunication is not sufficient anymore. Alternative topologies have been proposed. The classical bus architecture is often replaced by a crossbar switch type interconnection where cores can simultaneously access memory banks and I/O modules, increasing parallelism.

Through this thesis we will study the LEON 3 processor architecture and study a way to increase the speed of the intercommunication network used by LEON 3 systems. We will adapt ARM's Advanced Microcontroller Bus Architecture (AMBA) High-performance Bus (AHB), which is the protocol used by LEON 3 and which is inherently implemented as a multiplexed bus, to work with a crossbar-switch structure instead. The resulting module, named AHB Crossbar Controller, will replace the existing AHB Controller and will maintain compatibility with existing AMBA AHB devices.

We will then test the circuit's fault tolerance to Single Event Upset faults to study its reliability in critical applications.

This thesis work is finally intended to be part of a bigger project which aims to the creation of multi-processing partitioned systems. Such systems would include multiple core-

SUMMARY (continued)

memory couples executing in parallel, enabling, through the means of an hypervisor, to have several computing environments running concurrently in these logical partitions. The final objective of this future project is to provide a software alternative to hardware fault-tolerance methods, like radiation hardening, by concurrently running redundant computing environments in a partitioned system and checking via software for correctness.

CHAPTER 1

INTRODUCTION

1.1 Motivations and aims

Interconnection networks are a critical part of computing systems. They have become even more critical in the era of very large-scale integration (VLSI) circuitry because of the drive characteristics of MOS transistors combined with the relatively high capacitance of on-chip interconnects (1).

The interconnection networks, used to connect functional units within a chip, can have a significant – indeed, dominating – effect on the chip's performance. Buses, although the simplest form of interconnect, are a poor choice from a density or power standpoint as the power and space required to drive them at maximum speed grow exponentially with the capacitance of the bus (2). Furthermore, multi-point connection networks are a poor choice because the entire length of the bus must be driven even when only a single conversation may be going on at a time, or where the communication is between direct neighbors. A crossbar is an optimal solution, up to a maximum size determined by the underlying device and wiring technology (1).

The LEON 3 system, a moderately multiprocessing-enabled computing architecture, theoretically supports up to 16 cores, but in reality is used in configurations with up to 4 cores and a limited number of other master modules. The relatively small number of masters in this system leads us to think that a crossbar switch topology intercommunication network would be a good compromise between performance increase and implementation complexity to increase the parallelism with respect to the actual shared bus implementation. In this thesis we will investigate the related performance improvements introduced by an innovative crossbar switch, in the form of the AHB Crossbar Controller. The new module, which will replace the shared bus structure used in the LEON 3, will support existing hardware transparently as it is completely compatible with any AHB-compliant device. We will also investigate the reliability of the resulting entity by performing an in-depth series of fault injections to simulate Single Event Upset (SEU) faults. SEUs are a kind of non-destructive fault that is caused by electro-magnetic radiation striking a circuit, and are frequent in microprocessors for space-applications.

We will start by focusing our case study on the LEON 3 processor architecture. We will consider the feasibility of a crossbar adaptation of the protocol. The LEON 3 system provides several AMBA-compliant IP cores and a full framework for testing. It is, in our opinion, the ideal choice for such a study because of the open source nature of its code and the fact that it is freely accessible and released under the GNU GPL open-source license. The LEON 3 processor implements the AMBA AHB 2.0 specification, which dictates a shared-bus topology for on-chip intercommunication.

In this chapter we are going to explore the existing building blocks upon which our work is based, and we are introducing the main differences between the shared-bus and crossbar switch topologies.

1.2 On-chip interconnection networks

The components in a Multi Processing SoC design invariably need to communicate with each other during application execution (3). For instance, a microprocessor fetches instructions from memory components, or writes to external memories by sending data to an on-chip memory controller. It is the responsibility of the on-chip communication architecture to ensure that the multiple, co-existing data streams on the chip are correctly and reliably routed from the source components to their intended destinations. In addition to correctness, the on-chip communication architecture must provide latency or bandwidth guarantees to ensure that the application performance constraints are satisfied. A latency guarantee implies that a data unit must traverse the communication architecture and reach its destination within a finite amount of time, determined by a latency bound (e.g., 40 ns from source to destination). Depending on the performance requirements of an application, various types of on-chip communication architectures can be used, as described in following subsections.

1.2.1 Shared Bus

A basic building block of most on-chip communication architectures in MPSoC designs is the single shared bus. This is the simplest on-chip communication architecture, consisting of a set of shared, parallel wires to which various components are connected. Only one component on the bus can have control of the shared wires at any given time to perform data transfers. The bus is accessed in a time-sharing manner by the masters of the system.

This limits the parallelism and achievable performance, which makes it unsuitable for most highly multi-processing SoC applications which can have tens to hundreds of components. Consequently, the single shared bus architecture is not scalable to meet the demands of MPSoC applications.

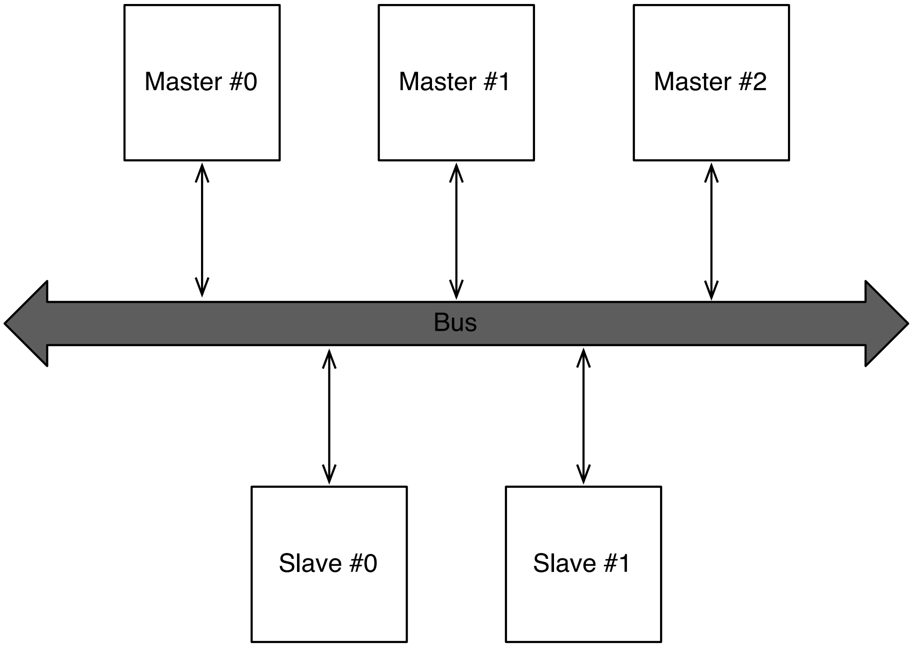

Figure 1. Shared-bus topology

1.2.2 Crossbar Switch

A crossbar switch, also called full bus matrix, is an extension of the shared bus where each master element of the system is connected to all the slaves and vice versa, as shown in Figure 2. This network topology drastically increases parallelism. For instance, two processing elements can access two separate memory banks concurrently. At the same time this structure requires more silicon area and increases power consumption.

A crossbar switch is generally, but not necessarily, implemented through multiplexers, as is the case in my implementation, which is detailed in Section 3.1.

1.3 LEON 3 and GRLIB: an AMBA compliant processor and framework

The GRLIB IP Library is an integrated set of reusable IP cores (4), designed for SoC development. The IP cores are centered around a common on-chip bus, the AMBA AHB 2.0 bus, and use a coherent method for simulation and synthesis. The library is vendor-independent, with support for different CAD tools and target technologies. A unique plug&play method is used to configure and connect the IP cores without the need to modify any global resource.

The Library is organized as a collection of VHDL entities, divided in multiple VHDL libraries. Each library provides components from a particular vendor (e.g. Gaisler, Gleichmann), or a specific set of shared functions or interfaces (e.g. grlib/amba, techmap). Data structures and component declarations to be used in a GRLIB-based design are exported

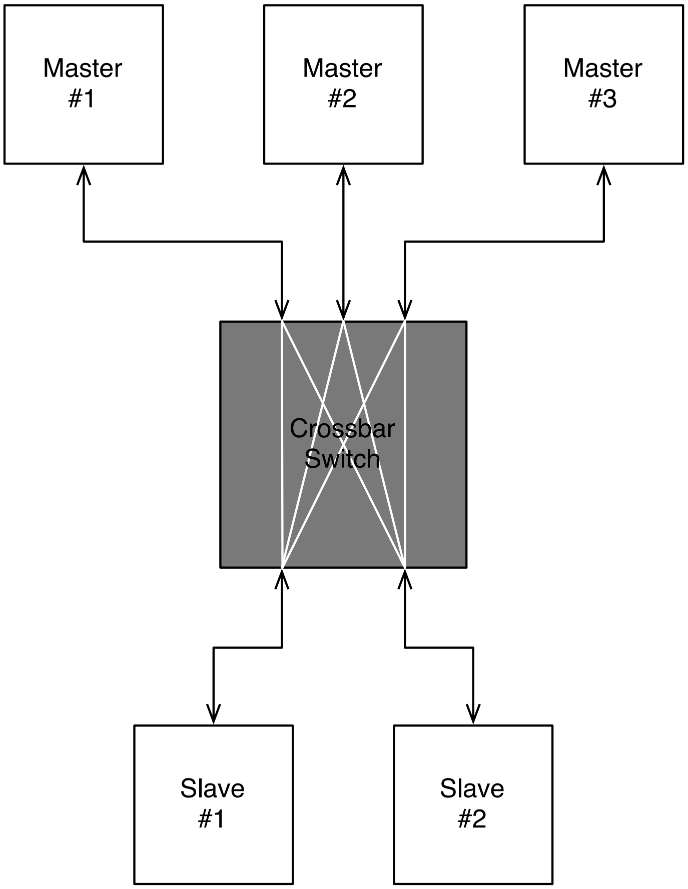

Figure 2. Crossbar switch topology

through library-specific VHDL packages. All GRLIB cores use the same data structures to declare the AMBA interfaces, and can then easily be connected together.

GRLIB is designed to be 'bus-centric', i.e. it is assumed that most of the IP cores be connected through an on-chip bus. The AMBA 2.0 AHB bus was selected as the common on-chip bus, due to its market dominance (ARM processors) and because it is well documented and can be used for free without license restrictions (4). The multiplexed shared-bus structure of AHB is condensed inside the AHB Controller module, which will be replaced, in my implementation, by the AHB Crossbar Controller module. An example of a single processor LEON3 system designed with GRLIB is shown in Figure 3.

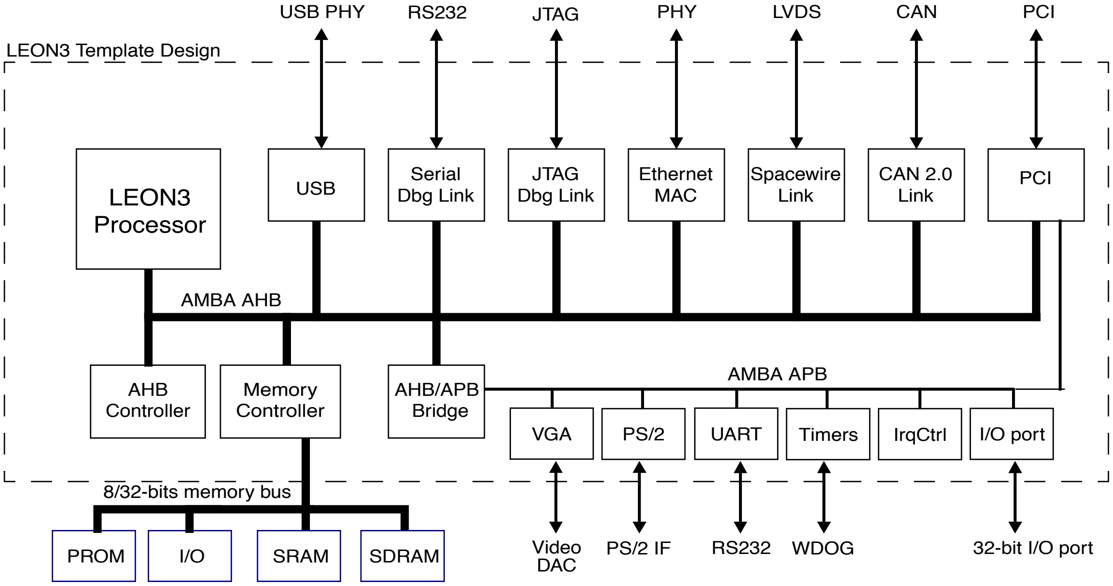

Figure 3. LEON3 GR-XC3S-1500 Template Design

1.3.1 LEON 3 IP Core

The LEON3 processor core, one of the main design modules of GRLIB, is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 (IEEE-1754) architecture (5). It is designed for embedded applications, combining high performance with low complexity and low power consumption. The core is highly configurable and particularly suitable for SoC designs. The configurability allows designers to optimize the processor for performance, power consumption, I/O throughput, silicon area and cost. The processor can be efficiently implemented on FPGA and ASIC technologies and uses standard synchronous memory cells for caches and register file.

The LEON3 core has the following main features (Figure 4): 7-stage pipeline with Harvard architecture, separate instruction and data caches, hardware multiplier and divider, on-chip debug support and multi-processor extensions.

The LEON3 processor's separate instruction and data buses are connected to two independent cache controllers. Both instruction and data cache controllers can be separately configured to implement a direct-mapped cache or a multi-set cache with set associativity of 2–4 and multiple replacement policies.

The LEON3 processor uses one unique AHB master interface for all data and instruction accesses. Instructions are fetched with incremental bursts if the IB bit is set in the cache control register, otherwise single READ cycles are used.

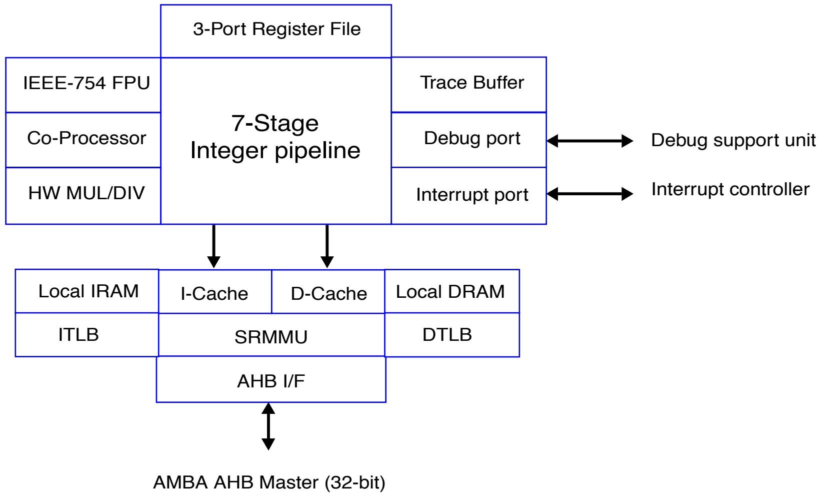

Figure 4. LEON3 processor core block diagram

1.3.2 SMP and Coherency

The LEON3 processor supports synchronous multi-processing (SMP) configurations, with theoretically up to 16 processors attached to the same AHB bus. In practice, the maximum allowed processing cores in GRLIB is 4.

With multi-core support comes the need for cache-coherency. Cache-coherency is the problem of keeping the cache contents of the cores coherent within themselves. So to say, even if there are multiple copies of a memory item in different caches, and the values of one of this items changes, the change will be propagated among the copies. It is worth noticing that the problem arises only on write operations to memory items, not on reads.

In the LEON 3, cache coherency among the cores is maintained by the use of cache snooping and a write-through policy on cache write access. Cache snooping, also known as bus sniffing, is a technique for achieving cache coherence based on the continuous monitoring of transfers on the bus done by every processor. In its most simple implementation, each cache line of a processor can be either 'valid' or 'invalid'. When processor X performs a memory write, the data is written in its cache(X), and at the same time it is also written in the system memory, accessing the AHB bus (write-through policy). Every other processor in the system, for instance processor Y, monitors the AHB bus for writes. If processor Y detects that some other processor has performed a write of a memory element that is, at that moment, in cache(Y), then processor Y marks the corresponding line of its cache as 'invalid'. Reading a memory address that is in a cache line marked as 'invalid', will result in a read miss. In this way cache coherency is guaranteed.

To perform cache snooping, every LEON3 processor, which is a master on the AHB bus, has also access to the slave inputs of the AHB bus. It is not considered by the system a slave and it uses the slave input lines only to perform snooping. In the next section, it will be clear how the AMBA AHB signals are structured, as master outputs and inputs, and slave outputs and inputs. Further details of the signals as they are implemented in the LEON 3 system and GRLIB's AHB Controller are provided in Section 1.5.

1.4 The AMBA AHB 2.0 Architecture

The Advanced Microcontroller Bus Architecture (AMBA) is a set of architectural and protocol specifications widely used in the industry. Since its inception, its scope has gone far beyond microcontroller devices and is now widely used in a number of ASIC and SoC parts, including most of the application processors used in modern portable devices such as smartphones and tablets. It was first introduced by ARM, and in its second version expanded its specification to include the AMBA High-Performance Bus (AHB), a single clock-edge protocol, which is one of the subjects of this thesis. The AMBA protocol is today the de facto standard for 32-bit embedded processors, as it is open source, well documented and finally usable without royalties. Another important feature of the standard is that it promotes reusability of designs by defining a common back-bone through specifications.

The AMBA Specification 2.0 (6) defines an on-chip communication standard for designing high-performance embedded microcontrollers. Three distinct buses are defined within the AMBA specification:

- the Advanced High-performance Bus (AHB)
- the Advanced System Bus (ASB)
- the Advanced Peripheral Bus (APB)

We will focus on AHB, as it is the standard used in the LEON 3 processor for communication among cores, memories and high speed peripherals. This section will give a not-too-brief introduction to the wider AHB 2.0 standard, to make matters covered in detail in the next chapters more comprehensible.

1.4.1 Introducing the AMBA AHB

The AMBA AHB is a high-performance, high clock frequency system backbone bus for system modules. AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macrocell functions. AHB is also specified to ensure ease of use in an efficient design flow using synthesis and automated test techniques. AMBA AHB implements many features required for high-performance, high clock frequency systems, including:

- burst transfers
- split transactions
- single-cycle bus master handover
- single-clock edge operation
- non-tristate implementation
- wider data bus configurations (64/128 bits).

An AMBA AHB design may contain one or more bus masters, typically a system would contain at least the processor and test interface. However, it would also be common for a Direct Memory Access (DMA) or Digital Signal Processor (DSP) to be included as bus masters.

The external memory interface, APB bridge and any internal memory are the most common AHB slaves. Any other peripheral in the system could also be included as an AHB slave. However, low-bandwidth peripherals typically reside on the APB.

A typical AMBA AHB system design contains the following components:

AHB master	A bus master is able to initiate read and write operations by providing an address and control information. Only one bus master is allowed to actively use the bus at any one time.
AHB slave	A bus slave responds to a read or write operation within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer.
AHB arbiter	The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as highest priority or fair access can be implemented depending on the application requirements. An AHB would include only one arbiter, although this would be trivial in single bus master systems.
AHB decoder	The AHB decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer. A single centralized decoder is required in all AHB implementations.

It is important to note that the GRLIB implementation of the AHB arbiter, also includes the AHB decoder and bus structure, all of which is condensed in the AHB Arbiter entity. In the GRLIB framework, each slave and master provide the plug&play information statically by the means of a configuration record that goes to the AHB Controller. The plug&play information includes the memory mapping areas, cacheability, device identification, and the like. This mechanism will be further explored in Section 1.5.

1.4.2 Bus Interconnection

The AMBA AHB bus protocol is designed to be used with a central multiplexer interconnection scheme. Using this scheme all bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all of the slaves. A central decoder is also required to control the read data and response signal multiplexer, which selects the appropriate signals from the slave that is involved in the transfer. Figure 5 illustrates the structure required to implement an AMBA AHB design with three masters and four slaves.

1.4.3 Overview of AMBA AHB operation

Before an AMBA AHB transfer can commence the bus master must be granted access to the bus. This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master will be granted use of the bus.

A granted bus master starts an AMBA AHB transfer by driving the address and control signals. These signals provide information on the address, direction and width of the transfer, as well as an indication if the transfer forms part of a burst. Two different forms of burst transfers are allowed:

- incrementing bursts, which do not wrap at address boundaries
- wrapping bursts, which wrap at particular address boundaries.

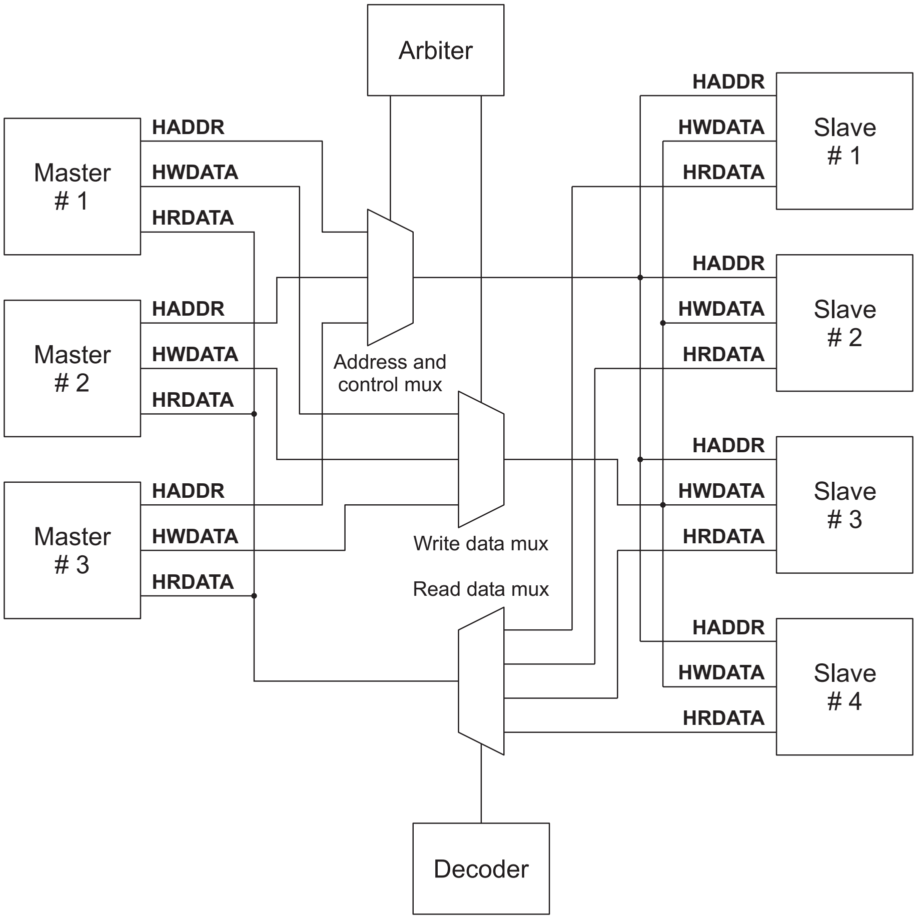

Figure 5. Multiplexer interconnection

A write data bus is used to move data from the master to a slave, while a read data bus is used to move data from a slave to the master.

Every transfer consists of:

- an address and control cycle
- one or more cycles for the data.

The address cannot be extended and therefore all slaves must sample the address during this time. The data, however, can be extended using the **HREADY** signal. When LOW this signal causes wait states to be inserted into the transfer and allows extra time for the slave to provide or sample data.

During a transfer the slave shows the status using the response signals, **HRESP[1:0]**:

OKAY	The OKAY response is used to indicate that the transfer is progressing normally and when **HREADY** goes HIGH this shows the transfer has completed successfully.
ERROR	The ERROR response indicates that a transfer error has occurred and the transfer has been unsuccessful.
RETRY and SPLIT	Both the RETRY and SPLIT transfer responses indicate that the transfer cannot complete immediately, but the bus master should continue to attempt the transfer.

In normal operation a master is allowed to complete all the transfers in a particular burst before the arbiter grants another master access to the bus. However, in order to avoid excessive arbitration latencies, it is possible for the arbiter to break up a burst and in such cases the master must re-arbitrate for the bus in order to complete the remaining transfers in the burst.

1.4.4 Basic transfer

An AHB transfer consists of two distinct sections:

- The address phase, which lasts only a single cycle.
- The data phase, which may require several cycles. This is achieved using the **HREADY** signal.

Figure 6 shows the simplest transfer, one with no wait states.

In a simple transfer with no wait states:

- The master drives the address and control signals onto the bus after the rising edge of **HCLK**.
- The slave then samples the address and control information on the next rising edge of the clock.
- After the slave has sampled the address and control it can start to drive the appropriate response and this is sampled by the bus master on the third rising edge of the clock.

This simple example demonstrates how the address and data phases of the transfer occur during different clock periods. In fact, the address phase of any transfer occurs

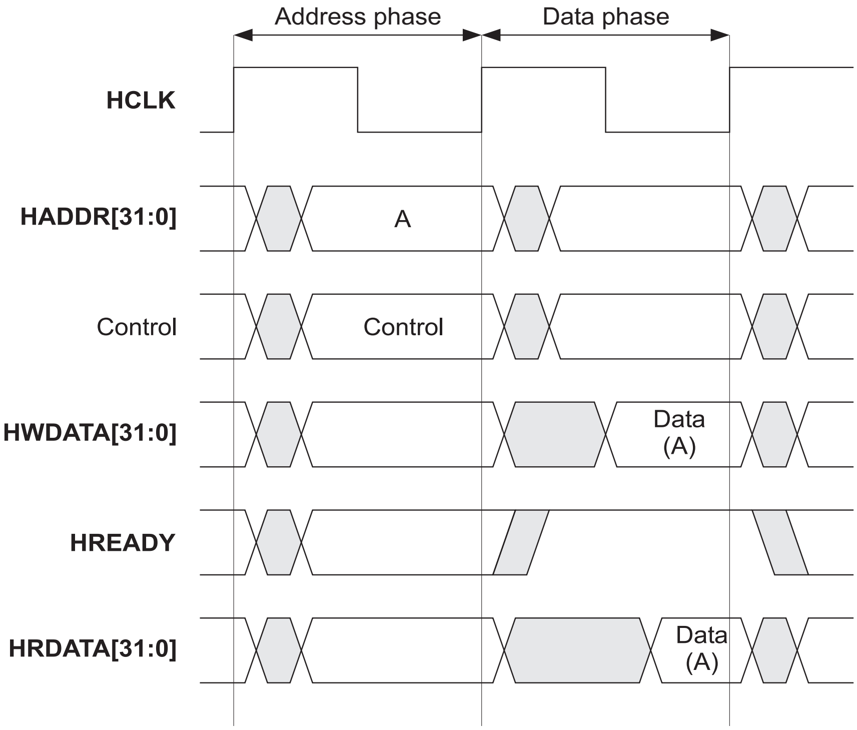

Figure 6. Simple transfer

during the data phase of the previous transfer. This overlapping of address and data is fundamental to the pipelined nature of the bus and allows for high performance operation, while still providing adequate time for a slave to provide the response to a transfer.

A slave may insert wait states into any transfer, as shown in Figure 7, which extends the transfer allowing additional time for completion.

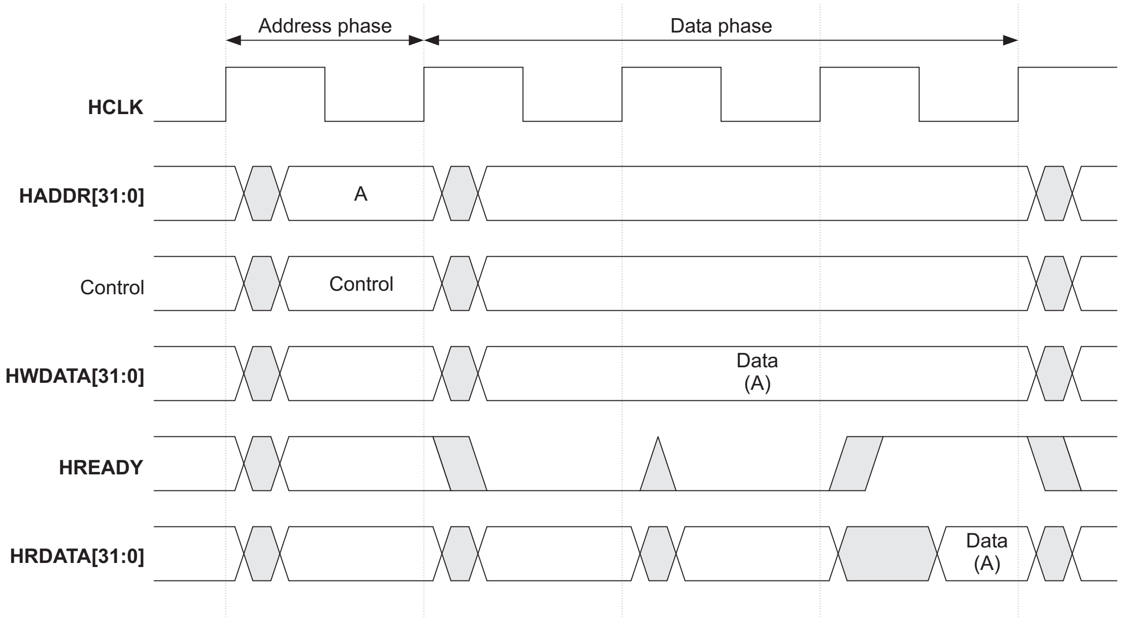

Figure 7. Transfer with wait states

Note: For write operations the bus master will hold the data stable throughout the extended cycles. For read transfers the slave does not have to provide valid data until the transfer is about to complete.

When a transfer is extended in this way it will have the side-effect of extending the address phase of the following transfer. This is illustrated in Figure 8 which shows three transfers to unrelated addresses, A, B & C.

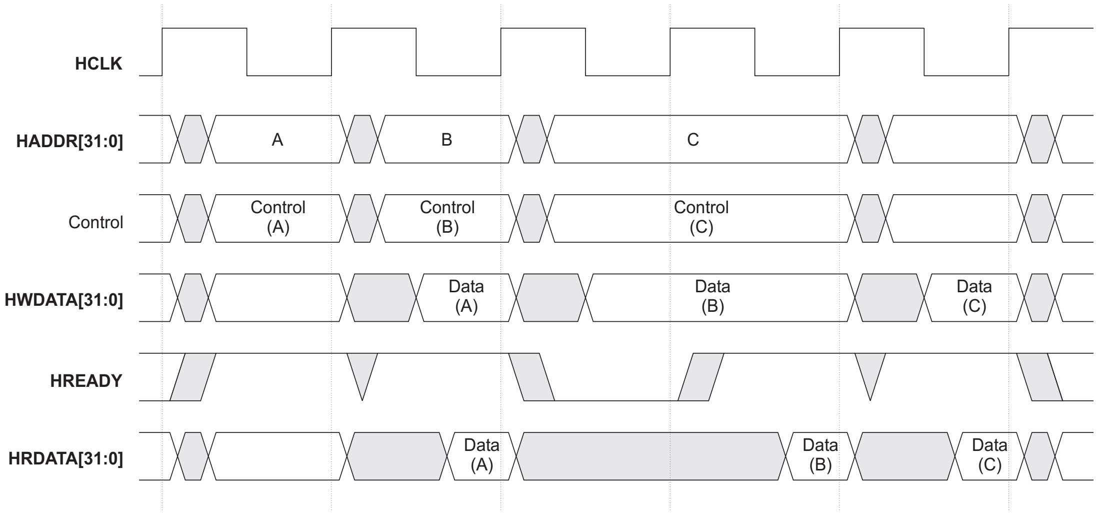

Figure 8. Multiple transfers

In Figure 8:

- the transfers to addresses A and C are both zero wait state
- the transfer to address B is one wait state
- extending the data phase of the transfer to address B has the effect of extending the address phase of the transfer to address C.

1.4.4.1 Transfer type

Every transfer can be classified into one of four different types, as indicated by the **HTRANS**[**1:0**] signals as shown in Table I.

