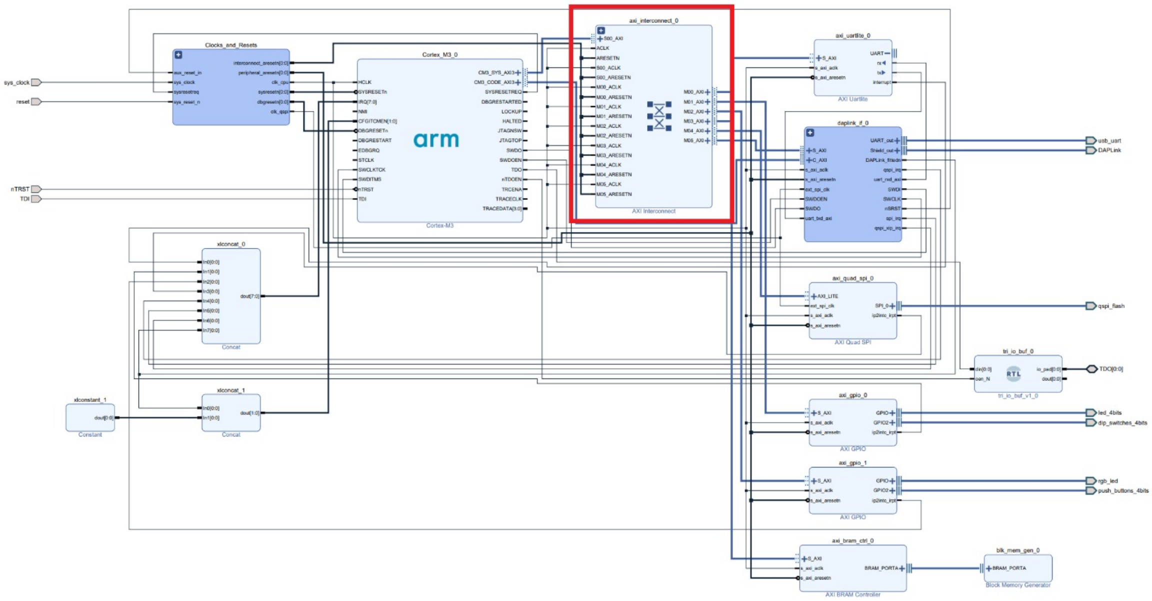
Embedded Systems AA [2020-21]
Colombi, Corbetta, Consonni



AXI Project

Prof. William Fornaciari
PhD Davide Zoni
PhD Std Andrea Galimberti

Deliverable:	Report
Title:	Project Report
Authors:	Marco Colombi 10631973 marco3.colombi@mail.polimi.it, Giorgio Corbetta 10570080 giorgio1.corbetta@mail.polimi.it, Cesare Consonni 10476810 cesare.consonni@mail.polimi.it
Version:	1.0
Date:	23-January-2021
Download page:	CCC-AXI git repo

Contents

1 Introduction

1.1 AXI Protocol

The AMBA AXI protocol supports high-performance, high-frequency system designs for communication between multi masters and multi slaves components. **AXI4** is widely adopted, providing **benefits** to **productivity** (standardization simplifies the work of developers), **flexibility** (there are slightly different protocols, eachone of them with their peculiarity), and **availability** (it's an industrial standard, and there is a world wide community that uses and support it).

AXI4-Lite, the procol that we studied, is a **subset of AXI4** for communication with simpler control register style interfaces within components.

Some **key features** of the AXI4-Lite protocol are: **separate address/control and data phases**, support for unaligned data transfers, using byte strobes, **separate read and write data channels**, all **transactions** are of burst **length 1**, all data accesses are non-modifiable, non-bufferable and use the full **width of the data bus** (the supported busses are the ones with width of **32-bit** (in our case) or 64-bit), exclusive accesses are not supported.

One of the features of **AXI4-Lite** is the **Handshake protocol**, that is done **for each payload exchanged and address line** and it ensures the reading of the right values. The signals involved are **valid**, asserted by the sender, when the data is stable and available, and **ready**, asserted by the reciever, when it is able to recieve the information. For more detailed explaination, please see [1, Section-A3.2.1].

