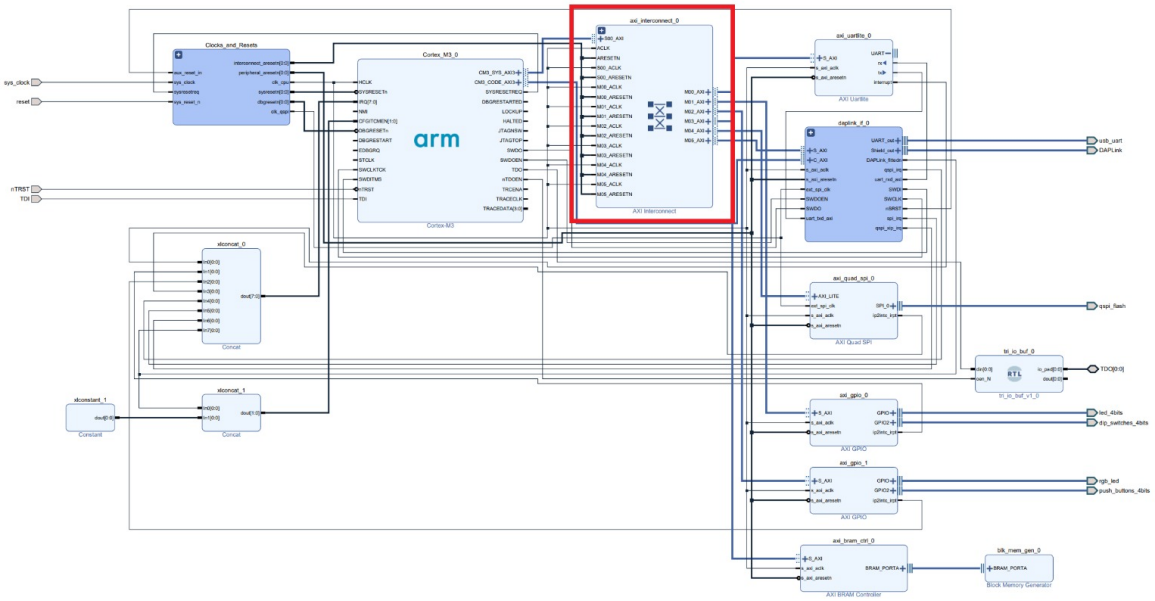
POLITECNICO
MILANO 1863

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 very interesting feature of **AXI4-Lite** is the **Handshake protocol**, that is done **for each data and address line** and that ensures the reading of the valid values. The signals involved are **valid**, asserted by the sender when the data is valid, and **ready**, asserted by the reciever when ready to recieve data. For more detailed informations, please see [1, Section-A3.2.1].

