Politecnico di Torino

Master's Degree Thesis

Advanced High-performance Bus (AHB) architecture verification

Master's Degree in Computer Engineering

Supervisor:
Prof. Ernesto Sanchez

Co-Supervisors:
M.Sc. Erwin Knittel (Apple)
M.Sc. Jonathan Burdalo (Apple)

Candidate:
Emilio Vivenzio (S262266)

October 2021

"A robot must obey the orders given it by human beings except where such orders would conflict with the First Law."

from Handbook of Robotics, 56th Edition, 2058 A.D.
Isaac Asimov

Abstract

Nowadays, semiconductor devices, also referred as integrated circuits and microchips are essential components in everyday electronic products. Silently they are the heroes of the technological world: they work, power, pilot, control an ever-increasing number of objects - from toys to wrist watches, from smartphones to cars and home automation products. Thanks to technological improvements such as the increasing computational capacity of chips, ever larger and faster memories, the ability to execute artificial intelligence algorithms and the reduction of their production cost, a radical change has been observed in the way we live and work. This improvement has brought challenges never faced before for companies, which have had the need to produce increasingly complex chips, while maintaining high quality standards, trying not to fall into excessive delays.
If, on the one hand, we have excellent designers and increasingly advanced CAD tools, can we really be sure that what has been developed is qualitatively ready to be placed on the market?
For this reason, hand-in-hand with the figure of the designer, a second one has begun to get wider and wider: the *design verification engineer*.
In this thesis, the verification of a design and the underlying reasons were presented, applying them to a concrete example of a standard protocol for on-chip busses, developed by ARM: **Advanced High-Performance Bus (AHB)**. Firstly, its latest revision, the fifth one, has been explained in detail, also trying to make comparisons with other on-chip protocols.
Subsequently, it was worked on a *verification component* (VC). It can verify that an AHB design is actually consistent with the specifications. The process used is known as **functional verification**, widely adopted in the semiconductor industry, with the creation of a *verification plan* and the extension of a partially already implemented *verification environment* based on a standard methodology, known as **Universal Verification Methodology (UVM)**. The latter was presented and *SystemVerilog* was adopted for its implementation, as suggested by the standard itself.
Afterwards, some of the attributes of the verification plan were coded and *stimuli* to stimulate the design were created, drawing some conclusions. The concrete intent is to connect it later to the design of a real chip and verify that the design of AHB matches the specifications.
The final part of the thesis is aimed at the study of the behaviour *on* the bus, through the creation and **analysis** of a **log** and within some use cases. It was started by having a single master with a single slave, and then it was extended the analysis to a generic multi-master and multi-slaves architecture. Thanks to the logs, it was possible to make observations on some features such as: on which masters is given the priority, which slaves were accessed more frequently and how the wait states of the slaves can affect within a set of transfers.
Keywords: AMBA-AHB, UVM, verification IP, log analysis

Contents

List of Figures

CHAPTER 1

Introduction

Nowadays, integrated circuits (ICs) are essential components in daily electronics products. They power, pilot, control almost everything around us - from toys to wrist watches, from smartphones to cars and home automation products. Thanks to technological improvements such as the increasing computational capacity of chips, larger and faster memories, the ability to execute artificial intelligence algorithms and the reduction of their production cost, a radical change has been observed in the way we live and work. This improvement has brought challenges never faced before for companies, which have had the need to produce increasingly complex chips, while maintaining high quality standards, trying not to fall into excessive delays. Confirming this diffusion, Alsop[34] has estimated a 2020 revenue from the sale of integrated systems of 361.23 billion U.S. dollars and he predicted it to increase in the years to come. Of the same opinion, Foster estimated an ever-increasing complexity in the production of integrated circuits in the last seven years, highlighting two major trends[35]. On the one hand, there is a growing production of designs with *less than 1M gates* (logic gates, datapaths, excluding memories): this is due to smaller sensor chips for IoT and automotive devices. On the other hand, however, the production of large designs is always present: 36% of projects working *around 80K gates* and 31% working *between 80K and 1M gates*. Although it is only one dimension that expresses the complexity of design, it is possible to understand how these are becoming challenges and how there is a need to identify their quality.

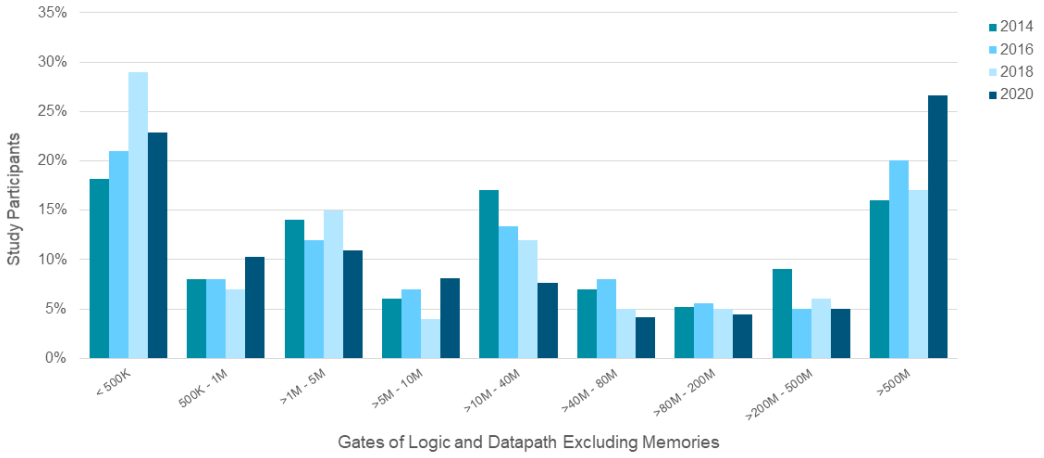


Figure 1.1: IC/ASIC Design size by gate count [35]

The reason behind is intuitively known that the sooner a bug is found, the cheaper it is to fix it. In "*Silicon Debug*", Gottlieb tried to summarize his findings giving a quantification of it[36]:

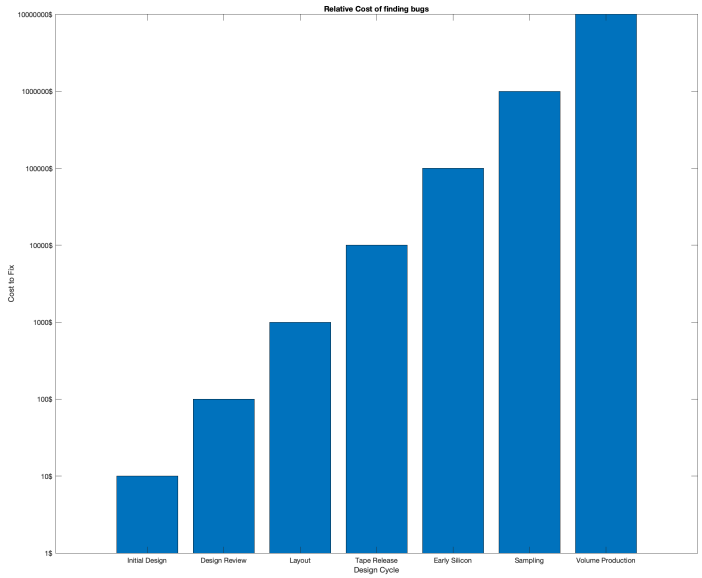


Figure 1.2: Relative costs of finding bugs [36]

A bug detected in the initial design phase is 10x times cheaper than a bug found during the tape out phase and hundreds or thousands of times if detected by the consumer. To ensure that an IC actually works and is reliable, there is a need to have verification procedures parallel to the product design path. The figure 1.3 adequately shows what is the mean peak number of engineers in the production and verification of an IC/ASIC. It is observable that there is a greater need for verifiers and that over the years there has been a growing need for both figures.

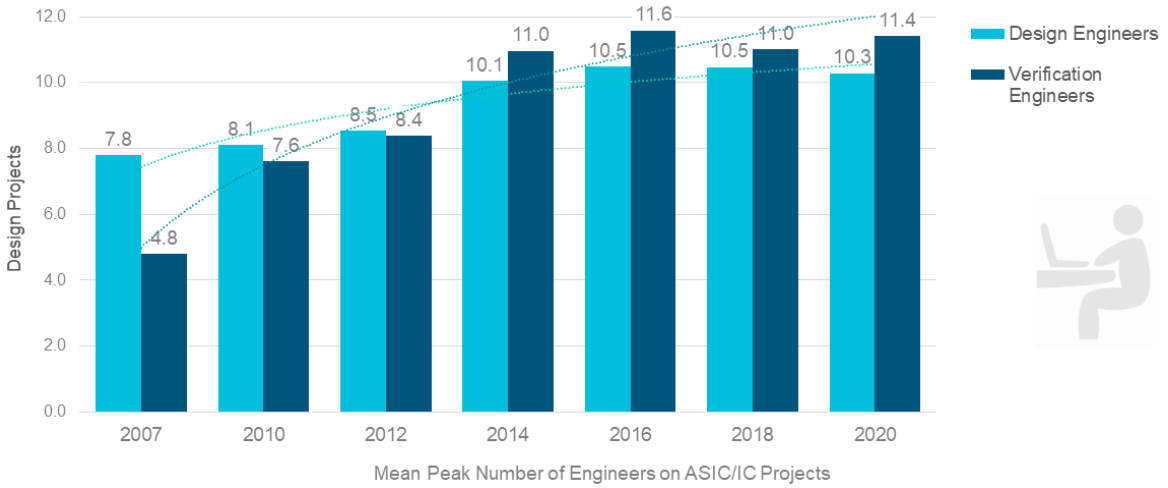


Figure 1.3: Mean peak number of engineers on a project [35]

In the figure 1.4 it is also possible to observe that not all projects need the same verification time. This is usually due to the ability of designers to use proprietary components - **intellectual property** (IP) - already verified and to integrate them into their new products, which therefore do not require significant time to verify, as it can happen with a brand new design.

Figure 1.4: Percentage of project time spent in verification [35]

In this thesis, the verification of a design and the underlying reasons are presented, applying them to a concrete example of a standard protocol for on-chip busses, developed by ARM: **Advanced High-performance Bus (AHB)**. It is a freely-available, open standard for interconnection and management of IP cores in a System-on-Chip (SoC). It allows the development of multi-processor chip designs in a modular, reusable, and scalable manner, which helps in avoiding costly re-designs and reduces time-to-market integrated designs[2]. Firstly, its latest revision, the fifth one, has been explained in detail, also trying to make comparisons with other on-chip protocols.
Subsequently, it was worked on a *verification component* (VC). It can verify that an AHB design is actually consistent with the specifications. The process used is known as **functional verification**, widely adopted in the semiconductor industry, with the creation of a *verification plan* and the extension of a partially already implemented *verification environment* based on a standard methodology, known as **Universal Verification Methodology (UVM)**. The latter was presented and *SystemVerilog* was adopted for its implementation, as suggested by the standard itself.
Afterwards, some of the attributes of the verification plan were coded and *stimuli* to stimulate the design were created, drawing some conclusions. The concrete intent is to connect it later to the design of a real chip and verify that the design of AHB matches the specifications.
The final part of the thesis is aimed at the study of the behaviour *on* the bus, through the creation and **analysis** of a **log** and within some use cases. It was started by having a single master with a single slave, and then it was extended the analysis to a generic multi-master and multi-slaves architecture. Thanks to the logs, it was possible to make observations on some features such as: on which masters is given the priority, which slaves were accessed more frequently and how the wait states of the slaves can affect within a set of transfers.

CHAPTER 2

Background

2.1 Advanced High-performance Bus (AHB)

2.1.1 AMBA

AMBA, Advanced Microcontroller Bus Architecture, is one of the most used type of communication bus in processor architectures. Developed by ARM Ltd in 1996, it was initially intended to be used in microcontrollers to effectively support communication between ARM processor cores. Nowadays it is widely used in both ASIC and system-on-chip designs, allowing the connection of a large number of peripherals (functional blocks): CPUs, GPUs, memory controllers, peripherals and many others. Over the years, five versions of the protocol have been released and it represents an open *"de facto" standard* to allow on-chip communications for embedded systems architectures. Its diffusion in the industry is due to its flexibility, scalability, dependability and compatibility between IP components from different vendors and designers, proving to be able to effectively connect both peripherals with the need for high performances and ones with lesser demands[1][2].
The standard includes multiple bus types: **Advanced System Bus** (ASB), **Advanced eXtensible Interface** (AXI), **Advanced High-performance Bus** (AHB) and **Advanced Peripheral Bus** (APB), as well as Lite versions of AHB and AXI. It is possible to visualize the bus in a hierarchical way, dividing it into two parts: a system bus (AHB) and a peripheral bus (APB), linked by a bridge that handles data and operations between the two domains.

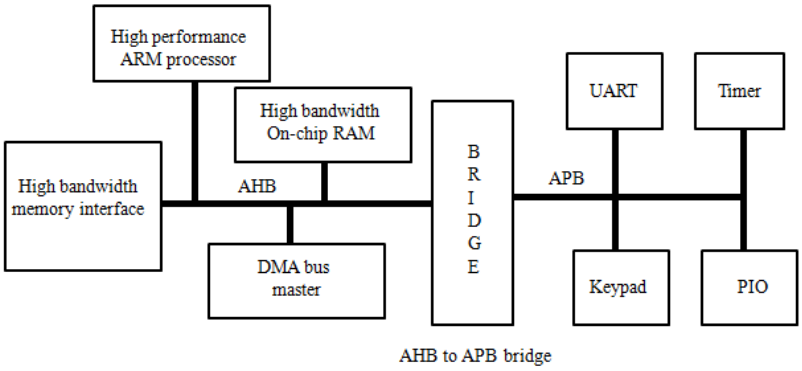


Figure 2.1: A typical use of AMBA in a system on chip [3]

One of the principles on which AMBA is based is to be **technology-independent**. It is a protocol described only at the "clock cycle" level, without detailing the electrical characteristics, which are therefore left to the discretion of the manufacturers.
The standard presented in this work is the last released in October 2015, AMBA version 5, which defines as buses/interfaces:

- AXI5, AXI5-Lite and ACE5 Protocols.
- Advances High-performance Bus (AHB5, AHB-Lite).
- Coherent Hub Interface (CHI).
- Distributed Translation Interface (DTI).
- Generic Flash Bus (GFB).

Previous standards define past versions of AXI, ACE and AHB, as well as ASB and APB.
AHB is a protocol introduced by the second version of the standard and it is dedicated to high performance transfers, to connect internal and external memories and high performance peripherals. It describes bus transactions as composed of an *addressing phase*, followed by a *data phase*. Usually, the first phase lasts one clock cycle, while the second phase can last one or more cycles (in case of *wait states*, a particular scenario described later on this chapter). In order to improve performances, these phases can be *pipelined*. Furthermore, it is possible to use *multiple masters* controlling the accesses to the target device with a multiplexer (non-tri-state), to have only one master at a time enabled to access the bus.
AHB-Lite represents a subset of AHB, defined by the third version of the standard, where the bus design is simplified for single masters.
APB is designed for designs with low requirements in terms of bandwidth transfers, performances, power, costs and low complexity, for example, to access I/O peripherals with non-pipelined transfers. The addressing phase is similar to the one defined for AHB, but the set of signals is reduced (e.g. it is not possible to perform burst).
AXI defines interfaces and it is addressed to high performance architectures, high clock rates and usually it is applied in high-speed sub-micrometer interconnections. AXI is not a standard bus, but it defines a protocol and interfaces, leaving the designers the possibility to define an implementation. ACE is an extension of AXI and introduces system wide coherency, allowing multiple processors to share memory and to enable technologies such as ARM big.LITTLE processing. ACE-Lite is a subset of ACE and it only allows one-way IO coherency[4].

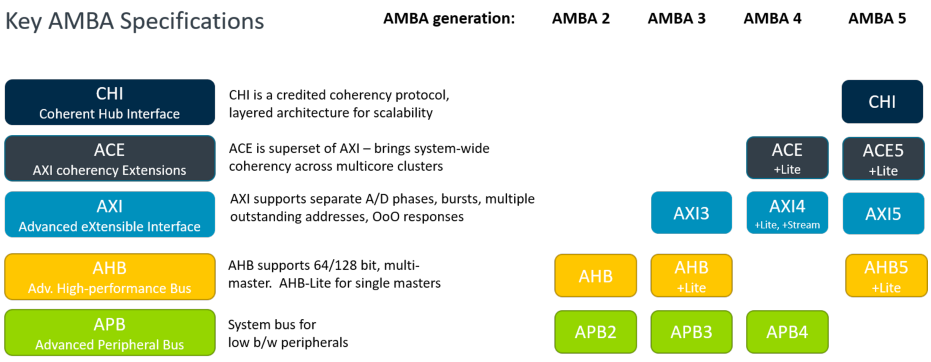


Figure 2.2: Overview diagram of AMBA [1]

2.1.2 AHB

AHB, Advanced High-performance Bus, defines the interfaces for master, slave and interconnections between them, allowing high performance and clock frequencies, thanks to the following characteristics:

- Single clock edge operations.
- Burst transfers.
- Non-tristate implementation.
- Broad data bus configurations such as 64, 128, 256, 512 and 1024 bits.

A transfer starts from the master, sending control signals and the address. The control signals include the direction, i.e. whether the operation is to write the data (from the master to the slave) or to read the data (from the slave to the master), the width of the transfer and whether the transfer is single or burst (incremental or wrapping to a particular address).
Transfers consist of two phases:

- **Address phase**: It lasts a clock cycle and the address and control signals are sent out.
- **Data phase**: It lasts at least one clock cycle (or more). A slave can request its extension using the HREADY signal, which when low, indicates the presence of wait states.

Master and Slave Interfaces

This paragraph introduces the interfaces of the masters and slaves and it explains the related signals.

Master Interface

The master provides information to perform read and write operations on the slaves. As detailed in the following when presenting the arbitration, each slave is selected by a selection signal (HSEL). In this way, the slave is able to understand when it is actually stimulated. It also receives two response signals HRESP and HREADY from the slaves, to know if a transfer has been performed successfully or if it has failed (HRESP) and if it is required an extension of the data phase (HREADY)[5].

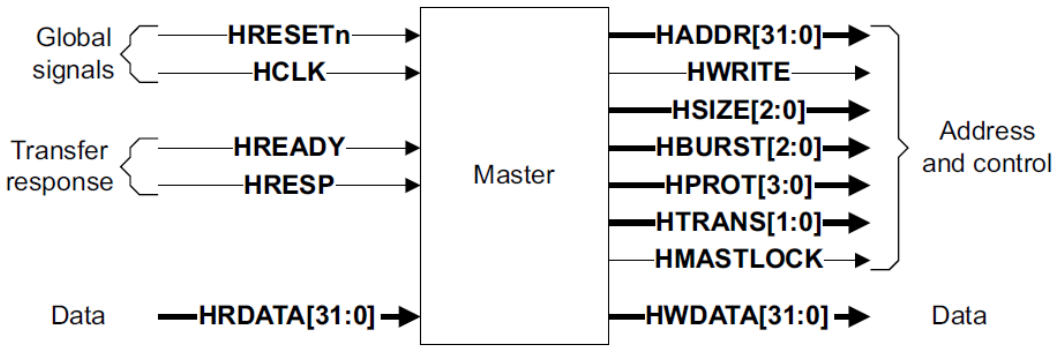


Figure 2.3: Master interface signals [5]

All the signals are presented below, most of which are connected to the slaves, and their meaning is briefly described:

- **HCLK**: Clock source. All transfers refer to the rising edge of this signal.
- **HRESETn**: Reset controller. It resets the bus. Active low.

- **HADDR[31:0]**: 32-bit system bus address. It is also connected to the decoder.
- **HBURST[2:0]**: It indicates the type of burst transfer. More details on the values it can assume are provided in the burst section.
- **HMASTLOCK**: When it is high it means that the transfer is part of a locked sequence. It has the same timing of the address signal.
- **HPROT[6:0]**: They provide additional information on the bus access and they indicate how the access should be managed within a system. The three most significant bits add "extended memory types", while the remaining least significant bits indicate whether a transfer is privileged or user mode and opcode fetch or data access. Its extension is new to this version, while the initial standard provided only 4 bits.
- **HSIZE[2:0]**: It indicates the size of the transfer (byte, halfword, word and extensions up to 1024 bits).
- **HNONSEC**: It indicates whether the transfer is secure or not. This signal also goes to the decoder. Added in this version of the standard.
- **HEXCL**: It indicates if the transfer is exclusive. The destination is the Exclusive Access Monitor. Added in this version of the standard.
- **HMASTER[3:0]**: Master identifier, unique for each master.
- **HTRANS[1:0]**: It indicates the type of transfer. It can be: IDLE, BUSY, NON SEQUENTIAL and SEQUENTIAL.
- **HWDATA[31:0]**: Data bus for writing. This bus contains what will be written to the slave. The size can be extended.
- **HWRITE**: It indicates the direction of the transfer. If the signal is high, a write operation is indicated, otherwise a read one. It has the same timing as the address signal, but it is kept constant during a burst.

Slave Interface

The slave receives signals from the master and produces the following responses which are mainly directed to the multiplexer[5]:

- **HRDATA[31:0]**: Read bus. During a read operation, the selected bus produces the data to the multiplexer which then provides it to the master. The bus can be extended to more than 32 bits.
- **HREADYOUT**: It indicates the end of a transfer (high) or the extension of a transfer (low).
- **HRESP**: It indicates the status of the transfer. If it is low, the signal assumes the meaning of OKAY, otherwise ERROR.
- **HEXOKAY**: It indicates the failure or success of an exclusive transfer. Added in this version of the standard.

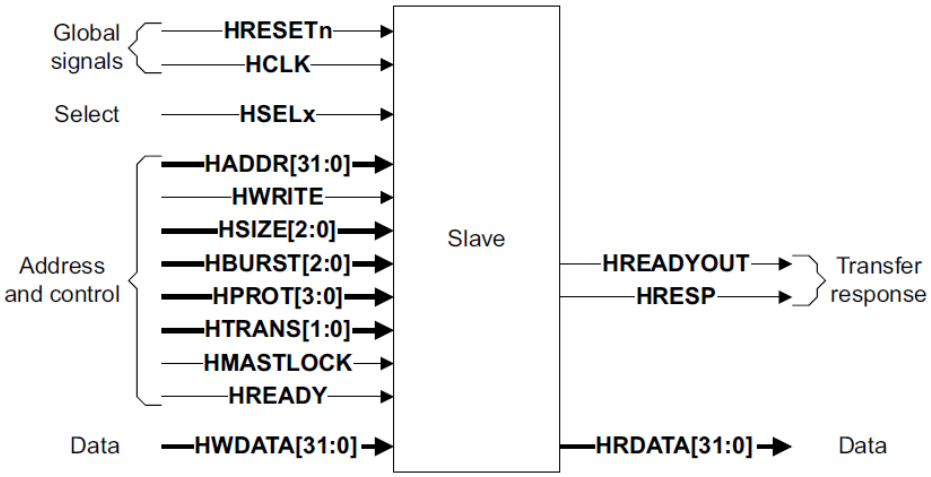


Figure 2.4: Slave interface signals [5]

Interconnections

The connection between master and slave is done by means of a decoder and a multiplexer, in simple case of a single master. Data and addresses are sent out to all slaves, but only the slave corresponding to the appropriate address will be selected (Fig. 2.5).

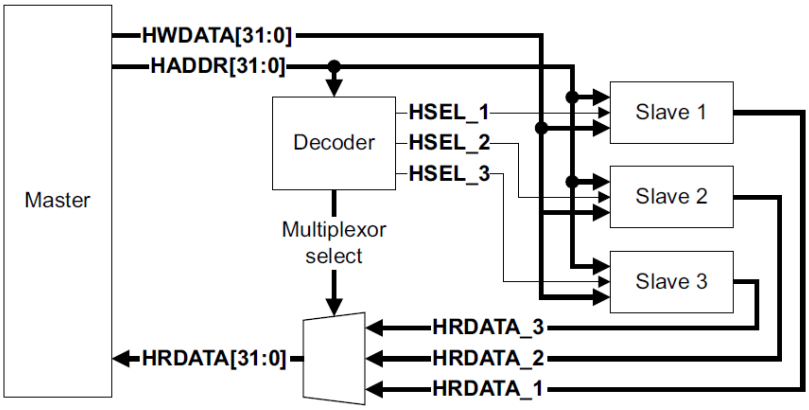


Figure 2.5: Interconnection for single master [5]

For the sake of completeness, hereon the signals of the decoder and multiplexer are specified:

- **Decoder signals**:
  - **HSELx**: Each slave has its own HSEL and in a combinatorial way, it takes an address and it produces a signal which enables the target slave.
- **Multiplexer signals**: The signals are directed to the master(s).
  - **HRDATA [31:0]**: Read data bus, selected by the decoder.
  - **HREADY**: It indicates to the master and to all the slaves that the previous transfer has ended.
  - **HRESP**: Response of the transfer.
  - **HEXOKAY**: Exclusive okay.