1.4.5 Burst Operation

Four, eight and sixteen-beat bursts are defined in the AMBA AHB protocol, as well as undefined-length bursts and single transfers. Both incrementing and wrapping bursts are supported in the protocol:

- Incrementing bursts access sequential locations and the address of each transfer in the burst is just an increment of the previous address.
- For wrapping bursts, if the start address of the transfer is not aligned to the total number of bytes in the burst (size x beats) then the address of the transfers in the burst will wrap when the boundary is reached. For example, a four-beat wrapping burst of word (4-byte) accesses will wrap at 16-byte boundaries. Therefore, if the start address of the transfer is 0x34, then it consists of four transfers to addresses 0x34, 0x38, 0x3C and 0x30.

TABLE I

TRANSFER TYPE ENCODING

HTRANS[1:0]	**Type**	**Description**
00	IDLE	Indicates that no data transfer is required. The IDLE transfer type is used when a bus master is granted the bus, but does not wish to perform a data transfer. Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer should be ignored by the slave.
01	BUSY	The BUSY transfer type allows bus masters to insert IDLE cycles in the middle of bursts of transfers. This transfer type indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. When a master uses the BUSY transfer type the address and control signals must reflect the next transfer in the burst. The transfer should be ignored by the slave. Slaves must always provide a zero wait state OKAY response, in the same way that they respond to IDLE transfers.
10	NONSEQ	Indicates the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer. Single transfers on the bus are treated as bursts of one and therefore the transfer type is NONSEQUENTIAL.
11	SEQ	The remaining transfers in a burst are SEQUENTIAL and the address is related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). In the case of a wrapping burst the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8 or 16).

Burst information is provided using **HBURST**[**2:0**] and the eight possible types are defined in Table II.

An incrementing burst can be of any length, but the upper limit is set by the fact that the address must not cross a 1kB boundary.

Note: The burst size indicates the number of beats in the burst, not the number of bytes transferred. The total amount of data transferred in a burst is calculated by multiplying the number of beats by the amount of data in each beat, as indicated by **HSIZE**[**2:0**].

All transfers within a burst must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries (that is A[1:0] = 00), halfword transfers must be aligned to halfword address boundaries (that is A[0] = 0).

Bursts must not cross a 1kB address boundary. Therefore it is important that masters do not attempt to start a fixed-length incrementing burst which would cause this boundary to be crossed.

It is acceptable to perform single transfers using an unspecified-length incrementing burst which only has a burst of length one.

1.4.6 Address Decoding

A central address decoder is used to provide a select signal, **HSELx**, for each slave on the bus. The select signal is a combinatorial decode of the high-order address signals,

TABLE II

BURST SIGNAL ENCODING

HBURST[2:0]	**Type**	**Description**
000	SINGLE	Single transfer
001	INCR	Incrementing burst of unspecified length
010	WRAP4	4-beat wrapping burst
011	INCR4	4-beat incrementing burst
100	WRAP8	8-beat wrapping burst
101	INCR8	8-beat incrementing burst
110	WRAP16	16-beat wrapping burst
111	INCR16	16-beat incrementing burst

and simple address decoding schemes are encouraged to avoid complex decode logic and to ensure high-speed operation.

A slave must only sample the address and control signals and **HSELx** when **HREADY** is HIGH, indicating that the current transfer is completing. Under certain circumstances it is possible that **HSELx** will be asserted when **HREADY** is LOW, but the selected slave will have changed by the time the current transfer completes.

The minimum address space that can be allocated to a single slave is 1kB. All bus masters are designed such that they will not perform incrementing transfers over a 1kB boundary, thus ensuring that a burst never crosses an address decode boundary.

In the case where a system design does not contain a completely filled memory map an additional default slave should be implemented to provide a response when any of the nonexistent address locations are accessed. If a NONSEQUENTIAL or SEQUENTIAL transfer is attempted to a nonexistent address location then the default slave should provide an ERROR response. IDLE or BUSY transfers to nonexistent locations should result in a zero wait state OKAY response. Typically the default slave functionality will be implemented as part of the central address decoder.

1.4.7 Slave transfer response

After a master has started a transfer, the slave then determines how the transfer should progress. No provision is made within the AHB specification for a bus master to cancel a transfer once it has commenced.

Whenever a slave is accessed it must provide a response which indicates the status of the transfer. The **HREADY** signal is used to extend the transfer and this works in combination with the response signals, **HRESP[1:0]**, which provide the status of the transfer. The slave can complete the transfer in a number of ways. It can:

- complete the transfer immediately
- insert one or more wait states to allow time to complete the transfer
- signal an error to indicate that the transfer has failed
- delay the completion of the transfer, but allow the master and slave to back off the bus, leaving it available for other transfers.

1.4.7.1 Transfer done

The **HREADY** signal is used to extend the data portion of an AHB transfer. When LOW the **HREADY** signal indicates the transfer is to be extended and when HIGH indicates that the transfer can complete.

Note: Every slave must have a predetermined maximum number of wait states that it will insert before it backs off the bus, in order to allow the calculation of the latency of accessing the bus. It is recommended, but not mandatory, that slaves do not insert more than 16 wait states to prevent any single access locking the bus for a large number of clock cycles.

1.4.7.2 Transfer response

A typical slave will use the **HREADY** signal to insert the appropriate number of wait states into the transfer and then the transfer will complete with **HREADY** HIGH and an OKAY response, which indicates the successful completion of the transfer.

The ERROR response is used by a slave to indicate some form of error condition with the associated transfer. Typically this is used for a protection error, such as an attempt to write to a read-only memory location.

The SPLIT and RETRY response combinations allow slaves to delay the completion of a transfer, but free up the bus for use by other masters. These response combinations are usually only required by slaves that have a high access latency and can make use of these response codes to ensure that other masters are not prevented from accessing the bus for long periods of time.

SPLIT operation support is not mandatory and not covered in this thesis, while we will frequently use the RETRY operation in the dummy slaves that we will describe further on in Chapter 3, a fundamental part of the crossbar design that we will implement.

The encoding of **HRESP**[**1:0**], the transfer response signals, and a description of each response are shown in Table III.

When it is necessary for a slave to insert a number of wait states prior to deciding what response will be given then it must drive the response to OKAY.

1.4.8 Two-cycle response

Only an OKAY response can be given in a single cycle. The ERROR, SPLIT and RETRY responses require at least two cycles. To complete with any of these responses then in the penultimate (one before last) cycle the slave drives **HRESP**[**1:0**] to indicate ERROR, RETRY or SPLIT while driving **HREADY** LOW to extend the transfer for an extra cycle. In the final cycle **HREADY** is driven HIGH to end the transfer, while **HRESP**[**1:0**] remains driven to indicate ERROR, RETRY or SPLIT.

If the slave needs more than two cycles to provide the ERROR, SPLIT or RETRY response then additional wait states may be inserted at the start of the transfer. During this time the **HREADY** signal will be LOW and the response must be set to OKAY.

The two-cycle response is required because of the pipelined nature of the bus. By the time a slave starts to issue either an ERROR, SPLIT or RETRY response then the address

TABLE III

RESPONSE ENCODING

HRESP[1]	**HRESP[0]**	**Response**	**Description**
0	0	OKAY	When **HREADY** is HIGH this shows the transfer has completed successfully. The OKAY response is also used for any additional cycles that are inserted, with **HREADY** LOW, prior to giving one of the three other responses.
0	1	ERROR	This response shows an error has occurred. The error condition should be signaled to the bus master so that it is aware the transfer has been unsuccessful. A two-cycle response is required for an error condition.
1	0	RETRY	The RETRY response shows the transfer has not yet completed, so the bus master should retry the transfer. The master should continue to retry the transfer until it completes. A two-cycle RETRY response is required.
1	1	SPLIT	The transfer has not yet completed successfully. The bus master must retry the transfer when it is next granted access to the bus. The slave will request access to the bus on behalf of the master when the transfer can complete. A two-cycle SPLIT response is required.

for the following transfer has already been broadcast onto the bus. The two-cycle response allows sufficient time for the master to cancel this address and drive **HTRANS**[**1:0**] to IDLE before the start of the next transfer.

For the SPLIT and RETRY response the following transfer must be canceled because it must not take place before the current transfer has completed. However, for the ERROR response, where the current transfer is not repeated, completion of the following transfer is optional.

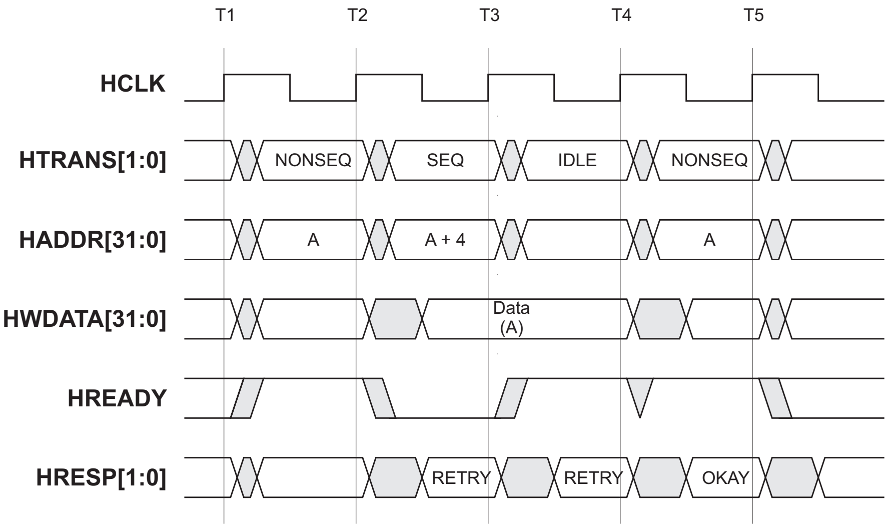

Figure 9. Transfer with retry response

Figure 9 shows an example of a RETRY operation where the following events are illustrated:

- The master starts with a transfer to address A.
- Before the response is received for this transfer the master moves the address on to A + 4.
- The slave at address A is unable to complete the transfer immediately and therefore it issues a RETRY response. This response indicates to the master that the transfer at address A is unable to complete and so the transfer at address A + 4 is canceled and replaced by an IDLE transfer.

Further details and coverage of the AMBA AHB 2.0 standard can be found in (6).

1.5 The AMBA AHB Controller

The AMBA AHB Controller, which is part of GRLIB, is the base upon which I conducted my feasibility studies for the AHB compliant crossbar structure and arbiter which I realized. This Section describes its structure and implementation, together with some key configuration parameters that were mostly transposed and used in my implementation. It is worth noticing that the code style of this entity is the one used in most of GRLIB, and which is described in (7), authored by Jiri Gaisler, one of the main contributors of GRLIB. I will further describe it in Section 2.1.

1.5.1 Overview

The AMBA AHB Controller is a combined AHB arbiter, bus multiplexer and slave decoder according to the AMBA 2.0 standard.

The controller supports up to 16 AHB masters, and 16 AHB slaves. The maximum number of masters and slaves are defined in the GRLIB AMBA package, in the VHDL constants NAHBSLV and NAHBMST. It can also be set with the *nahbm* and *nahbs* VHDL generics.

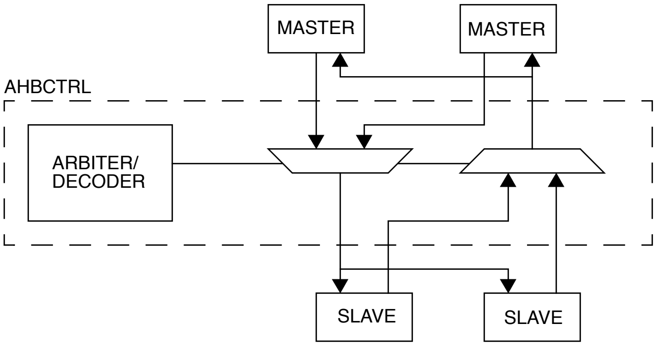

Figure 10. AHB controller block diagram

As shown in Figure 10, the master outputs are fed to the controller module as a vector. Once the master-slave couple is selected by the arbitration logic, the master's output will be selected by the appropriate multiplexer to be routed as the slave input signal. The signal is fed to all the slaves. The same happens for the slave outputs. The master's selected slave is routed by a multiplexer to be the input for all the masters. Only the granted master then reads its input, and only the selected slave reads its own input. The other modules in the system simply ignore their inputs when they are not granted (**HGRANTx** signal for masters) or selected (**HSELx** signal for slaves).

1.5.2 Arbitration

The AHB controller supports two arbitration algorithms: fixed-priority and round-robin (5). The selection is done by the VHDL generic *rrobin*. In fixed-priority mode (*rrobin* = 0), the bus request priority is equal to the master's bus index, with index 0 being the lowest priority. If no master requests the bus, the master with bus index 0 (set by the VHDL generic *defmast*) will be granted.

In round-robin mode, priority is rotated one step after each AHB transfer. If no master requests the bus, the last owner will be granted (bus parking). The VHDL generic *mprio* can be used to specify one or more masters that should be prioritized when the core is configured for round-robin mode.

During incremental bursts, the AHB master should keep the bus request asserted until the last access as recommended in the AMBA 2.0 specification, or it might loose bus ownership. For fixed-length burst, the AHB master will be granted the bus during the full burst, and can release the bus request immediately after the first access has started. For this to work however, the VHDL generic *fixbrst* should be set to 1.

1.5.3 Decoding

Decoding (generation of **HSEL**, see Subsection 1.4.6) of AHB slaves is done using the plug&play method explained in the GRLIB User's Manual (4). A slave can occupy any binary aligned address space with a size of 1 – 4096 Mbyte. A specific I/O area is also decoded, where slaves can occupy 256 byte – 1 Mbyte. The default address of the I/O area is 0xFFF00000, but can be changed with the *ioaddr* and *iomask* VHDL generics. Access to unused addresses will cause an AHB error response.

1.5.4 Plug & Play information

GRLIB devices contain a number of plug&play information words which are included in the AHB records they drive on the bus (see the GRLIB user's manual (4) for more information). These records are combined into an array which is connected to the AHB controller unit. The plug&play information is mapped on a read-only address area, defined by the *cfgaddr* and *cfgmask* VHDL generics, in combination with the *ioaddr* and *iomask* VHDL generics. By default, the area is mapped on address 0xFFFFF000 – 0xFFFFFFFF. The master information is placed on the first 2 kbyte of the block (0xFFFFF000 – 0xFFFFF800), while the slave information is placed on the second 2 kbyte block. Each unit occupies 32 bytes, which means that the area has place for 64 masters and 64 slaves. The address of the plug&play information for a certain unit is defined by its bus index. The address for masters is thus 0xFFFFF000 + n*32, and 0xFFFFF800 + n*32 for slaves.

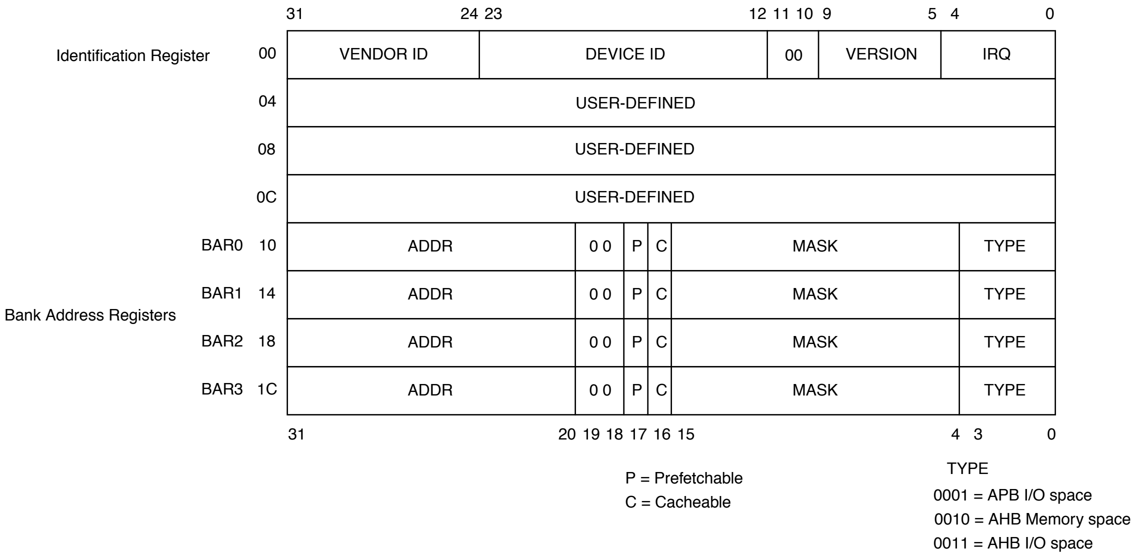

Figure 11. AHB plug&play information record

Further details, like configuration options, signal descriptions and how to instantiate the entity in a project can be found in (5).

CHAPTER 2

APPROACH

In order to complete the task of building and integrating my projected design entity inside GRLIB, I decided to base all of my work on an already available and productive design methodology. I chose to adhere to the style in which most of GRLIB is already written, that is the structured VHDL approach described by Jiri Gaisler [1] (7).

Courtesy of my home university, "Politecnico di Torino", I was given access to a license of Mentor Graphics' ModelSim and a remote account on a workstation on the internal university network. This chapter will describe in detail how I proceeded in laying down the basis for the project and creation of my final entity.

2.1 Structured VHDL method

The VHDL language was developed to allow modeling of digital hardware. When the language was first put to use, it was used for high-level behavioral simulation only (8). 'Synthesis' into VLSI devices was carried out by manually converting the models into schematics using gates and building blocks from a target library. However, manual conversion tended to be error-prone, and was likely to invalidate the effort of system simulation. To address this problem, VHDL synthesis tools that could convert VHDL code directly to

---

[1] the founder of Gaisler Research, upon which **GR**LIB takes its name.

a technology netlist started to emerge on the market at the beginning of 1990's. Since the VHDL code could now be directly synthesized, the development of the models was primarily made by digital hardware designers rather than software engineers. Hardware engineers were used to schematic entry as design method, and their usage of VHDL resembled the dataflow design style of schematics. The functionality was coded using a mix of concurrent statements and short processes, each describing a limited piece of functionality such as a register, multiplexer, adder or state machine. In the early 1990's, such a design style was acceptable since the complexity of the circuits was relatively low (<50 Kgates) and the synthesis tools could not handle more complex VHDL structures. However, today the device complexity can reach several millions of gates, and synthesis tools accept a much larger part of the VHDL standard. It should therefore be possible to use a more modern and efficient VHDL design method than the traditional 'dataflow' version.

In order to overcome the limitations of this classical 'dataflow' design style, a 'two-process' coding method is proposed: one process contains all combinational logic, whereas the other process infers all (and only) the registers.

2.1.1 Traditional VHDL design methodology

The most commonly used design 'style' for synthesizable VHDL models is what can be called the 'dataflow' style. A larger number of concurrent VHDL statements and small processes connected through signals are used to implement the desired functionality. Reading and understanding dataflow VHDL code is difficult since the concurrent statements and processes do not execute in the order they are written, but whenever any one of their input signals changes value. It is not uncommon that to extract the functionality of dataflow code, a block diagram has to be drawn to identify the dataflow and dependencies among the statements. The readability of dataflow VHDL code can be compared to an ordinary schematic where the wires connecting the various blocks have been removed, and the block inputs and outputs are just labeled with signal names!

A problem with the dataflow method is also the low abstraction level. The functionality is coded with simple constructs typically consisting of multiplexers, bit-wise operators and conditional assignments (if-then-else). The overall algorithm (e.g. non-restoring division) might be very difficult to recognize and debug.

Yet, there is another issue is simulation time: the assignment of a signal takes approximately 100 times longer than assigning a variable in a VHDL process. This is because the various signal attributes must be updated, and the driving event added to the event queue. With many concurrent statements and processes, a larger proportion of the simulator time will be spent managing signals and scheduling processes and concurrent statements.

2.1.2 Proposed structured design method

To overcome the limitations of the dataflow design style, a new ‘two-process’ coding method is proposed. The method is applicable to any synchronous single-clock design, which represents the majority of all designs. The goal of the two-process method is to:

- Provide uniform algorithm encoding
- Increase abstraction level

- Improve readability
- Clearly identify sequential logic
- Simplify debugging
- Improve simulation speed
- Provide one model for both synthesis and simulation

The above goals are reached with surprisingly simple means:

- Using record types in all port and signal declarations
- Only using two processes per entity
- Using high-level sequential statements to code the algorithm

The biggest difference between a program in VHDL and standard programming language like C, is that VHDL allows concurrent statements and processes that are scheduled for execution by events rather than in the order they are written. Indeed, this reflects the dataflow behavior of real hardware, but becomes difficult to understand and analyze when the number of concurrent statements passes some threshold (e.g. 50). On the other hand, analyzing the behavior of programs written in sequential programming languages does not become a problem even if the program tends to grow, since there is only one thread of control and execution is done sequentially from top to bottom.

In order to improve readability and provide a uniform way of encoding the algorithm of a VHDL entity, the two-process method only uses two processes per entity: one process that contains all combinational (asynchronous) logic, and one process that contains all sequential logic (registers). Using this structure, the complete algorithm can be coded in sequential (non-concurrent) statements in the combinational process while the sequential process only contains registers, i.e. the state.

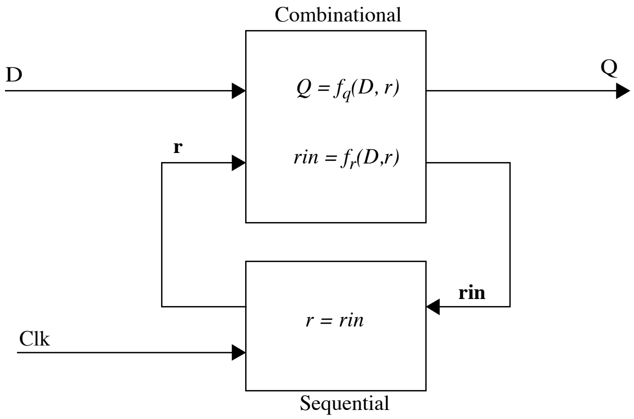

Figure 12. Generic two-process circuit

Figure 12 shows a block diagram of a two-process entity. Inputs to the entity are denoted by D and connected to the combinational process. The inputs to the sequential process are denoted by rin and driven by the combinational process. In the sequential process, the inputs (rin) are copied to the outputs (r) on the clock edge.

The functionality of the combinational process can be described by means of two equations:

$$Q = f_q(D, r)$$

$$rin = f_r(D, r)$$

Given that the sequential process only performs a latching of the state vector, the two functions are enough to express the overall functionality of the entity.

2.1.3 Using record types

The port interface list can, for complex IP blocks, consist of several hundreds of signals. Using the standard dataflow method, the signals are not grouped into more complex data types but just listed sequentially. The most common data types are scalar types and one-dimensional arrays (buses). Having a port list of several hundreds of signals makes it difficult not only to understand which signals functionally belong together, but also to add and remove signals. Each modification to the interface list has to be made at three separate locations: the entity declaration, the entity's component declaration, and the component instantiation (adding a port map).

By using record types to group associated signals, the port list becomes both shorter and more readable. The signals are grouped according to functionality and direction (in or out). The record types can be declared in a common global 'interface' package which is imported in each module. Alternatively, the record types can be declared together with the entity's component declaration in a 'component' package. This package is then imported into those modules where the component is used. A modification to the interface list using record types corresponds to adding or removing an element in one of the record types. This is done only in one single place, the package where the record type is declared. Any changes to this package will automatically propagate to the component declaration and the entity's component instantiation, avoiding time-consuming and error-prone manual editing. Similar problems arise when more registers are added. For each register, two signals have to be declared (register input and output), the register output signal has to be added to the sensitivity list of the combinational process, and an assignment statement added to the sequential process. By grouping all signals used for registers into one record type, this becomes unnecessary. The *rin* and *r* signals becomes records, and adding register is done by simply adding a new element in the register record type definition.

Below is the *count8* example using records for port and register signals. The load and count inputs are now latched before being used, and a zero flag has been added:

```
library ieee;
use ieee.std_logic_1164.all;
package count8_comp is          -- component declaration package
  type count8_in_type is record
    load : std_logic;
    count : std_logic;
    din : std_logic_vector(7 downto 0);
  end;

  type count8_out_type is record
    dout : std_logic_vector(7 downto 0); zero : std_logic;
  end;

  component count8
    port (
      clk : in std_logic;
      d : in count8_in_type; q : out count8_out_type);
  end component;
end package;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.count8_comp.all;

entity count8 is
  port (
    clk : in std_logic;
    d : in count8_in_type; q : out count8_out_type);
end;

architecture twoproc of count8 is
  type reg_type is record
    load : std_logic;
    count : std_logic;
    zero : std_logic;
    cval : std_logic_vector(7 downto 0);
  end;
  signal r, rin : reg_type;
begin
  comb : process(d, r)                                -- combinational process
    variable v : reg_type;
  begin
    v := r;                                           -- default assignment
    v.load := d.load; v.count := d.count;             -- overriding assignments
    v.zero := '0';
    if r.count = '1' then v.cval := r.cval + 1; end if; -- module algorithm
    if r.load = '1' then v.cval := d.data; end if;
    if v.cval = "00000000" then v.zero := '1'; end if;
    rin <= v;                                         -- drive register inputs
    q.dout <= r.cval; q.zero <= r.zero;               -- drive module outputs
  end process;

  regs : process(clk)                                 -- sequential process
  begin
    if rising_edge(clk) then r <= rin; end if;
  end process;
end;
```

Note the usage of variable *v* in the combinational process. The variable is of the register record type, and assigned at the beginning of the process with the value of *r*, i.e. the current register values. At the end of the process, the register inputs *rin* are assigned with *v*. This means that those elements of *v* which are not assigned during execution of the process will maintain their values, i.e. the register value will not change.

A large benefit with using record types for the register signals *rin* and *r*, is that elements can be added or removed without requiring any other modifications to the code. The sensitivity list of the combinational process does not have to be modified, and neither does the assignment of $r <= rin$ in the sequential process. This is because the operation is performed on the record as a whole, regardless of how many elements it has. In larger blocks with many registers, readability can be improved by defining separate record types for related registers.

The presented two-process method is a way of producing structured and readable VHDL code, suitable for efficient simulation and synthesis. By defining a common coding style, the algorithm can be easily identified and the code analyzed and maintained also by other engineers rather than the main designer. Using sequential VHDL statements to code the algorithm also allows the use of complex statements and a higher abstraction level. Debugging and analysis is simplified due to the serial execution of statements, rather than the parallel flow used in dataflow coding.

2.2 Framework structure and compilation

The basis for my practical understanding of the AMBA protocol was the behavior of the LEON3 processor. I run several simulations of a working LEON system and I analyzed the behavior of the AMBA bus signals. To do so, I configured the GRLIB framework to work in the ModelSim simulation environment. My objective was to configure a minimal LEON 3 system with a minimal set of modules plugged on the AHB bus, to be able to analyze the system's behavior in an easier way. This basic system needs to have two processing cores, to inquire the multi-processing aspects of the LEON 3 system.

The system hardware simulation provided by the framework includes a test bench which runs some simulated software tests on internal components on the simulated system. This tests were replaced with some simplified ad-hoc software, a simple 'hello world' program. The reason behind this was to study the compilation method used by the LEON 3 and to find a way to run software on the simulated system for possible further development of my work. The library organization and the process of configuration of the library are detailed below.

2.2.1 Directory organization

GRLIB is organized around VHDL libraries, where each IP vendor is assigned a unique library name. Each vendor is also assigned a unique subdirectory under *grlib/lib* in which all vendor-specific source files and scripts are contained. The vendor-specific directory can contain subdirectories, allowing for further partitioning among IP cores etc.

The basic directories delivered with GRLIB under *grlib-1.0.x/lib* are (4):

grlib	packages with common data types and functions
gaisler	Gaisler Research's components and utilities
tech/*	target technology libraries for gate level simulation

techmap	wrappers for technology mapping of macro cells (RAM, pads)
work	components and packages in the VHDL work library

Other vendor-specific directories are also delivered with GRLIB, but are not necessary for understanding the design concept. Libraries and IP cores are described in detail in separate documentation (5).

2.2.2 LEON 3 Overview

Implementing a LEON 3 system is typically done by using one of the template designs on the designs directory. I chose to start my configuration building on the already available LEON 3 template design for the GR-XC3S-1500 board. Implementation is typically done in three basic steps:

- Configuration of the design using *xconfig*
- Simulation of design and test bench
- Synthesis and place&route

The template design is located in *designs/leon3-gr-xc3s-1500*, and is based on three files:

config.vhd	a VHDL package containing design configuration parameters. Automatically generated by the *xconfig* GUI tool.
leon3mp.vhd	contains the top level entity and instantiates all on-chip IP cores. It uses *config.vhd* to configure the instantiated IP cores.

testbench.vhd	test bench with external memory, emulating the GR-XC3S-1500 board.

Each core in the template design is configurable using VHDL generics. The value of these generics is assigned from the constants declared in *config.vhd*, created with the *xconfig* GUI tool.

2.2.3 LEON 3 Configuration

By issuing the command ‘make xconfig’ in a remote bash shell on the workstation, I launched the *xconfig* GUI tool and modified the LEON 3 template design (the GR-XC3S-1500 board). When the configuration is saved and *xconfig* is exited, the *config.vhd* is automatically updated with the selected configuration. I configured a basic LEON 3 processor, disabling every module on the AHB bus that was unnecessary to a basic configuration. The AHB modules that I retained in my final system design are the following:

- two LEON 3 cores
- the AHB Controller
- the SDRAM memory controller
- the LEON 3 Debug Support Unit
- the AHB/APB bridge

In addition the following APB modules were enabled:

- the Generic UART

- the Multi-processor Interrupt Controller
- the Modular Timer Unit

2.2.4 Compilation of ad-hoc software

All software related to LEON 3 processor resides in the *grlib/software/leon3* directory. This directory holds the source code for the boot loader and the totality of the hardware tests. This is also where I will keep future code which I intend to run on the machine. The Makefile of this directory ensures that files will be compiled with the sparc-elf version of GCC, and that the resulting executable and library files are packed into an S-record binary file, an ASCII hexadecimal binary format that can be loaded directly into the PROM/SDRAM. The sparc-elf version of GCC is part of BCC, the LEON Bare-C Cross Compilation System, an essential part of the LEON toolchain. To build the software package for the LEON 3 core, I had to run the 'make soft' command from inside the *designs/leon3-gr-xc3s-1500* directory. This command compiles a simple boot loader from *prom.s* and the hardware tests. Everything is packed into a binary. The boot loader is loaded into PROM, while the software is loaded into both SDRAM and SRAM. Eventually, I modified the Makefile to build my Hello world software program and have it automatically create the S-record file which is loaded during test bench simulation of my basic system. The addition to the Makefile of the *software/leon3* directory is the following:

```
hello.exe: hello.c
  $(XCC) $(XCFLAGS) $(VPATH)/hello.c $(XLDFLAGS) -o hello.exe

my-soft: prom.srec my-sram.srec my-sdram.srec
```

```
my-sram.srec: hello.exe
  sparc-elf-objcopy -O srec hello.exe sram.srec

my-sdram.srec: hello.exe
  sparc-elf-objcopy -O srec hello.exe sdram.srec
```

My Hello world program consists of a single printf line. Again, by running the ‘make my-soft’ command from inside the *designs/leon3-gr-xc3s-1500* directory, I compiled and built the SDRAM image which was then executed during the simulation.