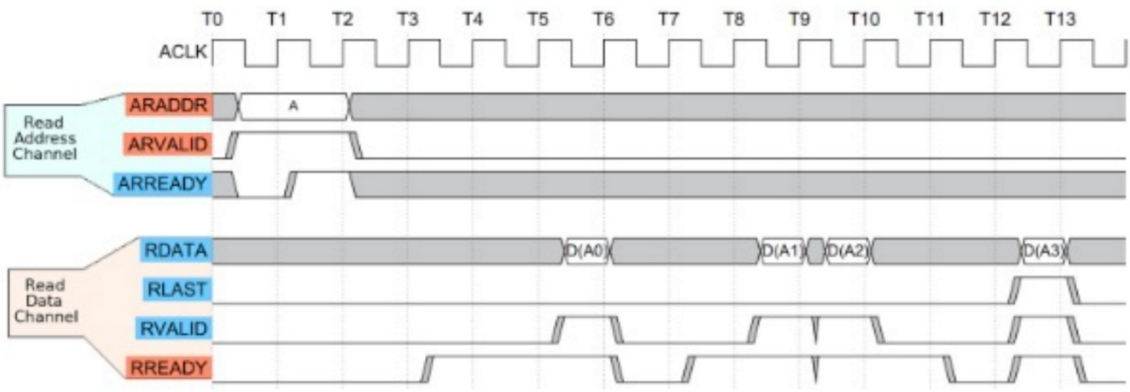


Figure 1: AXI4 (**non lite**) read

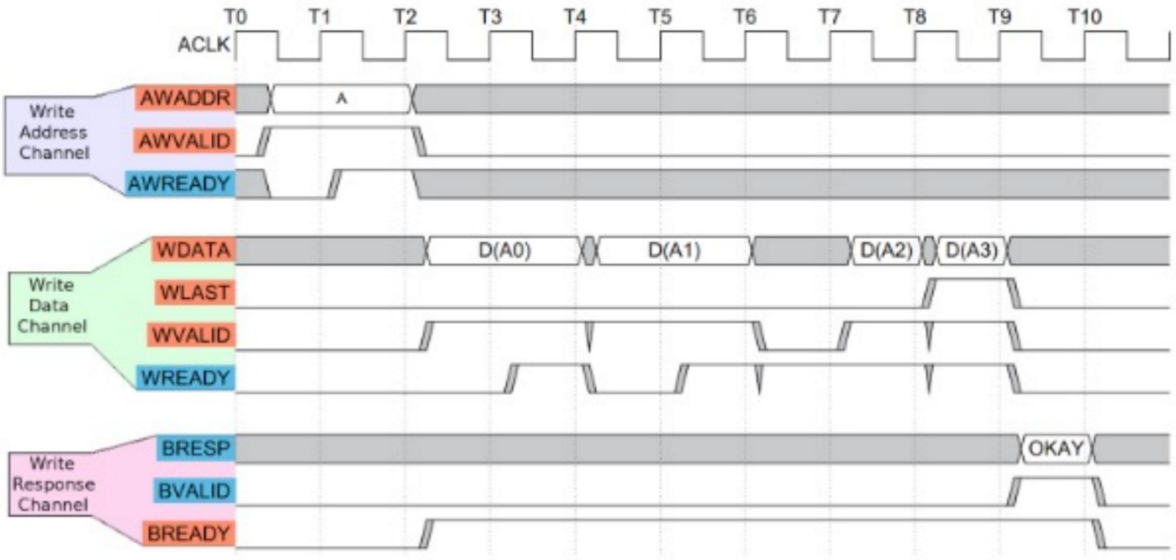


Figure 2: AXI4 (**non lite**) write

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.

2.2 Coupling

The coupling is achieved by the Muxers and Demuxers, driven by the Decoders which reads the Address and couple the address to the correspective slave.

2.3 Handshake

The handshake's signals are checked by the FSM that will drive the Decoders to signal when the address is valid and when it's not.

2.4 Errors

There are 2 types of error: The ones raised by the slave (and are generated by slave and so pass through the AXI interconnect) The ones raised if the address isn't mapped; in such a case we have an ad-hoc fake slave that sends the error to the master.

2.5 Integration inside the processor

We managed to put our AXI right inside the processor without any fake component.