In the case of several masters, the structure becomes more complicated and it is required a system that guarantees arbitration, as well as routing of address, control and write signals, from the numerous masters to the numerous slaves (Fig. 2.6).

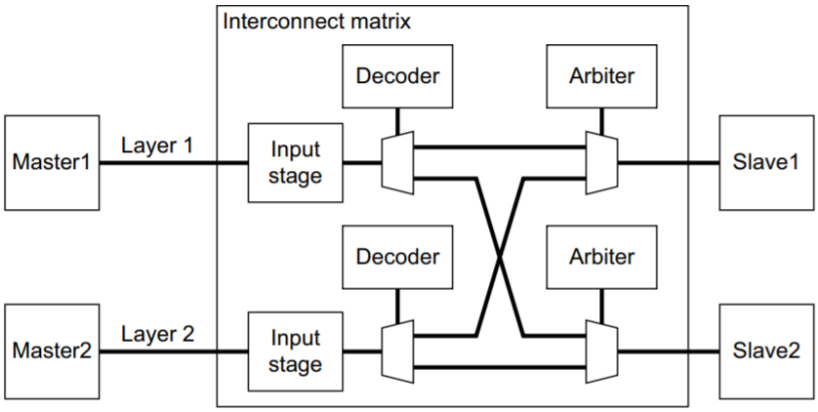


Figure 2.6: Interconnection for multi-masters [6]

The protocol also provides the possibility of having multiple slave select. An example is when a peripheral has the datapath and the control registers at different locations of the address space. Regarding the addressing space, the protocol defines a **default slave**. If not all memory addresses are mapped, then an additional dummy slave is provided to respond to non-existent addresses. This slave generates an error response for NONSEQ and SEQ transfers and ok with zero wait states for IDLE and BUSY transfers.

Transfers

Simple transfers

As previously introduced, a transfer consists of two phases: an *address phase*, which lasts a single HCLK cycle, and a *data phase*, which may require several clock cycles. According to the value of HWRITE, it is possible to get the direction of the transfer: if HWRITE is high, then it is indicated a write transfer on the HWDATA bus, while if it is low, a read operation from the HRDATA bus. The simplest transfers are read and write without wait states, so both the addressing phase and the data phase are performed in a single clock cycle each.

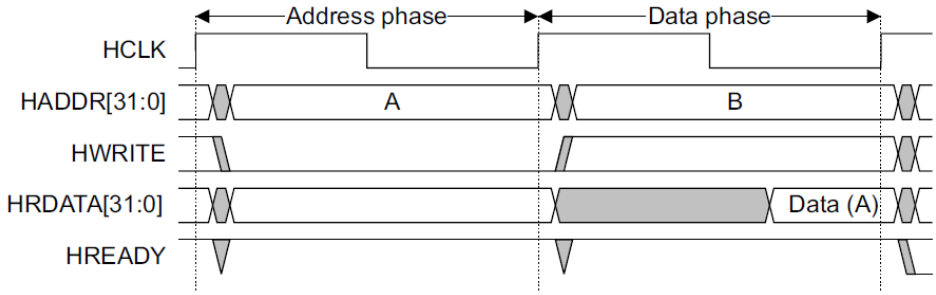


Figure 2.7: Simple read transfer [5]

A master guides the address and control signals after the HCLK rising edge. The slave at the next rising edge of the clock samples these signals. The slave can drive low HREADYOUT, in case it needs longer to be able to sample the data. All HREADYOUTs are combined by the interconnection that

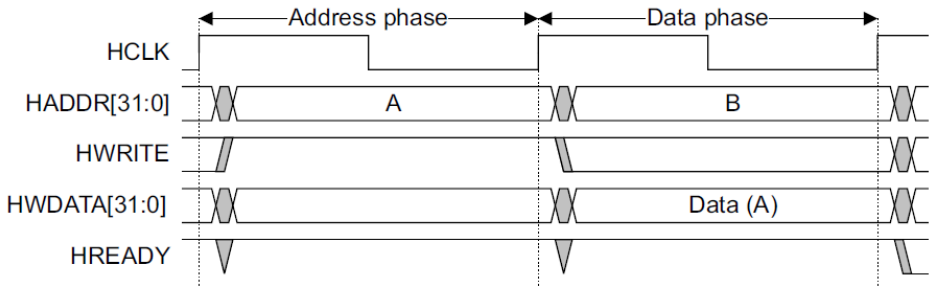


Figure 2.8: Simple write transfer [5]

generates a single HREADY for the master, which would understand that a *wait state* has occurred. This is shown in figure 2.9 and 2.10. It is also possible to observe that the transfers are *pipelined*: while a data phase of a transfer is occurring, the addressing phase of the next transfer takes place, which leads to higher bus performances.

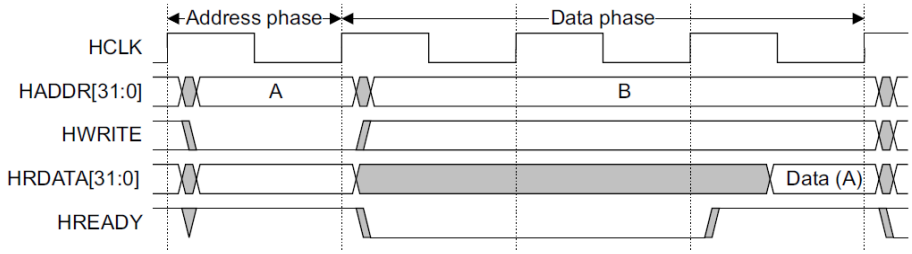


Figure 2.9: Read transfer with two wait states [5]

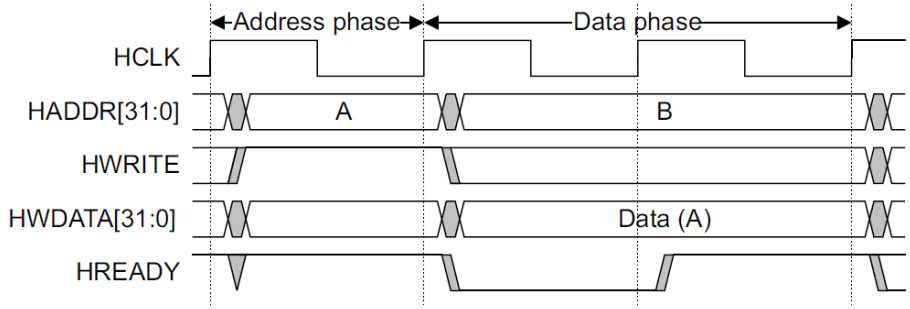


Figure 2.10: Write transfer with one wait state [5]

Advanced transfers

Using HTRANS and HSIZE signals, it is possible to define other more advanced transfers.
HTRANS signal can assume the following four values:

- **IDLE (00_2)**: It indicates that there is no need for any data transfer. The slave responds with a zero wait state OKAY.
- **BUSY (01_2)**: It allows the insertion of idle cycles when a burst is performed. This means that the transfer cannot take place immediately but that it is required to wait.
- **NONSEQ (10_2)**: It indicates the first transfer of a burst or a single transfer.

- **SEQ (11_2)**: The subsequent transfers of a burst are considered as sequential and the address and control signals are related to the previous transfer.

Locked transfers

A locked transfer is usually used to maintain the integrity of a semaphore. In this way, the target slave does not perform other operations from other masters. A pratical example could be a swap instruction (SWP), which carries out a read from memory followed by a write to memory[5]. To signal this intention, the master uses the HMASTLOCK signal. Nevertheless, not being strictly required by the protocol, typically not all the slaves in the system accept the HMASTLOCK signal. The protocol suggests inserting an IDLE transfer after a locked transfer and it forces these types of transfers to be addressed to the same slave address region.

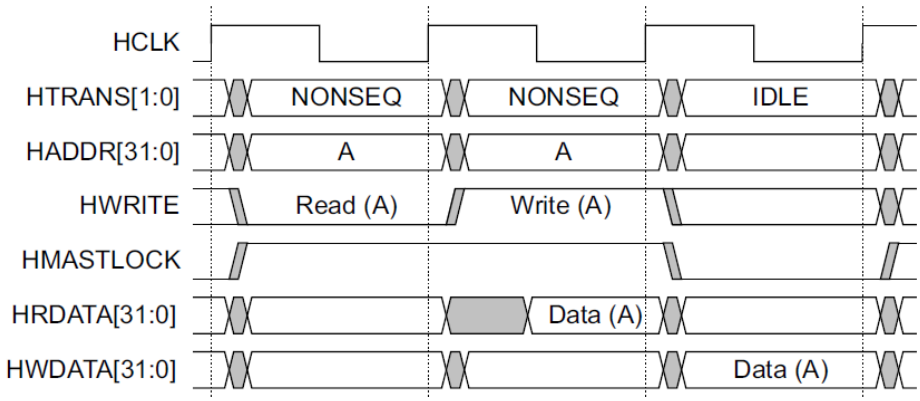


Figure 2.11: Locked transfer [5]

Burst transfers

A burst transfer consists of transmitting data repeatedly, without going through all the steps necessary in case of a single transfers. Bursts can be of two types: *incremental* and *wrapping*.
A burst is *incremental* if it accesses the memory locations sequentially, incrementing the value of the previous address by an offset. It is *wrapped*, instead, if the value of the previous address is incremented until a boundary value is reached (calculated as the number of beats by the transfer size). Afterwards, the pointed address is going to assume again the initial value. Two examples are shown in figures 2.12 and 2.13. The types of bursts, expressed with the HBURST signal, can be SINGLE, INCR (undefined length), WRAP4-8-16, INCR4-8-16.

Waited transfers

Waited transfers happen when the slave needs more time to sample data. The slave uses HREADY-OUT to notify the master and to obtain some clock cycles more. During this type of transfers, the master is limited in changing control and address signals. It can only change the *transfer type* and *address* in three cases:

- **IDLE Transfers**: The master can change HTRANS from IDLE to NONSEQ and the address can be changed until HTRANS becomes NONSEQ. Then they must remain constant until HREADY becomes high.
- **BUSY Transfers with fixed and undefined length burst**: The master can change HTRANS from BUSY to SEQ (for fixed burst) or to any other transfer types (for undefined length burst), but then it must remain constant until HREADY becomes high.

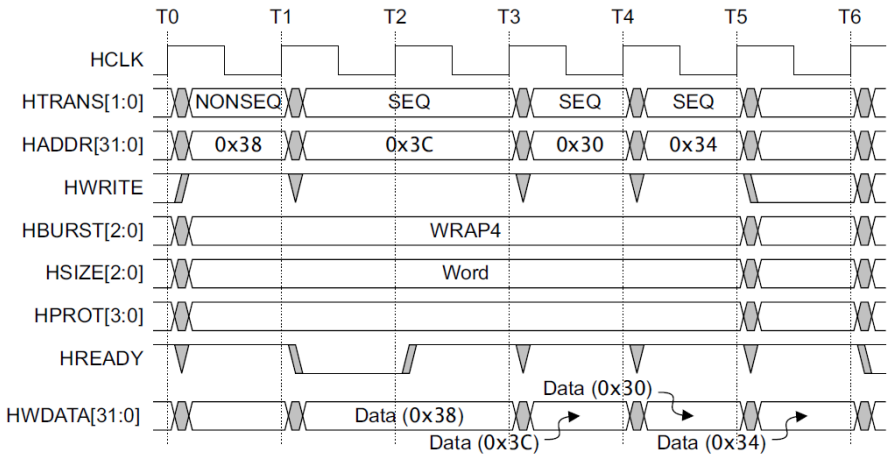


Figure 2.12: Four-beats wrapping burst (WRAP4) [5]

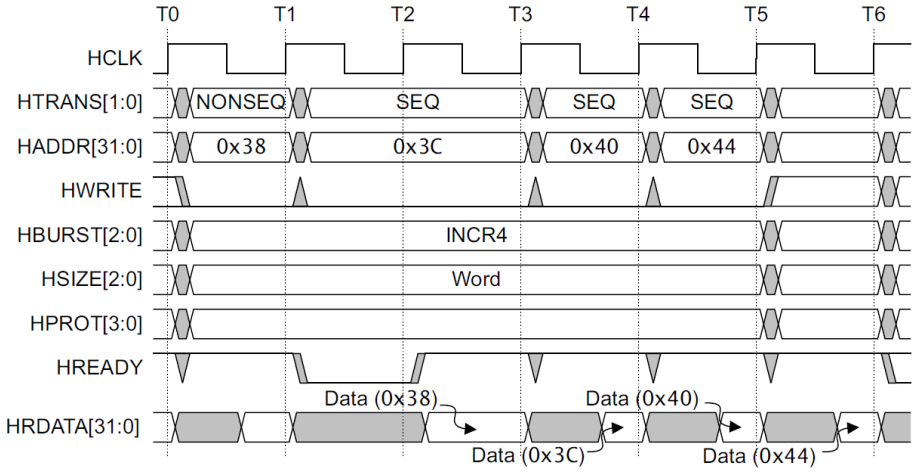


Figure 2.13: Four-beats incrementing burst (INCR4) [5]

- **After an ERROR response**: In case the slave responds with an error, the master can change the address while HREADY is low.

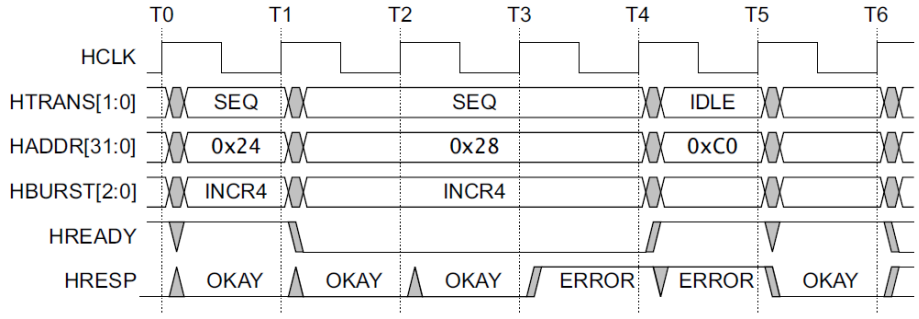


Figure 2.14: Change of address after an error response from the slave [5]

Slave responses

After a master initiates a transfer, it monitors the progress of it and waits for a response from the target slave. The answer is a composition of two signals: HRESP and HREADYOUT.
The transfer can be in three conditions:

1. **Transfer done**: The transfer was immediately completed with success. HREADYOUT is *1* and HRESP is *OKAY*.
2. **Transfer error**: The transfer was not completed, reporting an error. HRESP is *ERROR*. Two cycles are required to recover from an error and the second one contains HREADYOUT equal to *1*.
3. **Transfer pending**: The transfer needs one or more clock cycles (wait states) to be completed. HREADYOUT is *0* and HRESP is set to *OKAY* at the end of the transfer. The number of recommended wait states is a maximum of 16, to avoid degrading performances.

HRESP/HREADYOUT	0	1
0 - OKAY	Transfer pending	Successful transfer completed
1 - ERROR	ERROR response, first cycle	ERROR response, second cycle

Table 2.1: All the possible slave transfer responses.

Enhancements of the latest version of the protocol

The AMBA AHB protocol has evolved defining an interface protocol most widely used with Cortex-M processors, for embedded designs and other low latency SoCs [1][7]. The latest version is built on the previous generation of AHB-Lite and AXI and it has added some features such as:

- **Secure/Non-secure** signaling in address phase to indicate secure or non-secure transactions. An interface supports secure transfers if the additional HNONSEC signal is present. This signal is asserted for non-secure transfers and the opposite for secure transfers. It has the same constraints that apply to addresses.
- **Multi-copy atomicity** enabling scaling to multiple cores.
- **Extended memory types** to support more complex systems.
- **Exclusive transfers** that support semaphore-type operations. For instance, a master can:
  1. Perform an exclusive read from an address.
  2. Calculate a new value using the received data. In the meantime, there may also be non-exclusive transfers.
  3. Carry out an exclusive write to the same address with the new calculated data.

  Writing can be successful or not. In particular, if another master has written to that location, the transfer fails and the memory location is not updated. In cases like this one, it is repeated the operation from the beginning, performing a new read. To understand weather a write is successful or not, an Exclusive Access Monitor (EAM) is required. To perform this, three signals have been added, HEXCL. HMASTER[m:0], HEXOKAY.
  - **HEXCL**: This signal indicates that the transfer is part of an exclusive access.
  - **HMASTER[m:0]**: This signal indicates the unique identifier of the master. A master can have several exclusive threads.
  - **HEXOKAY**: This signal indicates the success or failure of an exclusive transfer.

2.1.3 Alternative on-chip bus standards

The purpose of a system bus is to allow communication of software, which runs on a processor, with other SoC hardware. Each component of an SoC (IP) has an interface to the outside world which consists of pins, that are used to send and receive control signals, addresses and data. In order to allow communication between the different components, it is necessary to find a common standard, which sometimes involves the creation of "logical wrappers", adaptations of the IP interfaces. Through the definition and use of a standard bus architecture it is possible to:

- Define the interface between IPs and bus architecture.
- Define some specifications of the bus architecture that implements the data transfer protocol.

Ideally, designers would like to have a single standard to connect all IPs, but in reality there are many different implementations and protocols with their advantages and disadvantages. The idea of this section is to present some of the major alternatives offered by the embedded systems market and to provide some comparisons with AMBA.

IBM CoreConnect

CoreConnect is an on-chip bus designed by IBM that allows the interconnection of chip cores.
The CoreConnect bus architecture includes the **Processor Local Bus** (PLB), the **On-chip Peripheral Bus** (OPB), a bus bridge, two arbitrators, and a **Device Control Register bus** (DCR).
IBM makes the CoreConnect bus available as a royalty-free architecture to leading tool vendors, IP and chip development companies, such as Cadence, Nokia, Ericsson, Synopsys and Siemens[9][10][11][12][13].

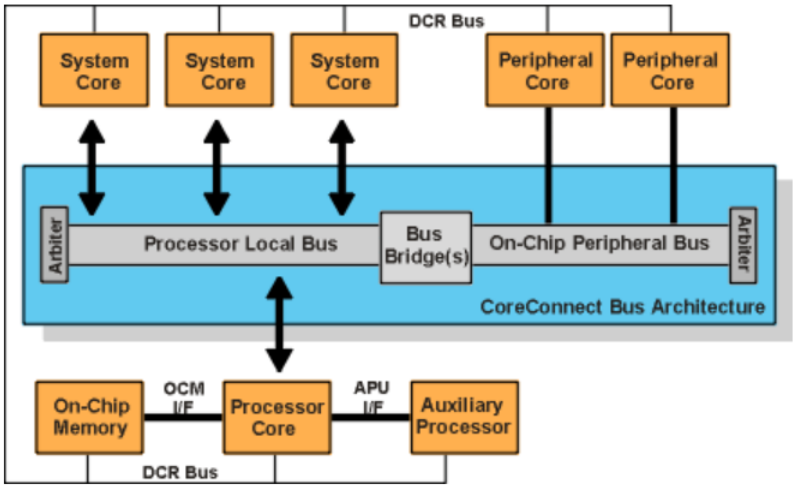


Figure 2.15: CoreConnect bus architecture [12]

The **PLB** (Processor Local Bus) on-chip bus is used in highly integrated systems. It supports read and write data transfers between master and slave devices equipped with a PLB interface and connected through PLB signals.
It is fully synchronous and it supports up to eight masters. Like in AHB, it provides burst transfers with variable and fixed length and it is pipelined. There are two busses, one for reading data and one for writing and the address and data phases are decoupled.
The address phase takes three cycles: Request, Transfer and Address-acknowledge, while the data phase takes two cycles: Transfer and Data-acknowledge[13].

Lower-performance peripherals are attached on the **OPB (On-chip Peripheral Bus)**. It is fully synchronous and it is able to arbitrate up to 4 OPB master peripherals. A bridge is provided between the PLB and OPB to enable data transfer by PLB masters to and from OPB slaves. The **DCR (Device Control Register)** bus is used primarily for accessing status and control registers within the various PLB and OPB masters and slaves. It is meant to off-load the PLB from the lower-performance status and control read and write transfers[9][14].
Some of the major features are compared in the table 2.2.

	IBM CoreConnect PLB	**ARM AHB 5.0**
Bus Architecture	32-, 64- and 128 bits (up to 256 bits)	32-, 64- and 128 bits (up to 1024 bits)
Data Buses	Separate Read and Write	Separate Read and Write
Key Capabilities	Multiple Bus Masters 4 Deep Read/2 Deep Write Pipelining Split Transactions Burst Transfers Line Transfers	Multiple Bus Masters Pipelining Split Transactions Burst Transfers Line Transfers
	IBM CoreConnect OPB	**ARM APB**
Master Supported	Supports Multiple Masters	1 Master: APB Bridge
Bridge Function	Master on PLB or OPB	APB Master Only
Data Buses	Separate Read and Write	Separate or 3-state

Table 2.2: A comparison between CoreConnect and AMBA[14]

STMicroelectronics STBus

The STBus, defined by STMicroelectronics, is a set of protocols, interfaces and architectures specifying an interconnection system, which aims to be versatile in terms of performance, architecture and implementation[15]. The components connected to the STBus are functionally similar to the master and slave, but they are defined as:

- **Initiator**: It generates traffic towards the interconnection to access other target resources (memories, peripherals, etc).
- **Target**: Any resource that is accessed by the initiators.

In order to define the protocol, some concepts are defined:

- **Transaction**: An exchange of information between initiator and target.
- **Cell**: Basic amount of information that can be transferred in a clock cycle. It depends on the data bus size.
- **Packet**: A collection of cells, building a not interruptible transfer. It depends on the operation and data bus sizes.

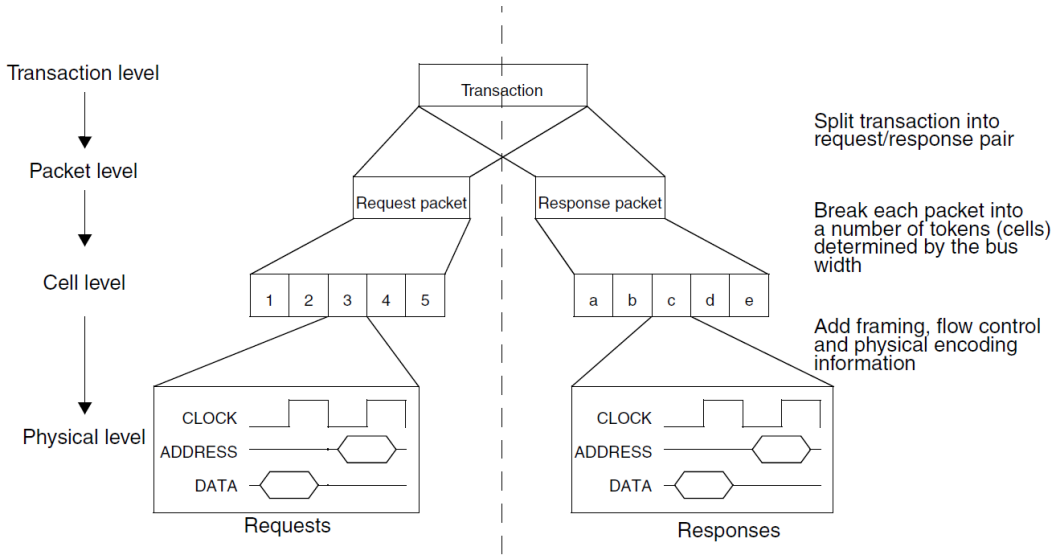


Figure 2.16: STBus protocol layers [15]

Three different types of the STBus protocol are defined[15][16]:

- **Type 1: Peripheral**: This is the simplest interface and supports a subset of the full transaction set. It is intended to be used for peripherals registers access and no pipeline is applied. Each request packet has a corresponding response packet of the same length and in the same order. Operation requests are repeated in each cell.
- **Type 2: Basic**: This adds support for split transactions to the peripheral interface and pipeline features. To simplify the module design, all request/response packets are symmetrical and the ordering of all operation transactions is enforced.
- **Type 3: Advanced**: This extends the basic to a full split transaction packet implementation with asymmetric request/response packets and full error support. It also allows the system to relax request/response ordering properties to allow modules to take advantage of system concurrency (out-of-order execution). In order to increase bus efficiency, initiators may request an operation only once, at the start of a packet, and then send several other cells of data in the same packet, receiving back a response packet of a different length.