2.2.5 Simulation

I simulated the design in a test bench emulating the basic prototype board described in Subsection 2.2.3. The test bench includes an external SDRAM which is pre-loaded with the ad-hoc program. The ‘make vsim’ command was issued to compile the VHDL code inside ModelSim, then the ‘vsim testbench’ command started the simulation of my design. A simulation log is shown below, including, at the very end, the output of my Hello world program [1].

```
# LEON3 GR-XC3S-1500 Demonstration design
# GRLIB Version 1.1.0, build 4113
# Target technology: spartan3  , memory library: spartan3
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 2, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Gaisler Research       LEON3 SPARC V8 Processor
# ahbctrl: mst1: Gaisler Research       LEON3 SPARC V8 Processor
# ahbctrl: slv0: European Space Agency  LEON2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
```

---

[1] The accelerated UART tracing option was enabled before compiling the VHDL system. The option ensures a faster hardware simulation of software functions like printf.

```
# ahbctrl:        memory at 0x20000000, size 512 Mbyte
# ahbctrl:        memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Gaisler Research        AHB/APB Bridge
# ahbctrl:        memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv2: Gaisler Research        LEON3 Debug Support Unit
# ahbctrl:        memory at 0x90000000, size 256 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   LEON2 Memory Controller
# apbctrl:        I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Gaisler Research        Generic UART
# apbctrl:        I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Gaisler Research        Multi-processor Interrupt Ctrl.
# apbctrl:        I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Gaisler Research        Modular Timer Unit
# apbctrl:        I/O ports at 0x80000300, size 256 byte
# gptimer3: GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 3, #cpu 2, eirq 0
# apbuart1: Generic UART rev 1, fifo 4, irq 2, scaler bits 12
# dsu3_2: LEON3 Debug support unit + AHB Trace Buffer, 2 kbytes
# leon3_1: LEON3 SPARC V8 processor rev 0
# leon3_1: icache 2*4 kbyte, dcache 1*4 kbyte
# leon3_0: LEON3 SPARC V8 processor rev 0
# leon3_0: icache 2*4 kbyte, dcache 1*4 kbyte
# clkgen_spartan3e: spartan3/e sdram/pci clock generator, version 1
# clkgen_spartan3e: Frequency 50000 KHz, DCM divisor 4/5
# Hello World from Polito!
#
# ** Failure: *** IU in error mode, simulation halted ***
#    Time: 311923 ns  Iteration: 0  Process: /testbench/iuerr File: testbench.vhd
```

Note that the simulation is terminated by generating a VHDL failure, which is the only way of stopping the simulation from inside the model.

CHAPTER 3

IMPLEMENTATION

This chapter will deal with the problems which were encountered while translating a protocol that was specifically created to work on a bus, the AMBA AHB protocol, on a concurrent parallel communication channel like a crossbar. Initially, I studied how the AHB Controller functioned, and using that module as an inspiration, I created the AHB Crossbar Controller.

3.1 Communication on the crossbar switch

In the standard AHB Controller bus design every master's output and every slave's output passes through a multiplexer and go to the slave input and the master input, respectively (confront Figure 5 on page 15). The master input and slave input signals are shared among the masters and among the slaves respectively. The crossbar switch topology that we are going to implement, on the other hand, connects every master to every slave. In the crossbar switch topology adaptation of the AHB Controller there will be an array of master inputs and slave inputs. The masters and slaves are connected by a crossbar switch, which is, in turn, made of a series of multiplexers. In a system with N masters and M slaves, there will be N+M multiplexers in total. An example crossbar switch network topology with three masters and four slaves is shown in Figure 13.

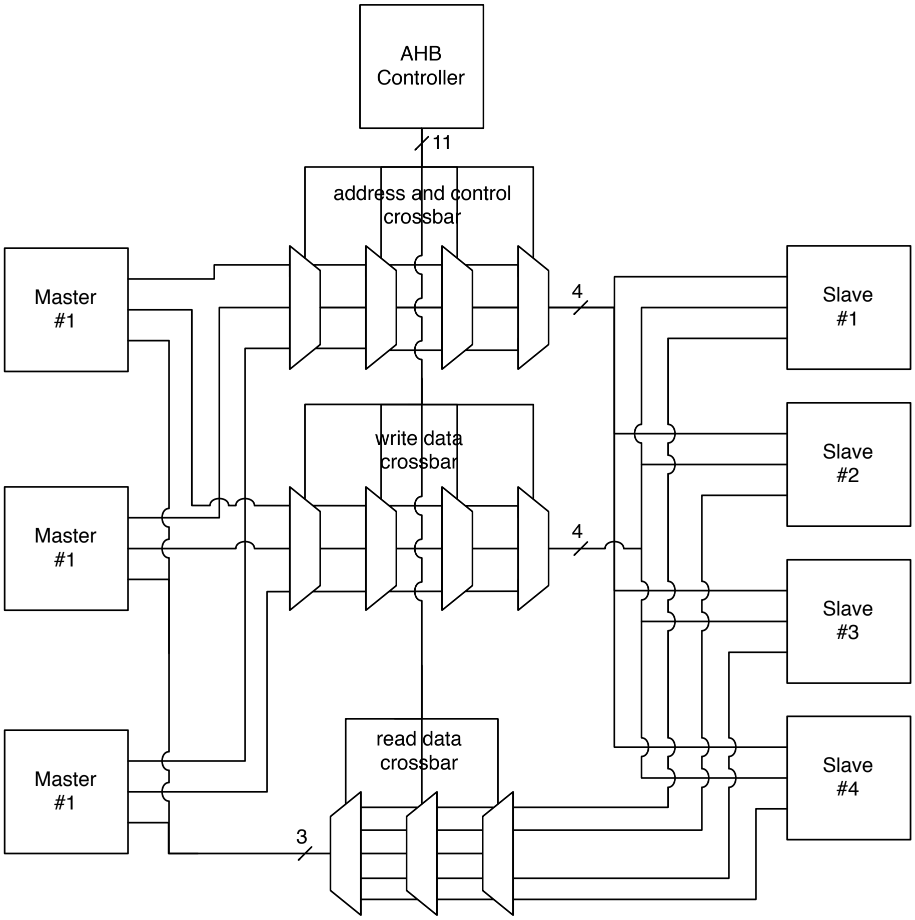

Figure 13. AHB Crossbar Topology

As seen in Section 1.4, every AHB protocol transfer consists of two different stages: the control cycle and the data cycle. During the control cycle the communication is mono-directional, from master to slave, and is handled by the crossbar mux shown at the top of Figure 13. The data cycle is bi-directional and the connection between the master and the slave (and vice versa) is handled by the middle and bottom multiplexers of Figure 13. Being bi-directional means that, for instance, if master 0 output is connected to slave 2 input, then slave 2 output is connected to master 0 input. At a certain instant of time, several concurrent connections between masters and slaves can happen in a crossbar switch. We will now show in detail how the crossbar switch network is implemented in hardware by means of multiplexers.

Figure 14 shows an example configuration with three masters and two slaves. In this case, we just show the data/cycle part of the crossbar, which is bi-directional. The control cycle crossbar can be easily extrapolated from this explanation. In this example configuration, we can see that the array of master outputs goes through two multiplexers, one for each slave input. As can be seen in the left part of Figure 14, a selection command, issued by the AHB Crossbar Controller's arbiter, routes the master output #0 to slave input #1, as highlighted by the green colored connection. Another selection command routes master output #2 to slave input #0, highlighted in blue. This completes the master-to-slave part of the crossbar. At the same time the slave-to-master connection must be connected coherently, as seen in the right part of Figure 14. Therefore two selection commands route the slave outputs to the correct master inputs, as highlighted by the blue and green colored

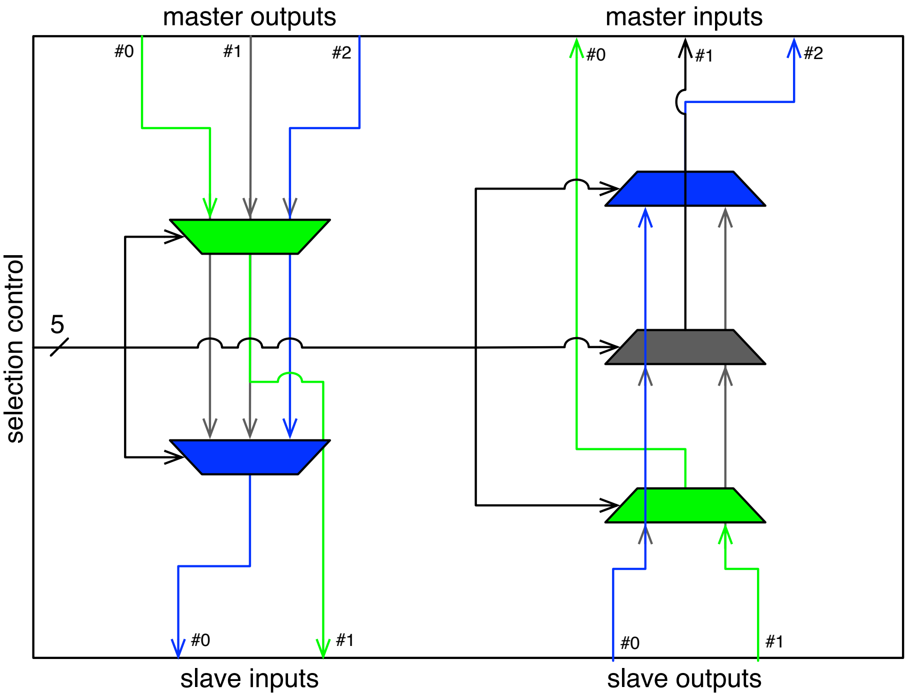

Figure 14. AHB Crossbar Topology Detail

wires. Master #1 input is not connected to a slave output, and is connected by the AHB Crossbar Controller to a dummy default slave output (not shown in figure).

Now that we can connect multiple masters to multiple slaves concurrently, we will have to deal with the problem of collisions. A collision happens when two masters will try to access the same slave simultaneously. Clearly, for protocol coherency reasons, this cannot happen.

3.2 Concurrent access and the AHB protocol

In a shared bus architecture, concurrent access to the same slave is dealt by limiting the use of the bus to one master at a time. As only one master can use the bus at every moment, it is not possible to have collisions. In a crossbar switch network topology this is not guaranteed by design. Our arbiter will have to manage collisions. As seen in Subsection 1.4.4, the AHB protocol is a two-phased protocol. Once the master is granted, it will output the address and control signals at the first phase, and then the read or write data at the second phase. From now on we will refer to this phases as the *control phase* and the *data phase*. With the bus topology, the prerequisite for a master to access the shared bus, is that the master be granted the bus. The granting process is the preliminary phase for any transfer on the bus. Subsequently a master requests a slave by outputting an address on the control bus during the control phase. The address is decoded, using the plug&play information, and the correspondingly mapped slave is then known.

But in order to detect collisions we have to control, at any moment, every master's request for any slave. Therefore, because a master can request a slave only when it is granted, all masters must be granted at any time in the crossbar system. In this way, when a master requests a slave, it does not need to initiate the granting process. It can directly output the address corresponding to the slave on the control bus.

My AHB Crossbar Controller therefore always grants every master access to the bus, it then proceeds at monitoring the control bus, for any control phase access by any master. By doing this, we are able to detect when two masters try to access the same slave by outputting on the bus an address that is mapped to said same slave.

3.3 Resolving collisions

The AHB protocol requires that the slave sample the address and control information during the *control phase* when the ready signal is high. But the AHB bus is pipelined, and the ready signal has the timing of the *data phase.* It is then output by the slave which was contacted by the master at the previous *control phase.* A simple basic transfer which involves two different slaves is shown in Figure 15.

The address and control signals cannot be extended and so must be sampled whenever they are present. This raises a problem when a collision occurs. In fact, whenever two masters are requesting the same slave, as shown in Figure 16, one of the two masters will have access to the slave, while the other master will have to be denied somehow. As we cannot extend the control phase of the denied master (confront Subsection 1.4.3), said master will have to retry the transfer at a later time. To implement this behavior we will connect the denied master to a *dummy slave* that will send the master a RETRY response.

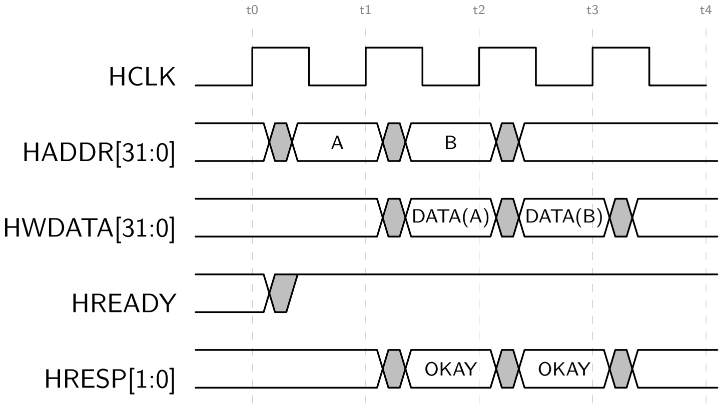

Figure 15. Transfer of data between one master and two different slaves on a bus

In this way the denied master will try again the transfer after two cycles (see two-cycle response in Subsection 1.4.8).

3.4 Dummy slaves

I envisioned the use of dummy slaves as this method is already used in the standard AHB Controller during some operations. For instance, a master which issues an address on the control bus that is not recognized by the plug&play decoder, is given an ERROR response by a dummy slave whose logic is situated on the AHB Controller. We expanded this concept and implemented several dummy slaves, implementing several different kind of responses. Each master can be connected to a dummy slave to have a certain type of response. In our implementation of the AHB Crossbar Controller we have, for each master, one dummy slave for each type present in the following list:

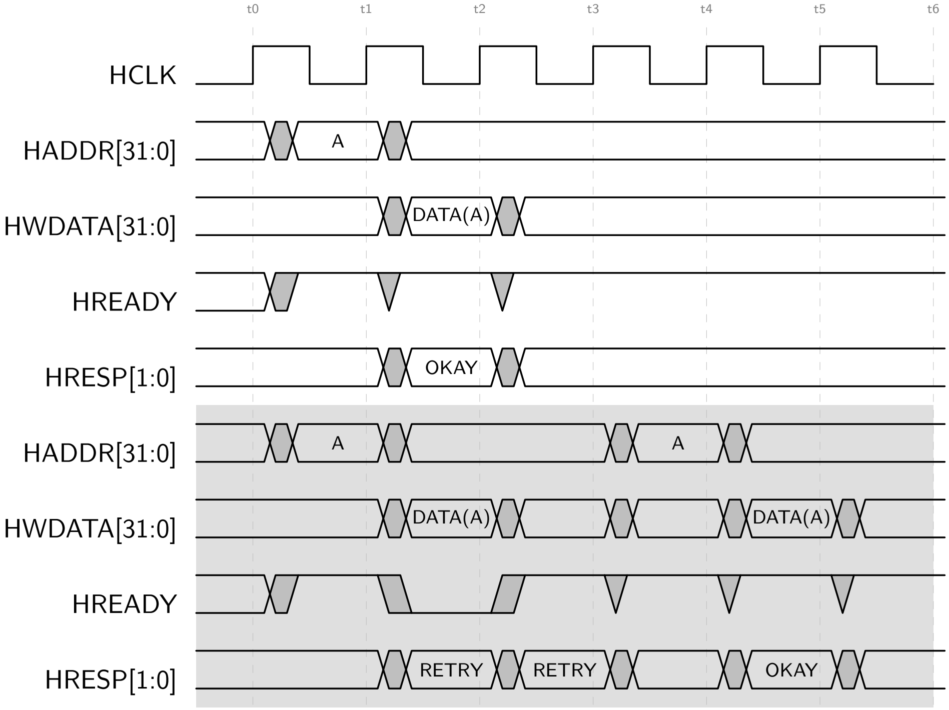

Figure 16. Two masters requesting the same slave concurrently

- retry slave
- error slave
- configuration slave.

The retry and error slaves will give a RETRY or ERROR type response, according to the two cycle response scheme that is discussed in Subsection 1.4.8. The configuration slave is used by the masters for read access of the plug&play information stored on the AHB Crossbar Controller. Concurrent access by several masters of this area is possible without collision or coherency problems because it is a read-only area.

3.5 Wait state behavior

A master is assigned a slave during the *control phase*, it will then sample the data at the following clock cycle during the *data phase*. The slave can delay the *data phase* by asserting a low ready signal. This cycle is called a 'wait state'. During a wait state the slave will not sample the *control phase* signals of the master. The master's control signals will be sampled when the ready signal is asserted high again. The control signal are ignored during a wait state, so in our AHB Crossbar Controller a master is stuck to a slave while the slave is in a wait state.

3.6 Arbitration

Arbitration in the AHB Crossbar Controller works accordingly to a round-robin scheme. This scheme was chosen to avoid starvation, a particular problem in parallel systems which arises whenever a master is never granted a resource. The round robin arbitration scheme here presented works according to a per-slave basis. During arbitration each slave will be assigned to the next requesting master, starting from master 0, if more than one master is requesting it.

An example is shown in Figure 17: master 0 is granted slave 0 at the cycle right before **t1** as it is the only master requesting it in the system. It receives an OKAY response at the corresponding data cycle, right before **t2**. Then master 0 and master 1 concurrently request slave 0 right before **t2**. Master 1 will be granted slave 0 and master 0 will be given a RETRY response (right before **t3**) because master 0 was already granted that slave at the previous cycle and because the system's round-robin policy.

3.7 Busy slaves

A slave must not be busy in order to be arbitrated between requesting masters. A slave is busy if it in a wait state because of a previous master access. It can also possibly be busy if it is in the middle of a burst and therefore locked on a master.

3.8 Burst operation

When a master issues a burst transfer on the control bus and is allowed the connection to the slave by the arbiter, it will lock the slave for the entire length of the burst. Locking a slave prevents any other master in the system from being able to establish a connection with said slave. If another master requests an address of a locked slave, it will receive a RETRY response. It will then try again the connection after the two cycles required by the RETRY response, and if the slave is still locked, it will receive another RETRY response, and so on. The internal implementation of fixed-length burst mode operation is done using

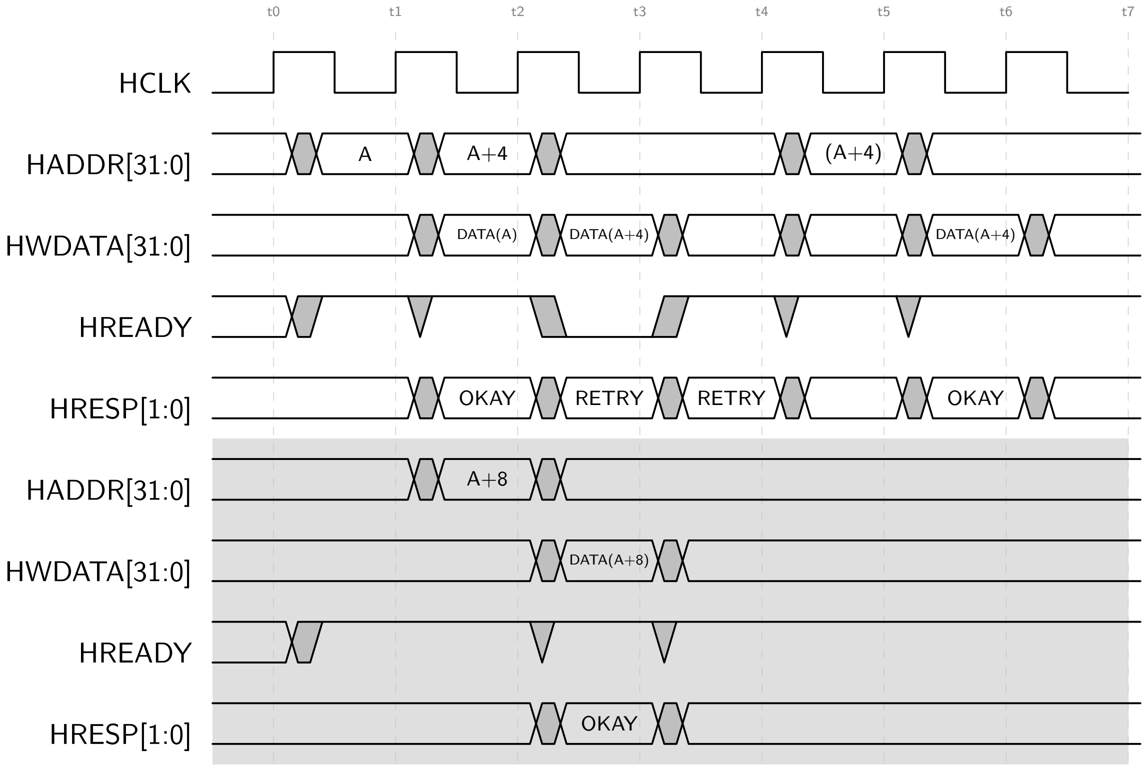

Figure 17. Example of simple concurrent request to same slave and the Round Robin policy

a counter. There is a per-master counter which increments with every successful burst transfer cycle in a burst. When, for instance, an 8 burst cycle reaches its eighth transfer, the AHB Crossbar Controller unlocks the slave for arbitration. This means that by the end of the last data cycle of the burst, the control cycle will be sampled and arbitrated among the requesting masters. In case of incrementing bursts (non fixed-length) the master will keep the **HBUSREQ** signal high during the burst, and lower it after the penultimate data cycle. This means that the last data cycle will be signaled to the arbiter by a lowered **HBUSREQ**. The AHB Crossbar Controller will then arbitrate between the master when it senses this event in an incrementing burst transfer.

3.9 The AHB Crossbar Controller implementation

My implementation of the AHB Crossbar Controller follows the two process method described in Section 2.1. The purely combinational process will consist of the proper crossbar structure (a group of multiplexers as described in Section 3.1) and the arbitration logic. The Decoder logic for the plug&play architecture is also purely combinational and included in this process. The sequential process will hold a control register which directly controls the crossbar selection values.

3.9.1 The crossbar control structure

The crossbar switch is a fundamental part of the circuit. Its task is to connect the master outputs to the slave inputs, and, vice versa, the slave outputs to the master inputs. As seen in Subsection 1.4.3, the AMBA signals are divided in two categories:

- control signals: from a master to a slave,
- data signals: both directions.

The pipelined nature of the AHB protocol implies that during a data cycle, a new control cycle will be issued by the master. Therefore these signals have two different timings: the control signals take one clock cycle, the data signals relative to the control signals take one or more clock cycles right after the control signals, depending on the number of wait states inserted by the slave. It is worth remembering that during a wait state, the control signals are withheld by the master, and they are not sampled by the slave that is issuing the wait state.

To ensure a collision free behavior of the crossbar, we envisioned a structure, implemented as two VHDL records, one for the control cycle and one for the data cycle, and hereon called the **crossbar control structure**. Each record in the structure contains, for each master and for each slave, their relative behavior. In other words, the record keeps information on the masters' and slaves' connections in the switch, so to say, what the master is connected to, and what the slave is connected to, in a consistent way. There is a record structure for each of the two timing cycles described in the previous paragraph: the control cycle and the data cycle. At a certain moment, a master can have its control bus connected to a certain slave, and its data bus connected to another slave, if, for example, it makes consecutive transfers to two different slaves, and both the transfers are allowed by the arbitration logic.

The structure's VHDL is described below:

```
type reg_master_type is record
  busy          : std_ulogic;
  hslave        : integer range 0 to nahbs - 1;
  retslv        : std_ulogic;
  errslv        : std_ulogic;
  cfgsel        : std_ulogic;
  idle          : std_ulogic;

  haddr         : std_logic_vector(15 downto 2);
  hrdatam       : std_logic_vector(31 downto 0);
  hrdatas       : std_logic_vector(31 downto 0);
  cfga11        : std_ulogic;
  hready        : std_ulogic;
  htrans        : std_logic_vector(1 downto 0);
  beat          : std_logic_vector(3 downto 0);
end record;

type reg_slave_type is record
  busy      : std_ulogic;
  hmaster   : integer range 0 to nahbmx - 1;
end record;

type reg_mvector_type is array (natural range <>) of reg_master_type;
type reg_svector_type is array (natural range <>) of reg_slave_type;
subtype reg_mvector is reg_mvector_type(nahbm - 1 downto 0);
subtype reg_svector is reg_svector_type(nahbs - 1 downto 0);

type cb_type is record
  m        : reg_mvector;
  s        : reg_svector;
end record;
```

Listing 3.1. Crossbar control structure records and types

It can be seen that each master has 5 basic, mutually exclusive fields:

busy	the master is busy connecting to a slave (whose AMBA slave number is kept in *hslave*),
retslv	the master is connected to its return dummy slave,
errslv	the master is connected to its error dummy slave,

cfgsel	the master is connected to the configuration slave,
idle	the master is currently idle.

Every slave, in turn, has the following field, consistent with the master entries:

busy	the slave is currently busy (connected to a master whose AMBA master number is kept in *hmaster*).

The other signals in the record are used by the internal dummy slaves and by the arbiter logic in case of burst operation.

The control structure record for the control cycle is implemented as a variable v of type *cb_type*. The control structure record for the data cycle is implemented as a signal r of type *cb_type*. Together v and r form the **crossbar control structure**. Furthermore, the r record is simply the sequential version of v. That is the v record at the previous cycle saved in a sequential memory structure.

For instance, if master 0 at cycle 0 is *busy*, then the corresponding field of the record v is set. At cycle 1, the corresponding field of r will maintain the value for *busy*. This ensures that the data cycle will be connected to the same slave as the control cycle, maintaining the correct pipelined timing.

It is the arbitration logic that will populate the v record. It fills the record after all master control signals are read and possible colliding masters and masters requesting a busy or locked slave are arbitrated. The crossbar reads the v record and then connect the control signals from the correct master to the correct slave. The v record is an output of the purely combinational arbitration logic (it is a subset of Q from Subsection 2.1.2 on page 37). It is important to note that arbitration and routing of the control signals happen in the same clock cycle in which the master has issued the signals (the control cycle).

The following clock cycle (or clock cycles, according to the number of wait states inserted by the slave), the r record is filled with the previous value of v. The r record corresponds to the output r of the sequential process as seen in Subsection 2.1.2. It will be read by the crossbar and used to connect the data signals, both ways, from master to slave.

3.9.2 Code organization

The combinational process is organized as follows:

- arbitration loops,
- interrupt merging,
- crossbar loops.

The interrupt-merging code merges, by performing an OR operation, all the hardware interrupts from the slaves. The output is a single interrupt signal that is fed to all the masters.

The crossbar is implemented as two 'for' loops. The loops here are completely concurrent, there is no inter-dependence between the two. One loop connects the slave output to the master input. The other loop, on the other hand, connects the master outputs to the slave inputs. The slaves which are not busy receive a de-asserted selection input, and are given a 'don't care' value on every other input.

The arbitration algorithm is divided into three loops which have a sequential dependence among one another, but which don't have a sequential dependence among the iteration of each loop. This means that every loop can be unfolded and executed concurrently, or in other words, can be synthesized in a completely parallel way.

The first loop iterates all the masters. It decodes the control output of every master and tries to decide which will be the attempted behavior of the master at the current clock cycle. The attempted behavior can be one of the following:

- idle,
- busy: trying to connect to a slave,
- configurational: trying to read the plug&play configuration from the AHB Crossbar Controller,
- error: trying to decode an address that is not mapped to any slave,
- retry: trying to connect to a slave that is not ready.

This information is kept in the variables: *busym*, *idle*, *cfgsel*, *errslv*, *retslv*, which are of type vector, meaning that each master has its set of variables. Optionally the variable lock can be set for a master that is in the middle of a locked transfer (burst transfer).

The second loop iterates the slaves and assigns to each master the requested slave, handling eventual collisions and busy slaves. The 'selmast' function is executed for each ready slave. The function ensures the arbitration between colliding masters. If several masters are requesting the same slave, the function uses a round robin policy to select which master will connect to the slave. The function also ensures that a master which started a locked transfer at the previous cycle, keeps the connection with the slave, in a prioritized way. The masters which are not granted a slave, are connected to their RETRY slave. At the end of this loop, the slave part of the crossbar v record will be correctly populated with the new information if the slave was ready. If the slave was not ready, the record will hold the old information. The slave record can be seen in Listing 3.1 on page 63.

The third loop iterates once again among the masters. It contains the logic for the dummy slaves (RETRY and ERROR) and also the logic for the access to the configuration slave and configuration plug&play information. At this cycle, if the master was connected to a slave that is ready, its master record information in the crossbar structure will be updated with new values, otherwise the old values will hold. The master record is shown in Listing 3.1 on page 63.

The information in the master record and the slave record is kept consistent by the arbitration logic. Inconsistent information in the crossbar control structure can lead to errors in the connections and invalid data access.

The complete code of the AHB Crossbar Controller can be found in Appendix A.

CHAPTER 4

SYSTEM VERIFICATION AND SIMULATION

Testing was an essential part throughout the development of the AHB Crossbar Controller. As I progressed in the completion of the entity, I would test any change made to the code on a set of purposely created tests, to ensure the proper behavior of the circuit after each modification to the code was made. The testing of my entity was not done on the complete LEON 3 system, as for testing and simulation purposes a reduced complexity system was preferred. The advantages of a reduced-complexity system are the ease of use of a limited number of masters, which could be directly controlled, and a limited number of slaves, which could be easily monitored. To simulate a complete LEON 3 system with my entity, more time would have been needed. Furthermore, there is no control over the initial startup sequence of the processor and there is a whole lot more complexity in the design of the masters, the LEON 3 cores, and the slaves, the memory controllers. For this reason I decided to create a test bench system with my AHB Crossbar Controller and the already available AHB test bench master and AHB test bench slave, from GRLIB. I made some modifications to the slave to allow a parametrized number of wait states. The setup and results of the testing of the circuit are discussed in this chapter.

4.1 Test bench entities

Fundamental for my test benches were the AHB test bench master and AHB test bench slave. These two entities are part of GRLIB and their code is located in the sub-directory *lib/gaisler/ambatest*. The test bench master implements a simple AHB master that can be controlled with a set of control signals to issue several kind of transfers on an AHB bus. The test bench slave implements a simple SRAM module that can be read or written with a configurable amount of bits and a configurable address. The test bench slave was modified to include a generic-mappable number of wait states. In real world usage scenarios, AHB slaves usually respond to AHB requests issuing wait states at the beginning of a transfer. If the transfer is sequential, like in a burst, usually the wait states are inserted only at the beginning of the response cycles. If the transfer is non-sequential, usually the slaves include wait states at every transfer response cycle. Normally, a slave responds with a number of wait states between 0 and 2.

4.2 Test bench formulation

I formulated several test benches for simulation. Every test bench has a varying number of masters and slaves and performs different kind of transfers (sequential, non-sequential, etc. . .). GRLIB provides a package with pre-compiled functions to more easily access the control input of the AHB test bench masters. This way I was able to easily configure and formulate a test bench for several configurations. Some of the functions needed to be modified to be able to finely tune some of the control signals. For example, I needed to add a function to control the master and have it output fixed-length burst transfers, as only incremental burst transfers were possibly started with the standard test bench package. Such function is reported below in Listing 4.2.

```
--------------------------------------------------------------------------------
-- AMBA AHB write access (htrans)
--------------------------------------------------------------------------------
procedure ahbwrite(
  constant address  : in  std_logic_vector(31 downto 0);
  constant data     : in  std_logic_vector(31 downto 0);
  constant size     : in  std_logic_vector(1 downto 0);
  constant htrans   : in  std_logic_vector(1 downto 0);
  constant hburst   : in  std_logic_vector(2 downto 0);
  constant debug    : in  integer;
  constant appidle  : in  boolean;
  signal   ctrl     : inout ahbtb_ctrl_type) is
begin
  --ctrl.o <= ctrlo_nodrive;
  wait until ctrl.o.update = ’1’ and rising_edge(ctrl.o.clk);
  ctrl.i.ac.ctrl.use128 <= 0;
  ctrl.i.ac.ctrl.dbgl <= debug;
  ctrl.i.ac.hburst <= ”000”; ctrl.i.ac.hsize <= ’0’ & size;
  ctrl.i.ac.haddr <= address; ctrl.i.ac.hdata <= data;
  ctrl.i.ac.htrans <= htrans; ctrl.i.ac.hwrite <= ’1’;
  ctrl.i.ac.hburst <= hburst;
  ctrl.i.ac.hprot <= ”1110”;
  if appidle = true then
    wait until ctrl.o.update = ’1’ and rising_edge(ctrl.o.clk);
    ctrl.i <= ctrli_idle;
  end if;
end procedure ahbwrite;
```

Every test bench VHDL entity includes a number of components: from 1 to 4 masters, from 1 to 4 slaves and the AHB Crossbar Controller. For the testing of the basic transfers, every test bench was run with all its slaves answering with 0, 1 and 2 wait states. In each test bench entity there is a sequential process for each master, compiled with the AHB test bench package functions for controlling its behavior. An example of said process code, simulating a single transfer request from the master, is the following:

```
-- testbench for master 0
tb0 : process is
begin

  -- Initialize the control signals
  ahbtbminit(ctrl0);

  -- Write 0x12345678 to address 0xA0000000.
  ahbwrite(x"A0000000", x"12345678", "10", "10", '0', 2, true, ctrl0);

  -- Stop simulation
  ahbtbmdone(1, ctrl0);

  wait;
end process tb0;
```

The test bench files were kept in the *designs/leon3-gr-xc3s-1500* folder.

