AXI4-Lite Interface Development

Author:
Jakub Szefer
Yale University
New Haven, CT 06511, USA

Revised:
November 14, 2019

This document is licensed under Creative Commons license: CC BY-SA 4.0
Source files are available at: bit.ly/cloudfpga
### Contents

1. **Introduction** 2  
   1.1 Prerequisites 2  

2. **AXI4-Lite Bus Interface** 2  
   2.1 AXI4-Lite Signals Going From Master 2  
   2.2 AXI4-Lite Signals Going To Master 3  
   2.3 AXI4-Lite Read and Write Transactions 3  

3. **Developing Simple Adder with an AXI4-Lite Interface and a Testbench** 5  
   3.1 AXI4-Lite Module Interface 5  
   3.2 Simple Adder with AXI4-Lite Interface 6  
   3.3 Testbench for the Adder with AXI4-Lite Interface 6  

4. **File Submission and Evaluation Information** 7  
   4.1 Tutorial Time Estimate 7  

5. **Acknowledgement** 7
1 Introduction

This document provides background about the AXI4-Lite protocol and interfaces. After reading the document and relevant information in other sources, students will be asked to develop an AXI4-Lite slave module that implements a simpler adder (with a delay), and a test bench that emulates the AXI4-Lite master module which interacts with the adder. The design should be tested and simulated using open-source iVerilog simulator and using open-source GTKWave for viewing simulation waveforms.

1.1 Prerequisites

This tutorial assumes that readers have access to a development computer, typically running Linux, where they have installed iVerilog simulator and GTKWave waveform viewer.

2 AXI4-Lite Bus Interface

AXI4-Lite bus protocol is a common protocol used for communication between modules. It is especially used in Cloud FPGAs, such as Amazon F1, to communicate between the “shell” and user’s logic. AXI4-Lite is a typical memory mapped address and data interface, which only supports sending single piece of data with each transaction. With AXI4-Lite data can move in both directions between the master and slave simultaneously, and data transfer sizes depends on the width of the data lines, usually it is 32-bits.

Details about AXI specification can be found in [https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf](https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf) and [https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf](https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf) which students should use as references for working on the AXI4-Lite interface. Wikipedia also contains article about AXI at [https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface](https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface).

2.1 AXI4-Lite Signals Going From Master

The signals going from the master module to the slave module are listed below:

- **awvalid** – Write address valid, indicates that valid write address and control information are available.
- **awaddr** – Write address, the write address bus gives the address of the transaction.
- **wvalid** – Write valid, this signal indicates that valid write data and strobes are available.
- **wdata** – Write data, actual data to be written.
- **wstrb** – Write strobes, his signal indicates which byte lanes to update in memory.

The **wstrb** controls which of the bytes in the data bus are valid and should be read by the slave. For example, if the master is performing an 8 bit write transaction to the slave then not all 32 bits of the data bus will be relevant. In the case of an AXI4-Lite transaction the data bus is 32 bits wide, and therefore the **wstrb** signal is four bits wide with each bit representing one byte (8 bits) of data.
• **bready** – Response ready, this signal indicates that the master can accept the response information.

• **arvalid** – Read address valid, when high, this signal indicates that the read address and control information is valid and remains stable until the address acknowledgement signal, **arready**, is high.

• **araddr** – Read address, the read address bus gives the address of a read transaction.

• **rready** – Read ready, this signal indicates that the master can accept the read data and response information.

### 2.2 AXI4-Lite Signals Going To Master

The signals going to the master module from the slave module are listed below:

• **awready** – Write address ready, this signal indicates that the slave is ready to accept an address and associated control signals.

• **wready** – Write ready, this signal indicates that the slave can accept the write data.

• **bvalid** – Write response valid, this signal indicates that a valid write response is available.

• **bresp** – Write response, this signal indicates the status of the write transfer: “00” = OKAY or “10” = SLVERR.

• **arready** – Read address ready, this signal indicates that the slave is ready to accept an address and associated control signals.

• **rvalid** – Read valid, this signal indicates that the required read data is available and the read transfer can complete.

• **rdata** – Read data, the actual data that is returned following the read request.

• **rresp** – Read response, this signal indicates the status of the read transfer: “00” = OKAY or “10” = SLVERR.

### 2.3 AXI4-Lite Read and Write Transactions

The AXI4-Lite specification gives the timing requirements for the different signals during reads or writes. Details are listed below, taken from “Digital Systems, Introduction to Advanced Extensible Interface (AXI)”, which is available online.

**Timing for WRITE transaction:**

1. The master puts an address on the write address channel **awaddr** and data on the write data channel **wdata**. At the same time it asserts **awvalid** and **wvalid** signals indicating the address and data on the respective channels are valid.

2. The master asserts the **bready** signal, indicating it is ready to receive a response.

3. The slave asserts **awready** and **wready** on the write address and write data channels, respectively.

4. Since both valid and both ready signals are present on the write address and write data channels, the handshakes on those channels occurs and the associated valid and ready

---

[4] See footnote 1

[5] https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ce4081
signals can be deasserted. (After both handshakes occur, the slave has captured the write address and data).

5. The slave asserts the \texttt{bvalid} signal, indicating there is a valid response on the write response channel \texttt{bresp}, typically it is “00” = \texttt{OKAY}.

6. The next rising clock edge completes the transaction, with both the ready and valid signals on the write response channel high.

**Timing for READ transaction:**

1. The master puts an address on the read address channel as well as asserts the \texttt{arvalid} signal, indicating the address is valid, and asserts \texttt{rready} signal, indicating the master is ready to receive data from the slave.