The topology of an STBus interconnect is very flexible and it can range from a simple shared bus, like AMBA AHB, to a full crossbar. A performance enhancer typical of this protocol is the overlapping of transfers, possible thanks to the presence of two data channels. One channel is devoted to transfers from masters, as previously said called *initiators* (e.g., processors), to slaves, called *targets* (e.g., memories, peripherals and other dedicated hardwares) and the second channel is used in the opposite direction. This allows an initiator to start a request while a target is transmitting a response. A practical example which points out the difference with AHB is related to the bursts. STBus is able to speed up transactions by requesting new burst transfers while previous ones are still finishing, therefore not having idle cycles in between the bursts. STBus features fast arbitration, and this makes possible to complete single read transfers in just two cycles - one cycle for arbitration/sending addresses and one for receiving data[17]. Furthermore, when inserting a wait state, the minimum latency becomes of three cycles[17].

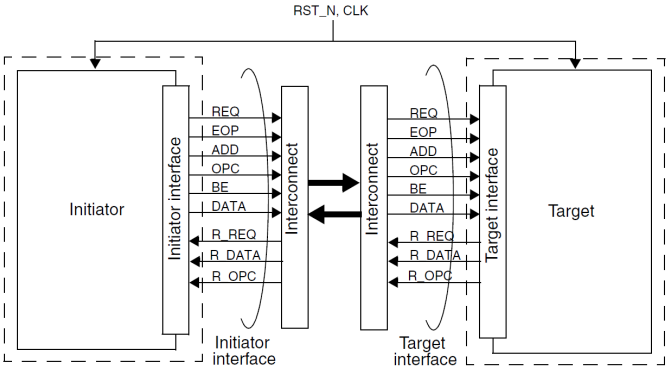


Figure 2.17: STBus Type 1 interfaces [16]

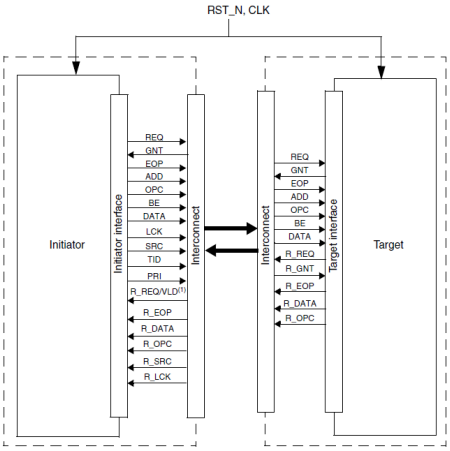


Figure 2.18: STBus Type 2 interfaces [16]

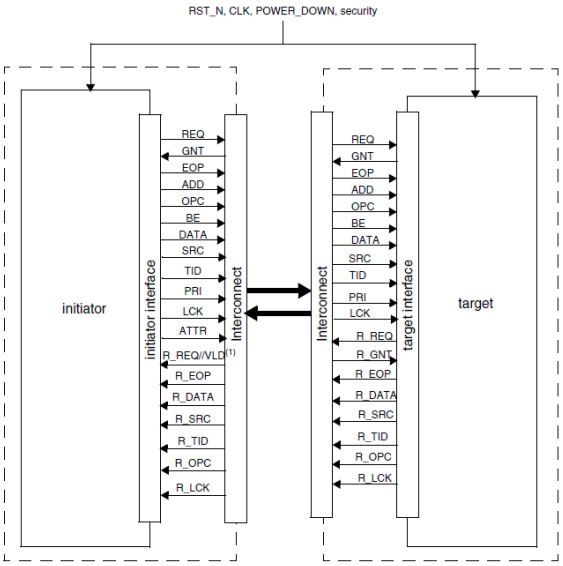


Figure 2.19: STBus Type 3 interfaces [16]

Silicore Corp Wishbone

This section presents an open-source bus architecture, called **Wishbone**, developed by Silicore Corp and maintained by OpenCores, to allow on-chip communication of various integrated components within an SoC. Like the solutions already specified above, the description is purely logical, in terms of signals, clock cycles and logic levels, while electrical information and the bus topology are left to the designers, who can combine hardware designs written in the languages of their liking as VHDL or Verilog. Being in the public domain, the idea of the protocol is to encourage the free disclosure of designs within a community of engineers and enthusiasts. This means that each Wishbone specification is accompanied by a datasheet that specifies what it does, its use, bus-width, facilitating its reuse by others [19]. One of Wishbone's most common specifications is called Simple Bus Architecture (SBA).

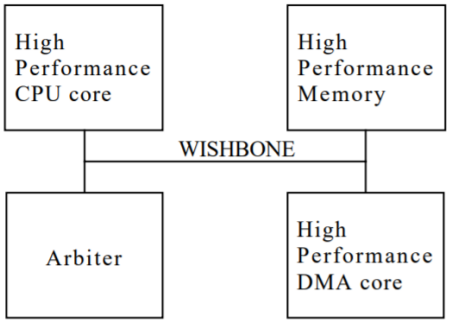


Figure 2.20: Wishbone simplified architectural view [18]

Wishbone has 8, 16, 32, and 64-bit data bus widths by definition. All signals are synchronous and only some slave responses are combinatorial, in order to improve performances. In addition, the protocol allows the possibility of defining an optional *bus tag*, which describes what is present in the bus. Only some signals cannot contain tags, such as reset, simple addressed reads and writes, movement of blocks of data, and indivisible bus cycles[19]. The protocol is based on two interfaces which, just like for AMBA, are defined as Master and Slave. Masters are able to initiate requests (bus cycles), while slaves accept requests (bus cycles). The peculiarity, however, lies in having a **single bus** for all applications: *a high speed bus*. This solution has the advantage of requiring a single interface for both high-speed and low-performance peripherals which, clearly, simplifies the design, than designing two different bus interfaces. The hardware implementations have compatibility with various types of interconnection topologies such as *crossbar switch*, *dataflow*, *point-to–point* and *shared bus* [19].

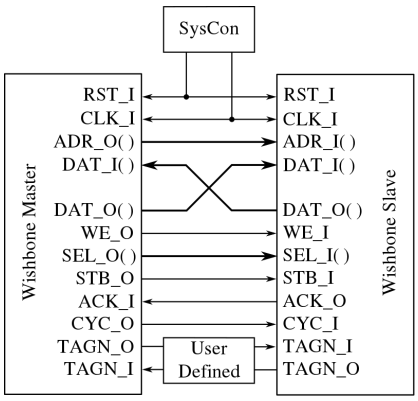


Figure 2.21: Point-to-point interconnection [19]

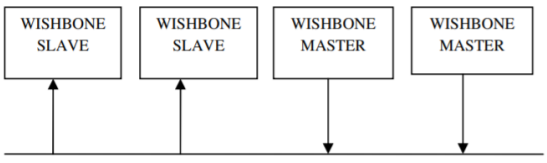


Figure 2.22: Shared bus interconnection [19]

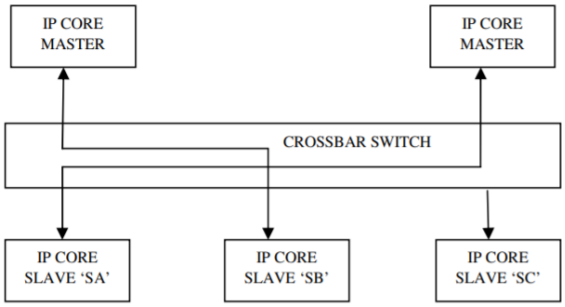


Figure 2.23: Crossbar switch interconnection [19]

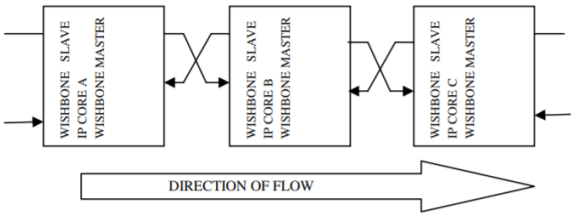


Figure 2.24: Data Flow interconnection [19]

A *point-to-point interconnection* consists of directly connecting master and slave. A *shared bus interconnection* consists of connecting all masters and all slaves on the same bus, but only one master at a time can use it. A *crossbar switch interconnection* is useful when several masters (or the same master) want to access different slaves, which is a typical case in integrated systems with multiple cores. A *data flow interconnection*, also called "pipelining", instead consists in processing data sequentially, where each IP core has both a master and a slave interface and each node in the middle of the chain receives from the previous IP core and, after having made some computations, it produces data for the next IP core. To conclude, a list of technical features of Wishbone is hereon provided[18][19]:

- One Bus Architecture for all applications.
- Simple and compact architecture.
- Multi-master support.
- 64-bit address space.
- 8-64 bit data bus (expandable).
- Single read and write, Read Write Memory and Event cycles.
- Supports memory mapped, FIFO and crossbar interface.
- Throttling of data for slower devices provided.
- User defined TAGs for identifying data transfer types.
- Arbitration defined by the end user.

2.2 Design verification

2.2.1 Context, principles and methodologies

The role of a design verification engineer

For the development of a complex hardware system, an equal number of *designers* and *verifiers* are required. Often the number of verification engineers is even double or, at project level, about 60-70% of the development cycle, requiring a large set of skills and competences from them.
A *verification engineer* must not only be able to understand the specifications but also the designs, following a different approach from that of the designers, to avoid making the same mistakes that the designers could potentially do. In particular, it can be thought that a designer tries to give shape to a design considering mainly the cases required by the *specifications*, while a verifier must check a design taking into consideration *all the possible cases*, even those which are not described (which ideally are infinite), trying not to extend the time to market[20].
A possible result would be having multiple tapeouts, before an actual presentation of the chip to customers. Being chip design still a creative process with unavoidable imperfections and as chip manufacturing is an onerous process for a company, design verification is here to stay.
A *designer* is responsible for shaping the *specifications*, which describe what a chip is supposed to do, creating an *implementation*, that is, detailing the functionality that the system must provide.
This is a *refinement process*: various steps are passed that lead to a concrete realization. They are functional specification, algorithmic description, register-transfer-level (RTL), gate netlist, transistor netlist and finally the layout. The skill of a designer lies in choosing the appropriate description, going down the levels: for example, from an RTL description, it is possible to have countless gate level implementations that describe the same RTL design.

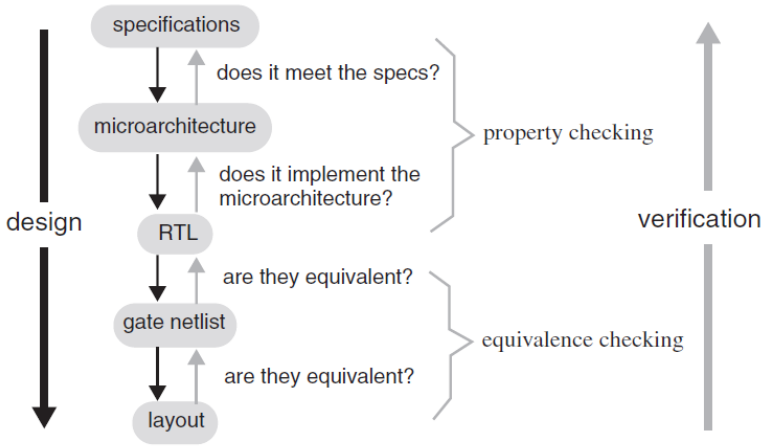


Figure 2.25: Relationship between design and design verification [20]

Design verification instead can be seen as an opposite direction process, meaning it starts with an implementation and verifies that the specifications are met. Considering the figure 2.25, it can be pointed out that a verification phase is required for each stage of the design phase. For example, it can be verified that an *algorithmic description* matches the *functional specification* or that a *layout* actually matches the *gate netlist* from which it derives. It is immediately clear how complex and varied the field of design verification is. This ranges from functional verification, to timing, layout, electrical verifications and so on.

Focusing on *functional verification*, it is possible to distinguish between:

- **Equivalence checking**: Verify that two versions of a design are equivalent in functionality. It is common practice to use it in two ways: for the same abstraction level and for different ones (such as RTL and gate level).
- **Implementation verification, model/property checking**: Verify that the implementation meets the specifications and that the model/property is respected. This usually happens when we are at two different levels of abstraction and details are introduced that are allowed, but are not specified in the upper abstraction levels (such as timing constraints).

Types of errors in a design

A verifier can look for two errors: errors introduced in the **design implementation** and errors related to **specifications**.
Regarding *implementation* errors, they are errors that are not present in the specifications but they are introduced during the implementation phase, for example by human mistakes or bugs in the software tools that are used. For human mistakes, it is meant that a designer may misunderstand the specifications. A solution could be to use software automation to produce an implementation, but it is not feasible because the specification is usually written in a loose language, i.e. English language. This last one is subject to interpretation and not as precise as a programming language or a mathematical model could be, therefore it is hard to have a direct translation. In particular, even if formal languages are used, it is not ensured that all the requirements, i.e. timing requirements, could be met while describing a system at such a high level.
Another solution commonly used is the **redundancy**, which means that the same specifications are implemented *twice* with different approaches and then compared.

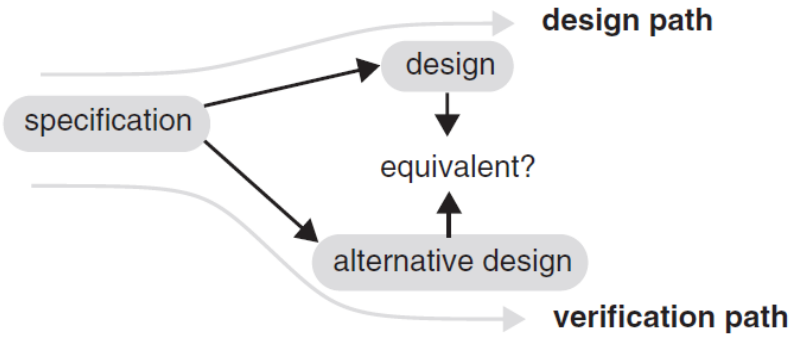


Figure 2.26: Verification by redundancy [20]

In particular, as shown in figure 2.26, an alternative design is created for verification which is then compared with the original one. To avoid that the design for verification is developed in the same way as the original one, risking to make the same mistakes, it is usually chosen a different approach. An example could be to use two different *descriptive languages*. VHDL or Verilog can be used for the description of the hardware by the designers and SystemVerilog, Vera, C/C++, for the verification counterpart. Usually to compare them, an intermediate form can be used to prove the equivalence: e.g. if we have a transistor-level circuit and an RTL implementation, we can resort to binary decision diagrams (BDDs).
The classic simulation-based verification paradigm is based on four components: circuit, test patterns, comparison mechanism, reference output. The circuit is stimulated with a test pattern and the reference output is compared with the actual one. The circuit is what derives from the path followed by the designers, while the test pattern and reference outputs are what derives from the path followed

by the verifiers. It is easy to understand that using a programming language for verification introduces

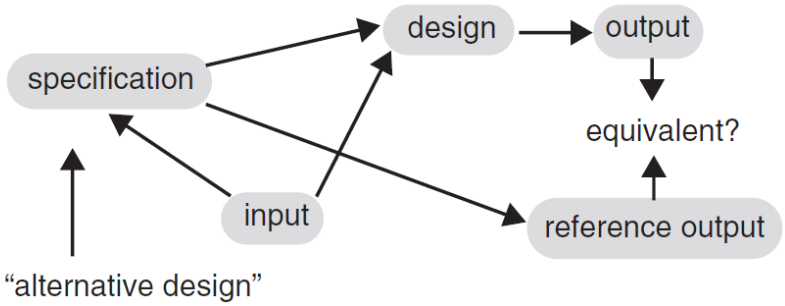


Figure 2.27: Verification by redundancy with simulation [20]

errors (because we are writing additional code). Furthermore, using a different one from the one used to describe the design involves inevitable difficulties between the two approaches. For example, languages like C/C++, used for verification, do not model timing and parallelism as easily as Verilog, used to design. For such reasons, verification engineers find themselves having to manage verification errors in addition to those probably present in the design: employing more time and costs in the project.
Regarding errors related to *specifications*, they can be colliding requirements, unrealized features or unspecified functional details. The problem with this type of errors is that, being at the top of the abstraction hierarchy, it is not possible to have a model to check with. In reality, what happens is to have a series of meetings and design reviews within the team of engineers, addressing the architecture that is going to be designed. In addition, requirements that are not feasible, for technological reasons or for reasons related to available or conflicting resources, they are going to emerge over time during implementation and therefore managed from time to time.

Verification methodologies

Verifying a design begins with a test plan detailing the features to be tested to meet the specification. A test plan consists of features, operations, corner cases and scenarios. Its progress is followed by a *scoreboarding* scheme, through the use of test cases [20].
This is detailed in the next chapter, when it is discussed about the *verification plan* which was developed for AHB. The verification plan is a test plan that uses functional coverage, but the simulations and tests are random, having even more advantages than a simple test plan.
During scoreboarding, test plan elements are marked as verified. In reality, it is not that easy to check specifications in a complete way, so metrics are used.
Some of the most common metrics are **code coverage** and **functional coverage**.
Functional coverage is a measure of which design features were exercised by the tests, while code coverage is the percentage of code exercised by the tests. However, neither of the two covers guarantees the absence of bugs. Through coverage metrics it is possible to understand if a part of the design has not been exercised. Parallel to a test plan, a verification methodology also requires choosing a programming language used for verification. Since there are different needs from the design - e.g. there is no need to synthesize the code - usually a high-level language is used that is closer to the software level than to the hardware one, such as SystemVerilog, Specman/e, SystemC, etc.
In literature there are two types of verification methodologies, based on the presence or absence of vectors: simulation-based verification and formal method-based verification. Another way to distinguish them is input or output oriented.

Simulation-Based Verification

Simulation-based verification consists in having a testbench, where the design is placed and input stimuli are applied and the design outputs are collected (Design Under Test) which are compared with those expected. Both the expected inputs and outputs can be calculated in advance and loaded during the simulation, or they can be dynamically generated. In the first place, a design is usually passed through a static code analysis tool, called a *lint* which looks for potential errors and violations, such as a bus without a driver, the width port of an instance that does not match the module definition or pending inputs of a gate. For what concerns the production of the input vectors, it is possible to talk about **directed tests**, when the input vectors are directly chosen by the engineers. The problem with choosing this way is that they limit the choice only on the basis of their knowledge, which often leaves unexplored regions and therefore with potential bugs. To avoid this usually a seed is provided and *random* vectors of input and expected outputs are produced, thus having tests also in the surroundings of the directed tests. These tests are used by a *simulator* that performs the simulations. There are many on the market and they can be **hardware simulators**, **cycle-based software** or **event-driven**.
Event-driven simulator makes evaluations whenever the inputs of a gate or the sensitivity variables change a block statement. The change is called an *event*. A *cycle-based simulator* partitions a circuit and evaluates at each clock edge. Clearly this type of simulator can only be used for synchronous circuits. A *hardware simulator* on the other hand uses hardware elements such as FPGAs (Field Programmable Gate Arrays) to model the design, i.e. they are programmed to emulate the behavior of the design gates. Using a simulation, outputs are obtained and if they do not comply with those expected, the cause of the problem is investigated. It is done by analyzing the code and checking the waveforms by observing the changes in the values assumed by variables over time, identifying the anomaly. Usually when a bug is found, it is communicated to the designer in order to fix it. This is done using a *bug tracking system* that informs the person who was in charge of the design and it tracks its progress, from verification to resolution. In large projects, a revision control software is also used to arbitrate the management between multiple users, in order to make files always accessible to all members of a team, avoiding losing changes.

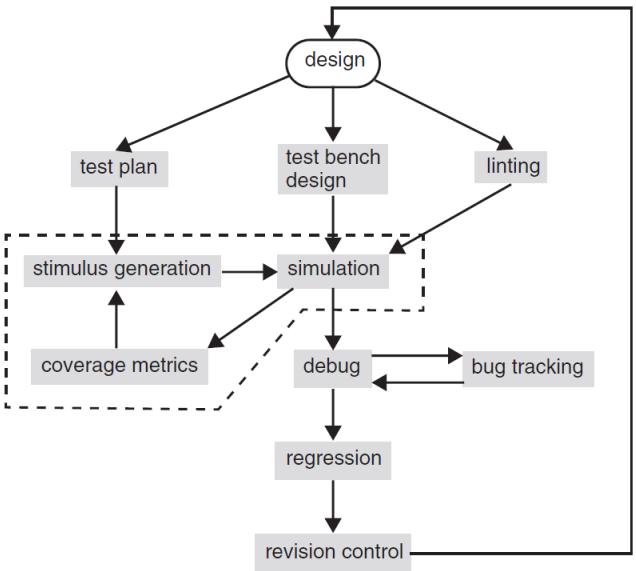


Figure 2.28: Simulation-based verification [20]

Formal verification

Although this method has not been adopted to verify AHB, it deserves a mention for completeness. In the verification based on formal methods there is no need to create test vectors and it is distinguished in: **equivalence checking** and **property verification**.
Equivalence checking checks if two implementations are functionally equivalent.
There are two approaches to address this problem:

1. A first one, called **SAT** (satisfability), consists in systematically finding an input vector for which the two circuits differ.

2. A second one consists in **representing** the two circuits (their logical functions) in a **canonical form**, e.g. ROBDD (Reduced Ordered Binary Decision Diagram) and to check if they are equivalent, which means that the two representations are *isomorphic*.

An example of application can be in verifying the integrity of an IC layout with its RTL version. Verification can be done by extracting the transistor netlist from the layout and then comparing it with the RTL version. If a check fails, then an input vector is generated which proves that the two circuits are different. Then the verifier can understand if the failure derives from a real error or if unintentional conditions have been stimulated. A second type of checking is called *property verification*. It takes a design and a property and it proves that the design has it or not. A program that checks this condition is called a **model checker**. The idea is to explore the entire state space for points that make the property fail. This point, if present, is a counterexample and, by analyzing the waveforms, the verifier is able to debug the failure. In order to avoid bugs that may not be relevant, it is necessary to constrain the exploration of the space of the assumable states and inputs. For example, a design inserted within a complex system may not assume all possible states or input values, so it could fall into irrelevant errors. Nevertheless, setting the constraints is not trivial and it can require a lot of effort. Another approach is called **theorem-proving** which uses deductive methods. A property is specified as a mathematical proposition and the design is described with a set of axioms, hence mathematical entities. The idea is to be able to determine if the proposition can be deduced from the axioms. If it is possible, then the property is proven. This approach requires a great knowledge of the tools' internal operations and familiarity with the mathematical proof process, but it can accept more complex properties.

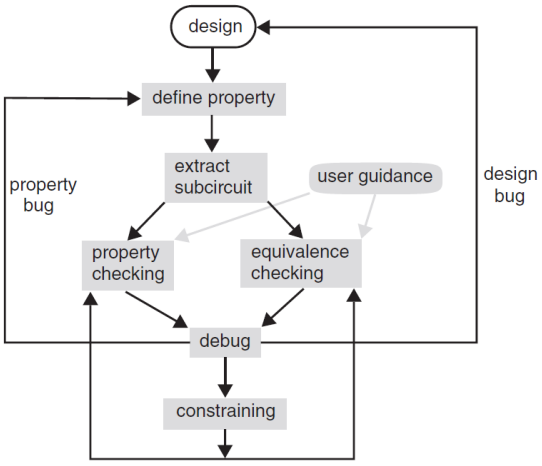


Figure 2.29: Formal verification [20]

2.2.2 SystemVerilog

The programming language adopted for the verification of AMBA AHB was SystemVerilog, widely used on an industrial level for design verification [21]. It is standardized as IEEE 1800 and it is a language based on Verilog, which is used for both design verification and hardware description at the register-transfer (RTL) level. Historically it was developed by Synopsys, and then, in 2005, it became an IEEE standard, with the latest version available dating back to 2017 [22]. Regarding its use for verification, it is a strongly object-oriented language, similar in this sense more to Java than to Verilog, allowing the creation of extensible and flexible testbenches and constructs that are not synthesizable. Many improvements have been done over Verilog, including:

- A **string** type with variable length.
- **Dynamic arrays** allocated at runtime, **associative arrays** and **queues**.
- **Object-oriented programming**: classes, single inheritance, polymorphism, interfaces, data hiding of methods and properties.
- **Random variables with constrain**: widely used in creating random verification scenarios.
- **Methods for randomization** (pre_randomize, randomize, post_randomize, constraint_mode, random_mode): methods available to programmers to randomize the variables of a class, to perform before and after that some manipulations and to control the randomization. In particular, the last two functions allow to exclude variables in the randomization or not to consider constraints.
- **Assertions**: they continuously verify that a design property is satisfied when a condition is met or a state of the system is reached. They are built from **sequences** and **properties**. The sequences are Boolean expressions augmented with time operators [21] and the properties are a superset of them. To stimulate them, stimulus tests are used and their coverage can be guaranteed with coverage groups.
- **Coverage group**: the coverage consists in the collection of statistics based on the sampling of events that occur during the simulation. A *bins database* is created which stores a histogram of values related to a variable. It is also possible to combine multiple variables in order to carry out *cross-coverage*. Furthermore, to catch the behaviour of a variable when needed, SystemVerilog offers a *sample* function.