4.3 Test bench results

In this section I will report the results of our simulation tests. The correctness of a test bench was checked by performing a manual analysis on the master outputs to slave inputs and slave outputs to master inputs. The content of the slave memory was also analyzed to check for correctness. The software used for the VHDL simulation was Mentor Graphics' ModelSim. The clock frequency used for the simulation was 50 MHz, giving a clock period of 20 ns.

The following list describes the typology of the tests I run. For each typology, whenever applicable, the tests were run on slaves in three configurations: 0 wait state response, 1 wait state response, 2 wait states response.

The test typologies are presented below, every typology is executed on a multi-master test bench:

- parallel slave access with collision,
- interlaced parallel multi-slave access,
- mixed timing multi-slave access,
- parallel slave non-sequential transfers access with collision,
- parallel slave burst access (incrementing and fixed length) with collision,
- parallel configuration access,
- parallel error access attempt,
- parallel slave burst incrementing access with collision.

The timing diagrams of the following subsections show the behavior that was both expected and verified during the tests. The key to read the diagrams is that they are shown from the masters' perspective. For each master the input-outputs are shown. The sequence of the signals in all the diagrams is the following: the master signals are shown first, going from master 0 to the highest-numbered master. Then the crossbar control record v is shown, with the signals organized in a similar manner. The v record shows the crossbar control signals of the masters, going from master 0 to the highest-numbered master, then it shows the signals of the slaves, going from slave 0 to the highest-numbered slave. In the diagrams every master at every cycle could potentially be received from or sent to a different slave. Every transfer cycle (control and data) is highlighted in the diagrams. So for every highlighted cycle it will be possible to understand what is the slave connected to the master, by observing the control signals output and then the data signal response. If the response is RETRY or ERROR, then a dummy slave was connected. This connection between the masters and the slave is stated clearly in the crossbar control structure that is shown at the bottom of the diagrams. In fact, the structure clearly states which connection the masters and the slaves are granted for the control cycle. For example Figure 18 shows a simple concurrent connection between master 0 and slave 0 and between master 1 and slave 1.

The diagram in Figure 18 shows that both master 0 and master 1 output an address on their control bus just before **t1**. Master 0 outputs address A, which maps to slave 0. Master 1 outputs address B which maps to slave 1. The AHB Crossbar controller recognizes the addresses, finds that there are no collisions, so no need for arbitration, and populates the v record of the crossbar with the correct information, as can be seen in the bottom part of the diagram. It is shown that master 0 is busy and connected to slave 0, and at the same time slave 0 is busy and connected to master 0. The same holds for the couple master 1 slave 1. The v record controls the control cycle crossbars which are correctly routed. At the following cycle, record r, which is not shown in the diagram as it is a one-clock delayed version of record v (see Section 3.9), correctly routes the data cycle crossbars as seen in the diagram just before **t2**.

Every test's behavior will be detailed in the following subsections.

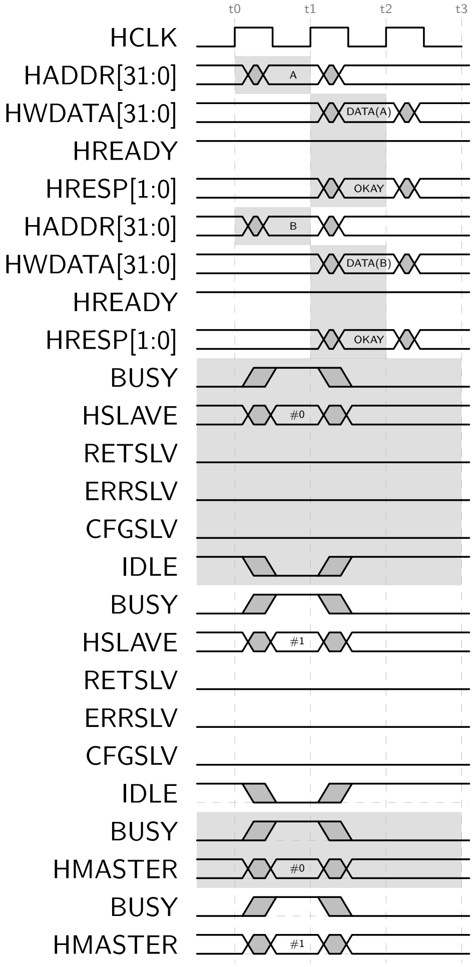

Figure 18. Simple concurrent transfers between two master-slave couples with crossbar control record v

4.3.1 Test 1, 2 and 3

These test benches are composed of two masters accessing the same slave at the same clock cycle. The correct behavior is that one of the two masters will be connected to its dummy REPLY slave, while the other will connect to the slave. After the two-cycle response the denied master will attempt another connection with the slave, and this time it will succeed. Test 1 is configured with a 0 wait state slave, test 2 with a 1 wait state slave, test 3 with a 2 wait states slave. The test results are shown in Figure 19, Figure 20 and Figure 21.

Address A maps to slave #1.

4.3.2 Test 4, 5 and 6

These tests are performed on two masters accessing two different slaves at each clock cycle, in an interlaced way. That means that master 0 will try to access slave 0 while at the same time master 1 will try to access slave 1. In a shared bus topology interconnection network, only one master at a time will be able to access a slave. Using the AHB Crossbar Controller both masters will be able to access the slaves concurrently. At a later moment master 0 will try to access slave 1 and master 1 will try to access slave 0. This transfer can also happen concurrently. The use of a crossbar halves the time it takes to complete all the transfers in this situation. As before, test 1 is configured with 0 wait state slaves, test 2 with 1 wait state slaves and test 3 with 2 wait states slaves. The test results are shown in Figure 22, Figure 23 and Figure 24.

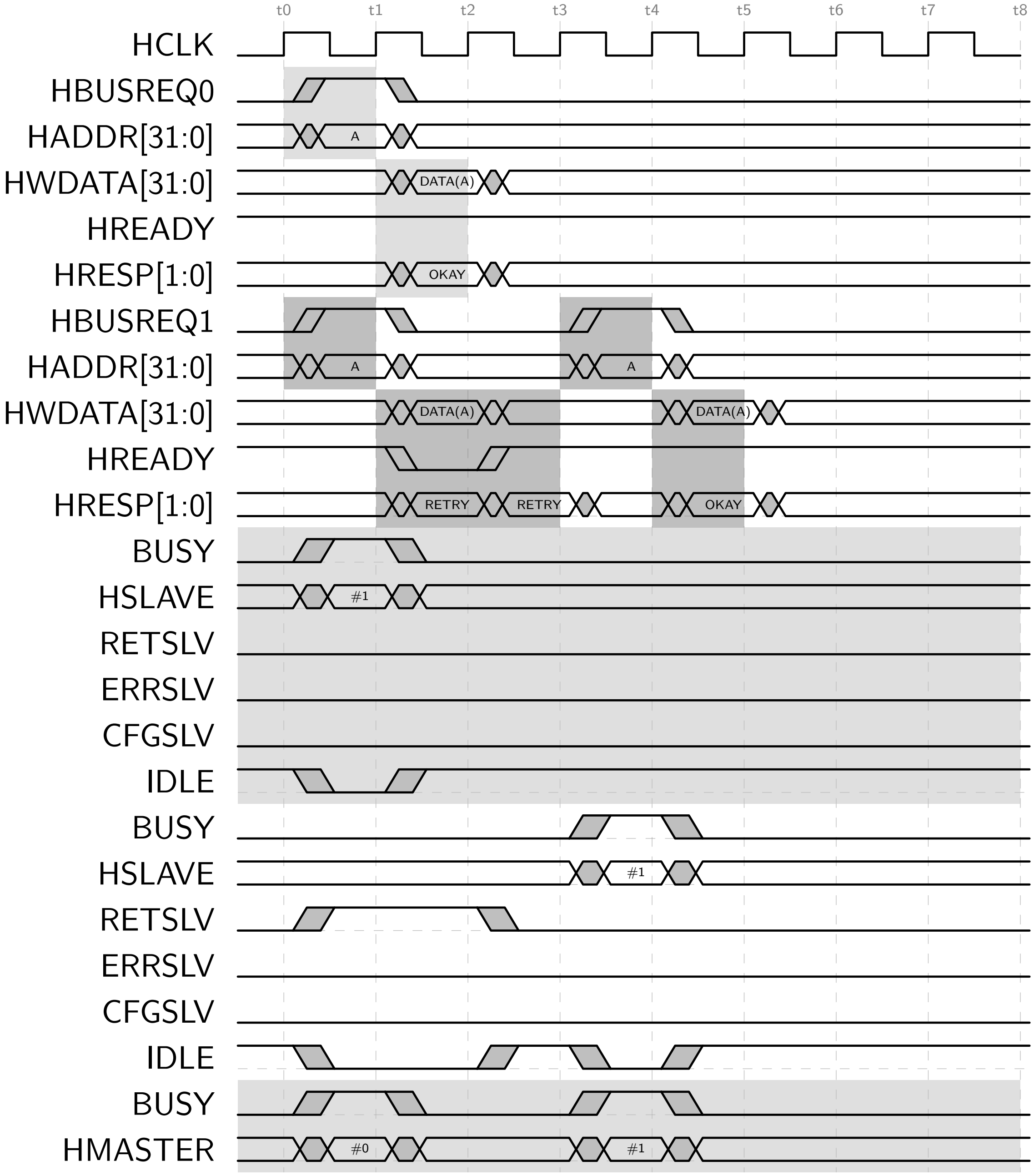

Figure 19. Test 1: two masters and one 0-ws slave

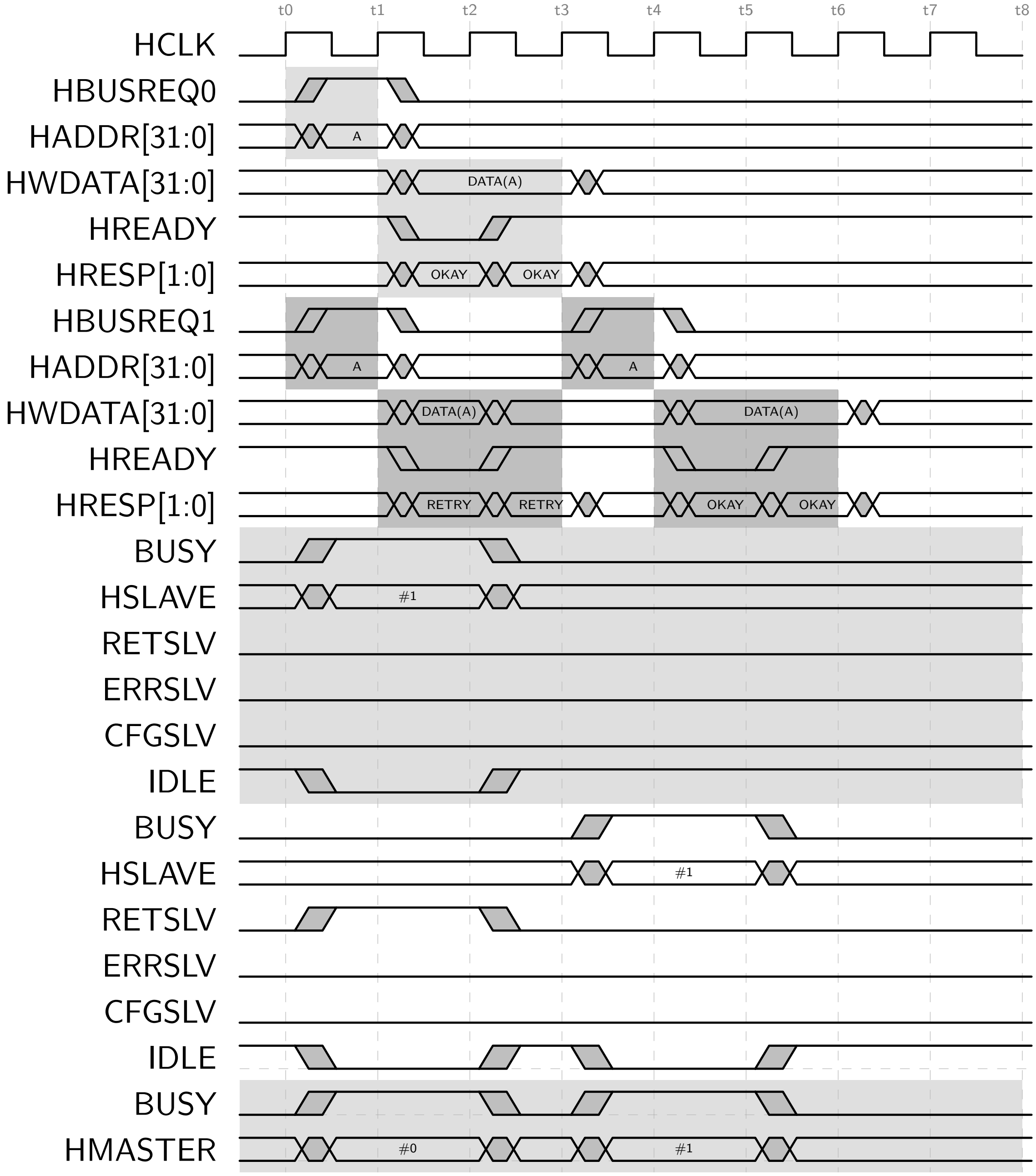

Figure 20. Test 2: two masters and one 1-ws slave

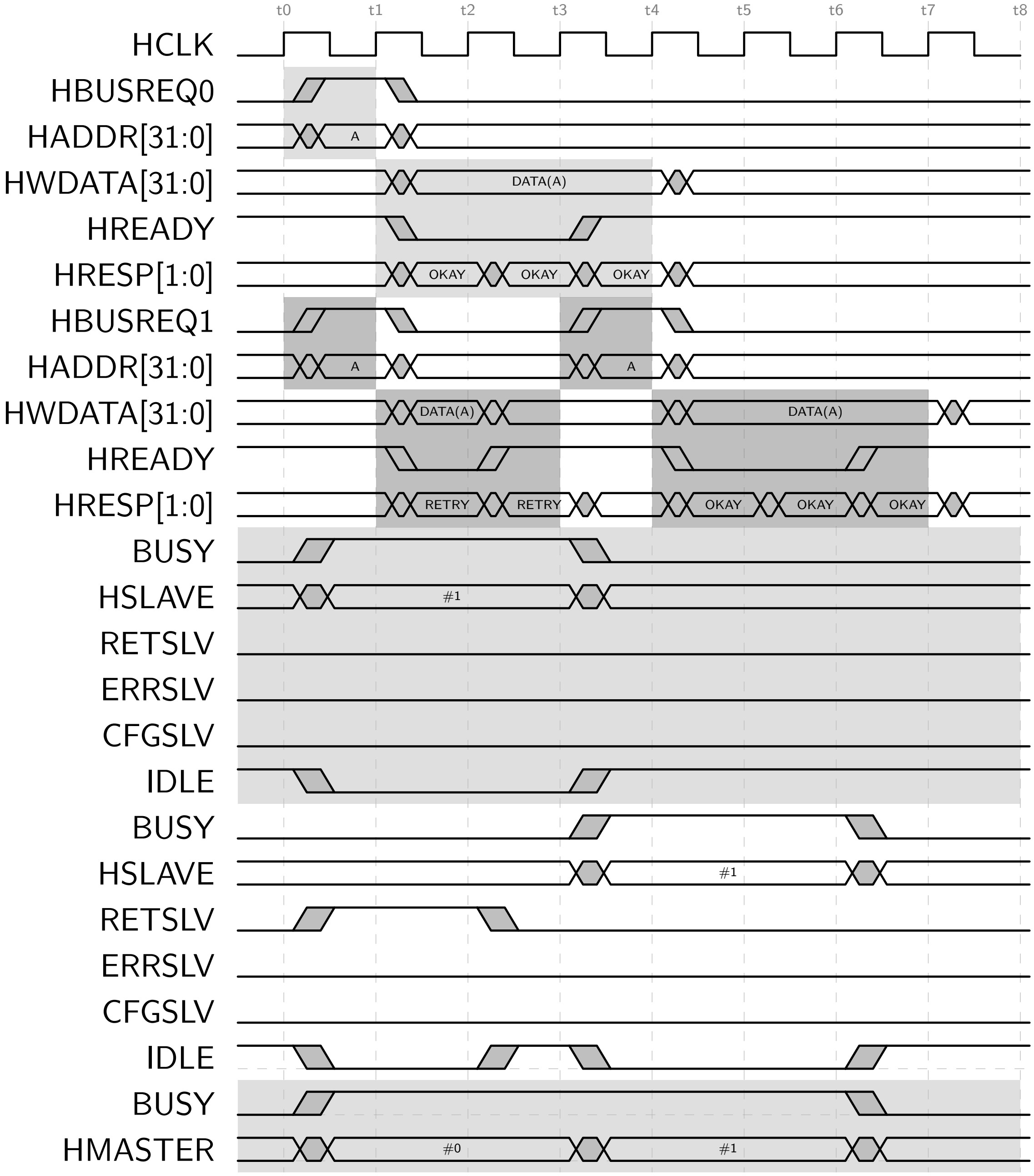

Figure 21. Test 3: two masters and one 2-ws slave

Address A maps to slave #0.

Address B maps to slave #1.

4.3.3 Test 7, 8 and 9

These test benches are composed of three masters accessing two slaves at different times. All the slaves respond to a transfer request with the same number of wait states. This test is done to control the behavior of the crossbar in a mixed access situation, with several masters accessing several slaves at several different times. The number of wait states implemented by the slave changes drastically the total completion time of the test, as it can be seen in the results. Tests 7, 8 and 9 are performed on slaves with 0, 1 and 2 wait states respectively. The test results are shown in Figure 25, Figure 26 and Figure 27.

Address A maps to slave #0.

Address B maps to slave #1.

4.3.4 Test 10, 11 and 12

This series of tests builds on the previous series described in Subsection 4.3.3 and increases the number of masters in the system connected to the crossbar to four. As before, tests 10, 11 and 12 are performed on slaves with 0, 1 and 2 wait states respectively. The test results are shown in Figure 28, Figure 29 and Figure 30.

Address A maps to slave #0.

Address B maps to slave #1.