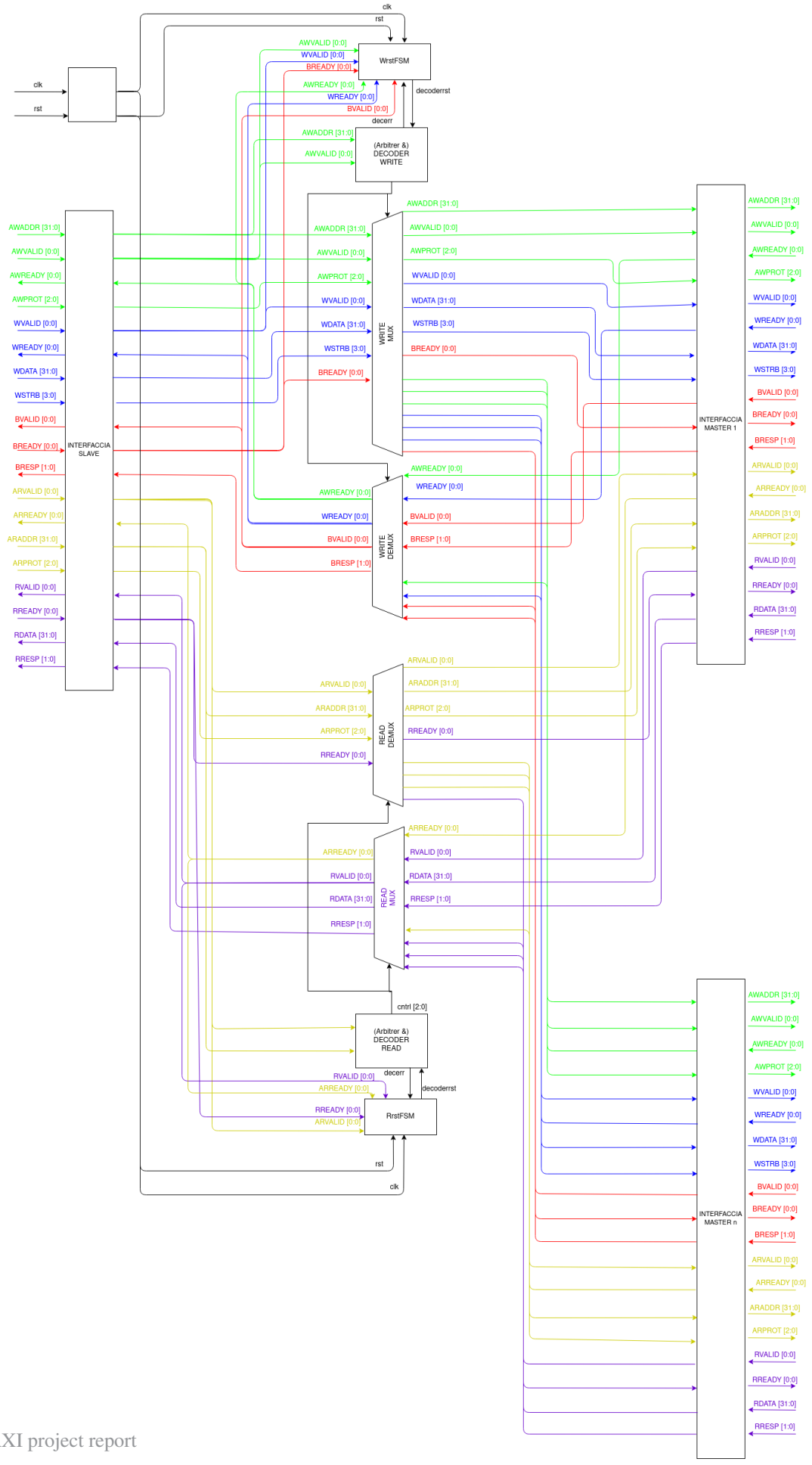


Figure 3: highLevel block diagram

3 Reading The OUTPUT

The first signals sent by Cortex M3 are transmitted to the uartlite element. The main scope of our studies on this component aims at understanding which the meaning of the bits sent from tx channel is and how the processor can control them. In order to reach our purpose, we studied the handshake protocol between these two elements of the design and we analysed the tx's output with the help of the documentation.

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 (how it is possible to see in the address editor of the block design), the last 16 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.

Unfortunately, to know the value of the words sent wasn't enough to understand the decoding procedure used on transmitter channel. We found also:

Baud Rate: After hypothesising a Baud Rate which was about 115740 bit/sec (in according to the sampling period used 8.34 us for the tx channel in *tb m3 for arty*), we discovered the more precise value of **175200 bit/sec** in the *IP configuration* of uartlite (the nearest value to 115740 bit/sec among the possible choices), like showed in Figure 4.

Data length: Defined in the *IP configuration* of the uartlite, the data length is **8 bits** without parity check, like showed in Figure 4.