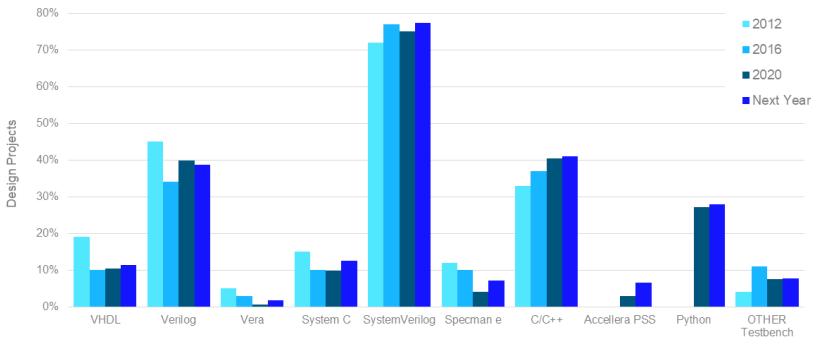


Figure 2.30: Current trend of use of ASIC verification languages [23]

It can be observed in the figure 2.30 how its use for the verification of ASIC/IC is predominant and how it is expected to be massively used also in the near future.

2.2.3 UVM

In order to verify the AHB, Universal Verification Methodology (UVM) was used. It has been an IEEE standard since 2017 and widely adopted within the electronics industry to unify the verification process [27]. UVM is managed by Accelera Systems, a standards organization in the EDA (Electronic Design Automation) industry but it has historically been developed by Synopsys and then extended by NVIDIA, Mentor Graphics, Cadence Design Systems and AMD and it is supported by numerous vendors [26][24]. Its reference implementation is in SystemVerilog and it extensively uses the factory pattern. The major benefits of UVM, as pointed out in [25] are:

- Defines a standard structure for testbenches.
- Defines a clear methodology to be followed for verification.
- Divides verification into three problems:
  1. Test scenarios definition.
  2. Pin-level protocols implementation with drivers and monitors which convert transaction-level stimulus into pin-level stimulus [27].
  3. Active verification with results analysis.

In the following, the main UVM components present in a testbench and the classes provided by the library are described according to the specifications [24].

UVM Testbench Architecture

UVM recommends a structure to be adopted when creating a testbench, as shown in the figure 2.31. In particular, two major components and the connection between these two can be observed: the instantiation of the Design Under Test (DUT) and a UVM Test. Each individual element of the test is described below and it is typically instantiated dynamically. This has an advantage for the testbench which does not need to be recompiled every time, as it can dynamically launch all the compiled tests and even make use of different parameters.

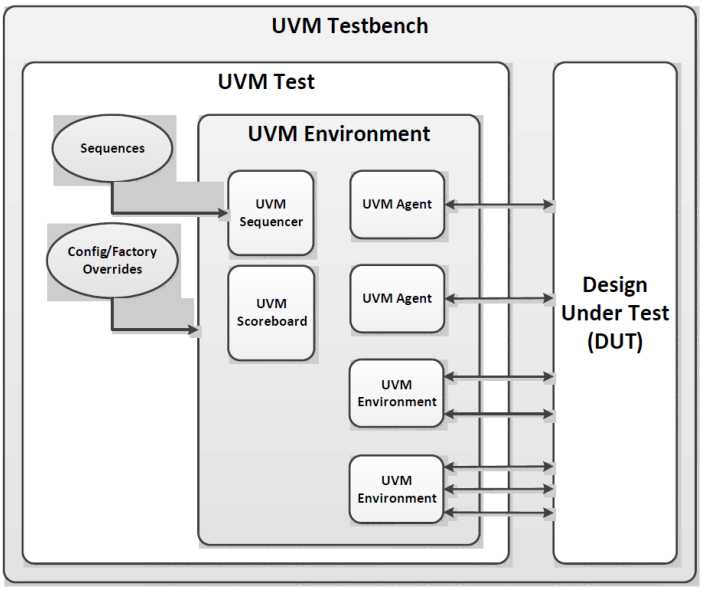


Figure 2.31: UVM Testbench architecture [27]

UVM Test

This is the top level component in the testbench. Basically it performs three operations:

- Instantiate the top-level environment.
- Configure the environment (using factory override and configuration database).
- Apply stimuli to the DUT by invoking UVM Sequences.

Typically there is a base test which is extended by other tests. This basic test instantiates a UVM Environment and other common test elements. Each individual test then customizes the environment or selects a different sequence to be run.

UVM Environment

It is a hierarchical component that groups together other components of the verification, linked to each other by some relationship. Typically there are other UVM Environments, Scoreboards or Agents.

UVM Scoreboard

It is a component that deals with the verification of the behavior of the DUT. In particular, it receives transactions, usually through *analysis ports* from agents, containing the inputs and outputs of the component to be verified. Using a DUT reference model, also known as a *predictor*, it compares the expected outputs with those actually produced by the component.

UVM Agent

An agent takes care of hierarchically grouping other components that communicate with a specific interface of the DUT. This is is shown in figure 2.32.
It usually contains:

- A UVM Sequencer.
- A UVM Driver.
- A UVM Monitor.
- Other components such as checkers, coverage collectors, etc.

An agent needs to operate both actively and passively. In active way, it is capable of generating stimuli, while in the passive way, it is limited only to observing the interface.

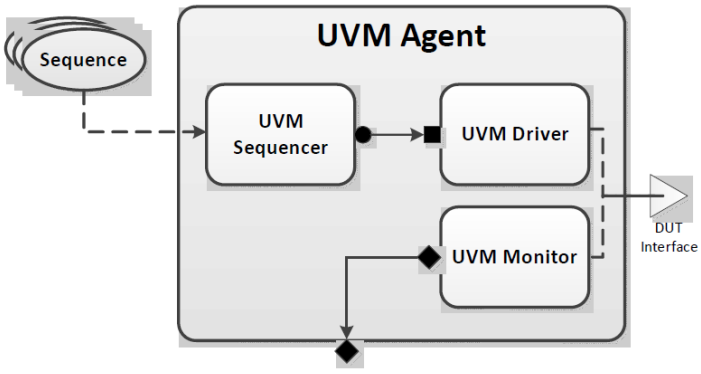


Figure 2.32: UVM Agent [27]

UVM Sequencer

A sequencer works as an arbiter to control the transaction flow from multiple sequences stimulus. In particular, it checks the flow of UVM Sequence items generated by UVM sequences. It is connected to the driver. Furthermore, it is typically the class that sequences can access to get information of the environment they are on (through p_sequencer).

UVM Sequence

It is an object that describes a particular behavior, defining the stimuli to stimulate the unit under observation. They can be *transient* or *persistent*: they can exist for a single transaction, for the duration of the entire simulation, or they can drive stimuli anywhere in the middle of the simulation. The sequences between them can also call each others. In literature, they are defined as *parent sequence* and *child sequence*. To work, a sequence needs to be linked to a sequencer which can also deal with more sequences.

UVM Driver

A driver receives sequence items transactions from the sequencer and drives them, or applies the stimuli to the interface of the DUT. Thus a driver converts transaction-level stimuli to pin-level stimuli. It also has a TLM port to receive transactions from the sequencer and to access the DUT interface.

UVM Monitor

A monitor captures the activities of the unit under observation, converting pin-level activities into transactions. Typically it has access to the DUT interface and it has an analysis port TLM to send the transactions created where needed within the testbench. A monitor can also internally perform some operations on captured transactions (such as verify the coverage, check, log) or it can delegate these operations to other components through analysis ports.

UVM Class Library

UVM provides a library of basic classes, utilities and macros, as shown in the figure 2.33, in order to build components and environments that can be developed quickly and they can be reusable. Components are instantiated hierarchically and they are located within a set of phases, which can be extended as needed. Among the advantages of using this class library there are:

- Having access to a set of built-in features such as printing, copying and other methods that facilitate the verification process and the use of the factory pattern.
- Being able to correctly implement the concepts related to the UVM standard, i.e. each element described above has a relative significant class that improves the readability of the code.
- Functions to support debugging, customizable functions with report messages or error reporting.
- Support for communication infrastructure from verification components (TLM).

Factory pattern

The factory pattern is a creational design pattern, commonly used in object-oriented programming and typical of software engineering. An object is used to instantiate other objects.
Specifically, UVM offers two ways to register objects[28]:

- In the declaration of a class X, the macros *uvm_object_utils (X)* or *uvm_component_utils (X)* can be invoked.

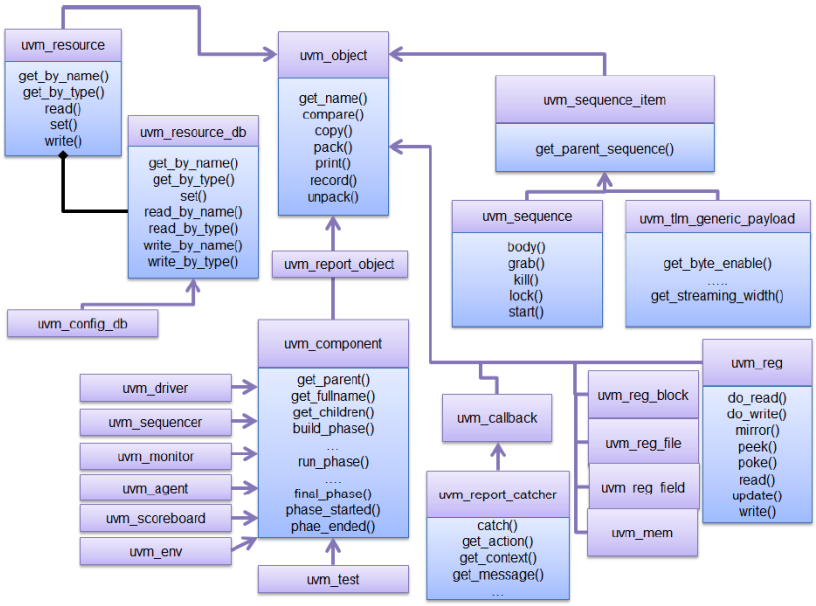


Figure 2.33: UVM class diagram [27]

The utility macros provide implementations for the methods:

- *create*: which allocates a component of type T using a two argument constructor. Needed for cloning.
- *get_type_name*: which returns type T as a string and it is used for several debugging features.
- *get_type*: which is used when configuring the factory.
- *get_object_type*: which works just like the static get_type() method, but operates on an already allocated object.

- Calling later *uvm_object_registry (X, S)* or *uvm_component_registry (X, S)* functions. This one maps a string S to a class type X.

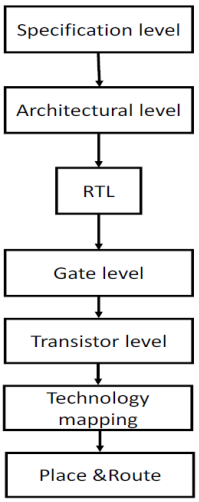


Figure 2.34: Abstraction levels in a digital system design [25]

2.2.4 Tools

Hierarchically, the development of a software system can be seen as composed of levels of abstraction (Figure 2.34). Usually at the *architectural level*, the system is modeled as composed of modules and the major interconnections between the systems are emphasized. Not much information about timing is provided, and descriptive languages such as SystemVerilog and SystemC are commonly used.
The lower level is called *Register transfer level (RTL)*, where the most important characteristics of the modules are modeled. They are cycle accurate, they contain the functional details of the design and there are more timing details. The most used languages for this phase are Verilog and VHDL.
In order to support engineers in these stages of the development of digital systems, on the market, numerous softwares that deal with emulating, accelerating and simulating hardware description languages (HDLs) have been presented. The most adopted are produced by large leading companies in the EDA industry, but there are also open-source and free alternatives.
Hereon some ones are detailed, where the first three are also referred as *the big 3*[29]:

- **Cadence Design Systems Xcelium Simulator**: Includes support for Verilog, VHDL, SystemVerilog, SystemC. It helps in the design and verification of ASICs, SoCs, and FPGAs. It is still referred as the previous name of the tool like: NCSim or ldv (logic design and verification).
- **Siemens Modelsim and Questa**: It supports Verilog, SystemVerilog, VHDL and it allows the use of advanced verification methodologies, such as assertion-based verification and functional coverage, as well as standard verification methodologies such as OVM and UVM. Modelsim is widely used for FPGA design and it is also present in Quartus Prime. Produced by Mentor Graphics, later acquired by Siemens.
- **Synopsys VCS**: It offers support for all functional verification methodologies and languages, from VHDL to Verilog, SystemVerilog, SystemC, C, C++, Verilog AMS and it provides support for advanced simulation technologies, fine-grain parallelism, unreachability analysis, X-propagation.
- **Xilinx ISE, XSIM (Vivado Suite)**: They are simulators that support VHDL, Verilog, SystemVerilog, SystemC codes. They provide advanced debugging tools such as: step through code, breakpoints, cross-probing, value probes, call stack and local variable windows. They allow engineers to view waveforms, support functional coverage, advanced verification methodologies (such as UVM at its latest version of the standard).
- **Intel Quartus Prime**: It supports Verilog, VHDL and AHDL (Altera). It offers features which allow developer to perform analysis, compilation, simulations with different stimuli, synthesis of designs, timing analysis, visualize RTL diagrams.
- **Aldec Active HDL/Riviera-PRO**: They support VHDL, Verilog and SystemVerilog and they are simulators dedicated to the development of FPGA applications. Riviera-PRO is an advanced version of Active HDL and it offers more debugging capabilities, compatibility with modern verification methodologies (such as UVM and assertion-based verification), as well as accepting the very last versions of the languages' standards.
- **Veripool Verilator**: It is an open-source simulator that supports Verilog and SystemVerilog and it compiles synthesizable Verilog code, but it has some limits, not accepting Verilog behavioral code for testbenches.
- **T. Gingold's GHDL**: It is an open-source simulator for VHDL-1987, 1993, 2002 and 2008, licensed under the GPL.
- **N. Gasson's NVC**: It is an open-source compiler and simulator for VHDL-1993 and VHDL-2008, available for Linux, macOS and Windows systems.

CHAPTER 3

Functional verification

3.1 Functional verification

The *functional verification* process is usually performed in parallel with the design process and it requires engineers to read the specifications and to create a verification plan to verify that the device actually achieves its intended goal. According to [30], it is composed of four steps:

1. Writing a verification plan.
2. Implementing a verification environment.
3. Performing a device bring-up.
4. Performing one or more device regressions.

In the following, these four steps have been presented, describing a generic approach for an integrated circuit. This one slightly differs from verifying a reusable IP such as AHB and the differences are mentioned when appropriate.

3.1.1 Verification plan

A verification plan is a natural language document which expresses two main purposes: *what* should be verified and *how* it should be verified. In order to quantify the verification of a component, coverage metrics are adopted that are based on elements of the specifications to be implemented within the verification environment.

Features extraction

Firstly, an **analysis of the design specifications** takes place, where the most important *features* of the design are *extracted*, following top-down or bottom-up approaches. This process requires the use of all available documentation, from the main design specifications to market information and standards upon which the architecture is based.
In practice this may mean having to go through the analysis of:

- **Design Specifications**: the specifications of an integrated circuit (IC).
- **Requirements Specifications**: technical document that specifies how a system is designed based on business requirements documentations.

- **Block Level Design Specifications** or High Level Design Document: document that provides an overview of the system.

As indicated in [25], if it is required to verify a particularly complex system such as a SoC (composed of memories, DMA, CPU, Timers, Controllers, I/O ports, busses, etc), it is needed to go through the features extraction in the following fashion:

1. Identify submodules within the target device.
2. Determine submodules stimulus and response methodology.
3. Determine stimulus traffic generation requirements.
4. Determine submodules response checking methodology.
5. Device interconnection analysis.
6. Power consumption requirements.
7. Assertion verification.
8. Functional coverage.
9. Co-verification possibilities.
10. Hardware acceleration.
11. Virtual platforms.
12. Analog/Mixed devices.

where the 1-6 elements concern the main features to be extracted from the device specification document, 7-9 elements are about how metrics are used in verification and the final ones of the list are about how it might be possible to speed up the development of the verification plan. What ultimately is generated is a document based on features and subfeatures each with a **verification goal** such as:

- Feature
  - Subfeature
  - Subfeature
  - ...
- Feature
  - Subfeature
  - Subfeature
  - ...
- ...

As previously mentioned AHB is an IP rather than an integrated circuit, therefore the verification process has been slightly different. For what has been presented so far, only the specification document provided by ARM has been used and the verification flow has been mainly like the following:

1. Creation of verification components (VC) for slave and master.
2. Replacement of RTL in place of the above elements.
   - Replacing the VC only per master with an RTL master and verification was performed.
   - Replacing the VC only for slave with an RTL slave and verification was performed.
3. Replacement of master and slave with RTL and plug of the IP verification in the chip.

In literature, the drafting of a verification plan for a simulation-based verification methodology, it is usually depicted as composed of three categories:

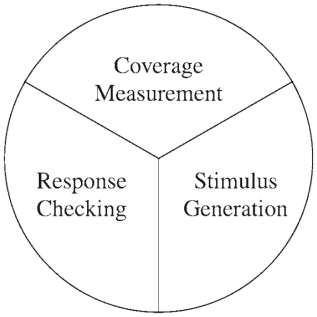


Figure 3.1: Functional verification aspects [30]

Coverage Measurement

The verification is measured based on *functional coverage, code coverage, assertion coverage* or hybrid and they compose the **coverage model** of the verification plan.

- To perform a *functional coverage*, usually for each feature obtained in the previous step, *coverage matrices* are defined that express the goals of the verification and keep track of their results. They determine how much functionality of a design has been exercised by a verification environment.
- *Code coverage* is a measurement, usually a percentage, of how many lines, blocks, statements of the RTL code are executed while the tests are running. Ideally, it should be performed when close to full functional coverage and the RTL is almost stable. The verification plan should express the coverage criteria that are going to be used and which specific set of stimuli should be provided. Some criterias for the code coverage can be statement, toggle, branch, conditions, path coverages.
- *Assertion coverage* measures which assertions have been activated. It is useful to check whether the assertion is coded correctly, and whether the test suite is capable of causing the condition that is being checked to occur [31].

Stimulus Generation

To verify the features specified in the verification plan, it is necessary to decide on the appropriate stimuli to be provided to the design under test. This is expressed in providing test patterns, transactions, device configurations, assembly programs and all that may be necessary to stimulate the device behaviors. Usually the type of stimulus is linked to the abstractive level in which the analysis takes place and to the type of design to be verified.
As presented in [30], it is possible to have two generation methodologies related to the use that is made of the measurements as a result. If inputs are applied to the design and the measurements obtained as an output are sent back to choose the right inputs, then it is defined a **closed-loop** system. On the other hand, without a feedback an **open-loop** system.
The *open-loop* methodologies begin by trying to abstract the design, creating a representation of it. This can be, for example, based simply on the possible constraints applied to the values of the design inputs or more complex models such as graphs indicating all the possible connections between the memory elements of the design[30]. Once this representation is obtained, a set of stimuli can be generated with a **manual strategy** or with **automatic algorithms**.
In addition to these steps, closed-loop methodologies have an *evaluator* in the feedback that verifies the quality of the stimuli. In particular, *potential stimuli* are generated which are evaluated and then

added or not to the *final* set of *stimuli*, because they are able to satisfy the qualitative parameters defined in the evaluator. This is therefore a great advantage, because it is possible to reach a wider range of stimuli. On the other hand, there is too much dependency on design. In fact, if design changes occur, as it usually happens, it is no longer certain that the stimulus, taken for example random with a particular seed, it may continue to be significant. The most adopted approach is therefore open-loop. Returning to how stimuli are generated, it is possible to have a generation that is based on human abilities, having a set of manually created stimuli and automatic approaches that are based on random, deterministic techniques and heuristic algorithms.
More attention will be given to the *manual* generation of stimuli, which is what was mainly used in verifying AHB. Usually, the main features are defined in functional coverage matrices and then cases of interest are identified (e.g. corner cases) to carry out the verification. It is understandable how a verification engineer must be very competent, having a deep knowledge of the design to be verified and with a strong background, thanks to the experience gained from the verification of previous designs. Clearly this translates into an economic and time expenditure for a company, as highlighted in the previous chapter.
As for the generation of stimuli **automatically**, this occurs using software tools. As indicated in [25], in the market there are tools which at first extract an internal representation to generate the stimuli, while other tools directly create stimuli based on the design under test as it is. Furthermore, several techniques are present: deterministic generation, evolutionary algorithms and random generation techniques. In the following, only the *random generation technique* is presented, used in combination with the manual creation of stimuli for AHB. To generate a set of stimuli in a simple way and without a lot of human intervention, it is possible to explore random generation techniques. In particular, an initial value is provided and repeatability is guaranteed in order to be able to repeat the test in case of bugs. It is important to note that, to produce valid stimuli, these can be constrainted or not. For example for AHB, the number of wait states has been limited to a maximum of 16, as recommended by the specifications[5]. In particular, this technique can be used to complete the set of stimuli, generating a large number of inputs not biased by human knowledge of the design, which however require high computational resources and also an understanding of the inputs generated, which may or may not be significant. In summary, during the verification of AHB it was proceeded to generate stimuli using a manual approach and complementary using an automatic approach, where the stimuli were randomized. In addition, the creation of the RTL components of the slaves, used to replace the verification components also have random behaviours (e.g. producing random answers), which has increased the confidence we have in the goodness of design.

Response Checking

This part of the verification plan describes how the behavior of the design under test is demonstrated to match that required by the specification. In the literature there are two approaches: one that uses a **reference model** and another one that uses **distributed data and temporal checks**. The approach adopted for the verification of AHB is the second, but for the sake of completeness both are presented. A *reference model* is usually an abstract model of the design under test. This means that not many details are provided and it is therefore more suitable for architectural verification rather than implementation one. For example, a gate level model can be verified with an RTL model, but the latter does not contain information on the technological library or its timing, so it can be used in a limited way for feature checks. Adopting *distributed data and temporal checks* instead uses data and temporal monitors to capture device behavior. This behavior is then compared with the expected one. The most common approach is to use a **scoreboard**, a structure also provided by UVM. A *scoreboard* is a data structure that stores both the expected outputs and it is able to get the outputs of the device. By providing the same inputs both to the design and to the scoreboard, the latter is able to evaluate whether the actual outputs comply with those expected.

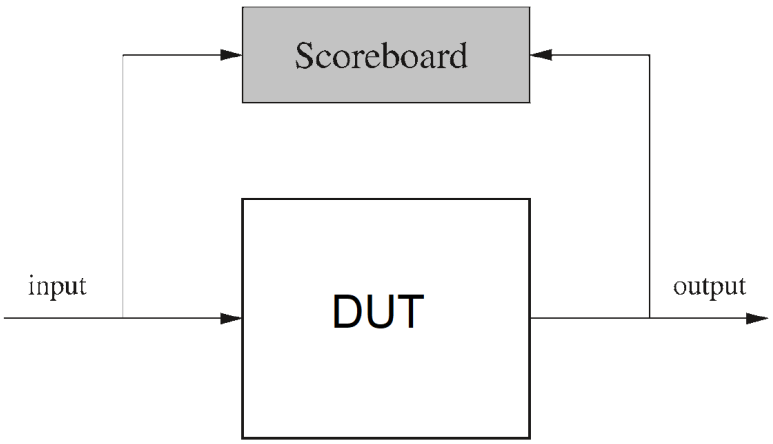


Figure 3.2: Scoreboard architecture [30]

3.1.2 Verification environment, device bring-up and regression

Once the verification plan is ready, it is used as a design specification document for implementing the **verification environment**. The idea is to perform coverage using stimuli and performing the verification as explained so far.
The environment architecture can be made from scratch or external tools on the market can be adopted. Nowadays, a verification IP is reuse-oriented, so it is common to resume environments already developed within the company, as happened for the verification of AHB.
One of the first applications of a verification environment is the **device bring-up** which consists in stimulating the design with simple steps, a small set of simulations to demonstrate basic functionality. Then by making assumptions and validating them, it is proceed with an incremental approach. For example, for a processor it may be possible to assert and negate the value of the reset pin and to check if it actually fetches an instruction after its initial state. Afterwards, it can possible to proceed with other tests, gradually removing the restrictions, going to stimulate the DUT with increasingly complex stimuli. The inputs can be varied both in time and in values, trying different permutations of the data and in different moments in time. Eventually, when the device is able to perform correctly with a high set of simulations covering all features, it can be referred to as *full coverage*.
These simulations and new ones are performed repeatedly, while the design continues to be developed, right up to the time of the tape-out. This process is called **regression** process. The regression aims to identify the reintroduction of bugs in the design. In order to detect them as soon as possible and remove them, a set of simulations, called *regressions*, are periodically launched.
As presented in [30] there are two approaches:

- **Classic regression**: It uses a regression test suite built incrementally over a period of time. It consists of direct tests, specified in the test plan, where each test verifies a feature, a fundamental function of the design. The test suite can also include other types of tests, if for example, they were able to find bugs in an earlier period during development.