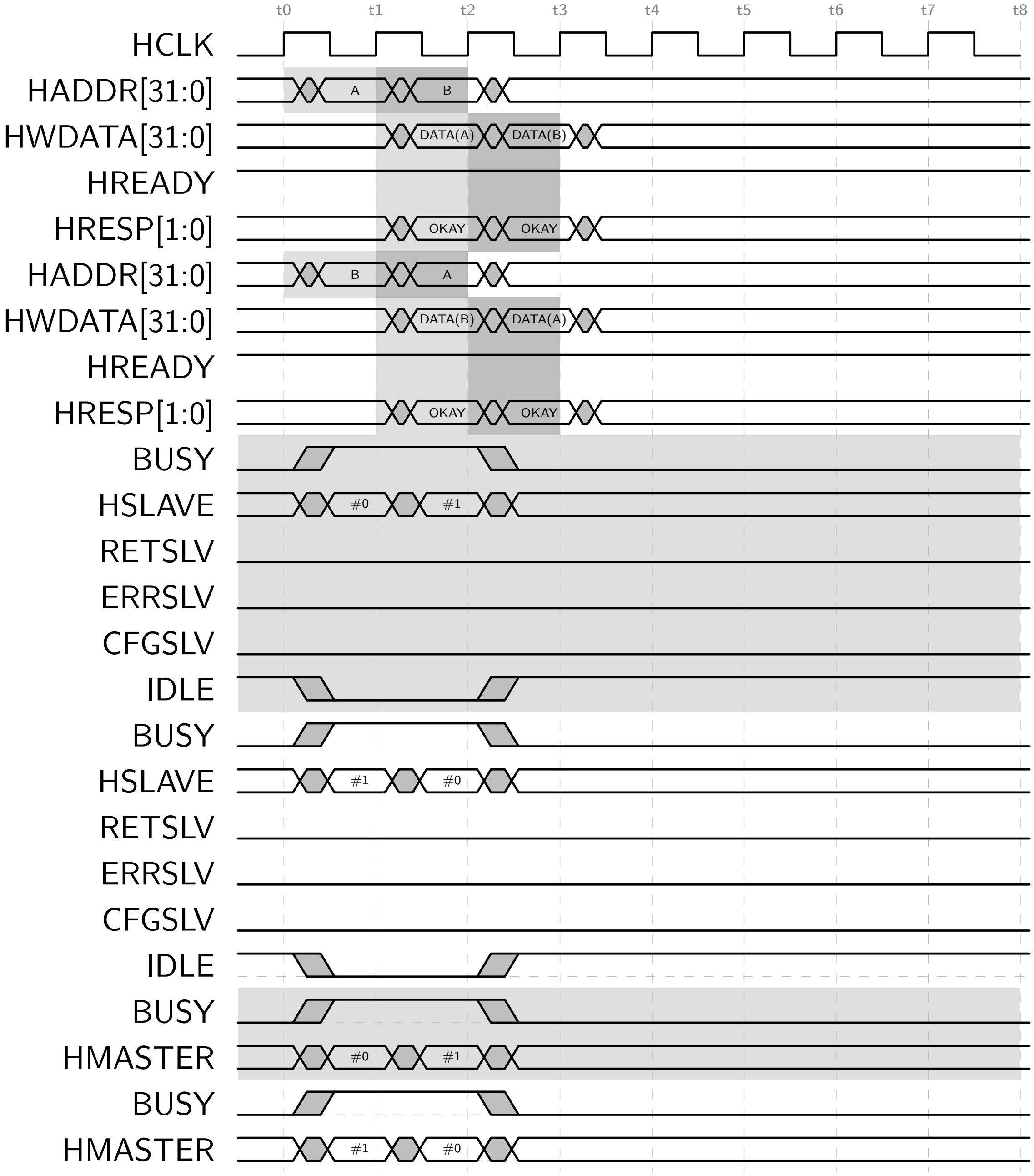

Figure 22. Test 4: two masters and two 0-ws slaves

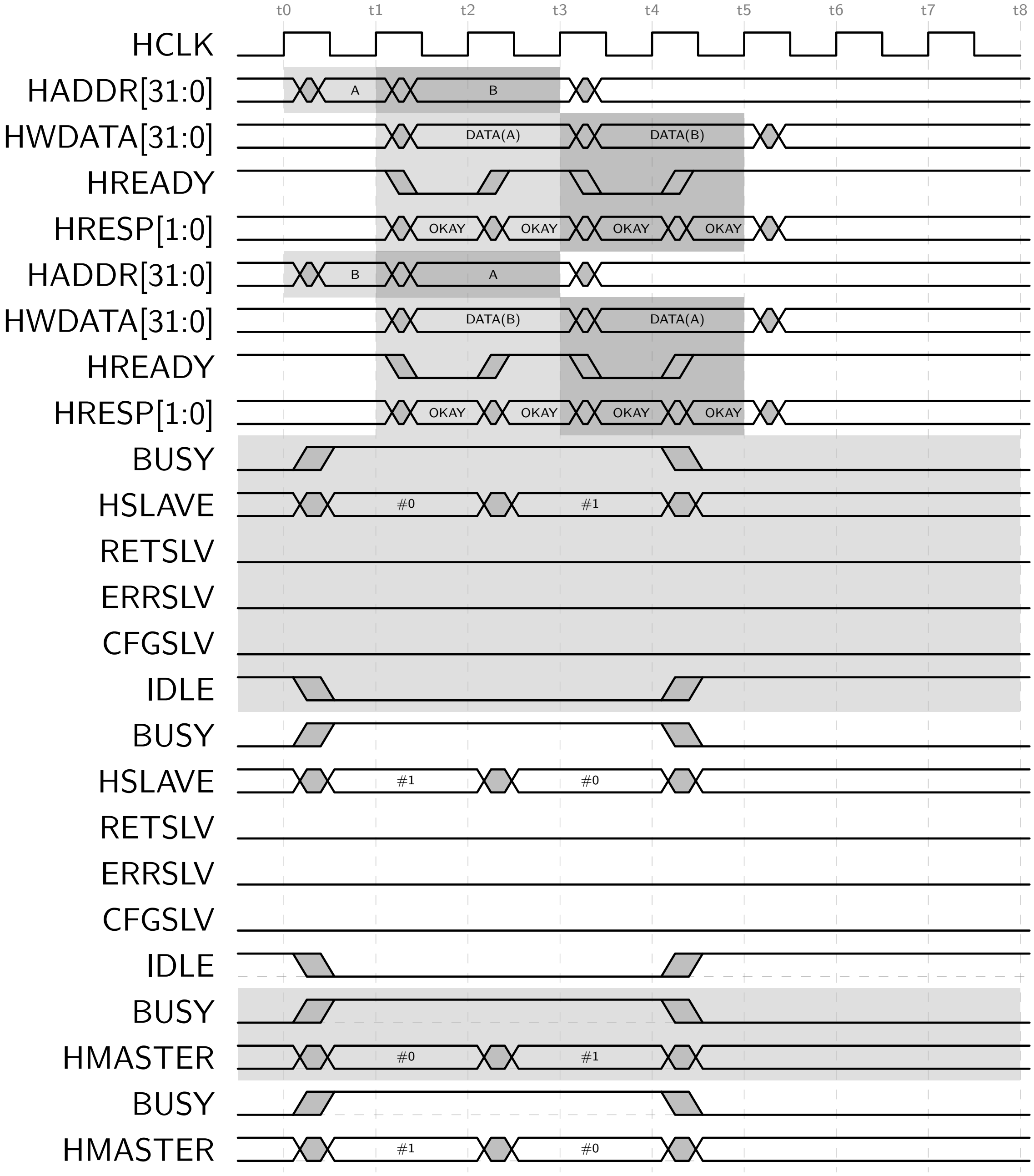

Figure 23. Test 5: two masters and two 1-ws slaves

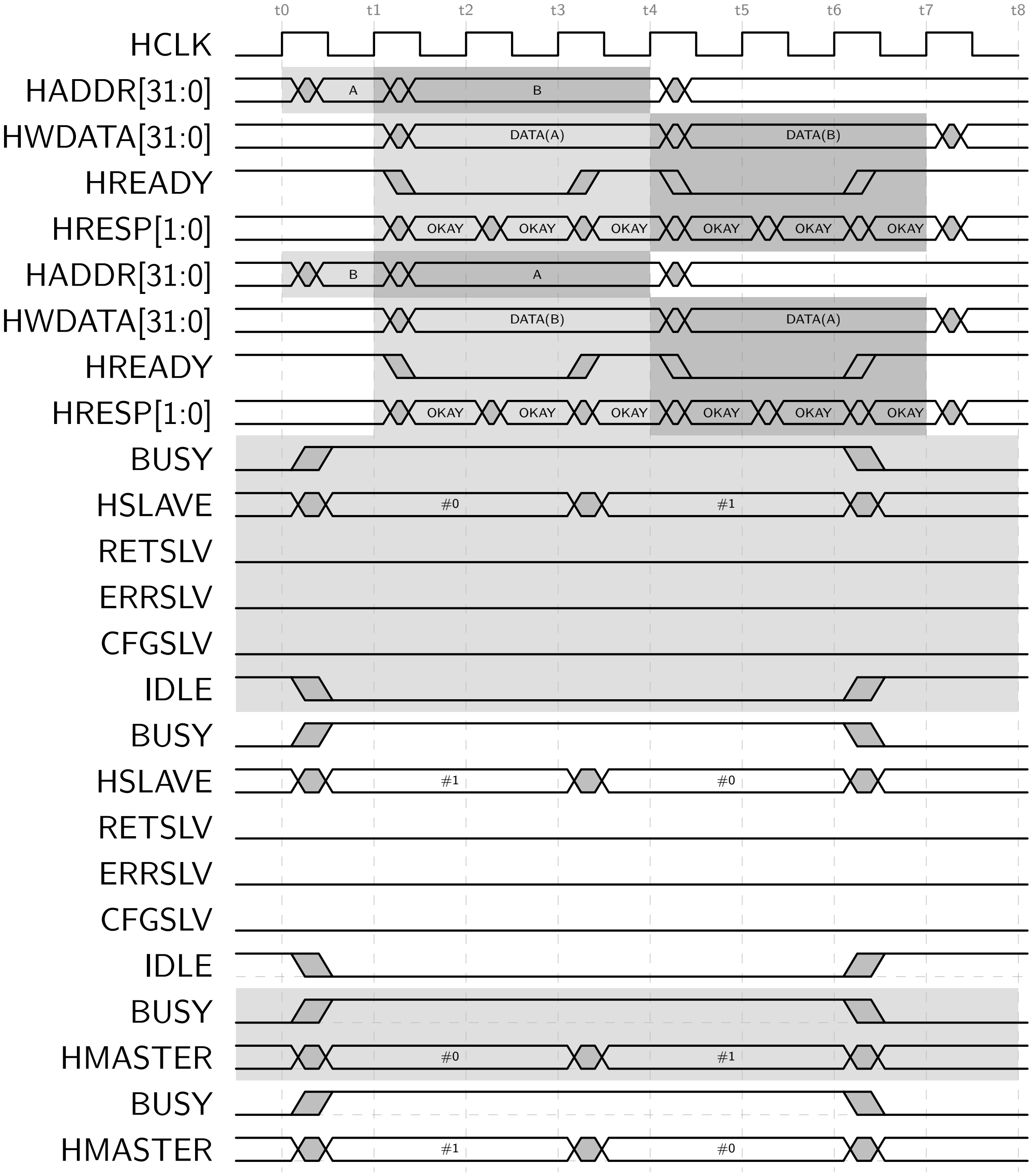

Figure 24. Test 6: two masters and two 2-ws slaves

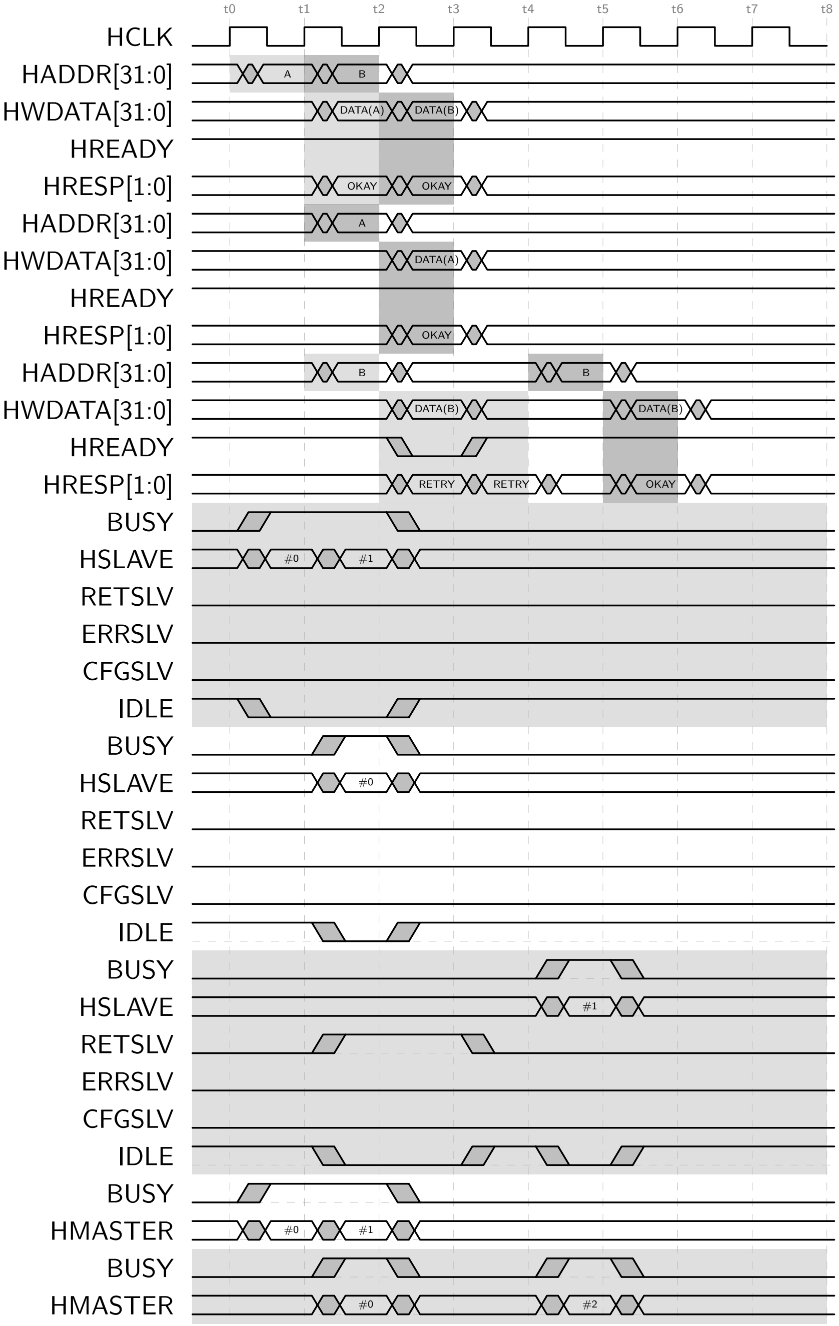

Figure 25. Test 7: three masters and two 0-ws slaves

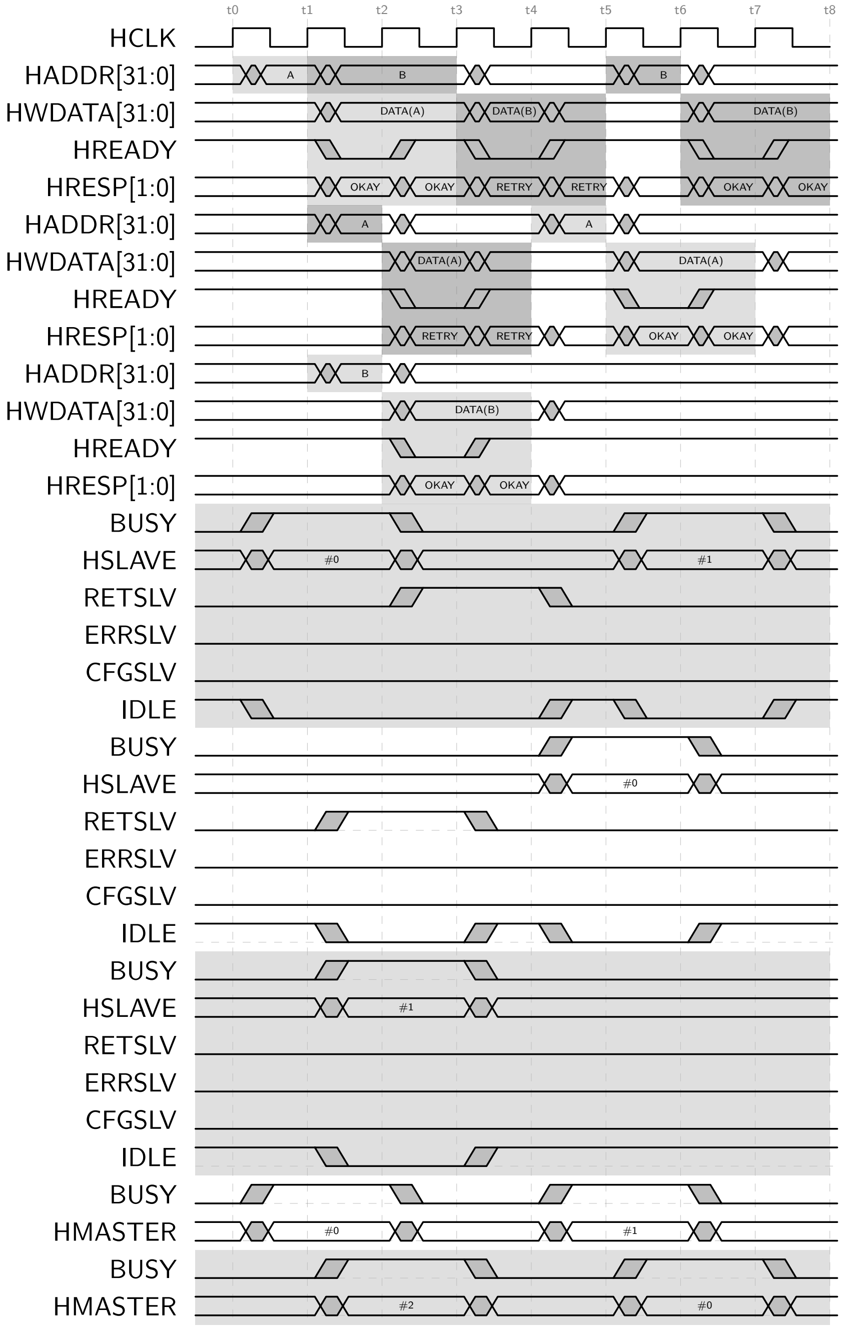

Figure 26. Test 8: three masters and two 1-ws slaves

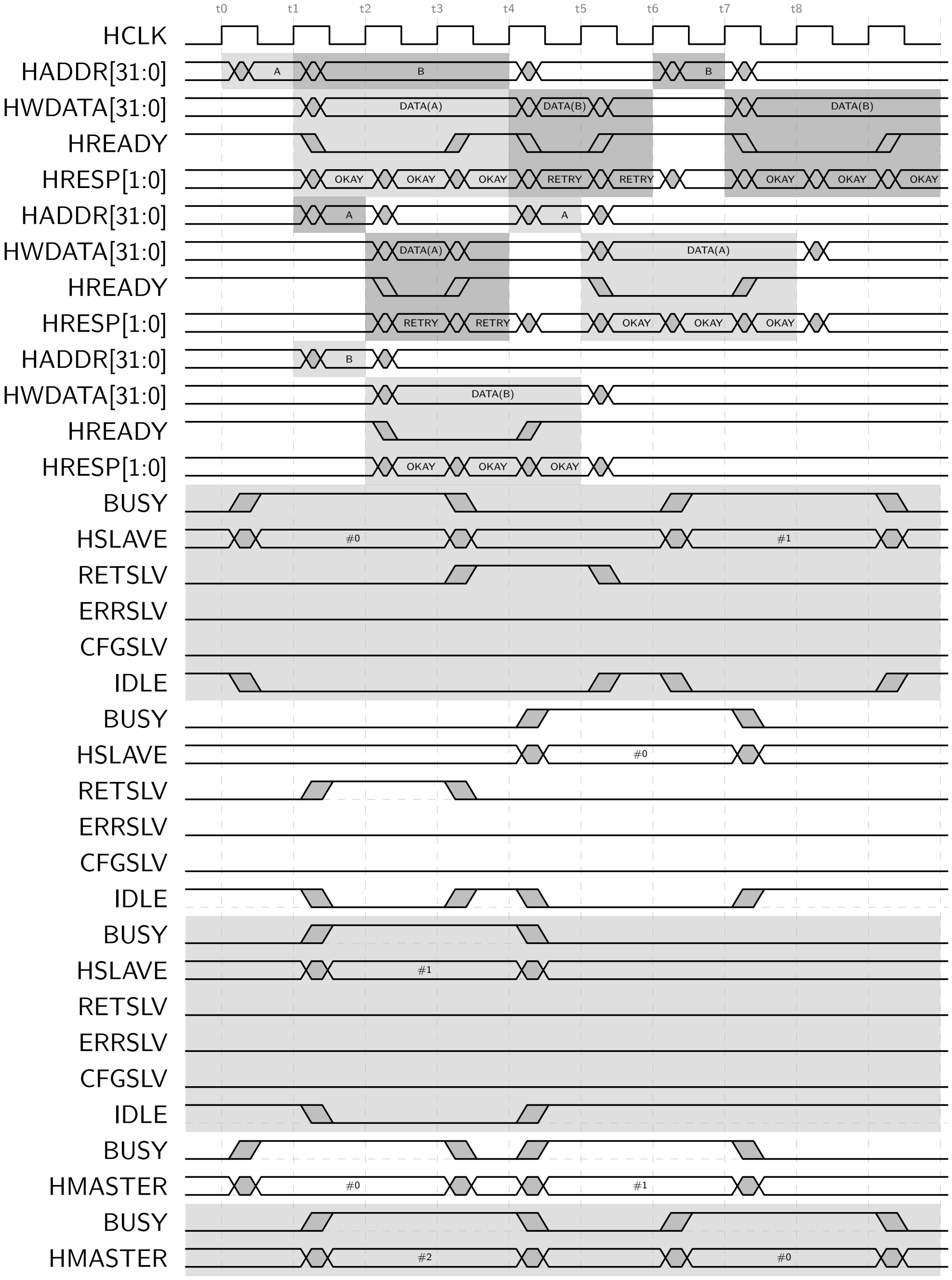

Figure 27. Test 9: three masters and two 2-ws slaves

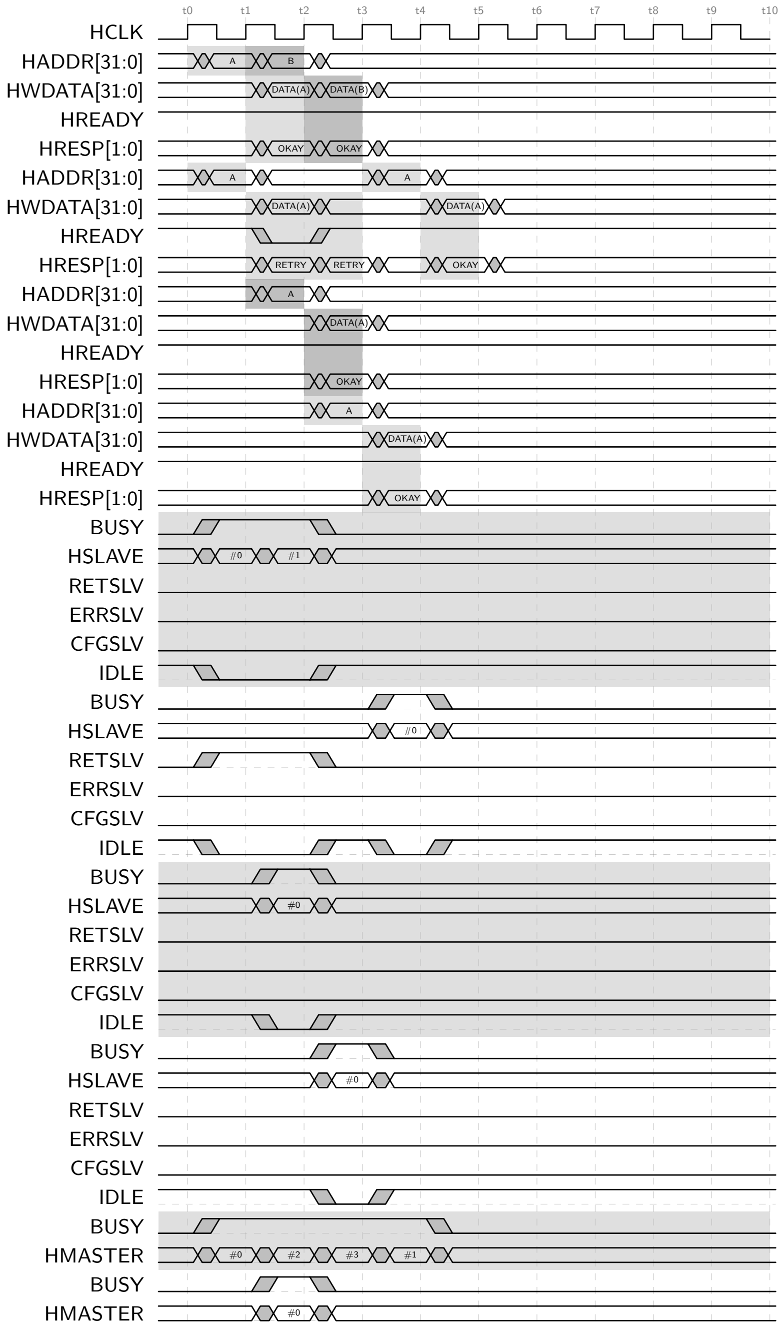

Figure 28. Test 10: four masters and two 0-ws slaves

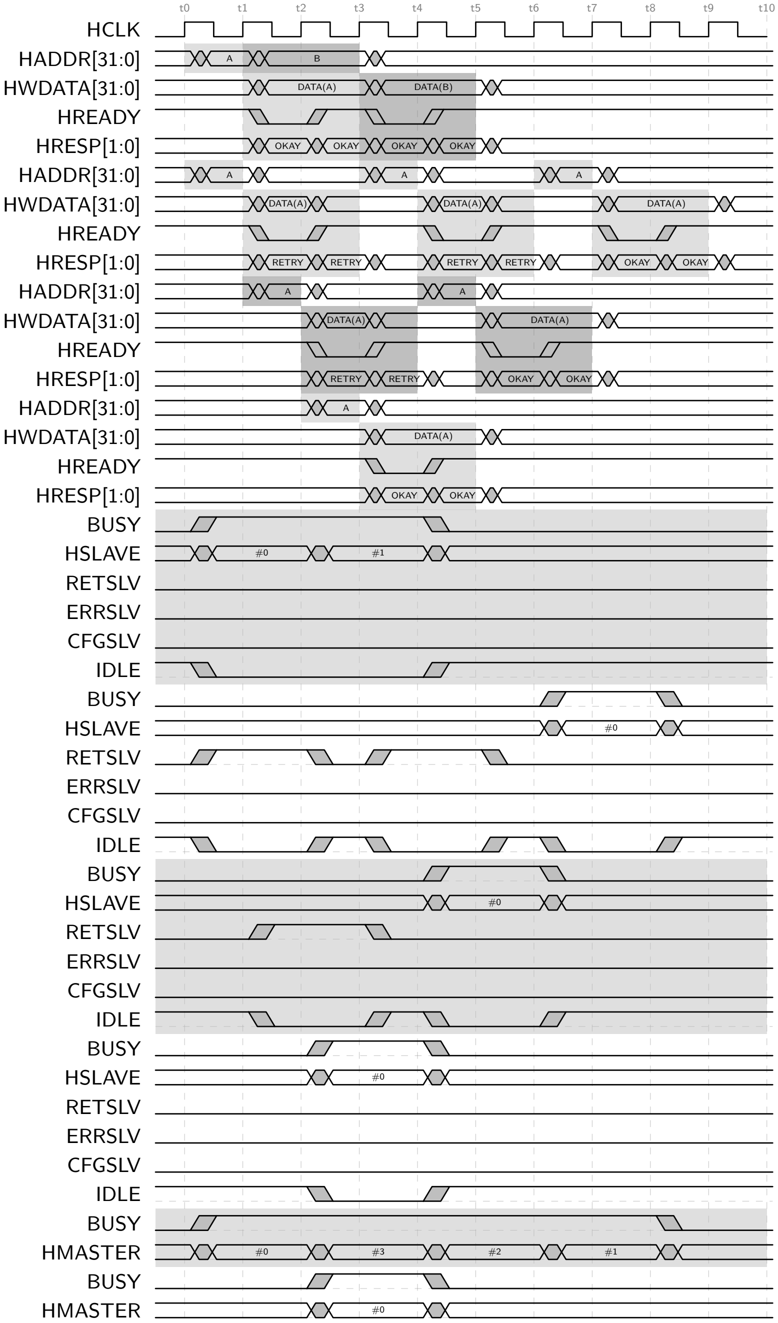

Figure 29. Test 11: four masters and two 1-ws slaves

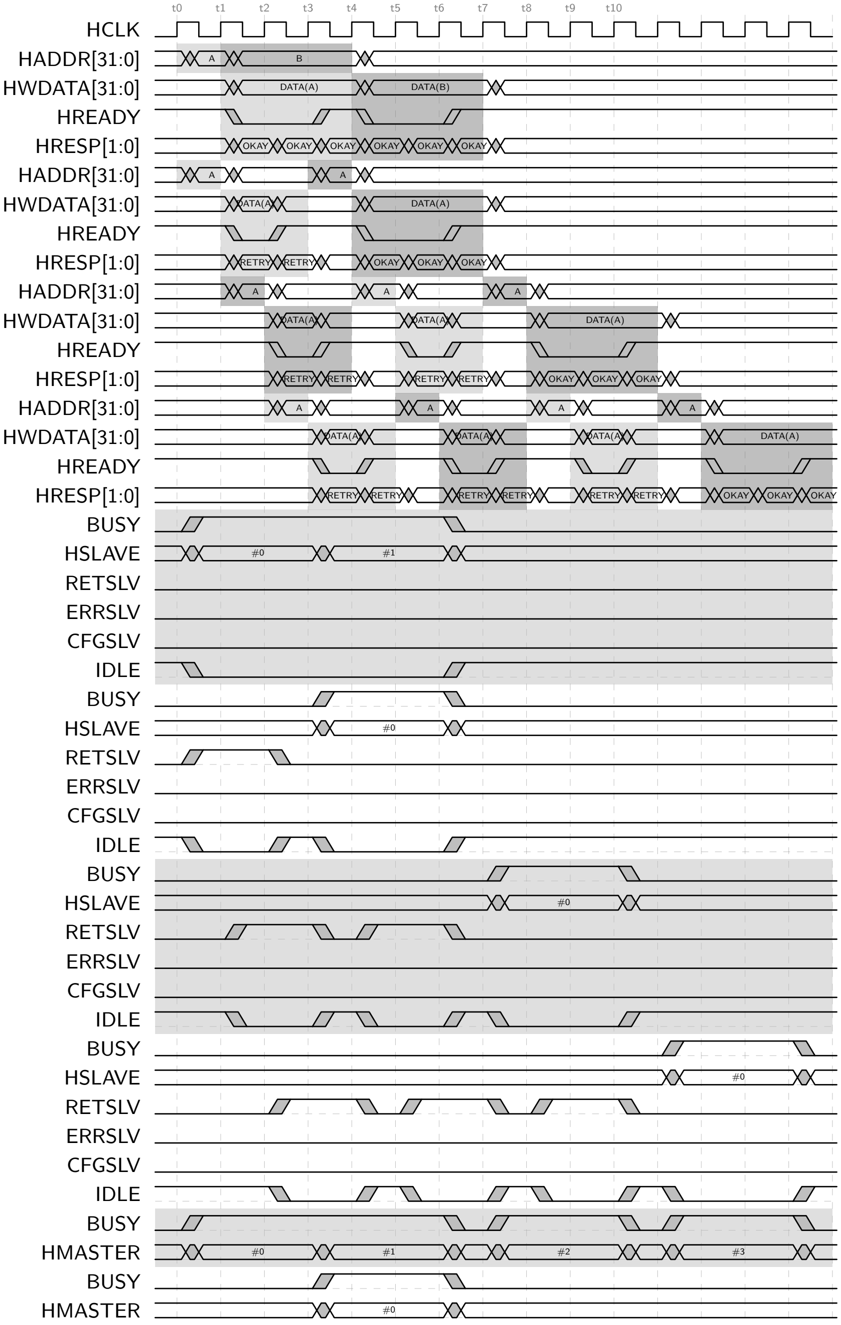

Figure 30. Test 12: four masters and two 2-ws slaves

4.3.5 Test 13

In this test two masters are requesting the same slave. The master which is granted access, begins a sequence of non-sequential single-length transfers to sequential addresses on the slave. The arbiter should not give priority to this kind of non-sequential bursts. Furthermore, a slave should answer with wait states at each transfer, if it requires it. The difference with a burst transfer is that in the burst, only the first transfer of the cycle is non-sequential and possibly receives wait states, at the following cycle. The remaining transfers of the burst will be of type sequential, will have 0 wait-state responses and will be prioritized. The slave in this test responds inserting 1 wait state. The test results are shown in Figure 31.

Address A maps to slave #0.

4.3.6 Tests 14 through 17

This series of tests was developed in order to test the behavior of the crossbar in case of a burst transfer. In fact, the purposed behavior is to lock the slave to the master which is instantiating the burst transfer, and unlock it only when the transfer is done. This ensures that the sequential accesses typical of a burst transfer are completed before the slave is re-assigned to another master according to the round-robin policy. A burst transfer has the potential of being much faster because it requests sequential accesses during its length. The sequential-access requests are served by the slave in a speedier way with respect to a non-sequential transfer, that could cause the insertion of wait-state at each

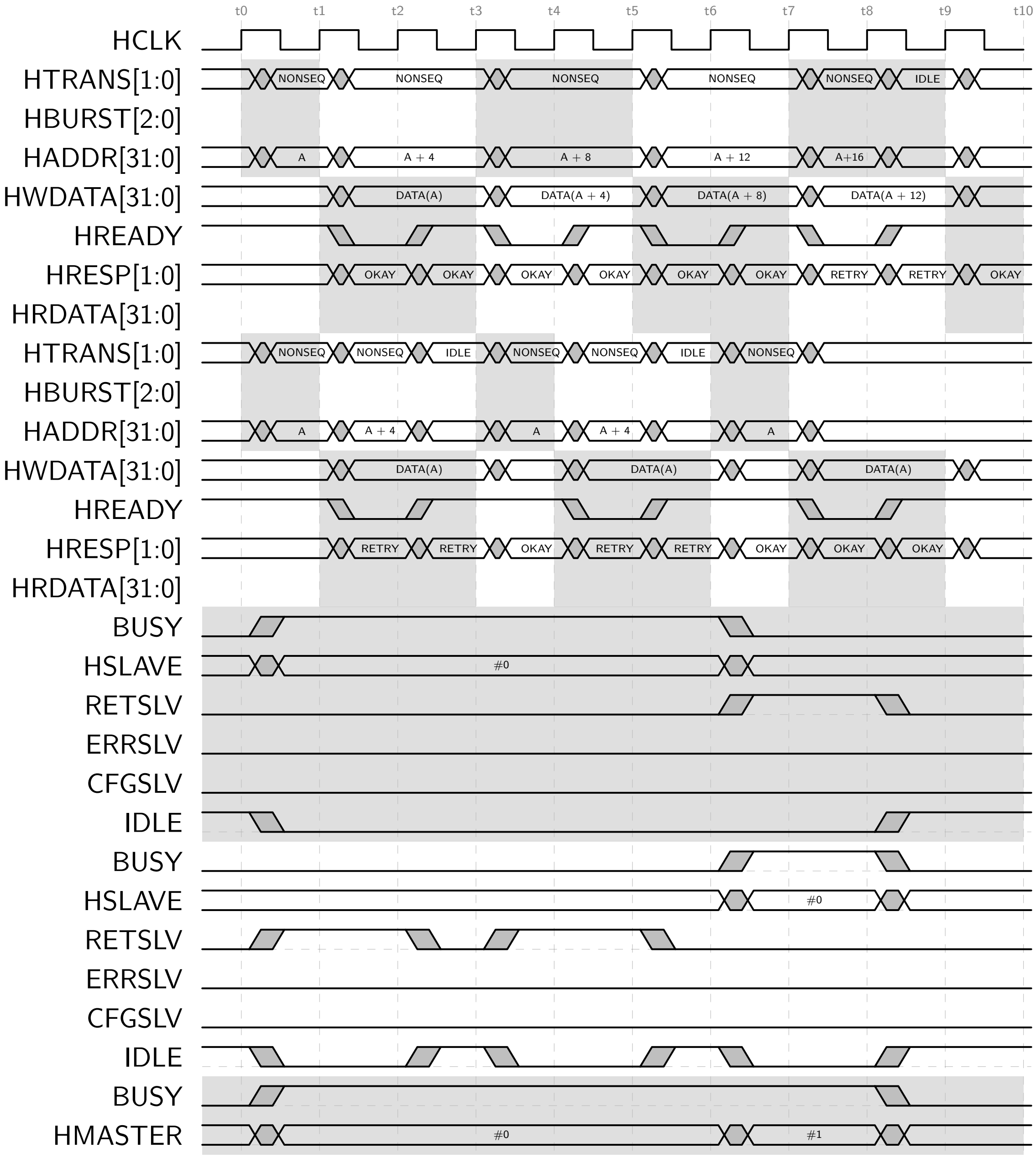

Figure 31. Test 13: two masters and one 1-ws slave

address request. The preferred behavior would be to let a master complete a burst transfer, without interrupting it if another master is requesting the slave. This is the implemented behavior as it enhances the overall performance of the system. In test 14, we have two masters requesting a transfer to the same slave. The master which is granted access to the slave performs a 5-burst incrementing burst. As this transfer is not fixed-length, the specifications of the AMBA AHB bus require that the master keep the bus request signal asserted during the burst, and de-asserts it one clock before completing the burst. This way the AHB Crossbar Controller knows when it is allowed to re-arbitrate the slave to another requesting master, as seen in this test. The test results are shown in Figure 32.

In test 15 through 17 we have fixed length incrementing bursts of size 4, 8 and 16. The size of the burst indicates the number of transfers which will be requested on the bus, and not the size in bytes (compare the specification in Subsection 1.4.5). During the transfer the slave must be locked to the requesting master and the address must increase at each transfer. The test results are shown in Figure 33, Figure 34 and Figure 35.

In all the previous tests address A maps to slave #0.