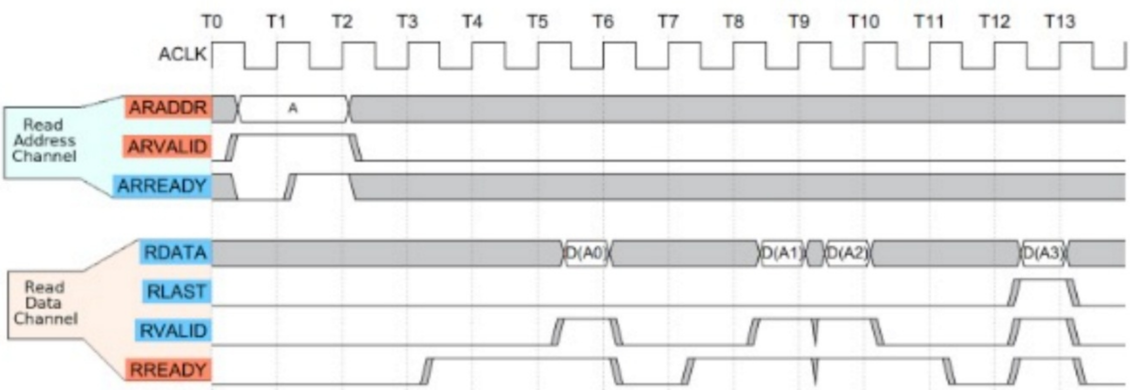


Figure 1: AXI4 (**non lite**) read

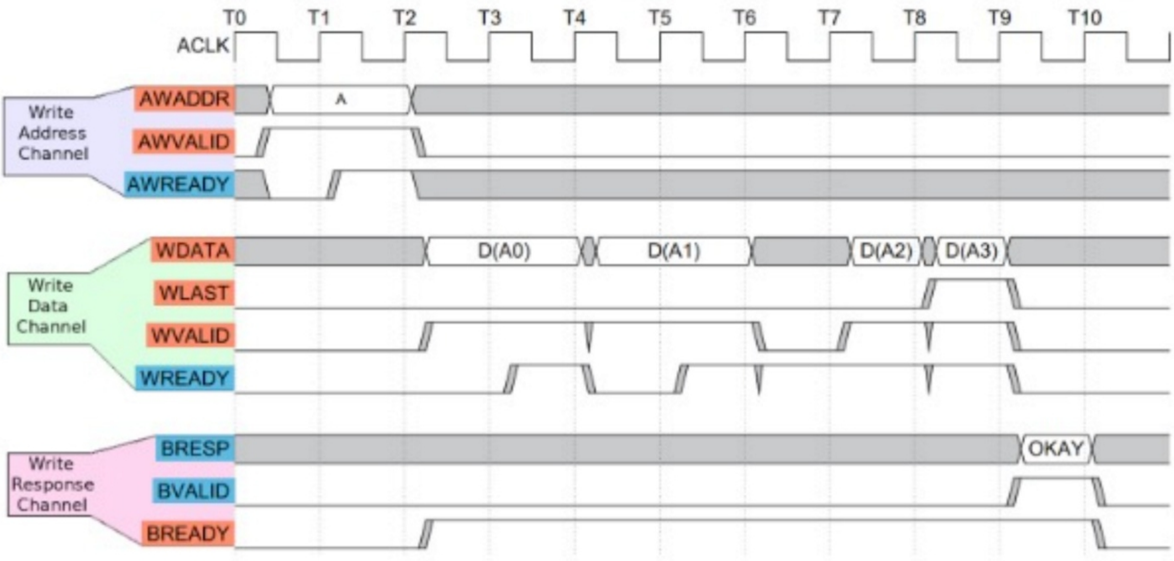


Figure 2: AXI4 (**non lite**) write

WRITE SIGNALS		
AW group		
AWADDR	[31:0]	//where the master want to write
AWVALID	[0:0]	//the address line is valid
AWPROT	[2:0]	//access permissions
AWREADY	[0:0]	//the slave is ready to recieve the address
W group		
WDATA	[31:0]	//what the master want to write
WSTRB	[3:0]	//which bytes of the WDATA are meaningful
WVALID	[0:0]	//the data line is valid
WREADY	[0:0]	//the slave is ready to recieve the data
B group		
BREADY	[0:0]	//the master is ready to recieve the response
BRESP	[1:0]	//slave's response about the operation
BVALID	[0:0]	//the response line is valid
READ SIGNALS		
AR group		
ARADDR	[31:0]	// where the master want to read
ARVALID	[0:0]	//the address line is valid
ARPROT	[2:0]	//access permissions
ARREADY	[0:0]	//the slave is ready to recieve the address
R group		
RREADY	[0:0]	//the master is ready to recieve the data
RDATA	[31:0]	//the data requested by the master
RVALID	[0:0]	//the data line is valid
RRESP	[1:0]	//slave's response about the operation

MASTER controlled
SLAVE controlled

1.2 Project Scope

The scope of the project is to understand **how works the AXI4-Lite** communication protocol, and **design by ourselves** an "AXI interconnect" **component** to **replace** the real one inside the *Arm Cortex-M3 DesignStart FPGA-Xilinx edition* and **observe** its **behaviour** and the **differences** between them.