- **Automated regression**: It is based on an autonomous verification environment, characterized by aspects of generation, checking and coverage. It is talked about *simulation farms* that receive hundreds of copies of the environment every evening (or at other scheduled intervals), where each copy of the simulation has the same inputs but it differs only by an initial random generation seed. Each regression contributes to verify the design, in terms of functional coverage, code and assertions. Checkers are used to make sure the device does not misbehave. In case of a test failure, it is possible to manually rerun the simulation with the seed that has generated it in

order to investigate if there is a bug in the design or a testbench issue.

In practice, both types of regressions are used for AHB verification. At the beginning a *classic* regression mode is used. A series of very simple tests are run (e.g. **sanity checks**, which quickly evaluate whether a claim or the result of a calculation can possibly be true). They then have been *automated* and launched several times, forming the coverage database of the verification plan.
The process is schematized in the figure 3.3.

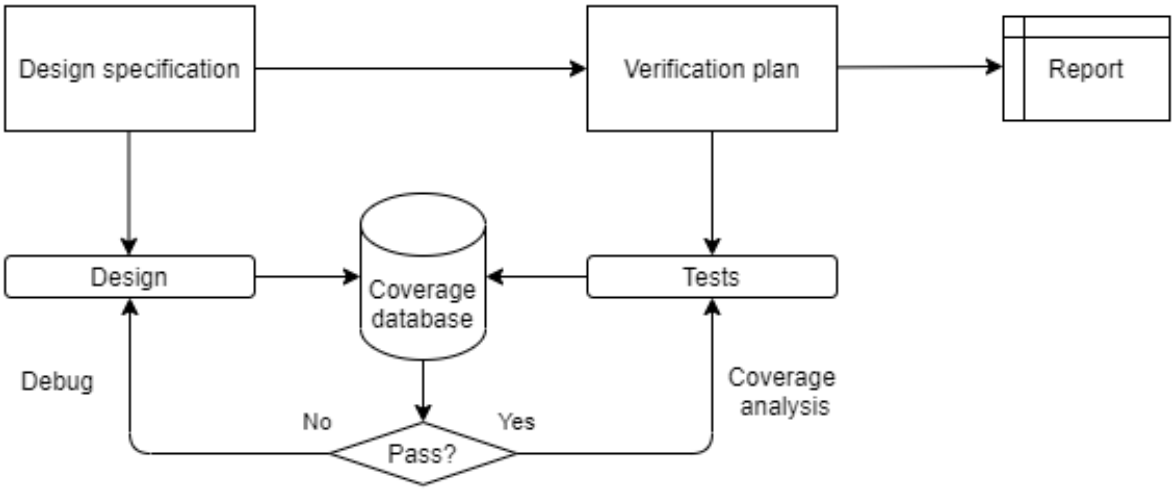


Figure 3.3: Verification architecture

To conclude this section, the methodology used is called **coverage driven** [32], which extend the four major steps presented so far, but it puts more emphasis on modeling based on coverage:

1. Create the verification plan.
2. Create the coverage model from the verification plan.
   - Creation of the verification environment.
   - Coding of checkers, cover groups, test cases.
   - Creation of tests with relevant stimuli.
3. Debug the verification environment, checkers, and coverage model.
4. Run tests with multiple random seeds until cumulative coverage flattens off.
5. Annotate coverage results back onto the verification plan.
6. Run further tests with modified stimulus constraints to close coverage holes.
7. Analyze and prioritize any unverified features and allocate resources accordingly.
8. Run directed tests for particularly hard-to-reach coverage holes.

3.2 Functional verification of AHB

3.2.1 Verification plan

In order to facilitate the integration of the steps just presented, EDA companies offer sophisticated software tools on the market. These ones guide the complete verification process from planning to closure. Some offered features may be the ability to analyze and rank tests to improve regression efficiency, optimize farm utilization, provide objective feedback about verification completeness against the plan and so on. The use that was made for AHB verification is related to definition of attributes, mapping of verification items and collection of verification metrics. In addition, the specification document was linked to the generated verification plan, in order to check constantly what has been verified (*specs annotation*). Therefore, the first step was to load the AHB specifications document into the tool, to extract and to describe all the features and functionality of the IP. To satisfy a complete functional check, three types of metrics were classified:

- **Checks**: Check and assert that a property is satisfied.
- **Coverage**: Cover that a property is satisfied when a sampling event occurs.
- **Test case**: Create stimuli and scenarios using a set of actions.

As explained in the previous chapter, other elements can be inserted to complete the verification plan such as interfaces with other blocks (e.g. busses or protocols), design requirements (e.g. robustness checks, FSM dead lock checks), test scenarios (sequence of test cases) and differences with previous projects. As mentioned, the tool is particularly useful in visualizing the degree of completeness of the verification. In particular, a feature can be said to be complete if:

- All its tests pass.
- Checks in any tests do not fail.
- Coverage goals are hit.

Usually a name is chosen for the element in the verification plan and a brief description is given of what it should do. To view the complete verification plan that was developed for AHB, please refer to the appendix A.

3.2.2 Verification environment

As explained when presenting UVM, the whole testbench is based in connecting a DUT to be tested and a test, as shown in figure 2.31. In our case the DUT to be verified can be viewed as composed of:

- 3 Masters
- 8 Slaves
- 1 AHB Interconnect Fabric

The initial architecture consisted of a 16-bit data bus with a single master and a single slave connected to it. It has been extended to work with three masters and eight slaves with a 32-bits data width, to have a verification as generic as possible. A representation of the DUT can be the following:

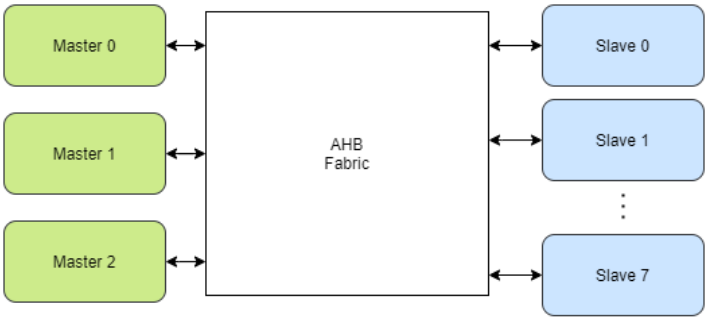


Figure 3.4: DUT to verify

A view of the AHB Fabric has already been seen in figure 2.6, but a simplied way is taken up here and is the following:

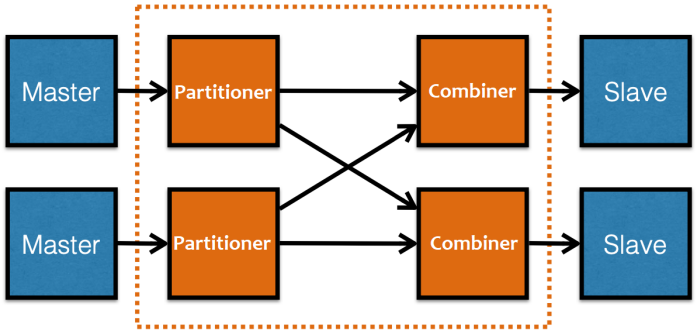


Figure 3.5: Simplified AHB Fabric architecture

Where a **partitioner** is needed when a master wants to connect to several slaves. Instead, when multiple masters share access to a single slave, then a **combiner** is necessary.

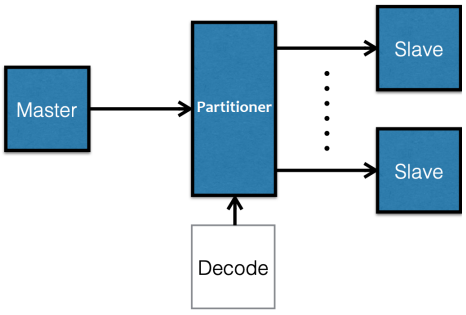


Figure 3.6: AHB Fabric partitioner

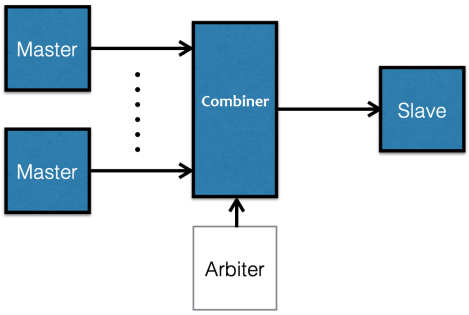


Figure 3.7: AHB Fabric combiner

Each master and each slave have an **agent**, as previously shown in 2.32. It is composed of a driver, a monitor, a sequencer (which receives various sequences) and in addition also of an instance of the *coverage class*, used to collect coverage measurements.

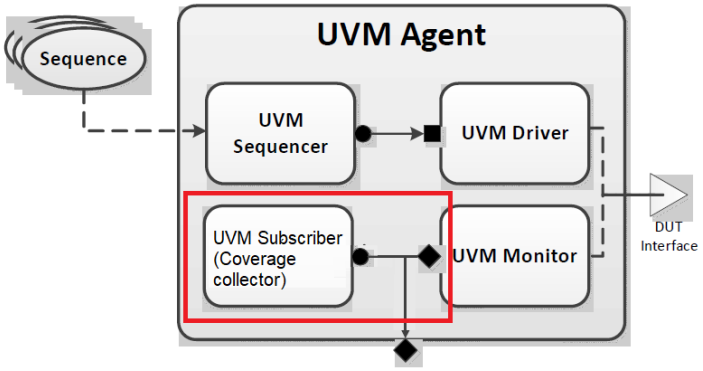


Figure 3.8: UVM Agent in VC

To define a verification environment, UVM offers a base class (uvm_env), which is used as a hierarchical containers of other components that together comprise a complete environment. This one has been inherited and the elements shown in the figure 3.9 have been coded, using the exact same classes offered by UVM in a partially already existing environment.

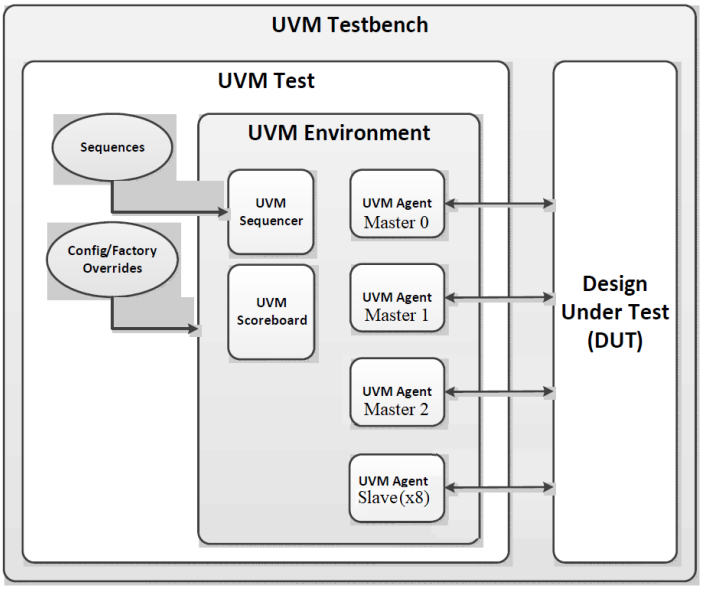


Figure 3.9: UVM Testbench with more slaves and masters

3.2.3 Implementation of an attribute

In the following it is presented how an attribute has been coded using SystemVerilog in the verification environment and how the coverage occurred. As an example, it is possible to cover two *waited transfers*, which are transfers where the slaves enter wait states because they take longer to sample the data from the bus. These have been presented in figures 2.9 and 2.10 where respectively it has been shown a **read transfer with two wait states** and a **write transfer with a wait state**. What it is interesting to check is that a reading and a writing take place correctly even in presence of wait states and that all the signals in our design behave as indicated in the specifications.
First of all, it was chosen to define these two attributes of the verification plan with a **coverage model**. Considering the appendix A, here it is shown the flow for the 1.1.1.1 and 1.1.1.2 attributes.

Within the coverage class present in each agent (Figure 3.8), a **covergroup** has been created. It is a user-defined type in SystemVerilog that contains a coverage model, which is expressed with:

- A set of coverage points, which specifies an expression that is required to cover.
- An event that defines when the covergroup is sampled.

In particular, every time a new transaction is monitored on the bus, the covergroup samples. Since the specifications indicate a maximum of 16 wait states, a coverpoint has been created within the coverage class for:

- Direction (READ/WRITE).
- Type of transfer (IDLE, BUSY, NONSEQ, SEQ).
- Number of wait states: taken as range 0, 1-4, 8-16, external intervals.

```
covergroup ahb_item_cg;
   option.per_instance = 1;
   option.name = "ahb_covergroup";
   coverpoint ahb_item.transfer;
   coverpoint ahb_item.direction;

   coverpoint ahb_item.num_wait_states{
      bins zeros = {0};
      bins low_values = {[1:4]};
      bins middle_values = {[5:7]};
      bins high_values = {[8:16]};
      bins others = default;
   }

endgroup // ahb_item_cg
```

Figure 3.10: Covergroup and coverpoints example

The next step is to stimulate the design. This happens by using **sequences**. A sequence creates a transaction, randomizes it and sends it to a sequencer and then it is fetched by a driver. In the driver, the generated transaction causes reactions on the interface pins [33]. For example, a **write/read_sequence** generates a random write/read transaction and sends it to the sequencer and it is fetched by the driver. The driver interprets the transaction payload and it causes a write/read with the specified address and data.
Enabling the wait states, they are going to be randomly generated per slave within configurable values.

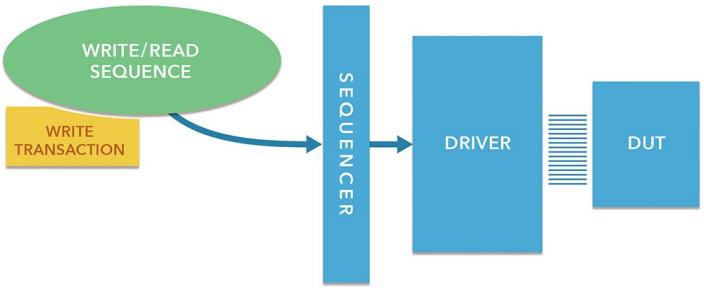


Figure 3.11: Sequence and Sequencer representation [33]

This structure is part of the environment and it is called when a test is defined. The test and the DUT together are part of the testbench.

Several sequences have been realised to test AHB, but for this case it would have been sufficient to use a read and a write sequences, enabling the wait state property. For making sequences, initially a basic sequence was created with all the attributes randomized and some global constraints related to the specifications. Then, based on the type of transfer it is required to test, the sequences have additional constraints. For example for a write sequence, all attributes have been left free to randomize, but the direction has been constrained to be WRITE.
Some examples of sequences which have been realised are:

- **idle_seq**: Idle transfer sequence.
- **write_seq**: Write transfer sequence.
- **write_user_seq**: Write transfer sequence with HPROT equal to non cacheable, non bufferable, user access, data access.
- **write_priv_seq**: Write transfer sequence with HPROT equal to non cacheable, non bufferable, privileged access, data access.
- **read_seq**: Read transfer sequence.
- **read_user_seq**: Read transfer sequence with user access enabled.
- **read_priv_seq**: Read transfer sequence with privileged access enabled.
- **read_opcode_seq**: Read transfer sequence with privileged access and opcode fetch enabled.
- **write_burst_incr4_seq**: Write 4-beats burst incremental transfer sequence.

and they have been used in creating tests like the following ones:

- Basic reads.
- Basic writes.
- Generic test of all the sequences randomizing everything.
- Burst transfers.

Using a simulator for design verification, like the ones presented in the previous chapter, it was possible to visualize the waveforms (Figures 3.12 and 3.13), where it is already possible to check that the component actually performs as required by the specifications.

Figure 3.12: Write transfer with three wait states

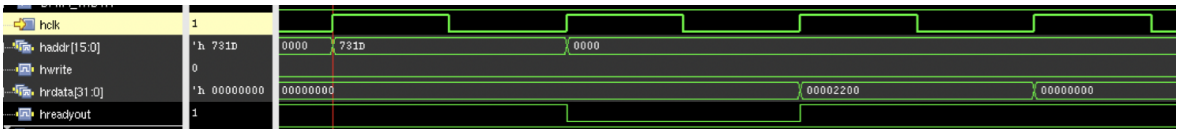


Figure 3.13: Read transfer with one wait state

In order to retrieve metrics, regressions were launched, with different seeds and different sequences and the results were imported in our verification plan tool, creating the coverage database. Subsequently, the code was annotated with the various attributes of the verification plan and it was possible to understand if the required coverage was achieved.

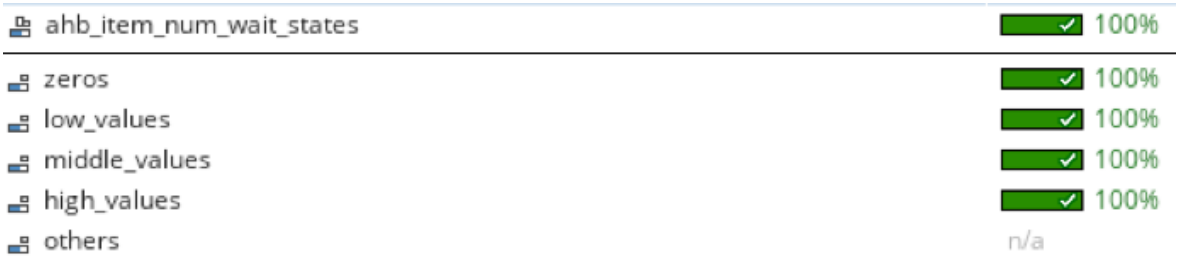


Figure 3.14: Covered coverage bins

3.2.4 Conclusions

What was done for the AHB verification was to create an entire verification plan starting from the most recent specifications released by ARM. This resulted in the creation of checkers, coverage groups and test cases, related sequences and other tests to be able to stimulate them. Furthermore, a previously existing verification environment has been extended as previously explained. It should be emphasized that the verification plan was not completely coded but that at the outset it was chosen to limit to the functions present in the chips of competence of the team where I interned. In addition, we have preferred to dedicate some time in verifying the behavior of the bus through a log-based analysis. To conclude at the moment, the AHB verification component has:

- Fully randomized tests.
- It is capable of verifying simple and advanced transfers (idle transfers, burst transfers, etc.).
- It is capable of verifying masters and slaves.
- It is capable of being connected to real chips.
- It is reusable and can be extended by implementing the remaining features already specified the verification plan.

CHAPTER 4

Log Verification Analysis

4.1 Introduction

Log files provide meaningful information about the implicit and explicit activities of any computer hardware and software system. This type of record reports all the information on the normal operation of a machine or program, helping to intercept anomalies and problems. It was decided to record the transactions that occurr on the bus and to analyse them, trying to get useful information for an IP core such as AHB. Considering the architecture of AMBA, a log can be used to:

- Serve as a **support for design verification**. It is useful to check whether a specific master actually performs a wanted operation towards a certain register of a slave and vice versa.
- Serve as a support for **debugging errors** that may arise during a transfer (backtrace).
- **Stress the interconnect fabric** and see if it can be improved. In particular, this means understanding whether the arbiter's RTL implementation is adequate or it can be improved, thus changing the arbitration policy. For example, it is possible to observe if the arbiter prefers a master rather than another one.
- It can be useful to **understand** if a particular **master completely takes possession** of a bus and how often.
- It can be useful to **understand** which **slave causes more delays** on the bus (e.g. it generates too many wait states, affecting next transactions).
- Understanding slower and predominant masters and higher latency slaves, it may be used in **changing the arbitration policy** of the arbiter giving them a lower or a fair priority. For example, it can be thought of giving a different priority to different masters based on the number of requests they send or the duration of these requests or to use a more equitable algorithm such as **round robin**, in order to avoid an excessive use of the bus by a single master.
- Estimate how much the bus is actually used, in the presence of wait states (**bus throughput**).
- Estimate other features of the bus usage (e.g. power consumptions).

4.2 Implementation of the log

For the implementation of the log, a new class was created, called **recorder**, which is always inserted within the UVM methodology, exploiting the observer pattern. It inherits from the *uvm_subscriber* class and it is called whenever a transaction is needed, writing the transaction into a text file. In particular, a subscriber provides:

1. An *uvm_analysis_port*. It is connected to the *data_port* of the masters' agents.
2. It requires to define a *write()* method that handles the data received. In this case, it is where the log writing is actually done. The one which actually calls the write function is the monitor contained within the agent of every master.

The connection of this component takes place within the environment class and it was chosen that it was sufficient to collect transactions only from the point of view of the masters.

```
// If logging is enabled, connect log port
if(log_enabled) begin
   master0_agent.data_port_i.connect(vc_ahb_recorder_i.analysis_export);
   master1_agent.data_port_i.connect(vc_ahb_recorder_i.analysis_export);
   master2_agent.data_port_i.connect(vc_ahb_recorder_i.analysis_export);
end
```

Figure 4.1: Ports connections between masters' agents and recorder classes

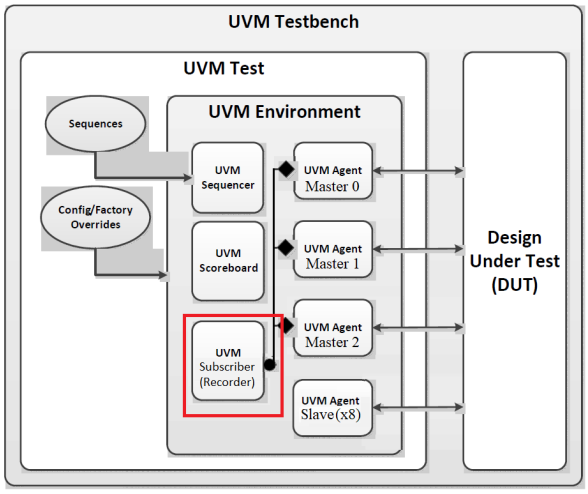


Figure 4.2: Recorder class inside the environment

In particular, it is collected:

- **Start time**: Transaction start time (in ns).
- **End time**: Time of conclusion of the transaction (in ns).
- **Master name**: Name of the master who initiates the transaction. In our case, fictitious names have been provided (master0, master1, etc.), but when inserted within an integrated system, these names are mapped with the real ones (CPU, DMA, GPU, etc.).
- **Operation**: Specifies whether it is a read or a write.

- **Address**: Indicates the slave address, in hexadecimal on 16 bits.
- **Slave name**: Name of the target slave (corresponding to the address). In our case, the names are fictitious (slave0, slave1, etc.), but when inserted within a real integrated system, a mapping with the real names of the slaves can be provided.
- **Register name**: Name of the register within the slave. The mappings are fictitious but when inserted within a real integrated system, a mapping with the real names of the registers can be easily provided.
- **Data**: Data to be transferred in reading/writing. Expressed in hexadecimal on 32 bits.
- **Transfer**: Indicates the type of transfer which can be IDLE, BUSY, NONSEQUENTIAL or SEQUENTIAL.
- **Response**: Indicates if a transfer is OK or if an error occurred.
- **Size**: Indicates the size of the transfer as specified by the protocol (typically byte, halfword, word).
- **Nonsec**: Indicates whether the transfer is secure (0) or non secure (1).
- **Wait states**: Indicates the number of wait states requested by the slave. The maximum achievable is 16 as recommended by the standard.
- **Prot**: Indicates the protection control signals, as specified when the protocol was introduced.