Start and End Bits: Reading the documentation about the UART communication protocol[2] we discovered the presence of a **start bit 0** and an **end bit 1** used to enclose the 8 bits word.

Data Order: Analysing the sequence of bits on tx channel and the values written on address `0x4010_0004`, we found the character `0x0d` (carriage return in ASCII code) which is written **starting from the last significant bit** (unfortunately, the previous characters `0x2a` are too symmetric, thus we chose `0x0d`). In the Figure 5 it is possible to see the piece of waveform analised.

At the end, we can conclude that the transmission on tx channel is defined with a sequence of 8 bits words enclosed by the starting bit 0 and the ending bit 1 and taken from the address `0x4010_0004` starting by the last significant bit.

3.2 Decoding Software

After analysing the signals transmitted by the uartlite and after understanding their encoding, we wrote in the file Verilog of the testbench some line of code in order to make automatic the decoding of the output and, at the same time, write it on a text file. The main operation of this code is executed by the always block showed by the following Figure 6.

It waits until the first 0 on output tx, then it writes the first eight bits and, at the end, it takes the nineth bit as ending bit and it waits the first 0 value to execute the block again. All the 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. It is important to explain that the writing on the text file is only for the buffer and it happens one clock's cycle after that all the buffer's bits are defined checking the `end_sig`'s value (1 for correct transmission) and only if the file is opening (the file is opened at the beginning of initial block and its closing is managed by another always block). Moreover, 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 quite aligned with that clock, in fact, during the communication the output tx changes the bit 1.170 us before the positive edge of the `clk_baud`.