In our case there is only one master with multiple slaves (we don't need arbitrer) and there is no need for clock gating (because the clock is shared between all components). For more informations, see: [5] [1]

2 Our Design

Here we will explain how we developed our AXI interconnect, first explaining how it works as a stand-alone component, and then how we inserted it into the Processor.

2.1 Block Design

In the figure 3 is reported the Block Diagram of the AXI interconnect we developed.
What we have implemented is essentially the **crossbar**-related part inside the **original AXI interconnection**: we specifically tailored our component to fit in the ARM Design Start, so **single-master multiple-slaves architecture**, but with few changes the number of slaves can be increased/decreased.
The architecture is divided into **two independent** mirrored **parts**, one for the **write** operation and one for the **read** operation, that are differentiated only by the inner logic of the two FSMs and by the number of wires that they manage. Their core is the **MUX-DEMUX-DECODER** sub-units for the **routing** of the signals and a supporting **FSM for** the handling of **handshakes**.
The MUX / *DEMUX* simply gathers the messages from slaves to master / *from the master to the slaves* and rally them for the master / *redirect them to the correct slave* under the explicit command of the **DECODER** that **maps the components** upon their **addresses**.
The FSM is in charge to follow the master in the write/read operations: it tracks the sequence of handshake (address -> data for read and address -> data -> response for write) keeping frozen the MUX / DEMUX states to preserve the connection of the interested parts of the system.
There are 2 types of error that we had considered: the ones raised by the slaves, which merely pass through the inteconnection and the ones raised if the address of the required slave is not mapped. The **FSM** is also a **supervisor** in case of the **errors** in the **addressing**: if the DECODER detects that the address provided by the master is not valid, the FSM will commute to an **error-handling state** and allow the AXI to show an appropriate behaviour.

2.2 Integration and Testing

We preferred to use the same enviroment of **ARM Design Start** respect to the implementation of a fake memory to **test the correctness** of our design: we make this choiche to make a **comparison between the original wave diagram** and the **one produced** with **our AXI** in. Aside from our component, we have had to insert an **ad hoc adapter** to match the **full AXI3** interface exposed by the processor to the **AXI4-Lite** implemented by us.