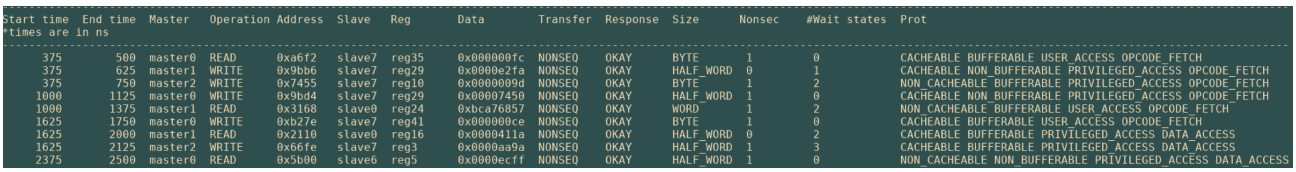
In order to perform analysis, several use cases have been generated. These were defined by varying parameters and coding them into tests that were run with different seeds. After having produced the logs, they have been collected and processed using a MATLAB script that was defined. It extracts the values from the log and then generates the plots that are presented in the following section.

script.m*

```
clear; clc;
S = string(importdata('/Users/emiliovivenzio/Desktop/transactions_log.txt')); %%importing from the log
S = S(5:size(S,1),:); %% Removing the first rows
[S,match] = split(S, "CACHEABLE" | "NON_CACHEABLE"); %splitting at Cacheable or non-cacheable

for index = 1:size(S,1)
  A(index, :) = strsplit(S(index,1));
end
A(:,1) = [];
A(:,14) = [];
startTime = str2double(A(:,1));
endTime = str2double(A(:,2));
master = str2double(erase(A(:,3),"master"));
operation = A(:,4);
address =  erase(A(:,5),"0x");
slave = str2double(erase(A(:,6),"slave"));
reg = str2double(erase(A(:,7),"reg"));
data = erase(A(:,8),"0x");
transfer = A(:,9);
response = A(:,10);
size = A(:,11);
nonsec = str2double(A(:,12));
wait_states = str2double(A(:,13));
prot = match + S(:,2); %Putting again the delimiter
A = [A prot];
clear index S match;
```

Figure 4.3: Snippet of the MATLAB script



Start time	End time	Master	Operation	Address	Slave	Reg	Data	Transfer	Response	Size	Nonsec	#Wait states	Prot
*times are in ns													
375	500	master0	READ	0xa6f2	slave7	reg35	0x000000fc	NONSEQ	OKAY	BYTE	1	0	CACHEABLE BUFFERABLE USER_ACCESS OPCODE_FETCH
375	625	master1	WRITE	0x9bb6	slave7	reg29	0x0000e2fa	NONSEQ	OKAY	HALF_WORD	0	1	CACHEABLE NON_BUFFERABLE PRIVILEGED_ACCESS OPCODE_FETCH
375	750	master2	WRITE	0x7455	slave7	reg10	0x0000009d	NONSEQ	OKAY	BYTE	1	2	NON_CACHEABLE BUFFERABLE PRIVILEGED_ACCESS OPCODE_FETCH
1000	1125	master0	WRITE	0x9bd4	slave7	reg29	0x00097450	NONSEQ	OKAY	HALF_WORD	1	0	CACHEABLE NON_BUFFERABLE PRIVILEGED_ACCESS OPCODE_FETCH
1000	1375	master1	READ	0x3168	slave0	reg24	0xbca76857	NONSEQ	OKAY	WORD	1	2	NON_CACHEABLE BUFFERABLE USER_ACCESS OPCODE_FETCH
1625	1750	master0	WRITE	0xb27e	slave7	reg41	0x000000ce	NONSEQ	OKAY	BYTE	1	0	CACHEABLE BUFFERABLE USER_ACCESS OPCODE_FETCH
1625	2000	master1	READ	0x2110	slave0	reg16	0x0000411a	NONSEQ	OKAY	HALF_WORD	0	2	CACHEABLE BUFFERABLE PRIVILEGED_ACCESS DATA_ACCESS
1625	2125	master2	WRITE	0x66fe	slave7	reg3	0x0000aa9a	NONSEQ	OKAY	HALF_WORD	1	3	CACHEABLE BUFFERABLE PRIVILEGED_ACCESS DATA_ACCESS
2375	2500	master0	READ	0x5b00	slave6	reg5	0x0000ecff	NONSEQ	OKAY	HALF_WORD	1	0	NON_CACHEABLE NON_BUFFERABLE PRIVILEGED_ACCESS DATA_ACCESS

Figure 4.4: Log file example

4.3 Tests and results

The scenarios that can be tested are countless. To narrow down the amount of data to analyse, it was proceeded with a set of representative and meaningful use cases, such as:

1. **Single master (1) - Single slave (1)**
   - Slave without wait states.
   - Slave with wait states (pre-assigned to: 0, 5, 10, 16).
   - Pipelined transfers (**back-to-back**) vs non-pipelined.
2. **Single master (1) - Multiple slaves (8)**
   From the analysis of the previous case, it was observed that *pipelined transfers* are more convenient. Therefore only those have been dealt with in the following.
   - Slaves without wait states.
   - Slaves with pre-assigned wait states on the most frequently accessed slaves (0 and 7):
     (a) Slave0 and slave7 without any wait state. The others without any wait state.
     (b) Slave0 and slave7 with 1 wait state. The others without any wait state.
     (c) Slave0 and slave7 with 2 wait state. The others without any wait state.
     (d) Slave0 and slave7 with 5 wait state. The others without any wait state.
   - Most accessed slaves with random wait states, always changing.
3. **Multiple masters (3) - Single slave (1)**
   - Pipelined transfers.
   - Non-pipelined transfers. Priority masters with randomness. Slave with pre-assigned wait states and equal to: 0, 5, 10, 16.
4. **Multiple masters (3) - Multiple slaves (8)**
   - Non-pipelined transfers. Priority masters with randomness.
     (a) Slaves without wait states.
     (b) Most accessed slaves with random wait states.
     (c) Slaves with pre-assigned wait states and equal to: 0, 5, 10, 16.

4.3.1 Single master (1) - Single slave (1)

The analysis begins with the simplest case: a single master (master0) making hundred transfers to a single slave (slave0). Some examples of what can be observed with the log are:

1. The **type of operation** that occurres most frequently between a particular master and a particular slave. In our simulations, the master performed more readings than writings (Figure: 4.5). Unfortunately, at this level of abstraction, both operations take the same amount of time, which is usually not always true. Therefore, when applied to a real design, delays can be provided.

2. The **size of the operation** requested more frequently. Between the allowed ones for our master (bytes, halfwords, words), it was observed that it has dealt more with byte transfers (Figure: 4.6).

3. The **registers of the slave** the master accesses most frequently. In our simulation mainly register 31 and 6 (Figure: 4.7).

4. The **overall completion time** without wait states and with wait states. In particular, it was possible to observe the difference for few transactions and then as the number of transactions increased, how the presence of wait states became exponentially significant (Figure: 4.10).

5. It was possible to observe how, by exploiting the pipelining offered by the protocol, it was possible to have a significant improvement in performance and a better use of the bus. This is because while the master waits for the response of the previous transfer, a new transfer request is already been sent (Figures 4.8 and 4.9).

It should be noted that if we had generated many more transactions, these results probably would not have been true anymore, but the probabilities would all have become equal. This is due to using the verification IP as master, but the intent here is to show that it is possible, when we are plugging it in a real context, to observe probabilities of occurrence and make analysis thanks to the log.

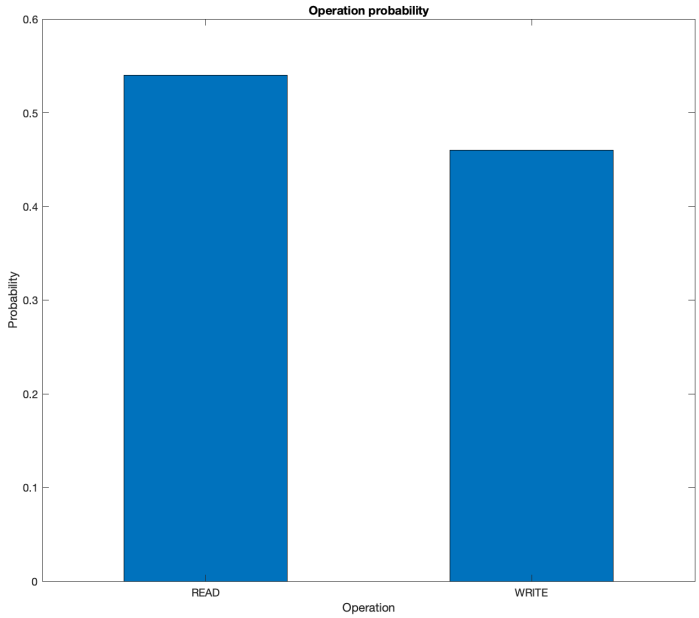


Figure 4.5: Results: Type of operations performed.

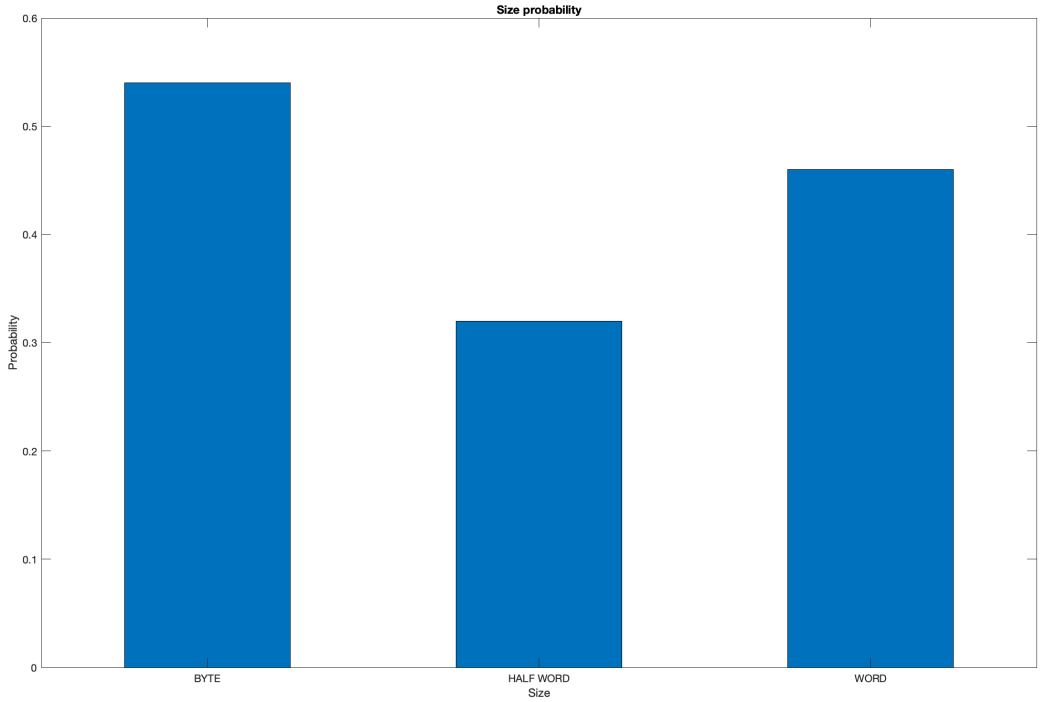


Figure 4.6: Results: Size of operations performed.

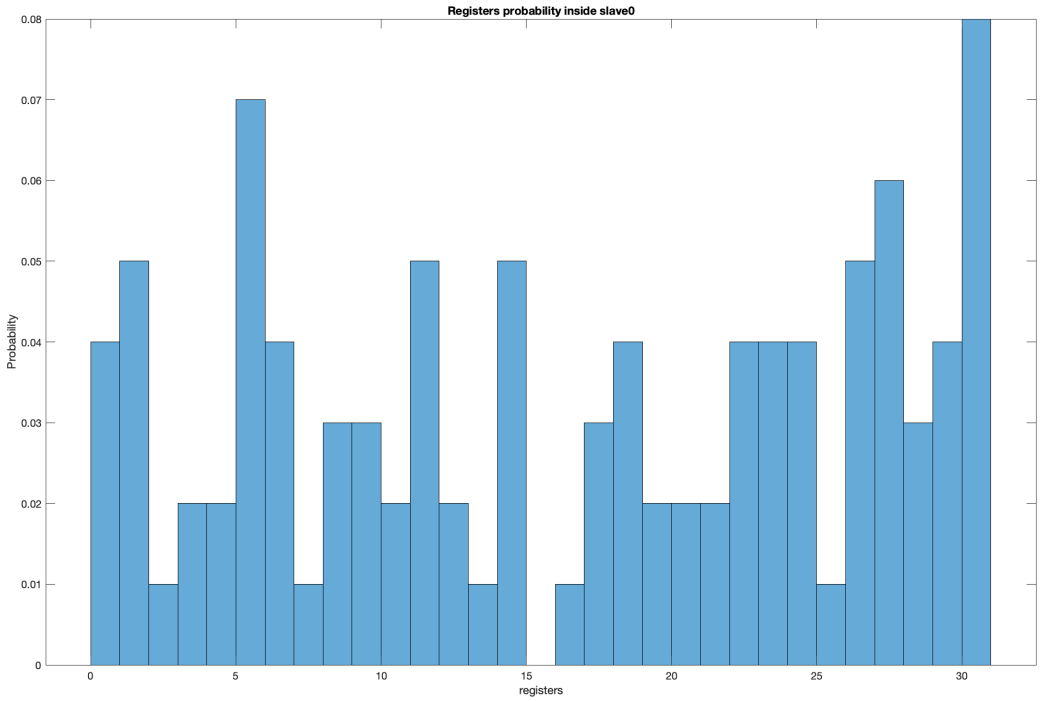


Figure 4.7: Results: Registers most accessed within the same slave.

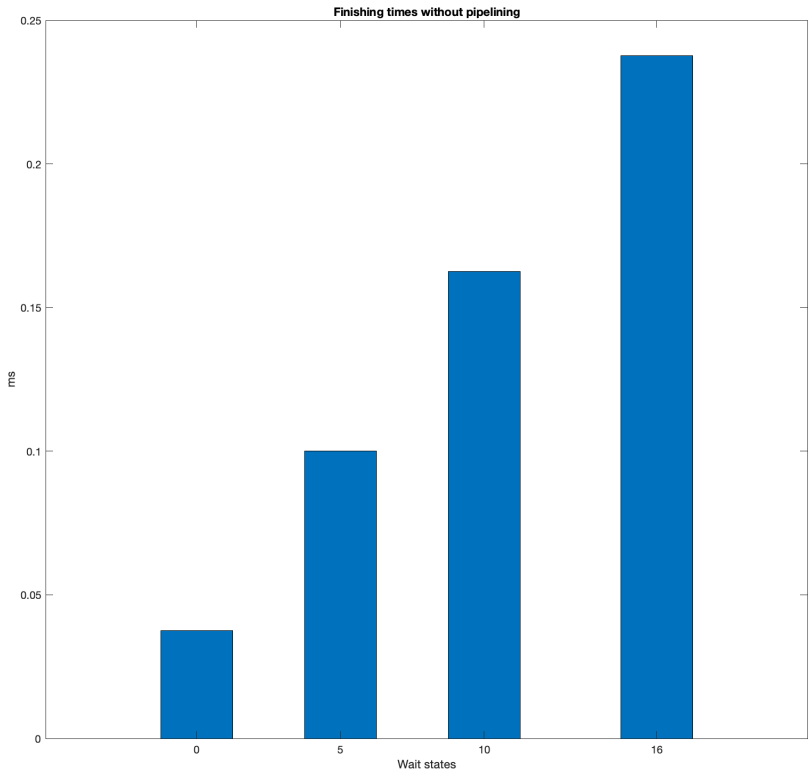


Figure 4.8: Results: Total time to perform 100 transfers without pipelining and increasing wait states.

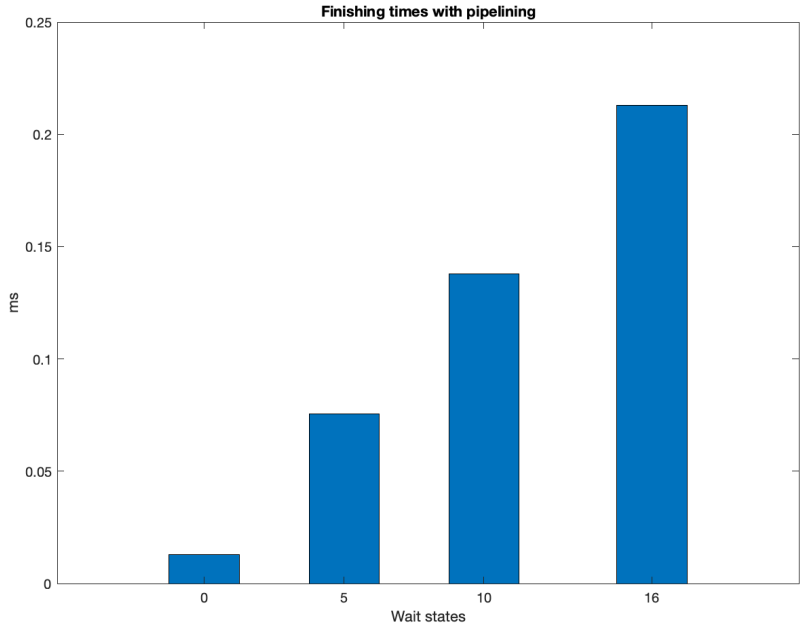


Figure 4.9: Results: Total time to perform 100 transfers with pipelining and increasing wait states.

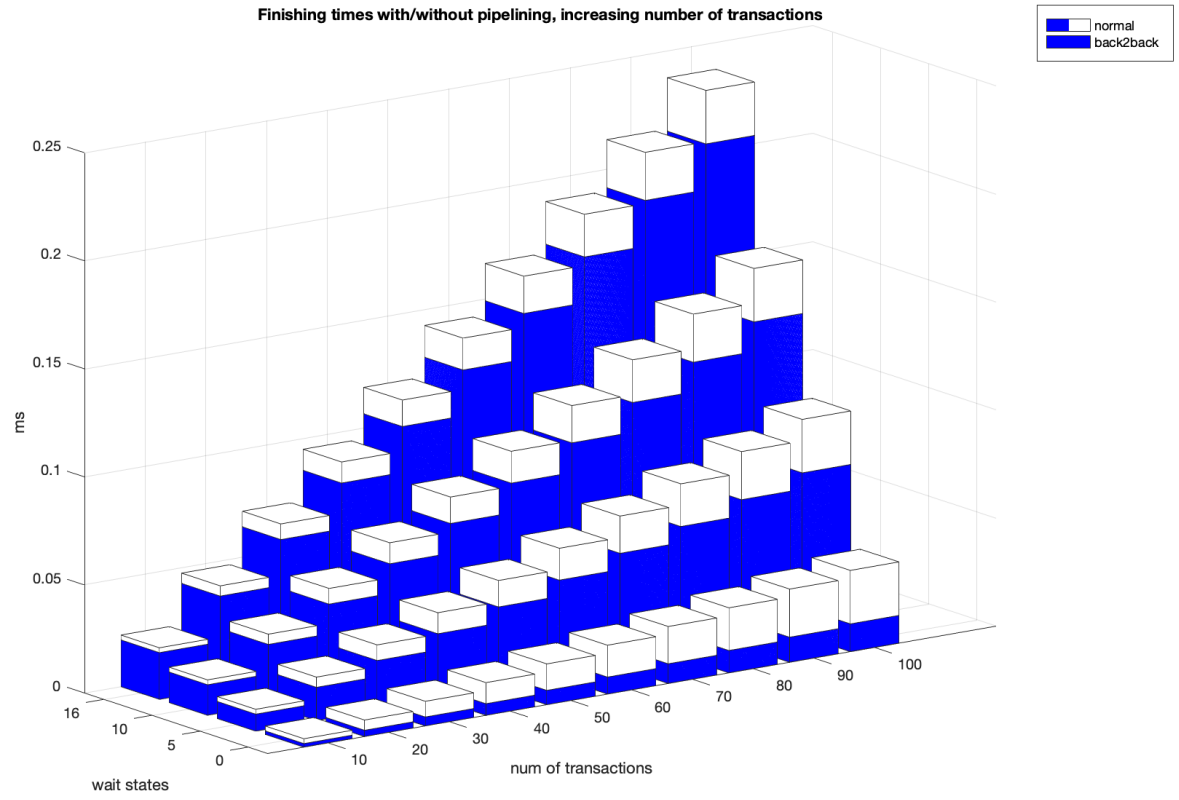


Figure 4.10: Results: Total time to perform 100 transfers with/without pipelining and increasing wait states.

4.3.2 Single master (1) - Multiple slaves (8)

Having observed that it is possible to improve performance with pipelining, subsequent cases developed make only use of this modality. As explained, cases were evaluated in which there are no wait states, random wait states and other cases where the number of wait states were configured. The analysis was carried considering a single master (master0) making hundred transfers to eight slaves (slave0 to slave7). Some examples of what can be observed with the log are:

1. The **type of operation** that occurres most frequently between a particular master and a particular slave. In our simulations, the master performed more operations to slave 7, performing more writings rather than readings (Figure: 4.11). Furthermore, the slave that had the second most number of accesses was slave 0, with more readings than writings. Knowing the access frequency of the slaves and the real access times of the peripherals, it could be possible to estimate which are the critical slaves and to use a scheduler that orders the number of transactions in order to obtain a better throughput of the bus.

2. The **size of the operation** requested more frequently. This allows us to understand how the master works. Between the allowed ones for our master (bytes, halfwords, words), it was observed that it has dealt more with byte transfers (Figure: 4.12).

3. The **overall completion time** without wait states and with fixed and random wait states. It was possible to vary the number of wait states of the most accessed slaves (0 and 7) from 0 to

1, 2, 5 and then randomly chosen during execution and it has been seen that also in this case there is an increasing exponential dependence between the number of wait states and the time required to complete all the transfers (Figure: 4.13).

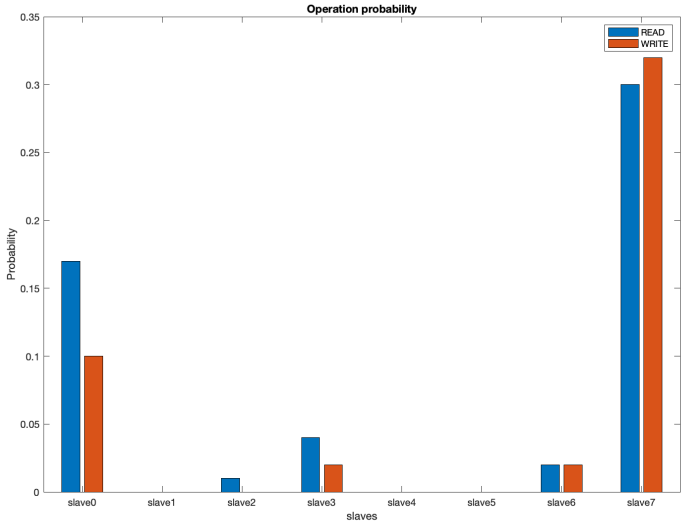


Figure 4.11: Results: Type of operations performed per slave.

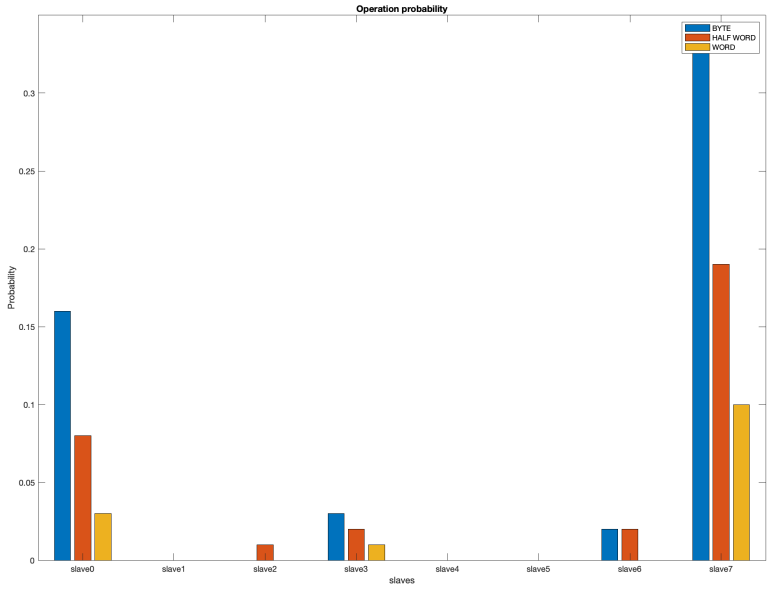


Figure 4.12: Results: Size of operations performed per slave.