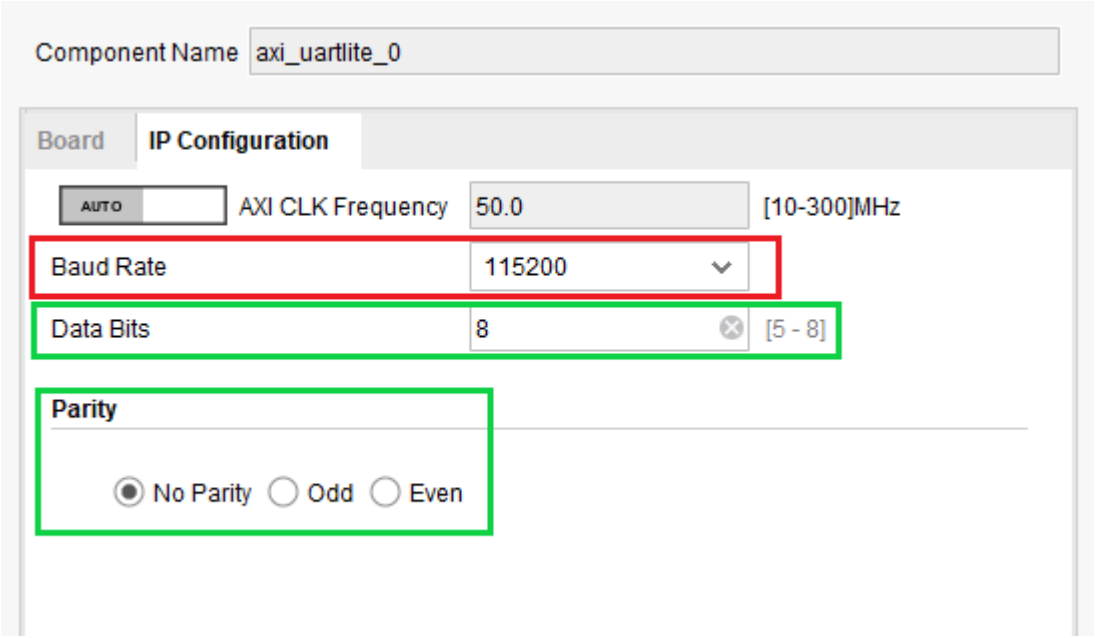


Figure 4: IP configuration with Baud Rate and bits definition

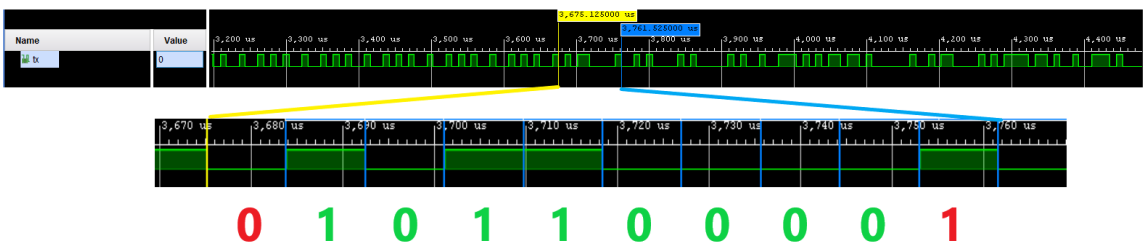


Figure 5

```
// write tx signal in char
always@(posedge clk_baud)
begin
    start_sig = uart_tx;
    end_sig = 1'bZ;
    if(start_sig == 1'b0)
    begin
        repeat(1)@(posedge clk_baud);
        start_sig = 1'bZ;
        while(bd_counter < 8 )
        begin
            buffer[bd_counter] = uart_tx;
            bd_counter = bd_counter + 1;
            repeat(1)@(posedge clk_baud);
        end
        end_sig = uart_tx;
        if (end_sig == 1)
        begin
            if(close_f == 0)
            begin
                $fwrite(fw, "%c", buffer);
            end
        end
        buffer = 8'b0000_0000;
        bd_counter = 0;
    end
end
```

Figure 6: The always block.

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 7 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].

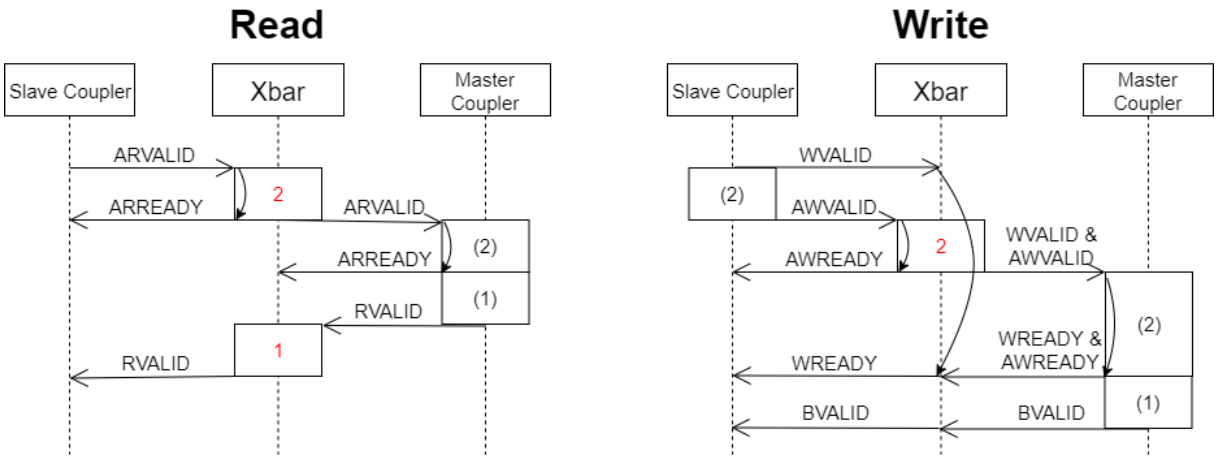


Figure 7: Read and write communication on Xbar with Xbar delays and other latencies (curly arrow to identify the communication's sections)

4.2 Signal Propagation

Our implementation is safer but more costly

References

[1] ARM. $IHI0022H_amba_axi_protocol_spec.pdf$. 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.