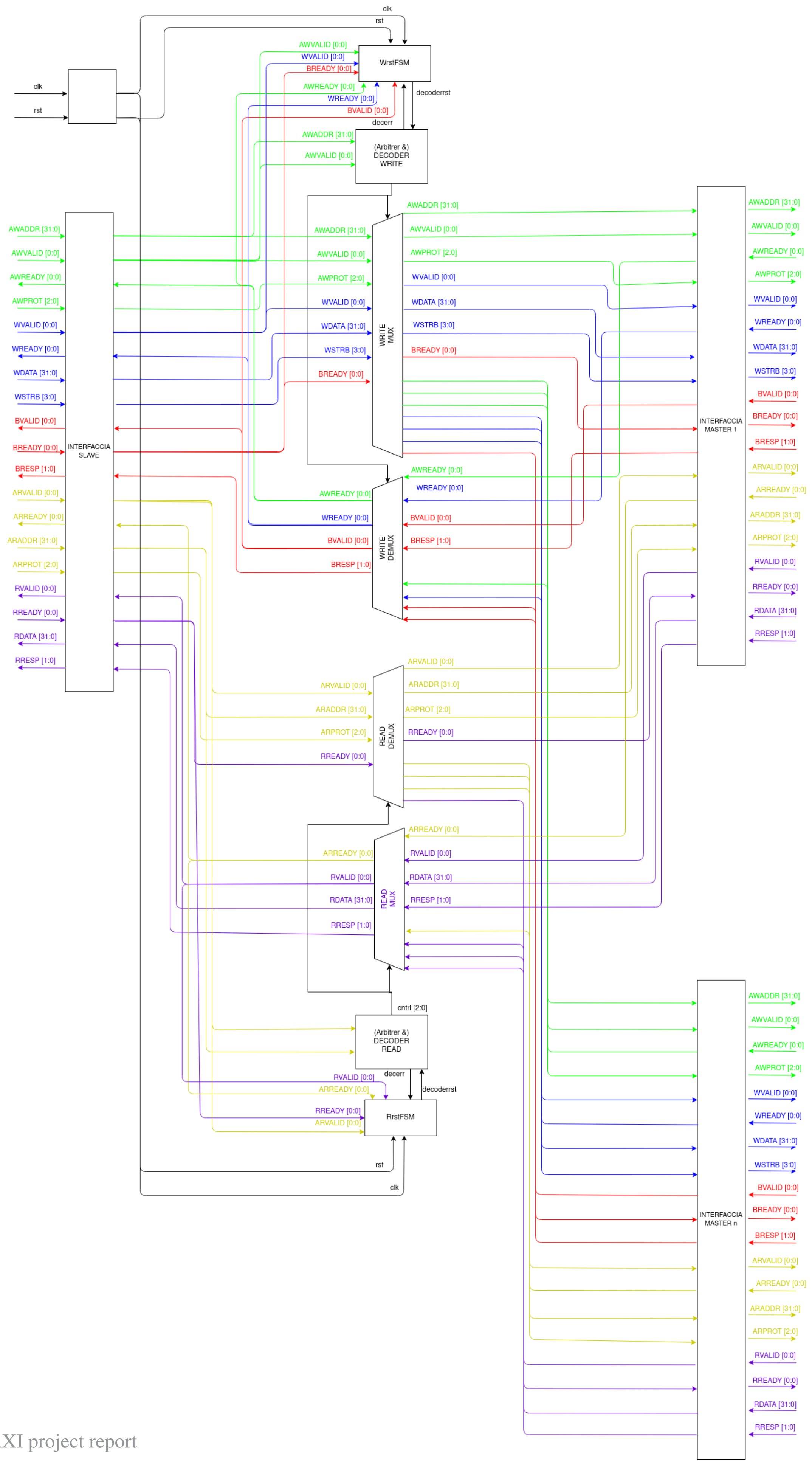


Figure 3: highLevel block diagram

3 Reading The OUTPUT

The **first signals** sent by **Cortex M3** are transmitted to the **uartlite** element. We studied the **meaning** of the **bits** sent on the **tx channel** and how the processor controls them.

3.1 Uart Decoding

The **communication** between Cortex M3 and uartlite is achieved using a **set of reads and writes** on the **same three addresses**: `0x4010_0004`, `0x4010_000c` (write only) and `0x4010_0008` (read only). While the first 16 bits are the base address for the communication with uartlite (as shown in the address editor of the block design), the last 16 bits define the offset specified in the uartlite's product guide [4, Chapter2.Register Space]. In particular, the offset `0x0004` is used as FIFO queue for the data which should be transmitted on tx channel.

Looking at the documentation[2] and at the IP custumization4, we can conclude that the **transmission on tx channel** is defined by a **sequence of 8 bits words**, enclosed by the **starting bit 0** and **the ending bit 1**, taken from the address `0x4010_0004` and **read starting from the last significant bit**. Despite what described in the figure 4, we found that the actual baudrate is about 115740 bit/sec (in according to the sampling period used 8.34 us for the tx channel in *tb m3 for arty*).