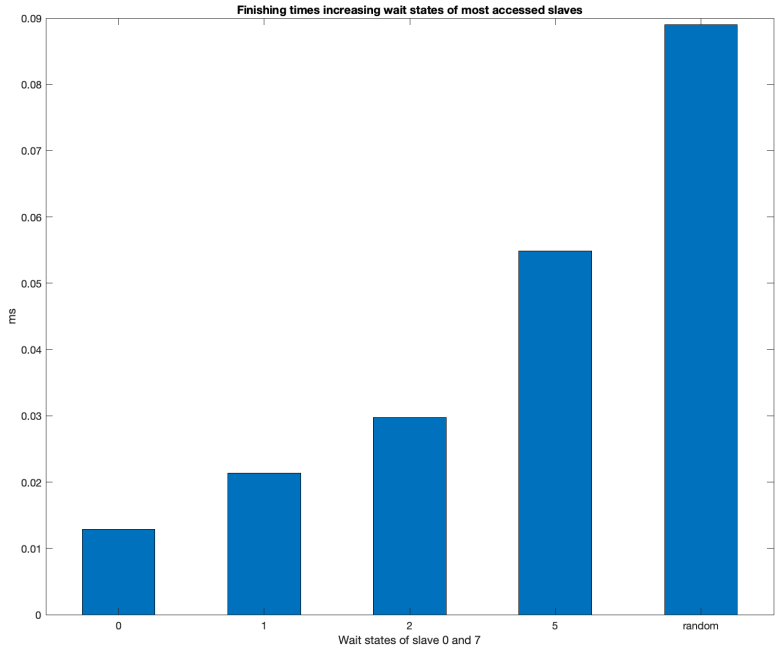


Figure 4.13: Results: End times by varying the wait states of slave0 and slave7.

4.3.3 Multiple masters (3) - Single slave (1)

This type of analysis was based on having three masters, called master0, 1, 2 sending 100 transactions each to the same slave (slave0). At the beginning a pipelined analysis was carried out. This one allowed us to observe the arbiter's priority. When three masters compete for the bus, it prioritizes master0, then master1, then master2. This means that if master0 constantly performs readings or writings, even if master 1 or 2 are waiting for the bus to be granted, it is always going to have a priority.

Start time	End time	Master	Operation	Address	Slave	Reg	Data	Transfer	Response	Size	Nonsec	#Wait states	Prot
*times are in ns													
375	500	master0	READ	0x2434	slave0	reg18	0x00000057	NONSEQ	OKAY	BYTE	1	0	CACHEAI
500	625	master0	WRITE	0x072e	slave0	reg3	0x00000087	NONSEQ	OKAY	BYTE	1	0	NON_CA(
...													
12750	12875	master0	WRITE	0x00f8	slave0	reg0	0xc43962fa	NONSEQ	OKAY	WORD	1	0	CACHEAI
375	13000	master1	WRITE	0x175d	slave0	reg11	0x000000bd	NONSEQ	OKAY	BYTE	0	100	NON_CA(
13000	13125	master1	READ	0x058c	slave0	reg2	0x8da85b54	NONSEQ	OKAY	WORD	1	0	NON_CA(
...													
25250	25375	master1	READ	0x23de	slave0	reg17	0x000000cb	NONSEQ	OKAY	BYTE	1	0	NON_CA(
375	25500	master2	WRITE	0x2753	slave0	reg19	0x00000012	NONSEQ	OKAY	BYTE	1	200	NON_CA(
25500	25625	master2	READ	0x2fa7	slave0	reg23	0x000000ba	NONSEQ	OKAY	BYTE	1	0	NON_CA(
...													
37750	37875	master2	WRITE	0x2cda	slave0	reg22	0x00005b8c	NONSEQ	OKAY	HALF_WORD	1	0	NON_CA(

Figure 4.14: Results: Back-to-back transfers. Detail of the priority given by the arbiter.

The log shows that firstly all the requests from master0 are served, then all those from master1 and finally all the ones from master2, although they all start at the same time. This is coherent with the arbiter's RTL implementation we have, but in a real context, it could allow us to understand which master takes possession of the bus without leaving it, preventing others from being able to make transfers. Therefore, a more fair scheduling technique (such as *round robin*) to grant the access to the bus would have been more appropriate.

A second analysis was made, allowing all the three masters to terminate their transactions, before allowing the other masters to be able to make the next transfer. In addition, randomness has been provided to the two priority masters, master0 and master1, which sometimes do not perform any transfers at all. An analysis without wait states was at first carried out, where it was observed that although everyone starts a transfer at the same time, the priority indicated above is always respected, then values of the wait states equal to 0, 1, 2, 5 clock periods have been provided to the slave. Also in this case, it has been seen that the presence of wait states degrades the performance of the system (Figure 4.17 and 4.18).

Start time *times are in ns	End time	Master	Operation	Address	Slave	Reg	Data	Transfer	Response	Size	Nonsec	#Wait states	Prot
375	500	master0	WRITE	0x02d9	slave0	reg1	0x00000070	NONSEQ	OKAY	BYTE	1	0	NON_CACHE
375	625	master1	WRITE	0x2fd9	slave0	reg23	0x000000df	NONSEQ	OKAY	BYTE	0	1	CACHEABLE
375	750	master2	WRITE	0x2cda	slave0	reg22	0x00005b8c	NONSEQ	OKAY	HALF_WORD	1	2	NON_CACHE
1000	1125	master2	READ	0x2678	slave0	reg19	0xffde8074	NONSEQ	OKAY	WORD	1	0	NON_CACHE
1375	1500	master0	WRITE	0x3fba	slave0	reg31	0x0000b69e	NONSEQ	OKAY	HALF_WORD	1	0	CACHEABLE
1375	1625	master1	WRITE	0x1916	slave0	reg12	0x0000c344	NONSEQ	OKAY	HALF_WORD	1	1	NON_CACHE
1375	1750	master2	READ	0x3514	slave0	reg26	0x000037d2	NONSEQ	OKAY	HALF_WORD	1	2	CACHEABLE
2000	2125	master0	WRITE	0x243c	slave0	reg18	0x00000088	NONSEQ	OKAY	BYTE	1	0	NON_CACHE
2000	2250	master2	WRITE	0x1142	slave0	reg8	0x00004389	NONSEQ	OKAY	HALF_WORD	1	1	NON_CACHE
2500	2625	master2	WRITE	0x3790	slave0	reg27	0x000000f2	NONSEQ	OKAY	BYTE	1	0	NON_CACHE
2875	3000	master1	WRITE	0x1c48	slave0	reg14	0x5b124c19	NONSEQ	OKAY	WORD	0	0	CACHEABLE
2875	3125	master2	READ	0x2151	slave0	reg16	0x000000d3	NONSEQ	OKAY	BYTE	1	1	NON_CACHE

Figure 4.15: Results: Detail on alternate and random transfers, without wait states by the slave, but only due to the competition between the masters.

Start time *times are in ns	End time	Master	Operation	Address	Slave	Reg	Data	Transfer	Response	Size	Nonsec	#Wait states	Prot
375	1125	master0	WRITE	0x02d9	slave0	reg1	0x00000070	NONSEQ	OKAY	BYTE	1	5	NON_CACHEABLE
375	1875	master1	WRITE	0x2fd9	slave0	reg23	0x000000df	NONSEQ	OKAY	BYTE	0	11	CACHEABLE NON
375	2625	master2	WRITE	0x2cda	slave0	reg22	0x00005b8c	NONSEQ	OKAY	HALF_WORD	1	17	NON_CACHEABLE
2875	3625	master2	READ	0x2678	slave0	reg19	0xffde8074	NONSEQ	OKAY	WORD	1	5	NON_CACHEABLE
3875	4625	master0	WRITE	0x3fba	slave0	reg31	0x0000b69e	NONSEQ	OKAY	HALF_WORD	1	5	CACHEABLE NON
3875	5375	master1	WRITE	0x1916	slave0	reg12	0x0000c344	NONSEQ	OKAY	HALF_WORD	1	11	NON_CACHEABLE
3875	6125	master2	READ	0x3514	slave0	reg26	0x000037d2	NONSEQ	OKAY	HALF_WORD	1	17	CACHEABLE NON
6375	7125	master0	WRITE	0x243c	slave0	reg18	0x00000088	NONSEQ	OKAY	BYTE	1	5	NON_CACHEABLE
6375	7875	master2	WRITE	0x1142	slave0	reg8	0x00004389	NONSEQ	OKAY	HALF_WORD	1	11	NON_CACHEABLE
8125	8875	master2	WRITE	0x3790	slave0	reg27	0x000000f2	NONSEQ	OKAY	BYTE	1	5	NON_CACHEABLE
9125	9875	master1	WRITE	0x1c48	slave0	reg14	0x5b124c19	NONSEQ	OKAY	WORD	0	5	CACHEABLE BUF
9125	10625	master2	READ	0x2151	slave0	reg16	0x000000d3	NONSEQ	OKAY	BYTE	1	11	NON_CACHEABLE

Figure 4.16: Results: Detail on alternate and random transfers, with 5 wait states by the slave summed to the competition between the masters.

4.3.4 Multiple masters (3) - Multiple slaves (8)

The last case study is the composition of the previously presented cases. Three masters with eight slaves have been connected and this represents the normal behavior of an integrated system. The analysis that was carried out is similar to that seen in the previous cases.
It was noted that:

- About the slaves:

  1. Which slaves were accessed most frequently by the three masters. In particular, it can be observed they have been slave 0 and slave 7 (Figure 4.19).
  2. Which were the most performed operations on the three slaves. In particular towards slave 7 more writes than reads have been made and the same for slave 0 (Figure 4.19).
  3. Which was the size of the transfers to the three slaves. In particular, more BYTE operations have been carried out than operations HALF WORD or WORD ones (Figure 4.20).

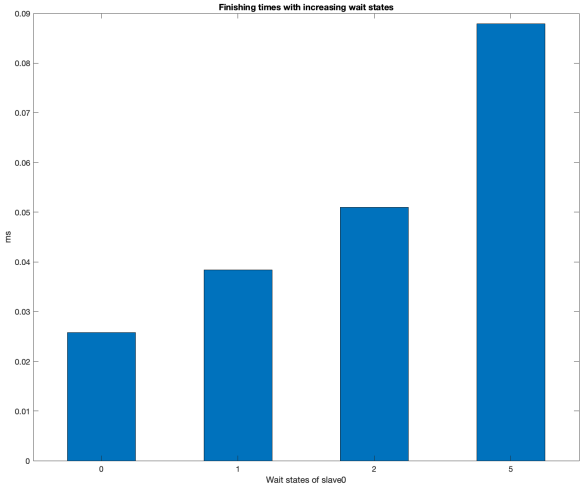


Figure 4.17: Results: Increasing number of wait states for the slave and resulting increasing ending time.

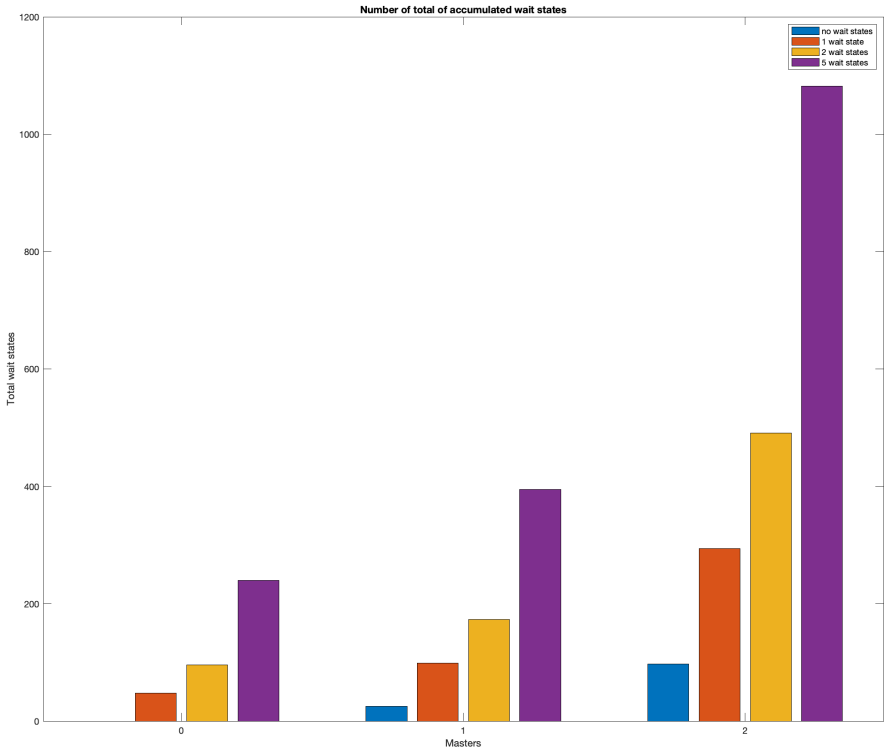


Figure 4.18: Results: Total wait states for the masters, increasing the number of wait states.

- About the masters:
  1. Which masters sent the most transfers. In particular, it was observed Master 2 (Figure 4.22).
  2. Which were the operations most carried out by the three masters. In the particular master 2 performed almost an equal number of reads and writes (Figure 4.22).

3. Which was the size of the transfers to the three masters. More BYTE operations were performed rather than HALF WORD and WORD (Figure 4.21).
4. It was then proceeded in increasing the number of wait states and they were compared cases in which:
   - The slaves did not have wait states.
   - The slaves had pre-assigned wait states:
     * Slave 0: 1 clock cycle
     * Slave 1-6: 2 clock cycles
     * Slave 7: 5 clock cycles
   - The slaves had pre-assigned wait states, but the slave accessed more frequently was improved:
     * Slave 7: 1 clock cycles
   - The slaves had random wait states.

It was possible to conclude that the master most influenced by the wait states, whether the slaves have them or not, is always master 2, as explained, previously because of the priorities assigned by the arbiter. From the figure 4.23, it is also evident how, even just by improving the most accessed slave (slave7) (orange bar), it leads to an overall improvement of the system.

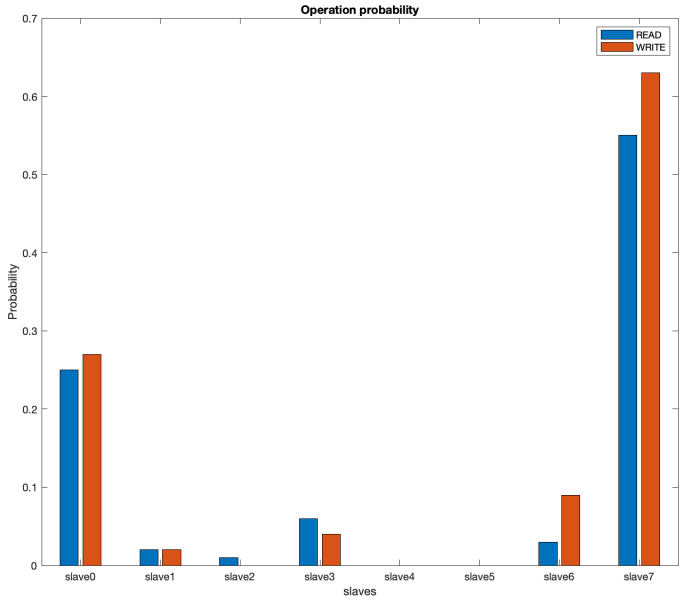


Figure 4.19: Results: Probability of operation type performed per slave.

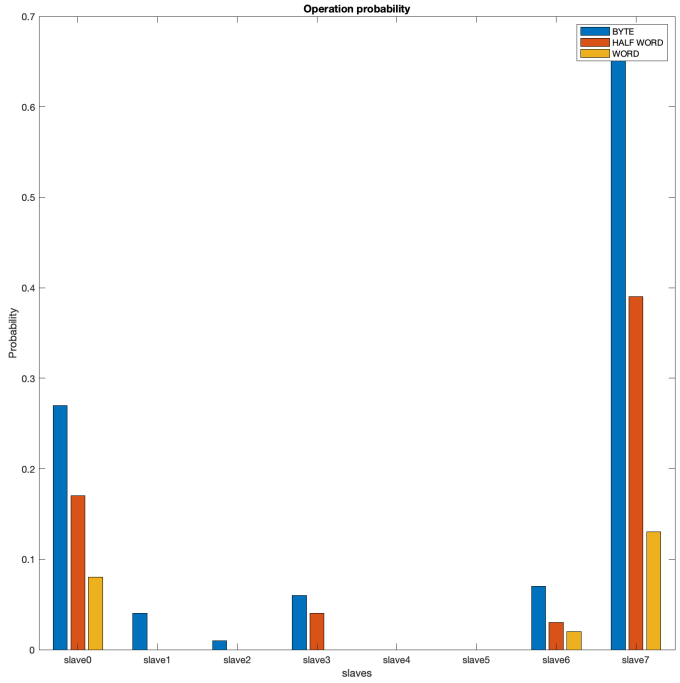


Figure 4.20: Results: Probability of operation size performed per slave.

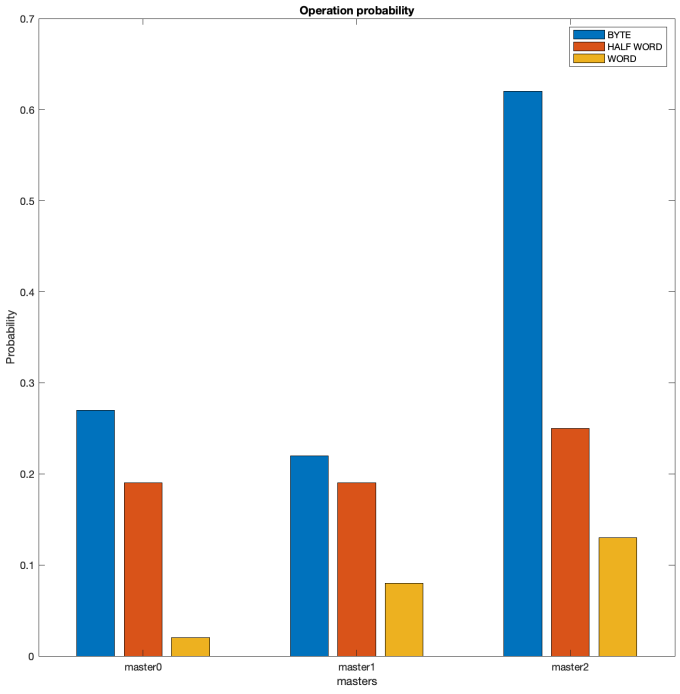


Figure 4.21: Results: Probability of operation size performed per master.

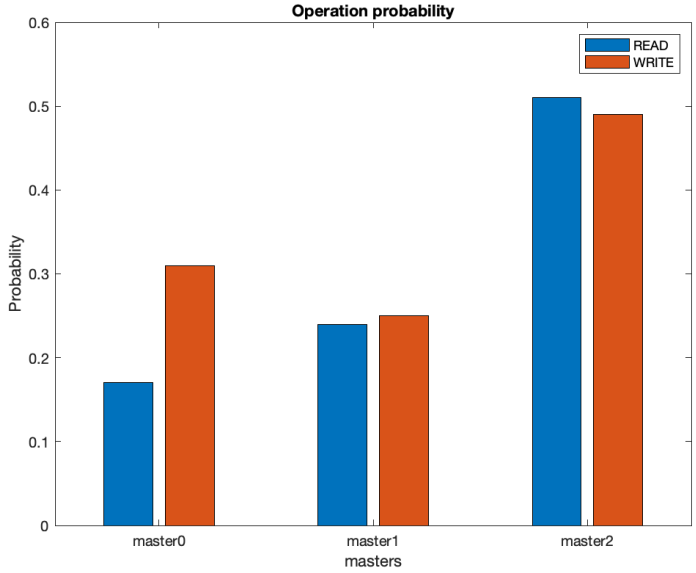


Figure 4.22: Results: Probability of operation type performed per master.

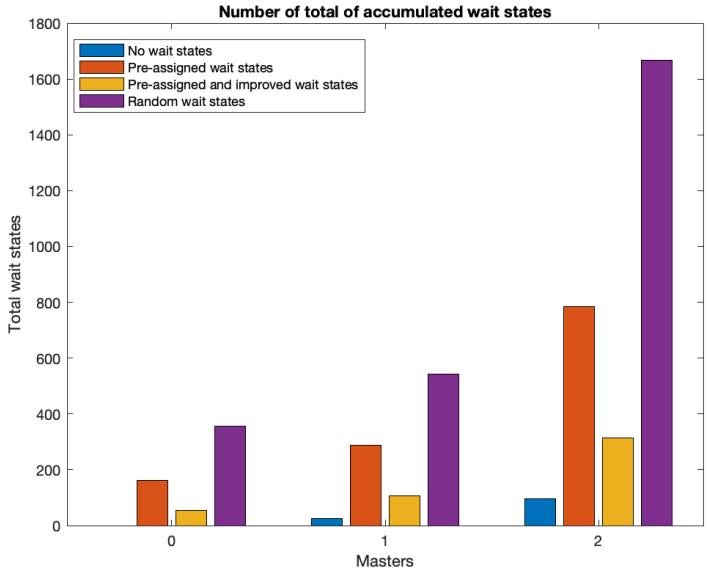


Figure 4.23: Results: Total wait states for the masters, for different wait states scenarios.

CHAPTER 5

Conclusions

In this thesis work, it was tried to give to the reader an overview of what means to verify and what has been verified. In the second chapter, it was proceeded with the presentation of the component to be verified: Advanced High-performance Bus (AHB), on-chip standard protocol, which is part of AMBA (Advanced Microcontroller Bus Architecture).
Its importance on the market and its major features have been highlighted as the possibility of offering features such as: single clock edge operations, burst transfers, non-tristate implementation, broad data bus configurations (up to 1024 bits), analyzing how transfers take place and what signals have been adopted.
After that, an overview of the alternatives on the market, proprietary and otherwise, was discussed by comparing the protocol with IBM CoreConnect, STMicroelectronics STBus, and Silicore Corporation Wishbone.
Then, it was explained what the verification of a design is, how it fits into the development cycle of an integrated system and what methodologies are usually adopted. Universal Verification Methodology (UVM) and SystemVerilog language have been detailed and used for performing the verification. Finally, an overview of the tools supporting the functional verification was provided, as one of them was used.
In the third chapter, the functional verification took shape, with the creation of a verification plan. It was explained what it was and how it generally happened for the design of an integrated system. Since AHB is an IP core, some differences have been highlighted in the course of the discussion. The verification plan, the verification environment and an example of the implementation of an attribute from the verification plan were presented, with the analysis of its behavior through waveforms and through the collection of bins within a covergroup.
As already mentioned in the course of the discussion, the verification plan has been completely defined for all AHB features, but only a subset of them has been implemented. Thanks to the verification of these features, a verification component was obtained able to estimate that the design of AHB is more robust, complete and more accurate. Furthermore, it was possible to estimate that both masters and slaves can work correctly.
In the final part of the thesis, a verification of the behavior on the bus was carried out, through the use of a log. It was possible to use it to analyze several factors such as:

- The actual behavior of the arbiter and its own policy and suggesting a fairer policy.
- The actual behavior of the slaves, with focus on wait states, understanding which ones were accessed most frequently and showing how much they affect the overall completition of the transfers.

- The actual behavior of the masters by understanding how they act in competition and some features about the most frequent requests (type of request, size of the request, etc.)
- The advantage of pipelining compared to sequential transfers.

Future developments could include the completion of the implementation of the verification plan, with the creation of more sequences/stimuli and the creation of even more use cases to show the actual use of the bus within the log. In this regard, it could be thought of making use of machine learning in order to identify patterns within the log and dependencies between the various parameters, expanding the observations already made on the influence of wait states on the transfers.

Appendix A: Verification plan attributes

This appendix presents the elements of the verification plan as defined in the chapter 3, to proceed with the verification. Given the complexity of the IP core, it was preferred to follow a subdivision of the attributes into lists and sublists which reflects the subdivision that occurs in the protocol specification index. Some empty sublists can be found mapped to chapters with only definitions and they can be potentially used for future refactories of the verification plan. It should be noted that a complete verification plan for the AHB protocol has been produced. Nevertheless, not all the features have been implemented, but only a subset supported by the integrated circuits of competence of the team in which I interned. Within the tool used, a reference to their specification document definition and references to the implemented code is practically provided. Thanks to these references, it was possible to collect metrics and make evaluations.