2. The slave asserts the \texttt{arready} signal, indicating that it is ready to receive the address on the bus.

3. Since both \texttt{arvalid} and \texttt{arready} are asserted, on the next rising clock edge the handshake occurs, after this the master and slave deassert \texttt{arvalid} and the \texttt{arready}, respectively. At this point, the slave has received the requested address.

4. The slave puts the requested data on the read data channel \texttt{rdata} and asserts the \texttt{rvalid} signal, indicating the data in the channel is valid. The slave can also put a response code on the \texttt{rresp} signal, typically it is “00” = \texttt{OKAY}.

5. Since both \texttt{rready} and \texttt{rvalid} signals are asserted, the next rising clock edge completes the transaction. \texttt{rready} and \texttt{rvalid} can now be deasserted.
3 Developing Simple Adder with an AXI4-Lite Interface and a Testbench

3.1 AXI4-Lite Module Interface

Code Listing 1 shows the interface of a module that uses AXI4-Lite interface. The inputs would come from a master module, e.g., the PCIe module in Amazon F1, or a simple testbench (as will be discussed later). The outputs go to the master module.

Listing 1: Inputs and outputs for a module with AXI4-Lite interface.

```verilog
module axi4lite_adder (
    input wire clk, main_a0,
    input wire rst, main_n_sync,
    // inputs to AXI-Lite slave
    input wire awvalid, awaddr,
    input wire wvalid, wdata,
    input wire [3:0] wstrb,
    input wire bready, bvalid,
    input wire arvalid, araddr,
    input wire rready,
    // outputs from AXI-Lite slave
    output wire awready, avalid,
    output wire wready, bvalid,
    output wire [1:0] bresp,
    output wire aready, rvalid,
    output wire [DATA_WIDTH-1:0] rdata,
    output reg [1:0] rresp,
 );
```
3.2 Simple Adder with AXI4-Lite Interface

Students should write a simple axi4lite_adder.v module that has the interface specified in code Listing 1. The module should have three 32-bit registers: arg1 for storing the first operand, arg2 for storing the second operand, and sum for storing the sum. The (byte) address of each register should be: ADDR_REG_ARG1, ADDR_REG_ARG2, and ADDR_REG_SUM. The values of this addresses should be specified in axi4lite_adder_defines.vh file, which should have the contents as listed in code Listing 2.

```
// Macros with useful parameters
#define DATA_WIDTH 32
#define ADDR_REG_ARG1 32'h0000_0500
#define ADDR_REG_ARG2 32'h0000_0504
#define ADDR_REG_SUM 32'h0000_0508
#define ADDER_DELAY 1000
```

Listing 2: Address definitions for simple adder module.

The axi4lite_adder_defines.vh file can be included in the axi4lite_adder.v file by using a `include "axi4lite_adder_defines.vh"` statement at the top of the file, so that the address macros can be used in the code.

The logic inside the axi4lite_adder.v should read the two operand registers and, after ADDER_DELAY write the result to the sum register. The addition should be triggered whenever there is change to the second operand register. E.g., each clock cycle check if the value in second operand has changed (assume the value is initially 0 and that addition will not be triggered when 0 is written to the register). Thus, simple `assign sum = arg1 + arg2;` should not be used in the code. Instead, a state machine needs to be written which is able to receive and send the correct AXI4-Lite signals, in order to implement read and write transactions (write transactions write to the operand registers, while read transactions are used to read from the sum register).

3.3 Testbench for the Adder with AXI4-Lite Interface

Students should write a simple axi4lite_adder_tb.v module which is the testbench for the adder. The testbench should test an instance of the axi4lite_adder.v module by writing two non-zero values to the two operand registers (using the addresses specified in the defines file) and then reading the result from the sum register (using the address specified in the defines file and after the delay cycles have passed – it is expected that if the testbench tries to read the sum register before the delay cycle have passed a zero value will be obtained).

For the testbench, further please follow the below requirements:

1. For the read and write operations, the timing of the signals generated by the testbench should follow the signals shown in Figures 2 and 1. For example, ARVALID signal should be high when ARADDR has a valid address, at all times both signals should be 0 as shown in Figure 2.

2. The testbench should perform 3 operations. I.e. there should be 3 sets of writes to set the operands, and three reads to read the sum result. The reset signal should not be used between the 3 operations; this is to test that both your state machine returns to some
initial IDLE state after each read and write completes and the adder is also able to handle more than one operation.

3. The delay cycles in the adder should be implemented as an extra state that has a counter and waits for ADDER_DELAY cycles. Do not use # delay statements in your adder logic.

4 File Submission and Evaluation Information

Students should complete this tutorial work as a group with their project partner(s). Three files should be uploaded to the project git repository in directory src/axi4lite_adder/. The submitted files should be: axi4lite_adder.v, axi4lite_adder_defines.vh, and axi4lite_adder_tb.v.

The designs will be tested and evaluated by modifying the parameters in the defines file, and also by using each group’s testbench file with each other group’s adder files.

4.1 Tutorial Time Estimate

To help improve the tutorial, please estimate the number of hours used to complete this tutorial (combined hours for all team members). Please round up your estimate to the nearest hour. Please put your answer as a single integer number of hours in your git repository in the file src/axi4lite_adder/TIME-ESTIMATE-TUTORIAL.txt.

5 Acknowledgement

Various online sources, listed in the footnotes, were used for information about the AXI4-Lite bus standard. The author would like to thank Shanquan Tian, Adam Wolnikowski, and Samantha Bleykhman for providing feedback on prior versions of this document.