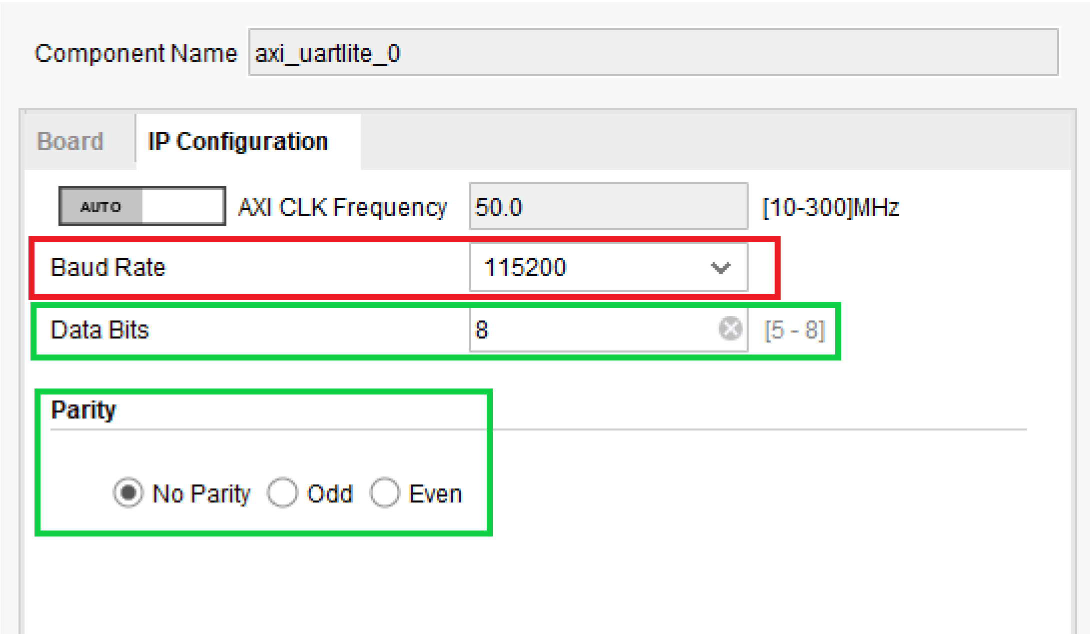


Figure 4: IP configuration with Baud Rate and bits definition

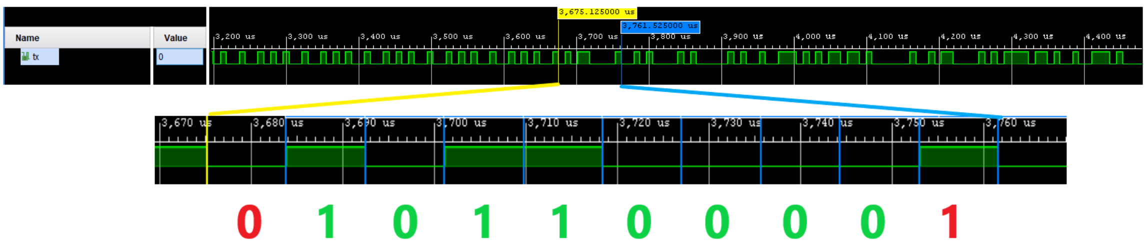


Figure 5: 0x0d (0000_1101 in binary) in the tx channel waveform with start and end bits

3.2 Decoding Software

After analysing the signals transmitted by the uartlite and after understanding their encoding, we wrote in the **Verilog file** of the testbench some lines of code in order to make **automatic** the **decoding of the output** and, at the same time, write it on a text file.

All the **read bits** are saved in **three register** variables (`start_sig`, `buffer` and `end_sig`, respectively) and cleaned a clock's cycle after being writtten or, for the `end_sig`, at the beginning of the always block using value Z for one bit variable and 0 for the buffer.

The **assigning** and the **checking** of the tx's values occur at the **positive edge** of the `clk_baud`. This is because we noticed that the bit changing is aligned with that clock.

4 Differences

In this section we would like to analyse the main differences between the *AXI interconnect* of the Cortex M3 example design and the *AXI connect*, the module which we had implemented.

In particular, the first paragraph explains how the signals pass through the two AXI and the delays given by them, while the second focuses on some post-synthesis features of our communication component.

4.1 Time Differences

Comparing the two implementation and the simulations it is possible to notice that the handshake protocol provided by our configuration is faster than that achieved by the other.
This is because, whereas we design a set of multiplexers and demultiplexers to make the connection among master and slaves, the AXI of the example is composed by three main type of components:

Slave Coupler: Used as interface for the Cortex M3, it converts the **AXI3 protocol** of the processor to the **AXI4-lite protocol** of *Xbar* and the peripherals. As a consequence, this conversion introduces a delay into the main signal of both write and read operations (In our implementation the conversion is provided by the component *AXI4lite adapter* which hides a simple wire connection for only the signals used).

Xbar: The most interesting element of *AXI interconnect* its main function is to initialize and manage the communication between a *Slave Coupler* and a *Master Coupler*. Therefore, the delay introduced by it is not only for choosing the right master interface to associate but also to achieve the handshake protocol among the interfaces (in our implementation the handshake protocol is managed by the master and slave themselves). The Figure 6 shows an example of read and write operations of the *Xbar*.

Master Coupler: Like the *Slave Coupler* but without conversion logic, it is used as interface for peripheral.

For more information about the *AXI interconnection* and its components, see [3].