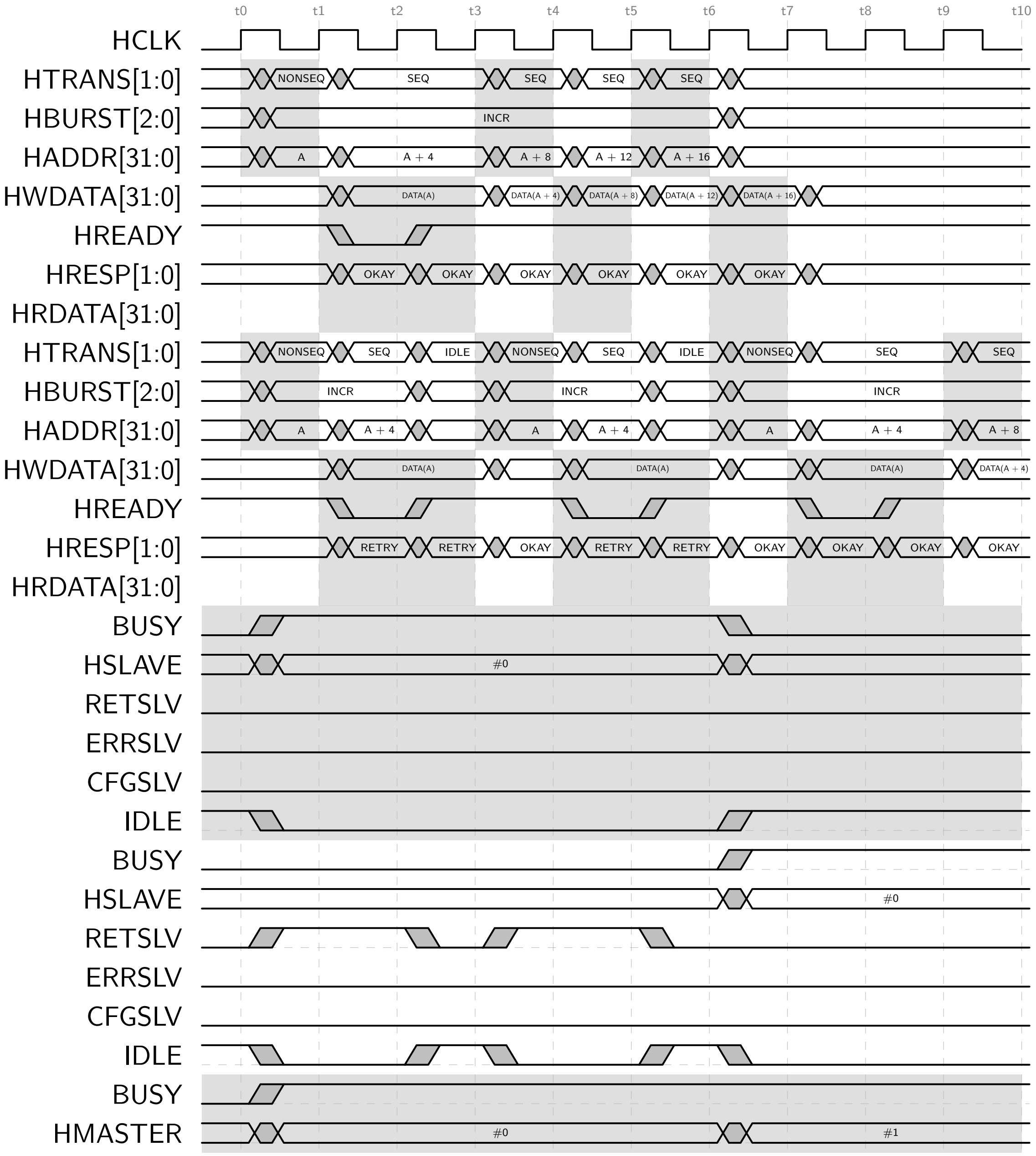

Figure 32. Test 14: two masters and one 1-ws slave

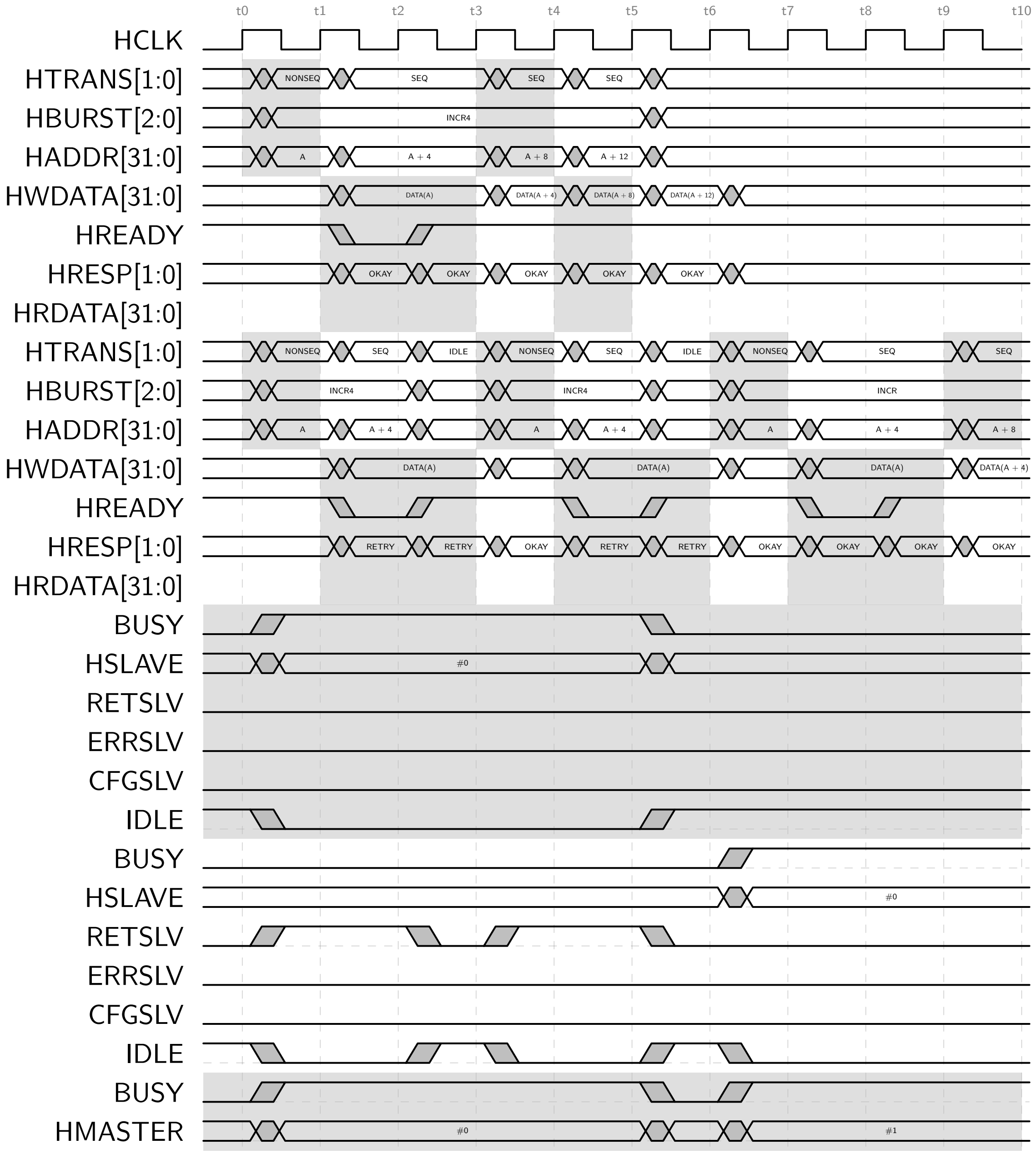

Figure 33. Test 15: two masters and one 1-ws slave

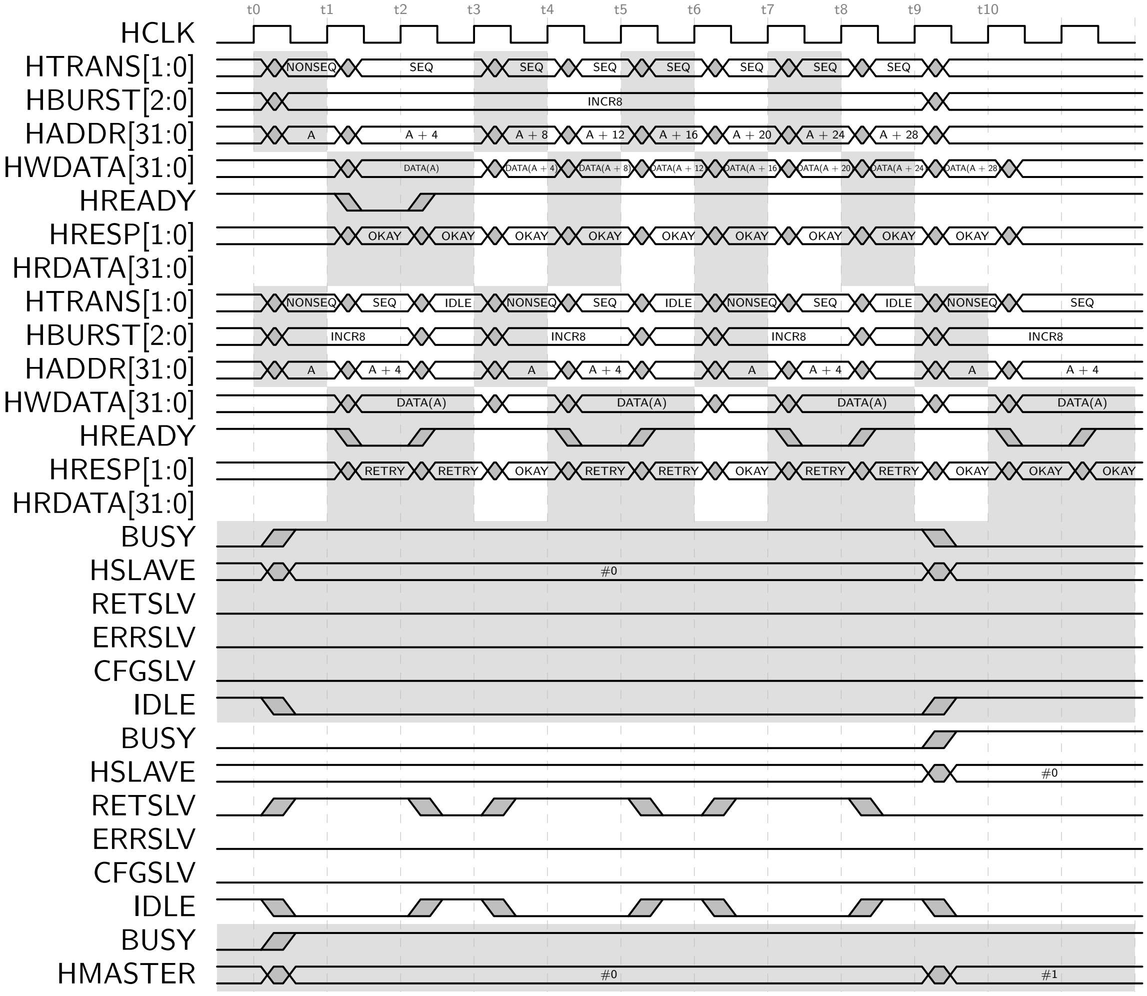

Figure 34. Test 16: two masters and one 1-ws slave

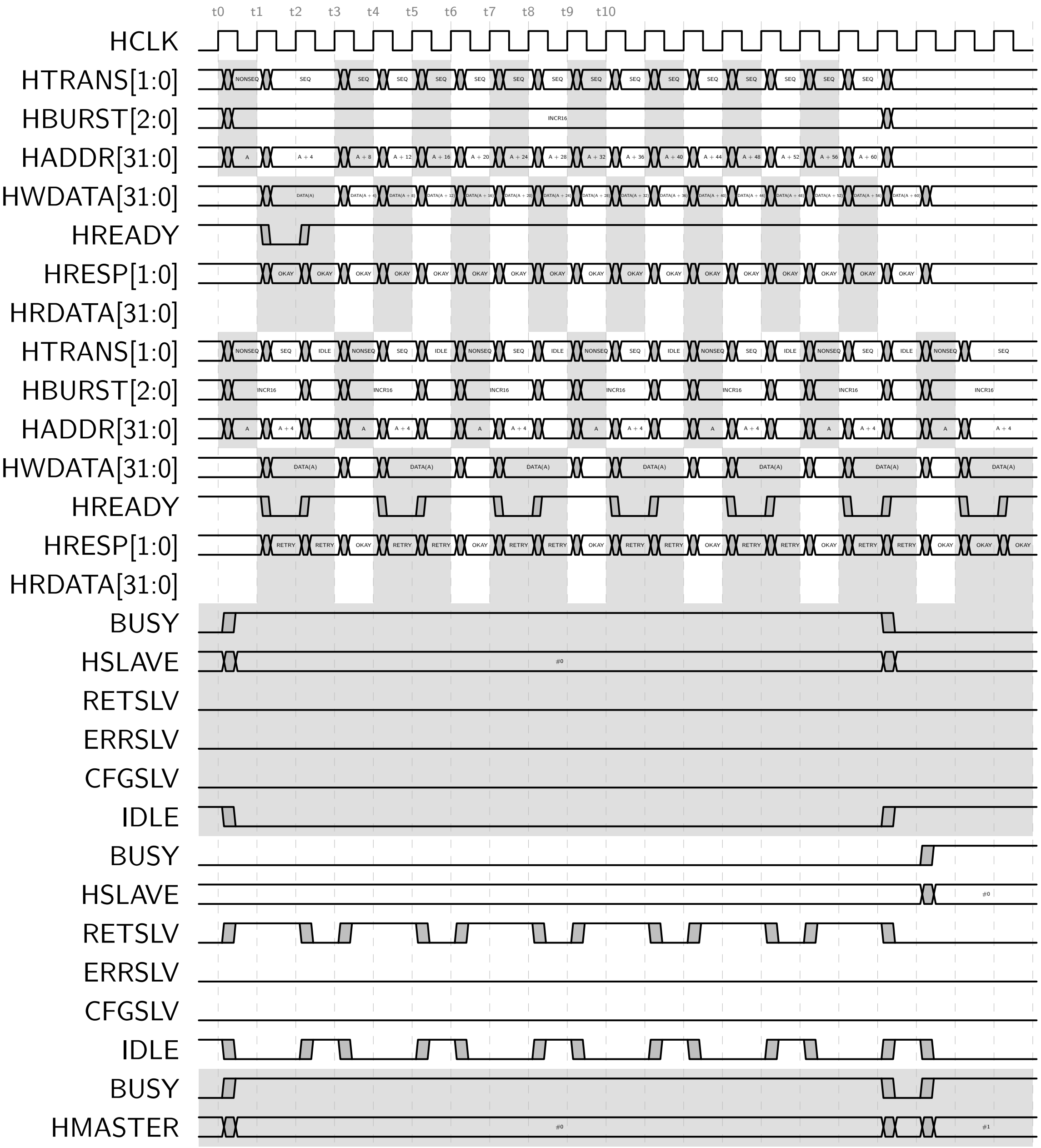
t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10
HCLK
HTRANS[1:0]
NONSEQ SEQ SEQ SEQ SEQ SEQ SEQ SEQ SEQ SEQ SEQ SEQ SEQ SEQ SEQ SEQ
HBURST[2:0]
INCR16
HADDR[31:0]
A A + 4 A + 8 A + 12 A + 16 A + 20 A + 24 A + 28 A + 32 A + 36 A + 40 A + 44 A + 48 A + 52 A + 56 A + 60
HWDATA[31:0]
DATA(A) DATA(A + 4) DATA(A + 8) DATA(A + 12) DATA(A + 16) DATA(A + 20) DATA(A + 24) DATA(A + 28) DATA(A + 32) DATA(A + 36) DATA(A + 40) DATA(A + 44) DATA(A + 48) DATA(A + 52) DATA(A + 56) DATA(A + 60)
HREADY
HRESP[1:0]
OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY OKAY
HRDATA[31:0]
HTRANS[1:0]
NONSEQ SEQ IDLE NONSEQ SEQ IDLE NONSEQ SEQ IDLE NONSEQ SEQ IDLE NONSEQ SEQ IDLE NONSEQ SEQ IDLE NONSEQ SEQ
HBURST[2:0]
INCR16 INCR16 INCR16 INCR16 INCR16 INCR16 INCR16
HADDR[31:0]
A A + 4 A A + 4 A A + 4 A A + 4 A A + 4 A A + 4 A A + 4
HWDATA[31:0]
DATA(A) DATA(A) DATA(A) DATA(A) DATA(A) DATA(A) DATA(A)
HREADY
HRESP[1:0]
RETRY RETRY OKAY RETRY RETRY OKAY RETRY RETRY OKAY RETRY RETRY OKAY RETRY RETRY OKAY RETRY RETRY OKAY OKAY OKAY
HRDATA[31:0]
BUSY
HSLAVE
#0
RETSLV
ERRSLV
CFGSLV
IDLE
BUSY
HSLAVE
#0
RETSLV
ERRSLV
CFGSLV
IDLE
BUSY
HMASTER
#0
#1

Figure 35. Test 17: two masters and one 1-ws slave

4.3.7 Tests 18 and 19

In tests 18 and 19 we do not have any slave in our test bench entity. The purpose of these tests is to control the proper behavior of the dummy ERROR slave and the configuration slave. In test 18 two masters request a read transfer from an address on the configuration area. The AHB Crossbar Controller provides this piece of information by reading it from the plug&play ROM and concurrently responds to the masters, inserting 1 wait state. In test 19 we have two masters requesting two different non-existing addresses at the same clock cycle. Each master is routed to its dummy ERROR slave and given a two-cycle ERROR response. The test results are shown in Figure 36 and Figure 37.

In Test 18 address X maps to the plug&play configuration area on the AHB Crossbar Controller.

In Test 19, addresses X and Y are not present in the system and are not mapped to any slave, therefore an ERROR response is generated.

4.3.8 Test 20

Test 20 is a special case of test 14 (seen in Subsection 4.3.6). In this test we check that the AHB Crossbar Controller keep the proper slave locked to a master which inserts BUSY states during an incremental burst (confront the specification in Subsection 1.4.4.1). The test results are shown in Figure 38.

Address A maps to slave #0.

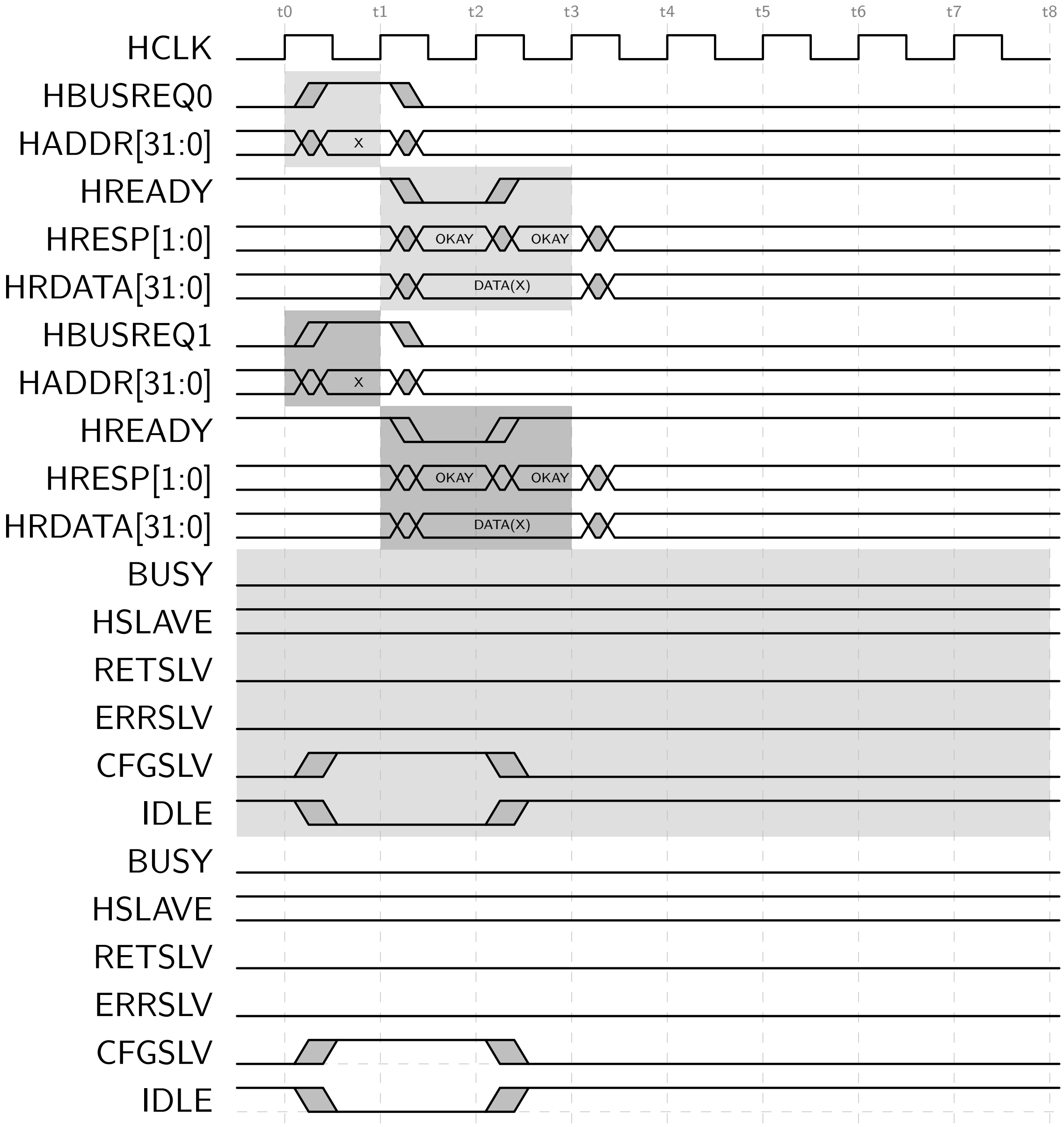

Figure 36. Test 18: two masters communicating with the AHB Crossbar Controller

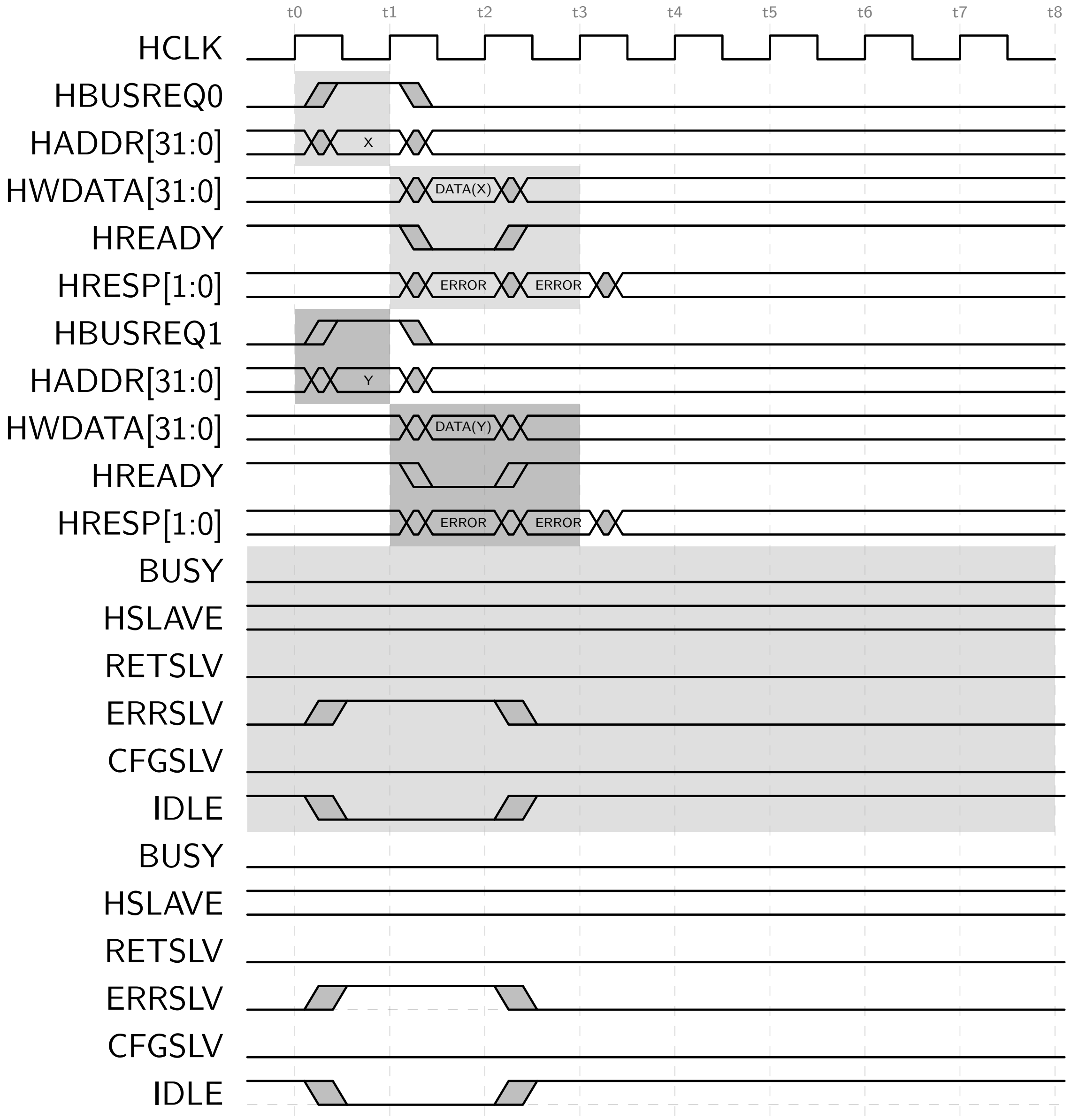

Figure 37. Test 19: two masters communicating with the AHB Crossbar Controller

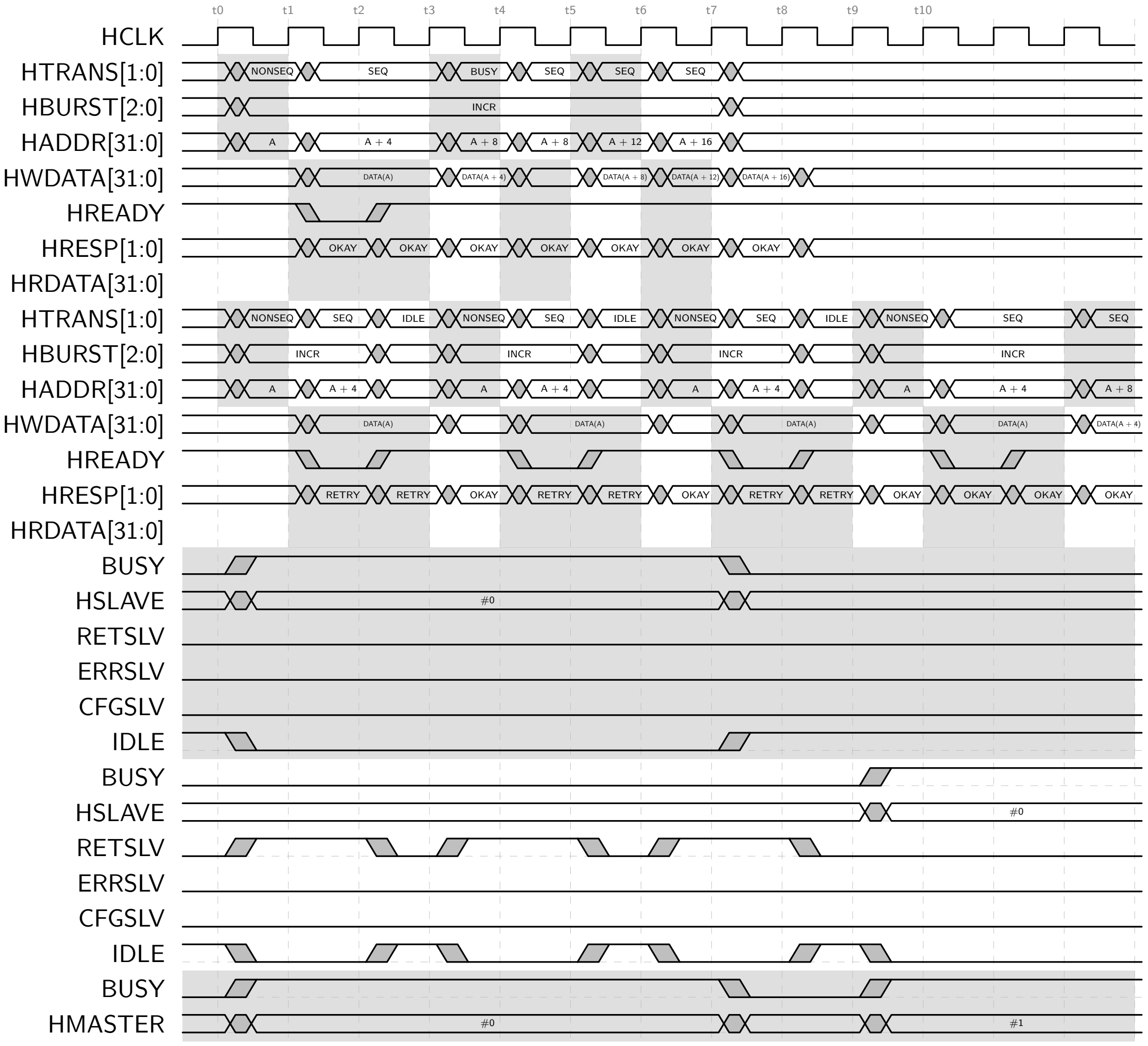

Figure 38. Test 20: two masters and one 1-ws slave

4.4 Transfer time comparison

An useful measure of performance gain of the AHB Crossbar Controller is the speedup obtained by comparing the total transfer time of the previous tests and comparing it with the total transfer time obtained in the same tests performed on the shared bus topology, by the means of the old AHB Controller. Table IV shows the total number of clock cycles required to complete all the transfers in the previous test benches. The time is measured in number of clock cycles. Each clock cycle lasts 20 ns, as the system clock used in the tests is 50 MHz, as stated before. The total time is measured from the beginning of the issuing of the address on the bus, by the earliest issuing master in the test bench, to the end of the issued data by the latest issuing master in the test bench. For instance, test 4 total duration is 3 clock cycles, as seen in Figure 22 on page 80. A speedup figure is also shown in the table for comparison.

Speedup was calculated as: $\frac{\{cycles\_on\_bus\}}{\{cycles\_on\_crossbar\}}$

Table IV shows drastic speedups in test benches which are highly parallel. For instance, test benches 4, 5 and 6, as seen in Subsection 4.3.2, consist of completely parallel transfers between two master-slave couples. These tests represent the perfect case where a crossbar improves the timing by a large factor. The theoretical speedup of 2 is in fact very much approximated by the speedup obtained in this series of tests.

Tests 7 through 12 are mixed access tests where parallelism is limited. Nonetheless these tests show an improvement in the total transfer time in the system using the crossbar, seen as a speedup of between 7% and 30%, except for test 7, where there is a 17% performance degradation, probably due to the fast response of the 0 wait state slave compared to the relative slowness of the two-cycle responses of the dummy slaves.

Tests 1 through 3, 13 through 17 and test 20 do not involve parallel transfer, but instead colliding accesses among masters were tested. The crossbar behavior in these cases does not degrade performance significantly. Instead, it shows an overall slight improvement in the total execution times of the tests. We can furthermore consider that performance degradation was consistently detected only for tests with 0 wait-state slaves.

Tests 18 and 19 show again how parallel access to slaves is significantly improved by the use of a crossbar, with speedups approximating the theoretical maximum of 2 (two masters are involved).

TABLE IV

NUMBER OF CLOCK CYCLES REQUIRED BY TEST BENCHES AND RELATIVE SPEEDUP

Test #	**# of cycles on crossbar (AHB Crossbar Controller)**	**# of cycles on bus (AHB Controller)**	**Speedup**
1	5	3	0.60
2	6	5	0.83
3	7	7	1.00
4	3	5	1.67
5	5	9	1.80
6	7	13	1.86
7	6	5	0.83
8	8	9	1.13
9	10	13	1.30
10	5	6	1.20
11	9	11	1.22
12	15	16	1.07
13	14	13	0.93
14	13	14	1.08
15	12	12	1.00
16	18	20	1.11
17	36	36	1.00
18	3	5	1.67
19	3	5	1.67
20	17	16	0.94

CHAPTER 5

FAULT TOLERANCE ANALYSIS

In this chapter we are going to explore the possible outcome of Single Event Upset faults on our AHB Crossbar Controller. We are going to create a model of the faults, able to replicate them in a deterministic way. We are then going to run a series of tests to determine the behavior of our entity in the event an SEU strike a memory element.

5.1 Single Event Upset faults

A Single Event Upset (SEU) fault is a type of soft error which generally occurs when deposited charge directly causes a change of state in dynamic circuit memory elements (e.g. flip-flop, latch) (9). In other words, an SEU occurs when a charged particle changes the stored value in a memory element from logic “1” to logic “0”, or vice versa.

SEU is the most common soft fault which can be observed. A soft fault is an error which is temporary, one that does not permanently destroy the hardware, and that, in theory, can be detected and corrected. The SEU most common cause is ionizing or electro-magnetic radiation, especially galactic cosmic rays. Energetic particles, for example, protons trapped in the Van Allen radiation belts, can deposit unwanted charge in a microelectronic device. The excess charge builds up and can manifest as an SEU. While this holds in space, in a terrestrial environment SEUs are caused by cosmic particles colliding with atoms in the atmosphere and creating cascades or showers of neutrons and protons, which in turn may interact with electronics.

5.2 Faults in the AHB Crossbar Control

As seen previously, the AHB Crossbar Controller makes use of memory elements, as it is implemented with the structured VHDL design approach explained in Section 2.1. Specifically, the *r* record, part of the crossbar control structure (refer to Subsection 3.9.1), is the output of the sequential part of the entity. As such, only the *r* record is affected by possible SEUs.

I envisioned a method to programmatically simulate a test bench entity and execute a bit-flip (from logic “1” to logic “0”, or vice versa) at a specified time. The method, which simulates an SEU, is implemented as a TCL/TK script for ModelSim. I run the script on a series of test benches and programmatically confronted the resulted signal outputs with the outputs generated during normal operation. This way I was able to see when an SEU changed the behavior of the AHB Crossbar Controller, generating a failure, or when the resulting behavior did not change, therefore generating what is called a ‘silent failure’.

5.3 Test methodology and results

To perform my tests I simulated an SEU fault which strikes record *r* of the AHB Crossbar Controller once, on a certain bit, at a certain time. Every possible bit in the sequential memory of the crossbar is flipped, and every occurrence of an SEU is run as a separate test. The record’s used memory bits for the masters are shown in Table V on page 106, while the memory bits used for the slaves are shown in Table VI. It can be seen that the total number of bits used in record r of the crossbar varies with the number of masters and with the number of slaves. For instance, a system which is configured with N masters and M slaves will have $N \times (91 + \lceil log_2 M \rceil) + M \times (1 + \lceil log_2 N \rceil)$ bits in the r record of the AHB Crossbar Controller. The system used in my tests has two masters and two slaves, so the total number of bits which can suffer from an SEU fault is 188. Furthermore, all the SEUs are tested for every clock cycle throughout the duration of the transmission. In the simplest case, for instance, which will be seen in Subsection 5.3.1, we have a single transfer to a 0 wait state slave, happening in parallel in two master-slave couples. The transfer lasts two clock cycles in total, one for the control phase, one for the data phase. Therefore all the SEUs are tested during the first clock cycle, and then during the second, for a total of $188 \times 2 = 376$ tests. This is explained more thoroughly in the next Subsections. The code for the test benches and for the scripts used in the following Subsections is shown in Appendix B.

5.3.1 Test 1: single transfer

The first batch of tests shows an SEU fault happening on a random location of the r record of the crossbar during a single write transfer transmission which happens in parallel in two master-slave couples. The proper behavior of the crossbar in the absence of the SEU is shown in Figure 39. I programmatically run a script simulating an SEU on every bit of the r record at time 80 ns and then, in a separate batch of tests, at 100 ns, one clock later. The SEU has the effect of possibly influencing the behavior of the AHB Crossbar Controller. The correct behavior of the test bench, as shown in Figure 39, was tested by

TABLE V

RECORD STRUCTURE REG_MASTER_TYPE ENTRIES FOR EACH MASTER AND NUMBER OF BITS USED

Field name	# of bits
busy	1
hslave	$\lceil log_2\{\#of slaves\}\rceil$
retslv	1
errslv	1
cfgsel	1
idle	1
haddr	14
hrdatam	32
hrdatas	32
cfga11	1
hready	1
htrans	2
beat	4

TABLE VI

RECORD STRUCTURE REG_SLAVE_TYPE ENTRIES FOR EACH SLAVE AND NUMBER OF BITS USED

Field name	# of bits
busy	1
hmaster	$\lceil log_2\{\#of masters\}\rceil$

matching the address and data input of the RAM which is situated on the slaves with the desired inputs. In case the signals differ, then the SEU has modified the behavior of the AHB Crossbar Controller, and we can state that we are experiencing a failure.

Table VII and Table VIII highlight the failures detected during the test for, respectively, the first and the second transmission cycles, with fault injections happening at 80 ns and 100 ns. For each record entry in the *r* record, we show the number of failures and of silent failures. If a record is multi-bit (e.g. *hrdatam*), the numbers refer to the aggregated number of tests/bits on said record.

Not only does the record control the data bus connections in the crossbar, it is also read by the arbiter inside the AHB Crossbar Controller. If an SEU strikes one of the bits of the record structure that primarily control the crossbar behavior, namely the *busy*, *retslv*, *errslv* or *cfgsel* fields, then some misbehavior might occur.

We have seen that, at timing 80 ns, a change on *retslv*, *errslv* or *cfgsel* modifies the data bus connection on the crossbar and connects the master to the dummy slaves, which are by default in *hready* state 0. This faulty connection starts a fake two-cycle response transfer which was not expected.