1. Attributes

1.1 Transfers

1.1.1 Basic transfer

1.1.1.1 Cover basic read transfer with two wait states.

1.1.1.2 Cover basic write transfer with one wait state.

1.1.1.3 Cover basic multiple transfers.

1.1.1.4 Cover basic read and write tranfers.

1.1.2 Burst transfer

1.1.2.1 Check that the control signals are identical to the previous (or first) transfer.

1.1.2.2 Check that the actual address is the previous + offset give by HSIZE.

1.1.2.3 Cover basic burst transfer (4-beats transfer).

1.1.2.4 Covers all the HBURSTs cases.

1.1.2.5 Cover single transfer of burst.

1.1.2.6 Check that all the addresses are aligned.

1.1.2.7 Check that the burst transfer doesn't finish with a BUSY transfer (if incrementing or wrapping) but with a SEQ.

1.1.2.8 Check that after a single burst there is an IDLE or a NONSEQ but not a BUSY transfer.

1.1.2.9 Four-beat wrapping burst, WRAP4.

1.1.2.10 Cover Four-beat incrementing burst, INCR4 example.

1.1.2.11 Cover Eight-beat wrapping burst, WRAP8 example.

1.1.2.12 Cover Eight-beat incrementing burst, INCR8 example.

1.1.2.13 Cover Undefined length bursts, INCR example.

1.1.2.14 Incrementing burst

1.1.2.14.1 Check that the address doesn't cross a 1KB boundary.

1.1.2.15 Error in burst

1.1.2.15.1 Check that HTRANS is changed to IDLE during the two cycle-error response.

1.1.3 Locked transfer

1.1.3.1 Cover simple locked transfer + optional IDLE transfer afterwards.

1.1.3.2 Check that all the transfers are to the same address region.

1.1.3.3 Check that a slave has the HMASTLOCK signal if it is accessed by more than one master.

1.1.4 Waited transfer

1.1.4.1 Check that when the slave is requesting wait states, the master must not change the transfer type (except for IDLE and BUSY).

1.1.4.2 IDLE transfer

1.1.4.2.1 Cover a waited transfer for a SINGLE burst, with a transfer type change from IDLE to NONSEQ.

1.1.4.2.2 Cover IDLE transfer when address changes during a waited transfer.

1.1.4.2.3 Check that when HTRANS changes to NONSEQ, the master keeps HTRANS constant, until HREADY is HIGH.

1.1.4.3 Busy transfer, fixed length burst

1.1.4.3.1 Cover a waited transfer in a fixed length burst, with a transfer type change from BUSY to SEQ.

1.1.4.3.2 Check that when HTRANS changes to SEQ, the master keeps HTRANS constant, until HREADY is HIGH.

1.1.4.3.3 Check that HTRANS is not BUSY when it is a single burst.

1.1.4.4 Busy transfer, undefined length burst

1.1.4.4.1 Cover a waited transfer during an undefined length burst, with a transfer type change from BUSY to NONSEQ.

1.1.4.4.2 Check that HREADY is LOW when HTRANS changes from BUSY to anything else.

1.1.4.5 After an ERROR response

1.1.4.5.1 Cover a waited transfer, with the address changing following an ERROR response from the slave.

1.1.5 Protection Control

1.1.5.1 Cover all the combinations of HPROT signals.

1.1.5.2 Check that HPROT signals are kept as long as an address signal. In case of a burst, they should be constant.

1.1.6 Memory types

1.1.6.1 Cover combinations of HPROT[3:0].

1.1.6.2 Cover combinations of HPROT[6:4].

1.1.6.3 Data or Instruction

1.1.6.3.1 Cover that when Data access, HPROT[0] is HIGH. When Instruction Fetch, HPROT[0] is LOW.

1.1.6.4 Unprivileged or privileged

1.1.6.4.1 Cover that when Privileged access, HPROT[1] is HIGH. When Unprivileged access, HPROT[1] is LOW.

1.1.6.5 Memory type

1.1.6.5.1 Cover that all the values of HPROT[6:2] which specify all the possible types of memories.

1.1.6.6 Device memory requirements

1.1.6.6.1 Check that read data is obtained from the final destination.

1.1.6.6.2 Check that transfers are not split into multiple transfers or merged with other transfers.

1.1.6.6.3 Check that reads are not prefetched or performed speculatively.

1.1.6.6.4 Check that writes are not merged.

1.1.6.6.5 Check that all read and write transfers from the same master to the same slave remain ordered.

1.1.6.6.6 Check that the size of the transfer, as indicated by HSIZE, is not changed.

1.1.6.6.7 Check that the total number of NONSEQ and SEQ transfers in the original burst is the same as the total number of NONSEQ and SEQ in the resultant small bursts.

1.1.6.6.8 Check that the only change permitted to HPROT is to convert a transfer from Bufferable to Non-bufferable.

1.1.6.6.9 Additionally, for Device-nE: Check that write response is obtained from the final destination.

1.1.6.6.10 Additionally, for Device-E: Cover that the response could be obtained from an intermediate point and that the transfers should be measurable in time and observable.

1.1.6.7 Normal memory requirements

1.1.6.7.1 Check that read and write transfers from the same master to addresses that overlap remain ordered.

1.1.6.7.2 Check that, for Shareable transactions, the response is only given when the transfer is visible to all.

1.1.6.7.3 For Normal Non-cacheable memory and : Check that write transfers are made visible at the final destination in a timely manner.

1.1.6.7.4 Check that read data is obtained either from: The final destination or a write transfer that is progressing to its final destination.

1.1.6.7.5 If read data is obtained from a write transfer: Check that it is obtained from the most recent version of the write and that the data is not cached to service a later read.

1.1.6.7.6 Check that reads do not cache the data obtained for later use.

1.1.6.7.7 Cover the behaviours of normal memories.

1.1.6.7.8 Additionally, for Write-through: Cover additional potential behaviours.

1.1.6.7.9 Additionally, for Write-back: Cover additional potential behaviours.

1.1.6.7.10 For Write-through: Check that write transactions are made visible at the final destination in a timely manner.

1.1.6.7.11 For Write-through: Check that a cache lookup is present for read and write transfers.

1.1.6.8 Allocate attribute

1.1.6.8.1 For Write-through and Write-back memories: Cover HPROT[5] when HIGH, transfer is in cache. When LOW it is not.

1.1.6.9 Legacy considerations

1.1.6.9.1 Cover the possible translations from HPROT[3:0] to HPROT[6:0] for legacy components.

1.1.7 Secure transfers

1.1.7.1 Check that HNONSEC is LOW when we want a se cure transfer and HIGH when we want a Non-Secure Transfer.

1.1.7.2 Check that HNONSEC follows the same constraints of an address (like HADDR).

1.2 Bus Interconnection

1.2.1 Interconnect

1.2.2 Address decoding

1.2.2.1 Default slave

1.2.2.1.1 Check that If a NONSEQUENTIAL or SEQUENTIAL transfer is attempted to a nonexistent address location then the default slave provides an ERROR response.

1.2.2.1.2 Check that an IDLE or BUSY transfers to nonexistent locations result in a zero wait state OKAY response.

1.2.2.2 Multiple slave select

1.2.2.2.1 Check that the minimum address space that can be allocated to a logical interface is at least 1KB.

1.3 Slave Response Signaling

1.3.1 Slave transfer responses

1.3.1.1 Cover the combinations of HRESP and HREADYOUT.

1.3.1.2 Transfer done

1.3.1.2.1 Check that a successful completed transfer is signaled when HREADY is HIGH and HRESP is OKAY.

1.3.1.3 Transfer pending

1.3.1.3.1 Check that when a slave inserts a number of wait states prior to completing the response, it must drive HRESP to OKAY.

1.3.1.3.2 Cover a transfer with wait states into the transfer.

1.3.1.4 ERROR response

1.3.1.4.1 Cover a transfer with an ERROR response.

1.4 Data buses

1.4.1 Data buses

1.4.1.1 HWDATA

1.4.1.1.1 Check that if the transfer is extended then the master must hold the data valid until the transfer completes, as indicated by HREADY HIGH.

1.4.1.1.2 For transfers that are narrower than the width of the bus, for example a 16-bit transfer on a 32-bit bus, check that the master drives the appropriate byte lanes.

1.4.1.2 HRDATA

1.4.1.2.1 Check that if the slave extends the read transfer by holding HREADY LOW, then the slave only has to provide valid data in the final cycle of the transfer.

1.4.1.2.2 For transfers that are narrower than the width of the bus, check that the slave is provides valid data on the active byte lanes.

1.4.1.2.3 Check that a slave provides valid data only when a transfer completes with an OKAY response.

1.4.2 Endianness

1.4.2.1 Little endian

1.4.2.1.1 When a little-endian component accesses a byte, check that the equation is applied.

1.4.2.2 Byte-invariant big-endian

1.4.2.2.1 When a byte-invariant big-endian component accesses a byte, check that the equation is applied.

1.4.2.2.2 When larger byte-invariant big-endian transfers occur, check that this is done as specified.

1.4.2.3 Word-invariant big-endian

1.4.2.3.1 When a word-invariant big-endian component accesses a byte, check that the data bus bits used follow the equation.

1.4.2.3.2 For halfword and word transfers using word-invariant big-endian, check that data is transferred as specified.

1.4.2.3.3 For transfers larger than a word using word-invariant big-endian, check that data is split into word size blocks and they follow the two specifications.

1.4.2.3.4 Cover Active byte lanes for a 32-bit little-endian data bus.

1.4.2.3.5 Cover Active byte lanes for a 32-bit byte-invariant big-endian data bus.

1.4.2.3.6 Cover Active byte lanes for a 32-bit word-invariant big-endian data bus.

1.4.2.4 Byte invariance

1.4.2.4.1 Cover an example of a data structure that requires byte-invariant access.

1.4.3 Data bus width

1.4.3.1 Implementing a narrow slave on a wide bus

1.4.3.2 Implementing a wide slave on a narrow bus

1.4.3.3 Implementing a master on a wide bus

1.5 Clock and Reset

1.5.1 Clock and reset requirements

1.5.1.1 Clock

1.5.1.1.1 Check that all input signals are sampled on the rising edge of HCLK.

1.5.1.1.2 Check that all output signal changes occurs after the rising edge of HCLK.

1.5.1.1.3 Check that signals that are described as being stable, remain stable (according to Stable_Between_Clock property).

1.5.1.2 Reset

1.5.1.2.1 Check that the reset signal, HRESETn, is the only active LOW signal in the protocol and is the primary reset for all the bus elements.

1.5.1.2.2 Check that the reset can be asserted asynchronously, but is deasserted synchronously after the rising edge of HCLK.

1.5.1.2.3 Check that a component defines a minimum number of cycles for which the reset signal must be asserted to ensure that the component and the outputs are reset.

1.5.1.2.4 Check that during reset all masters ensure the address and control signals are at valid levels and that HTRANS[1:0] indicates IDLE.

1.5.1.2.5 Check that during reset all slaves must ensure that HREADYOUT is HIGH.

1.6 Exclusive Transfers

1.6.1 Introduction

1.6.2 Exclusive Access Monitor

1.6.3 Exclusive access signaling

1.6.3.1 Check that HEXCL follows the same constraints of HADDR.

1.6.3.2 Check that HMASTER[m:0] follows the same constraints of HADDR.

1.6.3.3 Check that HEXOKAY Exclusive Okay is high when a successful exclusive transfer is performed otherwise it is low.

1.6.3.4 Response signaling

1.6.3.4.1 Cover assertion and deassertion cases of HEXOKAY signal.

1.6.3.4.2 Check that HEXOKAY is asserted in the same cycle as HREADY is asserted.

1.6.3.4.3 Check that HEXOKAY is asserted in the same cycle as HRESP is asserted.

1.6.4 Exclusive Transfer restrictions

1.6.4.1 Check that an exclusive transfer has a single data transfer.

1.6.4.2 Check that an exclusive transfer is indicated as burst type SINGLE or burst type INCR.

1.6.4.3 Check that an exclusive transfer does not include a BUSY transfer.

1.6.4.4 Check that an exclusive transfer address is aligned to data size as indicated by HSIZE.

1.6.4.5 Cover that the value of the HPROT signals must guarantee that the Exclusive Access Monitor has visibility.

1.6.4.6 Check that for an exclusive read-write transfer the indicated signals are the same.

1.6.4.7 Cover that when a master issues an Exclusive Write transfer fails deasserting HEXOKAY, when not preceded by an exclusive read transfer.

1.6.4.8 Check that a master does not have two Exclusive Transfers outstanding at the same point in time.

1.7 Atomicity

1.7.1 Single-copy atomicity size

1.7.1.1 Cover that a transfer is updated atomically for a number of data bytes equal to the single-copy atomicity size.

1.7.1.2 When a write transfer updates a memory location, cover that an observer sees either: no update to the location or an update to at least a single-copy atomicity size amount of data.

1.7.1.3 Check that an observer cannot see some data bytes updated within the single-copy atomic of someone else.

1.7.1.4 Check that byte strobes associated with a transfer do not affect the single-copy atomicity size.

1.7.1.5 Check that an atomic update happens for the complete size at the same time and not for portions of it in time.

1.7.2 Multi-copy atomicity

1.7.2.1 Check that the system works in multi-copy atomicity if the Multi_Copy_Atomicity property is set to True.

1.7.2.2 Cover a multi-copy atomicity behavior

1.8 User Signaling

1.8.1 User signal description

1.8.1.1 Check that the user signals for byte y are located at HWUSER[] and HRUSER[].

1.8.1.2 Check that the location of the user bits for a data channel is defined as indicated.

1.8.1.3 Check that the number of user bits are the recommended for data channel user signals.

1.8.1.4 Check that HAUSER, HWUSER and HRUSER signals have the same timing and validity requirements as the associated channel.

1.8.2 User signal interconnect recommendations

1.8.2.1 Cover single transfer conversion to multiple transfers.

1.8.2.2 Cover multiple transfers conversion to single transfer.

2 Interfaces

2.1 Global signals

2.1.1 HCLK

2.1.2 HRESETn

2.2 Master signals

2.2.1 HADDR[31:0]

2.2.1.1 Address lasts for a single HCLK cycle unless its extended by the previous bus transfer

2.2.2 HBURST[2:0]

2.2.3 HMASTLOCK

2.2.4 HPROT[3:0]

2.2.5 HPROT[6:4]

2.2.6 HSIZE[2:0]

2.2.6.1 Check that HSIZE is not larger than the databus width

2.2.6.2 Check that HSIZE doesn't assume values 6 and 7

2.2.6.3 Check that HSIZE is kept constant during a burst transfer

2.2.7 HNONSEC

2.2.8 HEXCL

2.2.9 HMASTER[3:0]

2.2.10 HTRANS[1:0]

2.2.10.1 When HTRANS is BUSY, check that the slave provides OKAY and the transfer is ignored by slave

2.2.10.2 Check that HTRANS is NONSEQ when it is a single transfer or a first transfer of a burst

2.2.10.3 Check that HTRANS is SEQ when it is a burst transfer

2.2.10.4 Check that when HTRANS is IDLE, no data is transferred

2.2.10.5 When HTRANS is IDLE, check that the slave provides OKAY and the transfer is ignored by slave

2.2.11 HWDATA[31:0] (can be larger)

2.2.11.1 Check that HWDATA lasts at least 1 clk cycle

2.2.11.2 Check that HWDATA contains a data to write when HWRITE is HIGH

2.2.12 HWRITE

2.2.12.1 Check that HWRITE is HIGH when writing

2.2.12.2 Check that HWRITE is LOW when reading

2.3 Slave signals

2.3.1 HRDATA[31:0] (can be larger)

2.3.1.1 Check that HRDATA contains the right data to be read, when a reading is performed

2.3.2 HREADYOUT

2.3.2.1 Check that the target slave drives HREADYOUT during the data phase of a transfer

2.3.3 HRESP

2.3.3.1 Reference When HTRANS is BUSY, check that the slave provides OKAY and the transfer is ignored by slave,When HTRANS is IDLE, check that the slave provides OKAY and the transfer is ignored by slave

2.3.3.1.1 When HTRANS is BUSY, check that the slave provides OKAY and the transfer is ignored by slave

2.3.3.1.2 When HTRANS is IDLE, check that the slave provides OKAY and the transfer is ignored by slave

2.3.4 HEXOKAY

2.4 Decoder signals

2.4.1 HSELx

2.5 Multiplexor signals

2.5.1 HRDATA[31:0]

2.5.2 HREADY

2.5.2.1 Check that HREADY is long enough to complete a full transfer

2.5.2.2 Check that HREADY is coherent to HREADYOUT signals from all slaves

2.5.3 HRESP

Bibliography

[1] ARM Holdings, *AMBA Overview* Available: arm.com, https://developer.arm.com/architectures/system-architectures/amba. [Accessed: 01 Jul. 2021].

[2] Wikipedia.org, *Advanced Microcontroller Bus Architecture* Available: Wikipedia.org, https://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture. [Accessed: 01 Jul. 2021].

[3] V.G, Kiran. (2015). *Modeling of FPGA and memory using AMBA AHB for DSP applications*, IJIRT, Volume 2 Issue 1

[4] ARM Holdings, *AXI protocol overview* Available: arm.com, https://developer.arm.com/documentation/102202/0200. [Accessed: 05 Jul. 2021].

[5] ARM Holdings, *ARM AMBA 5 AHB Protocol Specification* Available: arm.com, https://developer.arm.com/documentation/ihi0033/latest. [Accessed: 05 Jul. 2021].

[6] ARM Holdings, *Multi-layer AHB V2.0 - Technical Overview* Available: arm.com, https://developer.arm.com/documentation/dvi0045/b. [Accessed: 05 Jul. 2021].

[7] Cadence Design Systems, *What Is New in the Latest AMBA 5 ACE, AXI and AHB Protocol Specification Updates?* Available: community.cadence.com, https://community.cadence.com/cadence_blogs_8/b/fv/posts/what-is-new-in-the-latest-amba-5-ace-axi-and-ahb-protocol-specification-updates. [Accessed: 06 Jul. 2021].

[8] M. Mitić, M. Stojčev (2006), *A Survey of Three System-on-Chip Buses: AMBA, CoreConnect and Wishbone*, ICEST 2006.

[9] Alberto Macii, *SOC Slides: Embedded Busses* Available: polito.it. [Accessed: 07 Jul. 2021].

[10] Xilinx, *CoreConnect Technology* Available: xilinx.com, https://web.archive.org/web/20040616075914/http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=dr_pcentral_coreconnect. [Accessed: 07 Jul. 2021].

[11] Xilinx, *CoreConnect* Available: wikipedia.org, https://en.wikipedia.org/wiki/CoreConnect. [Accessed: 07 Jul. 2021].

[12] Xinping Zhu, *CoreConnect Model: A cycle-accurate IBM CoreConnect model using Operation State Machines (OSMs)* Available: berkeley.edu, https://ptolemy.berkeley.edu/projects/embedded/mescal/forum/7.html. [Accessed: 07 Jul. 2021].

[13] IBM, *32-bit Processor Local Bus - Architecture Specifications - Version 2.9* Available: berkeley.edu, https://ptolemy.berkeley.edu/projects/embedded/mescal/forum/7/coreconnect_32bit.pdf. [Accessed: 07 Jul. 2021].

[14] Dr. Gul N. Khan, Ryerson University, *SoC Bus Interconnect Structures* Available: ryerson.ca, https://www.ee.ryerson.ca/~courses/coe838/lectures/On-Chip-Bus-Interconnections.pdf. [Accessed: 07 Jul. 2021].

[15] STMicroelectronics, *UM0484: STBus communication system concepts and definitions* Available: st.com, https://www.st.com/resource/en/user_manual/cd00176920-stbus-communication-system-concepts-and-definitions-stmicroelectronics.pdf. [Accessed: 07 Jul. 2021].

[16] Olivier Cauvet, STMicroelectronics, *STBus complex interconnect design and verification for a HDTV SoC* Available: design-reuse.com, https://www.design-reuse.com/articles/16092/stbus-complex-interconnect-design-and-verification-for-a-hdtv-soc.html. [Accessed: 07 Jul. 2021].

[17] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini and R. Zafalon, *Analyzing on-chip communication in a MPSoC environment*, Proceedings Design, Automation and Test in Europe Conference and Exhibition, 2004, pp. 752-757 Vol.2

[18] Rudolf Usselmann, *OpenCores - SoC Bus Review 1.0*, Available: opencores.org, https://cdn.opencores.org/downloads/soc_bus_comparison.pdf. [Accessed: 08 Jul. 2021].

[19] M. Sharma, D. Kumar (2012), *Wishbone Bus Architecture - A survey and comparison*, International Journal of VLSI design & Communication Systems (VLSICS) Vol.3, No.2.

[20] W. K. Lam (2008), *Hardware Design Verification: Simulation and Formal Method-Based Approaches*, Prentice Hall, ISBN: 978-0137010929

[21] Wikipedia.org, *SystemVerilog* Available: Wikipedia.org, https://en.wikipedia.org/wiki/SystemVerilog. [Accessed: 13 Jul. 2021].

[22] IEEE, *1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language* Available: ieee.org, https://ieeexplore.ieee.org/document/8299595. [Accessed: 13 Jul. 2021].

[23] Harry Foster, Siemens, *The 2020 Wilson Research Group Functional Verification Study - IC/ASIC Language and Library Adoption Trend* Available: siemens.com, https://blogs.sw.siemens.com/verificationhorizons/2021/01/20/part-10-the-2020-wilson-research-group-functional-verification-study/. [Accessed: 13 Jul. 2021].

[24] Accellera, *Universal Verification Methodology (UVM) 1.2 User's Guide* Available: accellera.org, https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf. [Accessed: 14 Jul. 2021].

[25] E. Sanchez, *Silicon Systems For Wireless Lan (Advanced Series In Electrical And Computer Engineering Book 22)*, World Scientific Publishing Company, 2020, ISBN: 978-9811210716

[26] Wikipedia, *Universal Verification Methodology* Available: wikipedia.org, https://en.wikipedia.org/wiki/Universal_Verification_Methodology. [Accessed: 14 Jul. 2021].

[27] IEEE, *IEEE 1800.2-2017 - IEEE Standard for Universal Verification Methodology Language Reference Manual* Available: ieee.org, https://standards.ieee.org/standard/1800_2-2017.html. [Accessed: 14 Jul. 2021].

[28] Ray Salemi, *The UVM Primer: A Step-By-Step Introduction to the Universal Verification Methodology*, Boston Light Press, 2013, ISBN: 9780974164939

[29] Wikipedia, *List of HDL simulators* Available: wikipedia.org, https://en.wikipedia.org/wiki/List_of_HDL_simulators. [Accessed: 16 Jul. 2021].

[30] A. Piziali, *Functional Verification Coverage Measurement and Analysis*, Springer, 2004, ISBN: 978-1402080258

[31] Siemens, *Assertion coverage*, Available: verificationacademy.com, https://verificationacademy.com/cookbook/glossary/assertion-coverage. [Accessed: 20 Jul. 2021].

[32] Doulos, *Coverage-Driven Verification Methodology*, Available: doulous.com, https://www.doulos.com/httpswwwdouloscomknowhow/systemverilog/uvm/easier-uvm/easier-uvm-deeper-explanations/coverage-driven-verification-methodology/. [Accessed: 22 Jul. 2021].

[33] Siemens, *Fun with UVM Sequences - Coding and Debugging*, Available: verificationacademy.com, https://verificationacademy.com/verification-horizons/june-2019-volume-15-issue-2/fun-with-uvm-sequences-coding-and-debugging. [Accessed: 25 Jul. 2021].

[34] T. Alsop, *Integrated circuits semiconductor market size worldwide from 2009 to 2022* Available: statista.com, https://www.statista.com/statistics/519456/forecast-of-worldwide-semiconductor-sales-of-integrated-circuits/. [Accessed: 02 Aug. 2021].

[35] H. Foster, Siemens, *IC/ASIC Design Trends* Available: siemens.com, https://blogs.sw.siemens.com/verificationhorizons/2020/12/22/part-7-the-2020-wilson-research-group-functional-verification-study/. [Accessed: 02 Aug. 2021].

[36] H. Foster, Siemens, *Redefining Verification Performance* Available: siemens.com, https://blogs.sw.siemens.com/verificationhorizons/2010/08/08/redefining-verification-performance-part-2. [Accessed: 02 Aug. 2021].