4.2 Signal Propagation

Our implementation is safer but more costly

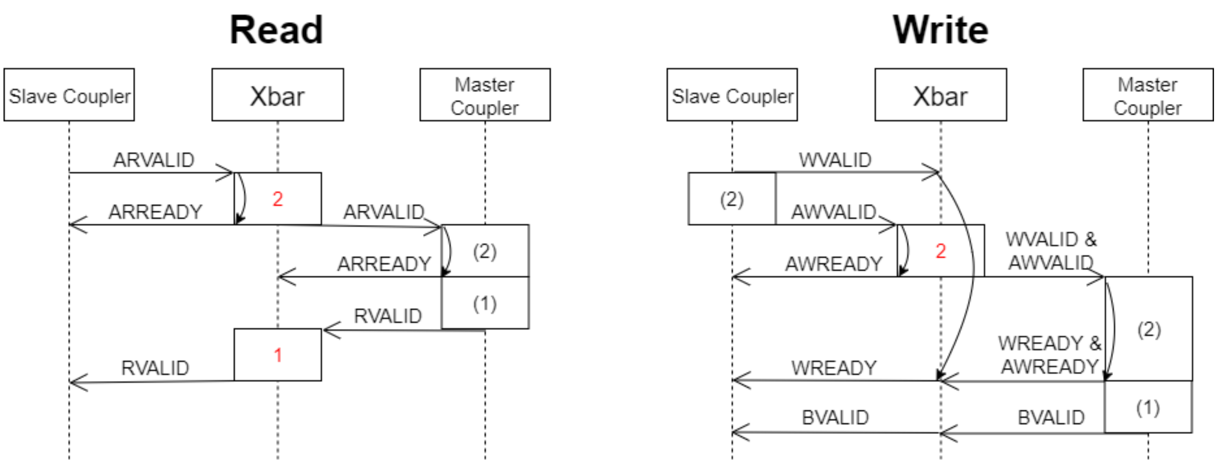


Figure 6: Read and write communication on Xbar with Xbar delays and other latencies (curly arrow to identify the communication's sections)

References

[1] ARM. $IHI0022H_{a}mba_{a}xi_{p}rotocol_{s}pec.pdf$. URL: https://beep.metid.polimi.it/web/2018-19-embedded-systems-1-william-fornaciari-/documenti-e-media?p_p_id=20&p_p_lifecycle=0&p_p_state=normal&p_p_mode=view&_20_struts_action=%2Fdocument_library%2Fview_file_entry&_20_redirect=https%3A%2F%2Fbeep.metid.polimi.it%2Fweb%2F2018-19-embedded-systems-1-william-fornaciari-%2Fdocumenti-e-media%3Fp_p_id%3D20%26p_p_lifecycle%3D0%26p_p_state%3Dnormal%26p_p_mode%3Dview%26_20_entryEnd%3D20%26_20_displayStyle%3Dlist%26_20_viewEntries%3D1%26_20_viewFolders%3D1%26_20_expandFolder%3D0%26_20_folderStart%3D0%26_20_action%3DbrowseFolder%26_20_struts_action%3D%252Fdocument_library%252Fview%26_20_folderEnd%3D20%26_20_entryStart%3D0%26_20_folderId%3D197422972%26%23p_20&_20_fileEntryId=197423679&#p_20.

[2] Scott Campbell. *Basics of UART communication*. URL: https://www.circuitbasics.com/basics-uart-communication.

[3] Vivado XIlinx. *AXI Interconnect v2.1 LogiCORE IP Product Guide*. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf.

[4] Vivado Xilinx. *AXI UART Lite v2.0 LogiCORE IP Product Guide*. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf.

[5] Vivado Xilinx. *ug1037-vivado-axi-reference-guide.pdf*. URL: https://beep.metid.polimi.it/web/2018-19-embedded-systems-1-william-fornaciari-/documenti-e-media?p_p_id=20&p_p_lifecycle=0&p_p_state=normal&p_p_mode=view&_20_struts_action=%2Fdocument_library%2Fview_file_entry&_20_redirect=https%3A%2F%2Fbeep.metid.polimi.it%2Fweb%2F2018-19-embedded-systems-1-william-fornaciari-%2Fdocumenti-e-media%3Fp_p_id%3D20%26p_p_lifecycle%3D0%26p_p_state%3Dnormal%26p_p_mode%3Dview%26_20_entryEnd%3D20%26_20_displayStyle%3Dlist%26_20_viewEntries%3D1%26_20_viewFolders%3D1%26_20_expandFolder%3D0%26_20_folderStart%3D0%26_20_action%3DbrowseFolder%26_20_struts_action%3D%252Fdocument_library%252Fview%26_20_folderEnd%3D20%26_20_entryStart%3D0%26_20_folderId%3D197422972%26%23p_20&_20_fileEntryId=197423705&#p_20.