At timing 100 ns a change in the slave control signals, mistakenly connects them to the wrong master, or to a default master, in case their *busy* record is flipped from 1 to 0. This is another faulty connection that is not expected and which results in a failure. Any other SEU is handled by the crossbar as a silent failure, because it either affects a field that is not used at that moment, like most of the failures in the dummy slave internal

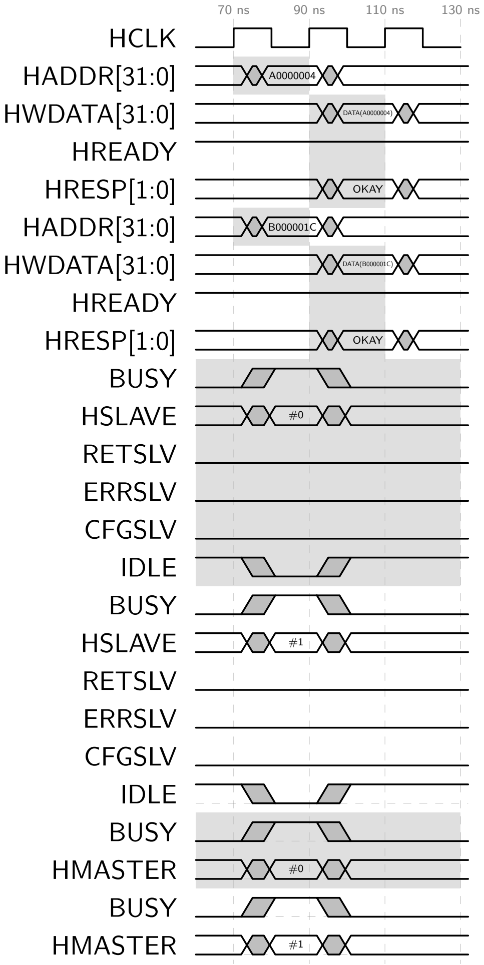

Figure 39. Test 1 correct behavior: two masters accessing two different slaves in parallel

TABLE VII

TEST 1 RESULTS: INJECTION AT 80 NS

Record entry	**# of failures**	**# of silent failures**
m(1).busy	0	1
m(1).hslave	0	1
m(1).retslv	**1**	0
m(1).errslv	**1**	0
m(1).cfgsel	**1**	0
m(1).idle	0	1
m(1).haddr	0	14
m(1).hrdatam	0	32
m(1).hrdatas	0	32
m(1).cfga11	0	1
m(1).hready	0	1
m(1).htrans	0	2
m(1).beat	0	4
m(0).busy	0	1
m(0).hslave	0	1
m(0).retslv	**1**	0
m(0).errslv	**1**	0
m(0).cfgsel	**1**	0
m(0).idle	0	1
m(0).haddr	0	14
m(0).hrdatam	0	32
m(0).hrdatas	0	32
m(0).cfga11	0	1
m(0).hready	0	1
m(0).htrans	0	2
m(0).beat	0	4
s(1).busy	0	1
s(1).hmaster	0	1
s(0).busy	0	1
s(0).hmaster	0	1

TABLE VIII

TEST 1 RESULTS: INJECTION AT 100 NS

Record entry	**# of failures**	**# of silent failures**
m(1).busy	0	1
m(1).hslave	0	1
m(1).retslv	0	1
m(1).errslv	0	1
m(1).cfgsel	0	1
m(1).idle	0	1
m(1).haddr	0	14
m(1).hrdatam	0	32
m(1).hrdatas	0	32
m(1).cfga11	0	1
m(1).hready	0	1
m(1).htrans	0	2
m(1).beat	0	4
m(0).busy	0	1
m(0).hslave	0	1
m(0).retslv	0	1
m(0).errslv	0	1
m(0).cfgsel	0	1
m(0).idle	0	1
m(0).haddr	0	14
m(0).hrdatam	0	32
m(0).hrdatas	0	32
m(0).cfga11	0	1
m(0).hready	0	1
m(0).htrans	0	2
m(0).beat	0	4
s(1).busy	**1**	0
s(1).hmaster	**1**	0
s(0).busy	**1**	0
s(0).hmaster	**1**	0

signals, or because of priorities inside the crossbar which ignore signals set to wrong values in case other signals are asserted, i.e. if a master is both *busy* and *retslv*, then *retslv* will be ignored because *busy* has a higher priority in the circuitry of the crossbar.

5.3.2 Test 2: burst transfer

A similar test was repeated with a different test bench configuration. This time the SEU injections were tested on the crossbar while two parallel 4-burst sequential write transfers were executed on the two master-slave couples. This time the total length of the transfer is 5 clock cycles, as the transfer is 4-cycle long and, being pipelined and 0 wait-state, we have to add one additional cycle. The correct behavior of the transfer is shown in Figure 40. In total we run $188 \times 5 = 940$ tests, whose results are shown in Table IX and Table X below.

As in the previous test, we have seen that, at clock cycle 1 (timing from 70 ns to 90 ns), a change on the master's *retslv*, *errslv* or *cfgsel* signals modifies the data bus connection on the crossbar and connects the master to the dummy slaves, which are by default in *hready* state 0. This faulty connection starts a fake two-cycle response transfer which was not expected. Additionally, a bit flip in the *busy* signal of slave 0 creates a failure in the system. We inspected the failure and found that the behavior of the crossbar connects the master requesting slave 0 to the retry dummy slave as it considers the slave 0 'busy'.

At clock cycles 2 and 3 (90 ns to 110 ns and 110 ns to 130 ns), failures are generated when an SEU strikes the sensitive masters' *retslv*, *errslv* or *cfgsel* signals or the slaves' *busy* or *hmaster* signals.

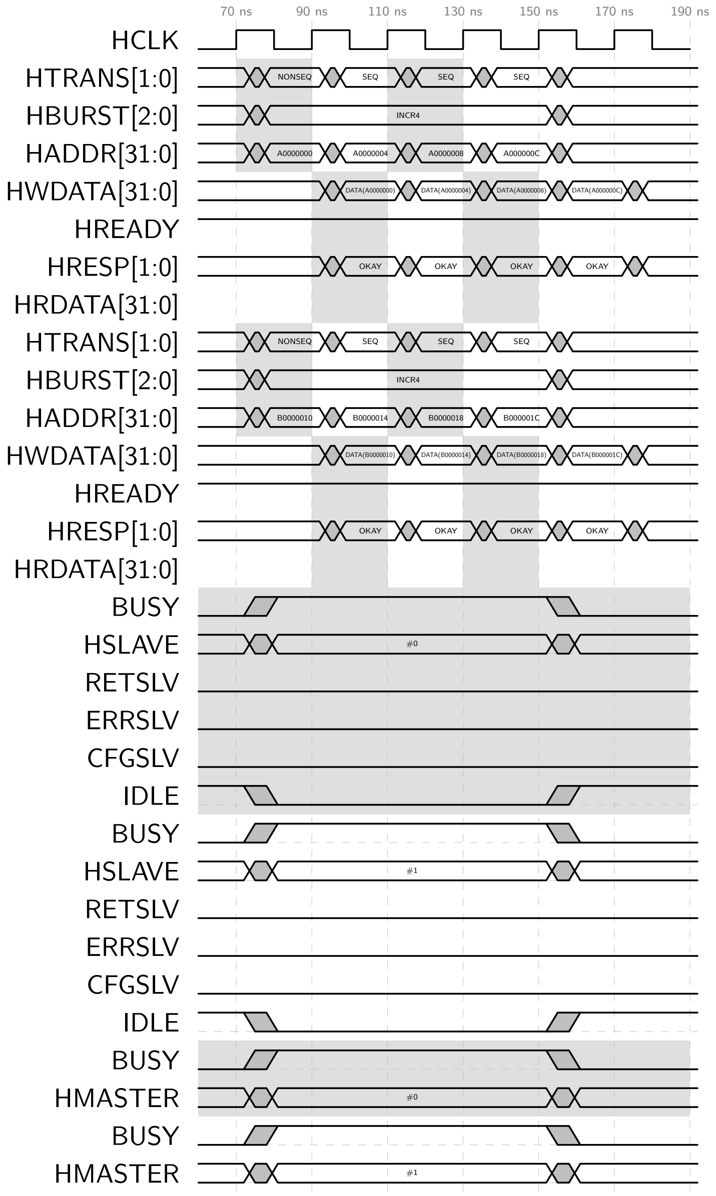

Figure 40. Test 2 correct behavior: two masters accessing two different slaves in parallel with 4-beat burst transfers

TABLE IX

TEST 2 RESULTS: INJECTIONS AT 80 NS, 100 NS AND 120 NS

Record entry	Cycle 1		Cycle 2		Cycle 3	
	# of f.	**# of s. f.**	**# of f.**	**# of s. f.**	**# of f.**	**# of s. f.**
m(1).busy	0	1	0	1	0	1
m(1).hslave	0	1	0	1	0	1
m(1).retslv	**1**	0	**1**	0	**1**	0
m(1).errslv	**1**	0	**1**	0	**1**	0
m(1).cfgsel	**1**	0	**1**	0	**1**	0
m(1).idle	0	1	0	1	0	1
m(1).haddr	0	14	0	14	0	14
m(1).hrdatam	0	32	0	32	0	32
m(1).hrdatas	0	32	0	32	0	32
m(1).cfga11	0	1	0	1	0	1
m(1).hready	0	1	0	1	0	1
m(1).htrans	0	2	0	2	0	2
m(1).beat	0	4	0	4	0	4
m(0).busy	0	1	0	1	0	1
m(0).hslave	0	1	0	1	0	1
m(0).retslv	**1**	0	**1**	0	**1**	0
m(0).errslv	**1**	0	**1**	0	**1**	0
m(0).cfgsel	**1**	0	**1**	0	**1**	0
m(0).idle	0	1	0	1	0	1
m(0).haddr	0	14	0	14	0	14
m(0).hrdatam	0	32	0	32	0	32
m(0).hrdatas	0	32	0	32	0	32
m(0).cfga11	0	1	0	1	0	1
m(0).hready	0	1	0	1	0	1
m(0).htrans	0	2	0	2	0	2
m(0).beat	0	4	0	4	0	4
s(1).busy	0	1	**1**	0	**1**	0
s(1).hmaster	0	1	**1**	0	**1**	0
s(0).busy	**1**	0	**1**	0	**1**	0
s(0).hmaster	0	1	**1**	0	**1**	0

TABLE X

TEST 2 RESULTS: INJECTIONS AT 140 NS AND 160 NS

Record entry	Cycle 4		Cycle 5	
	# of f.	# of s. f.	# of f.	# of s. f.
m(1).busy	0	1	0	1
m(1).hslave	0	1	0	1
m(1).retslv	0	1	0	1
m(1).errslv	0	1	0	1
m(1).cfgsel	0	1	0	1
m(1).idle	0	1	0	1
m(1).haddr	0	14	0	14
m(1).hrdatam	0	32	0	32
m(1).hrdatas	0	32	0	32
m(1).cfga11	0	1	0	1
m(1).hready	0	1	0	1
m(1).htrans	0	2	0	2
m(1).beat	0	4	0	4
m(0).busy	0	1	0	1
m(0).hslave	0	1	0	1
m(0).retslv	0	1	0	1
m(0).errslv	0	1	0	1
m(0).cfgsel	0	1	0	1
m(0).idle	0	1	0	1
m(0).haddr	0	14	0	14
m(0).hrdatam	0	32	0	32
m(0).hrdatas	0	32	0	32
m(0).cfga11	0	1	0	1
m(0).hready	0	1	0	1
m(0).htrans	0	2	0	2
m(0).beat	0	4	0	4
s(1).busy	**1**	0	**1**	0
s(1).hmaster	**1**	0	**1**	0
s(0).busy	**1**	0	**1**	0
s(0).hmaster	**1**	0	**1**	0

At clock cycles 4 and 5 (130 ns to 150 ns and 150 ns to 170 ns) the masters' signals are not sensitive to faults anymore because of the burst transfer locking behavior. Failures are generated nonetheless by SEUs which strike the slaves' signals, creating a misconnection on the crossbar and an erroneous data read by the slave.

From the data in Table VII through Table X, we can deduce that some values of the crossbar *r* record are more sensitive to SEU faults. In particular we see that masters' and slaves' signals which are direct inputs to the crossbar structure (i.e. masters' *retslv*, *errslv* and *cfgsel*, and slaves' *busy* and *hmaster*) are particularly sensitive to this kind of faults during a write transfer. We can infer that for a read transfer also the masters' *busy* and *hslave* signals will be sensitive to SEU faults, which generally cause a misconnection in the proper crossbar structure which routes the master inputs. Faults on the *idle* signal of the crossbar are easily recovered: a master whose idle is de-asserted by an SEU (bit flip from "1" to "0") is simply going to ignore the data on its inputs anyhow. A master which is not idle but gets an asserted *idle* signal from an SEU (bit flip "0" to "1") will ignore the *idle* signal as the other fields (*busy*, *retslv*, *errslv* and *cfgsel*) take a precedence over it.

The record fields used by the master's dummy slaves, which were not directly tested in these chapter, compose the vast majority of the bits of the *r* record structure, as can be seen in Table V (*hrdatam* and *hrdatas* for instance are 32-bit fields each one). These values are rarely used in a real-case scenario, but nonetheless are fundamental for the correct behavior in case of arbitration or special connections on the crossbar, such as error and configuration access.

CHAPTER 6

CONCLUSIONS AND FURTHER DEVELOPMENT

The study of the LEON 3 processor system, with its versatility and modularity, has been the base for the development of my work. By simulating the whole computing system I was able to analyze a real-case scenario for the use of an intercommunication network, namely the AHB bus. Furthermore, I studied the working details of a multi-processor system, and the features of the cache system of the LEON 3 and the way it handles coherency among the cores. This study was actualized in the adaptation of the AHB standard to work on a crossbar structure.

The use of a mature method of writing VHDL code enabled us to create a functional entity, the AHB Crossbar Controller, as described in Chapter 3, which is made up of more than 8 hundred code lines. The entity was tested with more than 20 test benches, for a total of more than 2 thousand lines of code, to simulate several real-case-usage models and different behaviors of masters and slaves communication in a system. The correctness of the test benches was then checked and the total transfer time of each test bench was compared with the transfer time of the same test bench run on a bus topology, instead of a crossbar.

After the proper functioning of the entity was ensured, we run several tests to characterize the behavior of the AHB Crossbar Controller in case of random SEU faults striking the circuit. A comprehensive set of more than 1300 tests was run where we simulated the bit flipping of every static memory cell of the circuit, at every cycle of the transfer, in two common test cases: two parallel single-beat transfers and two parallel burst transfers.

The results of performance testing of Chapter 4 confirmed that a crossbar network topology drastically improves parallel transfers among master-slave couples (see Table IV, tests 4, 5, 6, 18 and 19). We found that this type of transfer reports a speedup of, at minimum, 67% on two-master systems, quite close to the theoretical 100% speedup. Overall performance degradation was reportedly not a problem and even most transfers with low or no parallelism were subjected to a speedup.

The fault tolerance analysis highlighted some sensitive areas in the static memory used by the circuit, as seen in Chapter 5. Specifically, we reported that the AHB Crossbar Controller static-memory elements which directly feed the internal crossbar control signals, are more susceptible to Single Event Upset faults.

The AHB Crossbar Controller, as is, could work on a single-core single-master LEON 3 system but it would be unable to operate on a multi-core one, because of the way the LEON 3 handles cache coherency. As bus snooping is used to ensure coherency between the in-core caches (see Subsection 1.3.2 on page 9), the crossbar adaptation would not be able to serve the processors with the needed data, as we do not have a unique bus anymore, but several buses working in parallel. In theory, a two-core system could be adapted to work with the AHB Crossbar Controller, but there are theoretical considerations that prevent the system, as is, to work for three or more cores. The cache coherency part of the LEON

3 would therefore have to be modified. For instance a central directory scheme (10) could be implemented inside the AHB Crossbar Controller, or the cache could be moved out of the core and onto the intercommunication network.

Nonetheless, the AHB Crossbar Controller would be able to operate, and increase performance through parallelism, for non-cached memory accesses. So, for instance, a multi-core LEON 3 system with cache disabled could be operating with the crossbar as is. Otherwise we envision a single-core LEON 3 system which shares the crossbar with other AHB master devices accessing only non-cacheable memory entries, or accessing read-only memory or, else, not performing writes but only reads.

Finally the creation of the AHB Crossbar Controller and its fault tolerance analysis enables us to further our work towards software fault tolerance. In this thesis we have seen how a crossbar would be an efficient network topology for logically partitioned computing systems, because, in these systems, communication happens in parallel almost exclusively between cores and memories couples. Future expansions of the work could delve into the creation of an hypervisor which could manage multiple logical partitions of a computing system, running in parallel the same code we want to examine in the fault tolerance analysis. The hypervisor would then check the consistence of the data execution results of the code among the partitions, and would be able to detect the presence of a fault in one of the partitions if the results differ. This kind of software fault detection would have in the crossbar its weak spot, as possible SEUs on the crossbar hardware structure could not be detected by the hypervisor. The crossbar fault tolerance analysis we carried out in this thesis tries to analyze this unwanted situation in the pursuit of possible remedies for addressing this inconvenient in a more general setup.

APPENDICES

Appendix A

AHB_CB_CTRL - AMBA AHB CROSSBAR CONTROLLER CODE

```
--
   -------------------------------------------------------------------------------------------

--  This file is a part of the GRLIB VHDL IP LIBRARY
--  Copyright (C) 2003 - 2008, Gaisler Research
--  Copyright (C) 2008 - 2012, Aeroflex Gaisler
--
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 2 of the License, or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this program; if not, write to the Free Software
--  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
-----------------------------------------------------------------------------------------------
-- Entity:      ahbctrl
-- File:        ahbctrl.vhd
-- Author:      Jiri Gaisler, Gaisler Research
-- Modified:    Edvin Catovic, Gaisler Research
-- Description: AMBA arbiter, decoder and multiplexer with plug&play support
--
   -------------------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.stdlib.all;
use grlib.amba.all;
-- pragma translate_off
use grlib.devices.all;
use std.textio.all;
-- pragma translate_on

entity ahbctrl_cb is
```

Appendix A (continued)

```
generic(
  defmast      : integer                    := 0; -- default master
  split        : integer                    := 0; -- split support
  timeout      : integer range 0 to 255     := 0; -- HREADY timeout
  ioaddr       : ahb_addr_type              := 16#fff#; -- I/O area MSB
     address
  iomask       : ahb_addr_type              := 16#fff#; -- I/O area address
     mask
  cfgaddr      : ahb_addr_type              := 16#ff0#; -- config area MSB
     address
  cfgmask      : ahb_addr_type              := 16#ff0#; -- config area
     address mask
  nahbm        : integer range 1 to NAHBMST := NAHBMST; -- number of masters
  nahbs        : integer range 1 to NAHBSLV := NAHBSLV; -- number of slaves
  ioen         : integer range 0 to 15      := 1; -- enable I/O area
  disirq       : integer range 0 to 1       := 0; -- disable interrupt
     routing
  fixbrst      : integer range 0 to 1       := 0; -- support fix-length
     bursts
  debug        : integer range 0 to 2       := 2; -- report cores to console
  fpnpen       : integer range 0 to 1       := 0; -- full PnP configuration
     decoding
  icheck       : integer range 0 to 1       := 1;
  devid        : integer                    := 0; -- unique device ID
  enbusmon     : integer range 0 to 1       := 0; --enable bus monitor
  assertwarn   : integer range 0 to 1       := 0; --enable assertions for
     warnings
  asserterr    : integer range 0 to 1       := 0; --enable assertions for
     errors
  hmstdisable  : integer                    := 0; --disable master checks
  hslvdisable  : integer                    := 0; --disable slave checks
  arbdisable   : integer                    := 0; --disable arbiter checks
  mprio        : integer                    := 0; --master with highest
     priority
  mcheck       : integer range 0 to 2       := 1; --check memory map for
     intersects
  ccheck       : integer range 0 to 1       := 1; --perform sanity checks on
      pnp config
  acdm         : integer                    := 0; --AMBA compliant data
     muxing (for hsize > word)
  index        : integer                    := 0; --Index for trace print-
     out
  ahbtrace     : integer                    := 0; --AHB trace enable
  hwdebug      : integer                    := 0; --Hardware debug
  fourgslv     : integer                    := 0 --1=Single slave with
     single 4 GB bar
);
port(
  rst      : in  std_ulogic;
  clk      : in  std_ulogic;
```

```
    msti    : out ahb_mst_in_vector;
    msto    : in  ahb_mst_out_vector;
    slvi    : out ahb_slv_in_vector;
    slvo    : in  ahb_slv_out_vector;
    testen  : in  std_ulogic := '0';
    testrst : in  std_ulogic := '1';
    scanen  : in  std_ulogic := '0';
    testoen : in  std_ulogic := '1'
  );
end;

architecture rtl of ahbctrl_cb is
  constant nahbmx : integer := 2 ** log2(nahbm);
  type nmstarr is array (1 to 3) of integer range 0 to nahbmx - 1;
  type nvalarr is array (1 to 3) of boolean;

  type reg_master_type is record
    -- hmaster      : integer range 0 to nahbmx -1;
    -- hmasterd     : integer range 0 to nahbmx -1;
    busy          : std_ulogic;
    hslave        : integer range 0 to nahbs - 1;
    retslv        : std_ulogic;
    errslv        : std_ulogic;
    cfgsel        : std_ulogic;
    idle          : std_ulogic;

    haddr         : std_logic_vector(15 downto 2);
    hrdatam       : std_logic_vector(31 downto 0);
    hrdatas       : std_logic_vector(31 downto 0);
    cfga11        : std_ulogic;
    hready        : std_ulogic;
    htrans        : std_logic_vector(1 downto 0);
    beat          : std_logic_vector(3 downto 0);
  end record;

  type reg_slave_type is record
    busy      : std_ulogic;
    hmaster   : integer range 0 to nahbmx - 1;
  end record;

  type reg_mvector_type is array (natural range <>) of reg_master_type;
  type reg_svector_type is array (natural range <>) of reg_slave_type;
  subtype reg_mvector is reg_mvector_type(nahbm - 1 downto 0);
  subtype reg_svector is reg_svector_type(nahbs - 1 downto 0);


  type nhmaster_vector_type is array (natural range <>) of integer range 0 to
      nahbmx - 1;
  subtype nhmaster_vector is nhmaster_vector_type(nahbs -1 downto 0);
```

Appendix A (continued)

```
type nslave_vector_type is array (natural range <>) of natural range 0 to
    31;
subtype nslave_vector is nslave_vector_type(nahbmx -1 downto 0);

type hsel_vector_type is array (natural range <>) of std_logic_vector(0 to
    31);
subtype hsel_vector is hsel_vector_type(nahbmx -1 downto 0);

type hmbsel_vector_type is array (natural range <>) of std_logic_vector(0
    to NAHBAMR - 1);
subtype hmbsel_vector is hmbsel_vector_type(nahbmx -1 downto 0);

type sel_vector_type is array (natural range <>) of std_ulogic;
subtype sel_vector is sel_vector_type(nahbmx -1 downto 0);

type slave_vector_type is array (natural range <>) of std_ulogic;
subtype slave_vector is slave_vector_type(nahbs -1 downto 0);

type hresp_vector_type is array (natural range <>) of std_logic_vector(1
    downto 0);
subtype hresp_vector is hresp_vector_type(nahbmx -1 downto 0);

type bnslave_vector_type is array (natural range <>) of std_logic_vector(3
    downto 0);
subtype bnslave_vector is bnslave_vector_type(nahbmx -1 downto 0);

type data_vector_type is array (natural range <>) of std_logic_vector(31
    downto 0);
subtype data_vector is data_vector_type(nahbmx -1 downto 0);

type cb_type is record
  m       : reg_mvector;
  s       : reg_svector;
end record;

constant primst : std_logic_vector(NAHBMST downto 0) :=
    conv_std_logic_vector(mprio, NAHBMST + 1);
type l0_type is array (0 to 15) of std_logic_vector(2 downto 0);
type l1_type is array (0 to 7) of std_logic_vector(3 downto 0);
type l2_type is array (0 to 3) of std_logic_vector(4 downto 0);
type l3_type is array (0 to 1) of std_logic_vector(5 downto 0);

type tztab_type is array (0 to 15) of std_logic_vector(2 downto 0);

--returns the index number of the highest priority request
--signal in the two lsb bits when indexed with a 4-bit
--request vector with the highest priority signal on the
--lsb. the returned msb bit indicates if a request was
--active ('1' = no request active corresponds to "0000")
```

```
constant tztab : tztab_type := ("100", "000", "001", "000", "010", "000", "
    001", "000", "011", "000", "001", "000", "010", "000", "001", "000");

--calculate the number of the highest priority request signal(up to 64
--requests are supported) in vect_in using a divide and conquer
--algorithm. The lower the index in the vector the higher the priority
--of the signal. First 4-bit slices are indexed in tztab and the msb
--indicates whether there is an active request or not. Then the resulting
--3 bit vectors are compared in pairs (the one corresponding to (3:0) with
--(7:4), (11:8) with (15:12) and so on). If the least significant of the
     two
--contains an active signal a '0' is added to the msb side (the vector
--becomes one bit wider at each level) to the next level to indicate that
--there are active signals in the lower nibble of the two. Otherwise
--the msb is removed from the vector corresponding to the higher nibble
--and "10" is added if it does not contain active requests and "01" if
--does contain active signals. Thus the msb still indicates if the new
--slice contains active signals and a '1' is added if it is the higher
--part. This results in a 6-bit vector containing the index number
--of the highest priority master in 5:0 if bit 6 is '0' otherwise
--no master requested the bus.
function tz(vect_in : std_logic_vector) return std_logic_vector is
  variable vect             : std_logic_vector(63 downto 0);
  variable l0               : l0_type;
  variable l1               : l1_type;
  variable l2               : l2_type;
  variable l3               : l3_type;
  variable l4               : std_logic_vector(6 downto 0);
  variable bci_lsb, bci_msb : std_logic_vector(3 downto 0);
  variable bco_lsb, bco_msb : std_logic_vector(2 downto 0);
  variable sel              : std_logic;
begin
  vect                               := (others => '1');
  vect(vect_in'length - 1 downto 0) := vect_in;

  -- level 0
  for i in 0 to 7 loop
    bci_lsb := vect(8 * i + 3 downto 8 * i);
    bci_msb := vect(8 * i + 7 downto 8 * i + 4);
    --lookup the highest priority request in each nibble
    bco_lsb := tztab(conv_integer(bci_lsb));
    bco_msb := tztab(conv_integer(bci_msb));
    --select which of two nibbles contain the highest priority ACTIVE
    --signal, and forward the corresponding vector to the next level
    sel := bco_lsb(2);
    if sel = '0' then
      l1(i) := '0' & bco_lsb;
    else
      l1(i) := bco_msb(2) & not bco_msb(2) & bco_msb(1 downto 0);
    end if;
```

Appendix A (continued)

```
  end loop;

  -- level 1
  for i in 0 to 3 loop
    sel := l1(2 * i)(3);
    --select which of two 8-bit vectors contain the
    --highest priority ACTIVE signal. the msb set at the previous level
    --for each 8-bit slice determines this
    if sel = '0' then
      l2(i) := '0' & l1(2 * i);
    else
      l2(i) := l1(2 * i + 1)(3) & not l1(2 * i + 1)(3) & l1(2 * i + 1)(2
          downto 0);
    end if;
  end loop;

  -- level 2
  for i in 0 to 1 loop
    --16-bit vectors, the msb set at the previous level for each 16-bit
    --slice determines the higher priority slice
    sel := l2(2 * i)(4);
    if sel = '0' then
      l3(i) := '0' & l2(2 * i);
    else
      l3(i) := l2(2 * i + 1)(4) & not l2(2 * i + 1)(4) & l2(2 * i + 1)(3
          downto 0);
    end if;
  end loop;

  --level 3
  --32-bit vectors, the msb set at the previous level for each 32-bit
  --slice determines the higher priority slice
  if l3(0)(5) = '0' then
    l4 := '0' & l3(0);
  else
    l4 := l3(1)(5) & not l3(1)(5) & l3(1)(4 downto 0);
  end if;

  return (l4);
end;

--invert the bit order of the hbusreq signals located in vect_in
--since the highest hbusreq has the highest priority but the
--algorithm in tz has the highest priority on lsb
function lz(vect_in : std_logic_vector) return std_logic_vector is
  variable vect  : std_logic_vector(vect_in'length - 1 downto 0);
  variable vect2 : std_logic_vector(vect_in'length - 1 downto 0);
begin
  vect := vect_in;
  for i in vect'right to vect'left loop
```

Appendix A (continued)

```
    vect2(i) := vect(vect'left - i);
  end loop;
  return (tz(vect2));
end;

-- Find next master:
--   * 2 arbitration policies: fixed priority or round-robin
--   * Fixed priority: priority is fixed, highest index has highest
    priority
--   * Round-robin: arbiter maintains circular queue of masters
--   * (master 0, master 1, ..., master (nahbmx-1)). First requesting
    master
--   * in the queue is granted access to the bus and moved to the end of
    the queue.
--   * splitted masters are not granted
--   * bus is re-arbited when current owner does not request the bus,
--     or when it performs non-burst accesses
--   * fix length burst transfers will not be interrupted
--   * incremental bursts should assert hbusreq until last access

procedure selmast(r      : in  cb_type;
          -- msto   : in  ahb_mst_out_vector;
          slave  : in integer range 0 to nahbs - 1;
          busym  : inout sel_vector;
          nslave : in nslave_vector;
          lock   : in sel_vector;
          retslv : inout sel_vector;
          errslv : in sel_vector;
          cfgsel : in sel_vector;
          idle   : in sel_vector;
          mast   : out integer range 0 to nahbmx - 1;
          busy   : out  std_logic
          ) is
  variable nmst   : nmstarr;
  variable nvalid : nvalarr;

  variable rrvec : std_logic_vector(nahbmx * 2 - 1 downto 0);
  variable zcnt  : std_logic_vector(log2(nahbmx) + 1 downto 0);
  variable hpvec : std_logic_vector(nahbmx - 1 downto 0);
  variable zcnt2 : std_logic_vector(log2(nahbmx) downto 0);

begin
  nvalid(1 to 3) := (others => false);
  nmst(1 to 3) := (others => 0);
  mast   := r.s(slave).hmaster;

  rrvec := (others => '0');
  --mask requests up to and including current master. Concatenate
  --an unmasked request vector above the masked vector. Otherwise
  --the rules are the same as for fixed priority
```

Appendix A (continued)

```
for i in 0 to nahbmx - 1 loop
  if (i <= r.s(slave).hmaster)  then
    rrvec(i) := '0';
  else
    if (nslave(i) = slave) and (cfgsel(i) /= '1') and (retslv(i) /= '1')
       and (errslv(i) /= '1') and (idle(i) /= '1') then
      rrvec(i) := busym(i); -- msto(i).hbusreq;
    else
      rrvec(i) := '0';
    end if;
  end if;
  if (nslave(i) = slave) and (cfgsel(i) /= '1') and (retslv(i) /= '1')
     and (errslv(i) /= '1') and (idle(i) /= '1') then
    rrvec(nahbmx + i) := busym(i); -- msto(i).hbusreq;
  else
    rrvec(nahbmx + i) := '0';
  end if;
end loop;
--find the next master uzing tz which gives priority to lower
--indexes
zcnt := tz(rrvec)(log2(nahbmx) + 1 downto 0);
--was there a master requesting the bus?
if zcnt(log2(nahbmx) + 1) = '0' then
  nvalid(2) := true;
end if;
nmst(2) := conv_integer(zcnt(log2(nahbmx) - 1 downto 0));

-- if we have a locked slave (for instance when a burst transfer is in
   progress) we lock it to the previous master
if r.s(slave).busy = '1' and lock(r.s(slave).hmaster) = '1' then
  nmst(1)   := r.s(slave).hmaster;
  nvalid(1) := true;
end if;
--if no other master is requesting the bus select the current one
--          nmst(3) := r.s(slave).hmaster;
--          nvalid(3) := true;

--select the next master. If for round robin a high priority master
--(mprio) requested the bus if nvalid(1) is true. Otherwise
--if nvalid(2) is true at least one master was requesting the bus
--and the one with highest priority was selected. If none of these
--were true then the default master is selected (nvalid(3) true)
busy := '0';
for i in 1 to 3 loop
  if nvalid(i) then
    mast := nmst(i);
    busy := '1';
    for k in 0 to nahbmx - 1 loop
      if k /= nmst(i) and busym(k) = '1' and (nslave(k) = slave) then
        busym(k) := '0';
```

Appendix A (continued)

```
          retslv(k) := '1';
        end if;
      end loop;
      exit;
    end if;
  end loop;
end;

constant MIMAX    : integer                          := log2x(nahbmx) - 1;
constant SIMAX    : integer                          := log2x(nahbs) - 1;
constant IOAREA   : std_logic_vector(11 downto 0) := conv_std_logic_vector(
    ioaddr, 12);
constant IOMSK    : std_logic_vector(11 downto 0) := conv_std_logic_vector(
    iomask, 12);
constant CFGAREA  : std_logic_vector(11 downto 0) := conv_std_logic_vector(
    cfgaddr, 12);
constant CFGMSK   : std_logic_vector(11 downto 0) := conv_std_logic_vector(
    cfgmask, 12);
constant FULLPNP  : boolean                          := (fpnpen /= 0);

signal r, rin            : cb_type;  -- reg_vector;
signal rsplit, rsplitin  : std_logic_vector(0 to nahbmx - 1);
-- signal rcb, rcbin: cb_type;

-- pragma translate_off
signal lmsti : ahb_mst_in_type;
signal lslvi : ahb_slv_in_type;
-- pragma translate_on

begin
comb : process(rst, msto, slvo, r, rsplit, testen, testrst, scanen, testoen
    )
  variable area     : std_logic_vector(1 downto 0);

  variable hconfndx : integer range 0 to 7;

  variable defmst   : std_ulogic;
  variable tmpv     : std_logic_vector(0 to nahbmx - 1);

  -- no need to vectorize
  variable hgrant  : std_logic_vector(0 to NAHBMST - 1); -- bus grant
  variable v       : cb_type;       -- reg_vector;
  variable vsplit  : std_logic_vector(0 to nahbmx - 1);
  variable vslvi   : ahb_slv_in_type;
  -- variable vcb: cb_type;

  -- vectorized
  --        variable arb        : sel_vector;
  variable idle       : sel_vector;
  variable busym      : sel_vector;
```

Appendix A (continued)

```
  variable busys     : slave_vector;
  variable hready    : sel_vector;
  variable hresp     : hresp_vector;
  variable nhmaster  : nhmaster_vector;
  variable nslave    : nslave_vector;
  variable hsel    : hsel_vector; -- slave select
  variable hmbsel  : hmbsel_vector;
  variable cfgsel  : sel_vector;
  variable retslv  : sel_vector;
  variable errslv  : sel_vector;
  variable bnslave  : bnslave_vector;
  variable hrdata   : data_vector;
  variable hrdatam   : data_vector;
  variable hrdatas   : data_vector;
  variable lock      : sel_vector;

  variable hcache  : std_ulogic;
  --variable hrdata : std_logic_vector(AHBDW - 1 downto 0);
  variable haddr   : std_logic_vector(31 downto 0);
  variable hirq    : std_logic_vector(NAHBIRQ - 1 downto 0);

begin
  v       := r;
  hgrant := (others => '1');
  defmst := '0';

  -- loop masters and grant all masters (requesting)
  -- degrant also
  -- TODO: split logic
  for j in 0 to nahbm - 1 loop
    -- slave decoding
    haddr := msto(j).haddr;

    hsel(j)      := (others => '0');
    hmbsel(j)    := (others => '0');
    busym(j)     := '0';
    idle(j)      := '0';
    cfgsel(j)    := '0';
    errslv(j)    := '0';
    retslv(j)    := '0';
    lock(j)    := '0';

    if fourgslv = 0 then
      for i in 0 to nahbs - 1 loop
        for k in NAHBIR to NAHBCFG - 1 loop
          area := slvo(i).hconfig(k)(1 downto 0);
          case area is
            when "10" =>
              if ((ioen = 0) or ((IOAREA and IOMSK) /= (haddr(31 downto 20)
                  and IOMSK))) and ((slvo(i).hconfig(k)(31 downto 20) and
```

Appendix A (continued)

```
              slvo(i).hconfig(k)(15 downto 4)) = (haddr(31 downto 20)
              and slvo(i).hconfig(k)(15 downto 4))) and (slvo(i).
              hconfig(k)(15 downto 4
            ) /= "000000000000") then
            hsel(j)(i) := '1';
            hmbsel(j)(k - NAHBIR) := '1';
          end if;
        when "11" =>
          if ((ioen /= 0) and ((IOAREA and IOMSK) = (haddr(31 downto
              20) and IOMSK))) and ((slvo(i).hconfig(k)(31 downto 20)
              and slvo(i).hconfig(k)(15 downto 4)) = (haddr(19 downto
              8) and slvo(i).hconfig(k)(15 downto 4))) and (slvo(i).
              hconfig(k)(15 downto 4
            ) /= "000000000000") then
            hsel(j)(i) := '1';
            hmbsel(j)(k - NAHBIR) := '1';
          end if;
        when others =>
      end case;
    end loop;
  end loop;
else
  -- There is only one slave on the bus. The slave has only one bar,
      which
  -- maps 4 GB address space.
  hsel(j)(0) := '1';
  hmbsel(j)(0) := '1';
end if;

-- convert unary to binary representation
bnslave(j)(0) := hsel(j)(1) or hsel(j)(3) or hsel(j)(5) or hsel(j)(7)
    or hsel(j)(9) or hsel(j)(11) or hsel(j)(13) or hsel(j)(15);
bnslave(j)(1) := hsel(j)(2) or hsel(j)(3) or hsel(j)(6) or hsel(j)(7)
    or hsel(j)(10) or hsel(j)(11) or hsel(j)(14) or hsel(j)(15);
bnslave(j)(2) := hsel(j)(4) or hsel(j)(5) or hsel(j)(6) or hsel(j)(7)
    or hsel(j)(12) or hsel(j)(13) or hsel(j)(14) or hsel(j)(15);
bnslave(j)(3) := hsel(j)(8) or hsel(j)(9) or hsel(j)(10) or hsel(j)(11)
     or hsel(j)(12) or hsel(j)(13) or hsel(j)(14) or hsel(j)(15);

nslave(j) := conv_integer(bnslave(j)(SIMAX downto 0));

-- idle transfer
if (msto(j).htrans = HTRANS_IDLE) then
  idle(j)  := '1';
-- config access
elsif (((((IOAREA and IOMSK) = (haddr(31 downto 20) and IOMSK)) and (
    ioen /= 0)) or ((IOAREA = haddr(31 downto 20)) and (ioen = 0))) and
     ((CFGAREA and CFGMSK) = (haddr(19 downto 8) and CFGMSK)) and (
    cfgmask /= 0)) then
  cfgsel(j) := '1';
  hsel(j) := (others => '0');
-- decoding unknown slave? error!
elsif (nslave(j) = 0) and (hsel(j)(0) = '0') then
  errslv(j) := '1';
-- then we are busy if we are accessing a slave and doing a NONSEQ or
   SEQ or BUSY transfer
else --if msto(j).htrans = HTRANS_NONSEQ or msto(j).htrans = HTRANS_SEQ
     or msto(j).htrans = HTRANS_BUSY then
  -- if requested slave is not ready, then retry
  if slvo(nslave(j)).hready = '1' then
    busym(j) := '1';
    ---------------------------------------------- BURST PRIORITY LOGIC
    lock(j) := '1';
    case msto(j).htrans is
      when HTRANS_NONSEQ =>
        case msto(j).hburst is
          when HBURST_SINGLE => lock(j) := '0';
          when HBURST_INCR => lock(j) := msto(j).hbusreq; -- NOTE out
              of specs case??!!
          when others =>
        end case;
      when HTRANS_SEQ =>
        case msto(j).hburst is
          when HBURST_WRAP4  | HBURST_INCR4  => if (fixbrst = 1) and (r
              .m(j).beat(1 downto 0) = "11")   then lock(j) := '0'; end
               if;
          when HBURST_WRAP8  | HBURST_INCR8  => if (fixbrst = 1) and (r
              .m(j).beat(2 downto 0) = "111")  then lock(j) := '0'; end
               if;
          when HBURST_WRAP16 | HBURST_INCR16 => if (fixbrst = 1) and (r
              .m(j).beat(3 downto 0) = "1111") then lock(j) := '0'; end
               if;
          when HBURST_INCR => lock(j) := msto(j).hbusreq; -- last
              transfer of an INCR burst will have low busreq
          when others =>
        end case;
      when others => lock(j) := '1';
    end case;
    ----------------------------------------------
  else
    retslv(j) := '1';
  end if;
end if;

-- waitstates handling logic
-- TODO this code might be moved in a more appropriate location, like
   in the third loop
if(busym(j) = '1' and r.m(j).busy = '1' and slvo(r.m(j).hslave).hready
   = '0') then
  busym(j) := '0';
```

Appendix A (continued)

```
      retslv(j) := '1';
    elsif (busym(j) = '1' and (r.m(j).retslv = '1' or r.m(j).errslv = '1'
        or r.m(j).cfgsel = '1') and r.m(j).hready = '0') then
      busym(j) := '0';
      retslv(j) := '1';
    end if;
  end loop;


  -- now loop the slaves to detect conflicts
  -- TODO arbitrate between conflicting configuration accesses

  for j in 0 to nahbs - 1 loop
    if slvo(j).hready = '1' then
      busys(j) := '0';
      selmast(r, j, busym, nslave, lock, retslv, errslv, cfgsel, idle,
          nhmaster(j), busys(j));
      v.s(j).hmaster := nhmaster(j);
      v.s(j).busy := busys(j);
      --v.s(j).lock := lock(nhmaster(j));
    end if;
  end loop;

  -- now loop masters, answering with the embedded slaves

  for j in 0 to nahbm - 1 loop

    -- this if block can move to other previous for iterations
    if cfgmask /= 0 then
      -- plug&play information for masters
      if FULLPNP then
        hconfndx := conv_integer(r.m(j).haddr(4 downto 2));
      else
        hconfndx := 0;
      end if;
      if (r.m(j).haddr(10 downto MIMAX+6) = zero32(10 downto MIMAX+6)) and
          (FULLPNP or (r.m(j).haddr(4 downto 2) = "000")) then
        v.m(j).hrdatam := msto(conv_integer(r.m(j).haddr(MIMAX+5 downto 5))
            ).hconfig(hconfndx);
      else
        v.m(j).hrdatam := (others => '0');
      end if;

      -- plug&play information for slaves
      if (r.m(j).haddr(10 downto SIMAX+6) = zero32(10 downto SIMAX+6)) and
          (FULLPNP or (r.m(j).haddr(4 downto 2) = "000") or (r.m(j).haddr
          (4) = '1')) then
        v.m(j).hrdatas := slvo(conv_integer(r.m(j).haddr(SIMAX+5 downto 5))
            ).hconfig(conv_integer(r.m(j).haddr(4 downto 2)));
      else
```

Appendix A (continued)

```
      v.m(j).hrdatas := (others => '0');
    end if;

    -- device ID, library build and potentially debug information
    if r.m(j).haddr(10 downto 4) = "1111111" then
      if hwdebug = 0 or r.m(j).haddr(3 downto 2) = "00" then
        v.m(j).hrdatas(15 downto 0) := conv_std_logic_vector(
            LIBVHDL_BUILD, 16);
        v.m(j).hrdatas(31 downto 16) := conv_std_logic_vector(devid, 16);
      elsif r.m(j).haddr(3 downto 2) = "01" then
        for i in 0 to nahbmx-1 loop v.m(j).hrdatas(i) := msto(i).hbusreq;
            end loop;
      else
        for i in 0 to nahbmx-1 loop v.m(j).hrdatas(i) := rsplit(i); end
            loop;
      end if;
    end if;
  end if;

  v.m(j).hready := '0';

  if( r.m(j).busy = '1' ) then -- if master was busy, we read it's data
      bus slave ready.
    hready(j) := slvo(r.m(j).hslave).hready;
  else
    if( r.m(j).retslv = '0' and r.m(j).errslv = '0' and r.m(j).cfgsel =
        '0') then
      hready(j) := '1';
    else
      if (r.m(j).htrans = HTRANS_IDLE) or (r.m(j).htrans = HTRANS_BUSY)
          then
        hresp(j) := HRESP_OKAY;
        hready(j) := '1';
      else
      -- two cycle response from internal slaves
        if r.m(j).retslv = '1' then
          hresp(j) := HRESP_RETRY;
        elsif r.m(j).errslv = '1' then
          hresp(j) := HRESP_ERROR;
        elsif r.m(j).cfgsel = '1' then
          hresp(j) := HRESP_OKAY;
          hrdata(j) := (others => '0'); -- point of this line?
          if r.m(j).cfga11 = '0' then
            hrdata(j) := ahbdrivedata(r.m(j).hrdatam);
          else
            hrdata(j) := ahbdrivedata(r.m(j).hrdatas);
          end if;
        end if;

        hready(j) := r.m(j).hready;
```

Appendix A (continued)

```
        v.m(j).hready := not r.m(j).hready;
      end if;
    end if;
  end if;


  if (hready(j) = '1') then
    v.m(j).busy := busym(j);
    v.m(j).hslave := nslave(j);
    v.m(j).retslv := retslv(j);
    v.m(j).errslv := errslv(j);
    v.m(j).cfgsel := cfgsel(j);
    v.m(j).idle := idle(j);
    v.m(j).htrans := msto(j).htrans;
    v.m(j).haddr := msto(j).haddr(15 downto 2);
    v.m(j).cfga11 := msto(j).haddr(11);
    -- increment burst counter
    if (msto(j).htrans = HTRANS_NONSEQ) or (msto(j).htrans = HTRANS_IDLE)
        then
      v.m(j).beat := "0001";
    elsif (msto(j).htrans = HTRANS_SEQ) then
      v.m(j).beat := r.m(j).beat + 1;
    end if;
  end if;
end loop;


-- at this point every entry of nslave contains the correct slave. There
    won't be two entries with the same slave.
-- we can connect our crossbar.

-- interrupt merging
hirq := (others => '0');
if disirq = 0 then
  for i in 0 to nahbs - 1 loop
    hirq := hirq or slvo(i).hirq;
  end loop;
  for i in 0 to nahbm - 1 loop
    hirq := hirq or msto(i).hirq;
  end loop;
end if;


-- PROPER CROSSBAR

-- masters loop
for i in 0 to nahbm - 1 loop
  msti(i).hgrant <= (others => '0');           msti(i).hgrant(i) <= '1';

  if( r.m(i).busy = '1') then
```

Appendix A (continued)

```
      -- data bus
      msti(i).hready <= slvo(r.m(i).hslave).hready;
      msti(i).hresp <= slvo(r.m(i).hslave).hresp;
      msti(i).hrdata <= slvo(r.m(i).hslave).hrdata;
      msti(i).hcache <= slvo(r.m(i).hslave).hcache;
      msti(i).hirq <= hirq;
    elsif( r.m(i).retslv = '1' or r.m(i).errslv = '1' or r.m(i).cfgsel =
        '1') then
      msti(i).hready <= hready(i);
      msti(i).hresp <= hresp(i);
      msti(i).hrdata <= hrdata(i);
      msti(i).hcache <= 'X';
      msti(i).hirq <= hirq;
    else -- idle
      -- data bus
      msti(i).hready <= '1';
      msti(i).hresp <= HRESP_OKAY;
      msti(i).hrdata <= (others => 'X');
      msti(i).hcache <= 'X';
      msti(i).hirq <= hirq;
    end if;
    msti(i).testen  <= testen;
    msti(i).testrst <= testrst;
    msti(i).scanen  <= scanen and testen;
    msti(i).testoen <= testoen;
  end loop;


  -- slaves loop
  for i in 0 to nahbs - 1 loop
    if(v.s(i).busy = '1') then
      -- control bus
      slvi(i).hsel    <= (others => '0');     slvi(i).hsel(i) <= '1';
      slvi(i).haddr      <= msto(v.s(i).hmaster).haddr;
      slvi(i).hwrite     <= msto(v.s(i).hmaster).hwrite;
      slvi(i).htrans     <= msto(v.s(i).hmaster).htrans;
      slvi(i).hsize      <= msto(v.s(i).hmaster).hsize;
      slvi(i).hburst     <= msto(v.s(i).hmaster).hburst;
      slvi(i).hprot      <= msto(v.s(i).hmaster).hprot;
      slvi(i).hmaster    <= conv_std_logic_vector(v.s(i).hmaster, 4);
      slvi(i).hmastlock    <= msto(v.s(i).hmaster).hlock;
    else
      slvi(i).hsel       <= (others => '0');
      slvi(i).haddr      <= (others => 'X');
      slvi(i).hwrite     <= 'X';
      slvi(i).htrans     <= (others => 'X');
      slvi(i).hsize      <= (others => 'X');
      slvi(i).hburst     <= (others => 'X');
      slvi(i).hprot      <= (others => 'X');
      slvi(i).hmaster    <= (others => 'X');
      slvi(i).hmastlock <= 'X';
    end if;

    if(r.s(i).busy = '1') then
      -- data bus
      slvi(i).hwdata <= msto(r.s(i).hmaster).hwdata;
    else
      slvi(i).hwdata <= (others => 'X');
    end if;

    -- slvi(i).hcache <= ;
    slvi(i).hirq <= hirq;
    slvi(i).hready <= slvo(i).hready;
    slvi(i).testen <= testen;
    slvi(i).testrst <= testrst;
    slvi(i).scanen <= scanen and testen;
    slvi(i).testoen <= testoen;
  end loop;

  -- reset operation
  if (rst = '0') then
  -- reset master records
    for i in 0 to nahbm - 1 loop
      v.m(i).busy := '0';
      v.m(i).hslave := 0;
      v.m(i).cfgsel := '0';
      v.m(i).retslv := '0';
      v.m(i).errslv := '0';
      v.m(i).idle := '0';

      v.m(i).haddr     := (others => '0');
      v.m(i).hrdatam   := (others => '0');
      v.m(i).hrdatas   := (others => '0');
      v.m(i).cfga11    := '0';
      v.m(i).hready    := '0';
      v.m(i).htrans    := (others => '0');
    end loop;
  -- reset slave records
    for i in 0 to nahbs - 1 loop
      v.s(i).busy := '0';
      v.s(i).hmaster := nahbmx-1;
      --v.s(i).lock := '0';
    end loop;
  end if;
  rin       <= v;
end process;

reg0 : process(clk)
begin
  if rising_edge(clk) then
```

Appendix A (continued)

```
      r    <= rin;
    end if;
  end process;

end;
```

Appendix B

FAULT TOLERANCE TESTING CODE

B.1 Test 1 testbench code

```
-- two masters accessing same slave at the same time

library ieee;
use ieee.std_logic_1164.all;

library gaisler;
use gaisler.ahbtbp.all;

library grlib, techmap;
use grlib.amba.all;
use techmap.gencomp.all;

use work.config.all;

entity thesis_test_1 is
  generic(
    clkperiod : integer := 20        -- system clock period
  );
end entity thesis_test_1;

architecture RTL of thesis_test_1 is
  signal clk          : std_ulogic         := '0';
  signal rst          : std_ulogic;
  signal rstn         : std_ulogic;
  signal ctrl0, ctrl1 : ahbtb_ctrl_type;
  signal ahbsi        : ahb_slv_in_vector;
  signal ahbso        : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi        : ahb_mst_in_vector;
  signal ahbmo        : ahb_mst_out_vector := (others => ahbm_none);

  constant ct         : integer := clkperiod / 2;
  constant BOARD_FREQ : integer := 50000; -- input frequency in KHz
  constant CPU_FREQ   : integer := BOARD_FREQ * CFG_CLKMUL / CFG_CLKDIV; --
      cpu frequency in KHz
  constant IOAEN      : integer := CFG_CAN + CFG_ATA + CFG_GRUSBDC;
begin
  rstn <= not rst;

  rst <= '0' after 0 ns, '1' after 20 ns;
```

Appendix B (continued)

```
clk <= not clk after ct * 1 ns;

ahbtbm0 : ahbtbm
  generic map(hindex => 0)          -- AMBA master index 0
  port map(rst, clk, ctrl0.i, ctrl0.o, ahbmi(0), ahbmo(0));

ahbtbm1 : ahbtbm
  generic map(hindex => 1)          -- AMBA master index 1
  port map(rst, clk, ctrl1.i, ctrl1.o, ahbmi(1), ahbmo(1));

--   ahb0 : ahbctrl                          -- AHB arbiter/multiplexer
--     generic map(defmast => CFG_DEFMST, split => CFG_SPLIT, rrobin =>
    CFG_RROBIN, ioaddr => CFG_AHBIO, ioen => IOAEN, nahbm => maxahbm, nahbs
     => 8, ahbtrace => 1)
--     port map(rst, clk, ahbmi, ahbmo, ahbsi, ahbso);

ahb0 : ahbctrl_cb                         -- AHB arbiter/multiplexer - 2 masters
    & 2 slaves
  generic map(defmast => CFG_DEFMST, split => 0, ioaddr => CFG_AHBIO, ioen
      => IOAEN, nahbm => 2, nahbs => 2, ahbtrace => 1)
  port map(rst, clk, ahbmi, ahbmo, ahbsi, ahbso);

ahbtbs0 : ahbtbs
  generic map(
    hindex => 0,
    haddr  => 16#A00#,
    ws     => "00000000"
  )
  port map(
    rst   => rst,
    clk   => clk,
    ahbsi => ahbsi(0),
    ahbso => ahbso(0)
  );

ahbtbs1 : ahbtbs
  generic map(
    hindex => 1,
    haddr  => 16#B00#,
    ws     => "00000000"
  )
  port map(
    rst   => rst,
    clk   => clk,
    ahbsi => ahbsi(1),
    ahbso => ahbso(1)
  );

tb0 : process is                          -- testbench for master 0
```

```
  begin

    -- Initialize the control signals
    ahbtbminit(ctrl0);

    -- Write 0x12345678 to address 0x40000000. Print access.
    ahbwrite(x"A0000004", x"12345678", "10", "10", '0', 2, true, ctrl0);

    -- Stop simulation
    ahbtbmdone(1, ctrl0);

    wait;
  end process tb0;

  tb1 : process is                      -- testbench for master 1
  begin

    -- Initialize the control signals
    ahbtbminit(ctrl1);

    ahbwrite(x"B000001C", x"0FEDCBA9", "10", "10", '0', 2, true, ctrl1);

    -- Stop simulation
    ahbtbmdone(1, ctrl1);

    wait;
  end process tb1;

end architecture RTL;
```

B.2 Test 2 testbench code

```
-- two masters accessing same slave at the same time

library ieee;
use ieee.std_logic_1164.all;

library gaisler;
use gaisler.ahbtbp.all;

library grlib, techmap;
use grlib.amba.all;
use techmap.gencomp.all;

use work.config.all;

entity thesis_test_2 is
  generic(
```

Appendix B (continued)

```
    clkperiod : integer := 20        -- system clock period
  );
end entity thesis_test_2;

architecture RTL of thesis_test_2 is
  signal clk          : std_ulogic          := '0';
  signal rst          : std_ulogic;
  signal rstn         : std_ulogic;
  signal ctrl0, ctrl1 : ahbtb_ctrl_type;
  signal ahbsi        : ahb_slv_in_vector;
  signal ahbso        : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi        : ahb_mst_in_vector;
  signal ahbmo        : ahb_mst_out_vector := (others => ahbm_none);

  constant ct         : integer := clkperiod / 2;
  constant BOARD_FREQ : integer := 50000; -- input frequency in KHz
  constant CPU_FREQ   : integer := BOARD_FREQ * CFG_CLKMUL / CFG_CLKDIV; --
      cpu frequency in KHz
  constant IOAEN      : integer := CFG_CAN + CFG_ATA + CFG_GRUSBDC;
  constant maxahbm    : integer := CFG_NCPU + CFG_AHB_UART + CFG_GRETH +
      CFG_AHB_JTAG + CFG_SPW_NUM * CFG_SPW_EN + CFG_GRUSB_DCL +
      CFG_SVGA_ENABLE + CFG_ATA + CFG_GRUSBDC;
begin
  rstn <= not rst;

  rst <= '0' after 0 ns, '1' after 20 ns;

  clk <= not clk after ct * 1 ns;

  ahbtbm0 : ahbtbm
    generic map(hindex => 0)         -- AMBA master index 0
    port map(rst, clk, ctrl0.i, ctrl0.o, ahbmi(0), ahbmo(0));

  ahbtbm1 : ahbtbm
    generic map(hindex => 1)         -- AMBA master index 1
    port map(rst, clk, ctrl1.i, ctrl1.o, ahbmi(1), ahbmo(1));

--   ahb0 : ahbctrl                          -- AHB arbiter/multiplexer
--     generic map(defmast => CFG_DEFMST, split => CFG_SPLIT, rrobin =>
      CFG_RROBIN, ioaddr => CFG_AHBIO, ioen => IOAEN, nahbm => maxahbm, nahbs
       => 8, ahbtrace => 1)
--     port map(rst, clk, ahbmi, ahbmo, ahbsi, ahbso);

  ahb0 : ahbctrl_cb                       -- AHB arbiter/multiplexer
    generic map(defmast => CFG_DEFMST, split => 0, ioaddr => CFG_AHBIO, ioen
       => IOAEN, nahbm => 2, nahbs => 2, ahbtrace => 1)
    port map(rst, clk, ahbmi, ahbmo, ahbsi, ahbso);

  ahbtbs0 : ahbtbs
    generic map(
```

Appendix B (continued)

```
    hindex => 0,
    haddr  => 16#A00#,
    ws     => "00000000"
  )
  port map(
    rst   => rst,
    clk   => clk,
    ahbsi => ahbsi(0),
    ahbso => ahbso(0)
  );

ahbtbs1 : ahbtbs
  generic map(
    hindex => 1,
    haddr  => 16#B00#,
    ws     => "00000000"
  )
  port map(
    rst   => rst,
    clk   => clk,
    ahbsi => ahbsi(1),
    ahbso => ahbso(1)
  );

tb0 : process is                        -- testbench for master 0
begin

  -- Initialize the control signals
  ahbtbminit(ctrl0);

  -- Write 0x12345678 to address 0x40000000. Print access.
  ahbwrite(x"A0000000", x"11111111", "10", "10", "011", 2, false, ctrl0);
  ahbwrite(x"A0000004", x"22222222", "10", "11", "011", 2, false, ctrl0);
  ahbwrite(x"A0000008", x"33333333", "10", "11", "011", 2, false, ctrl0);
  ahbwrite(x"A000000C", x"44444444", "10", "11", "011", 2, true, ctrl0);

  -- Stop simulation
  ahbtbmdone(1, ctrl0);

  wait;
end process tb0;

tb1 : process is                        -- testbench for master 1
begin

  -- Initialize the control signals
  ahbtbminit(ctrl1);

  ahbwrite(x"B0000010", x"55555555", "10", "10", "011", 2, false, ctrl1);
  ahbwrite(x"B0000014", x"66666666", "10", "11", "011", 2, false, ctrl1);
```

```
    ahbwrite(x"B0000018", x"77777777", "10", "11", "011", 2, false, ctrl1);
    ahbwrite(x"B000001C", x"88888888", "10", "11", "011", 2, true, ctrl1);

    -- Stop simulation
    ahbtbmdone(1, ctrl1);

    wait;
  end process tb1;

end architecture RTL;
```

B.3 Injection script

The following code was used to simulate injections in the crossbar control structure in Test 1 and Test 2 in Chapter 5.

```
# bit flip script that flips bits of all signals present in
    thesis_signal_list_1 and saves data in thesis_test_results_1/
# Copyright Andrea Gianarro 2012
proc bit_flip {s} {
  if {[examine $s] == 0} {
    force -deposit $s 1
  } else {
    force -deposit $s 0
  }
}

proc assert {t s value} {
  if {[examine -radix Hex -time $t $s] == $value} {
    return 1
  } else {
    return 0
  }
}

proc tests {fp} {

  set test_list {}
  lappend test_list [assert 91 sim:/thesis_test_1/ahbtbs0/ramaddr 01]
  lappend test_list [assert 111 sim:/thesis_test_1/ahbtbs0/ramdata 12345678]

  lappend test_list [assert 91 sim:/thesis_test_1/ahbtbs1/ramaddr 07]
  lappend test_list [assert 111 sim:/thesis_test_1/ahbtbs1/ramdata 0FEDCBA9]

  set temp_bool 1
  foreach t $test_list {
```

Appendix B (continued)

```
    set temp_bool [expr $temp_bool && $t]
  }

  puts $fp "$temp_bool $test_list"
}

set fp [open "thesis_signal_list_1" "r"]
set fp_ck1 [open "thesis_test_results_1/summary_ck1.txt" "w"]
set fp_ck2 [open "thesis_test_results_1/summary_ck2.txt" "w"]
set file_data [read $fp]
close $fp
set signals [split $file_data "\n"]

#set s [lindex $signals 0]

# bit flip of control register bits during first clock cycle
foreach s $signals {
  log -r sim:/thesis_test_1/*
  run 80ns
  bit_flip $s
  run 40ns
  # simulation run, now check results
  tests $fp_ck1
  restart -force
}
close $fp_ck1

# bit flip of control register bits during second clock cycle
foreach s $signals {
  log -r sim:/thesis_test_1/*
  run 100ns
  bit_flip $s
  run 20ns
  # simulation run, now check results
  tests $fp_ck2
  restart -force
}
close $fp_ck2

unset signals file_data



# bit flip script that flips bits of all signals present in
    thesis_signal_list_2 and saves data in thesis_test_results_2/
# Copyright Andrea Gianarro 2012
proc bit_flip {s} {
  if {[examine $s] == 0} {
    force -deposit $s 1
  } else {
    force -deposit $s 0
```

Appendix B (continued)

```
  }
}

proc assert {t s value} {
  if {[examine -radix Hex -time $t $s] == $value} {
    return 1
  } else {
    return 0
  }
}

proc tests {fp} {

  set test_list {}

  lappend test_list [assert 91 sim:/thesis_test_2/ahbtbs0/ramaddr 00]
  lappend test_list [assert 111 sim:/thesis_test_2/ahbtbs0/ramdata 11111111]

  lappend test_list [assert 111 sim:/thesis_test_2/ahbtbs0/ramaddr 01]
  lappend test_list [assert 131 sim:/thesis_test_2/ahbtbs0/ramdata 22222222]

  lappend test_list [assert 131 sim:/thesis_test_2/ahbtbs0/ramaddr 02]
  lappend test_list [assert 151 sim:/thesis_test_2/ahbtbs0/ramdata 33333333]

  lappend test_list [assert 151 sim:/thesis_test_2/ahbtbs0/ramaddr 03]
  lappend test_list [assert 171 sim:/thesis_test_2/ahbtbs0/ramdata 44444444]

  lappend test_list [assert 91 sim:/thesis_test_2/ahbtbs1/ramaddr 04]
  lappend test_list [assert 111 sim:/thesis_test_2/ahbtbs1/ramdata 55555555]

  lappend test_list [assert 111 sim:/thesis_test_2/ahbtbs1/ramaddr 05]
  lappend test_list [assert 131 sim:/thesis_test_2/ahbtbs1/ramdata 66666666]

  lappend test_list [assert 131 sim:/thesis_test_2/ahbtbs1/ramaddr 06]
  lappend test_list [assert 151 sim:/thesis_test_2/ahbtbs1/ramdata 77777777]

  lappend test_list [assert 151 sim:/thesis_test_2/ahbtbs1/ramaddr 07]
  lappend test_list [assert 171 sim:/thesis_test_2/ahbtbs1/ramdata 88888888]

  set temp_bool 1
  foreach t $test_list {
    set temp_bool [expr $temp_bool && $t]
  }

  puts $fp "$temp_bool $test_list"
}

# list of signals subject to bit flipping
set fp [open "thesis_signal_list_2" "r"]
set file_data [read $fp]
```

```
close $fp
set signals [split $file_data ”\n”]

#set s [lindex $signals 0]

# number of clocks where bit flipping will happen
for {set x 1} {$x<=5} {incr x} {
  # results file
  set fp_res [open ”thesis_test_results_2/summary_ck$x.txt” ”w”]
  #set s [lindex $signals 0]
  foreach s $signals {
    log −r sim:/thesis_test_2/∗
    set t1 [expr 60 + 20∗$x]
    run  $t1 ns
     bit_flip $s
    run [expr 200 − $t1] ns
    # simulation run, now check results
     tests $fp_res
     restart −force
  }
  close $fp_res
}

unset signals file_data
```

CITED LITERATURE

1. Richard Venia, A.: Comparing IP integration approaches for FPGA implementation. EE Times Programmable Logic Designline, Feb 2008.

2. Brinkmann, A., Niemann, J., Hehemann, I., Langen, D., Porrmann, M., and Ruckert, U.: On-chip interconnects for next generation system-on-chips. In ASIC/SOC Conference, 2002. 15th Annual IEEE International, pages 211–215. IEEE, 2002.

3. Pasricha, S. and Dutt, N.: On-Chip Communication Architectures: System on Chip Interconnect. Morgan Kaufmann Series in Systems on Silicon. Elsevier / Morgan Kaufmann Publishers, 2008.

4. GRLIB IP library user's manual. Technical Report 1.1.0 B4100, Gaisler Research, October 2010.

5. GRLIB IP core user's manual. Technical Report 1.1.0 B4104, Aeroflex Gaisler, November 2010.

6. AMBA specification. Technical Report 2.0 A, ARM Limited, March 1999.

7. Gaisler, J.: A structured vhdl design method. Fault-tolerant microprocessors for space applications, pages 41–50, 2010.

8. Gaisler, J.: Fault-tolerant microprocessors for space applications. Gaisler, Research AB, Sweden, http://www. gaisler. com/doc/vhdl2proc.pdf, Last Accessed February, 6, 2008.

9. Iniewski, K.: Radiation Effects in Semiconductors. Devices, Circuits, and Systems. Taylor & Francis, 2010.

10. Gupta, A., Weber, W., and Mowry, T.: Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. 1990.

VITA

NAME: Andrea Gianarro

EDUCATION: Bachelor of Science in Computer Engineering, Polytechnic University of Turin, 2010

Master of Science in Computer Engineering, Polytechnic University of Turin, 2012

Master of Science in Electrical and Computer Engineering, University of Illinois at Chicago, 2013