

1 Objective

This laboratory work introduces basic information about the AXI4-Stream communication protocol. The main objective is to describe the general mechanism of this protocol and to show how it can be used to link various processing components in a computer system.

2 Introduction

Today's computer system components operate at different frequencies, which can result in imbalances in the load level of certain processing units. For example, if a component that has a higher processing frequency than the component to which it supplies input data, there are long time intervals in which the faster component has to wait until the slower component completes its processing before it can retrieve the provided data and process it. Thus, processing data in pipeline may become inefficient.

Therefore, it is necessary to implement a data flow control mechanism that regularizes the variable data flow traffic. One solution could be to temporarily store data in First In First Out (FIFO) buffers where it can be retrieved from by the components that need it, the time when faster modules wait for the slower ones to complete processing being significantly reduced in this way.

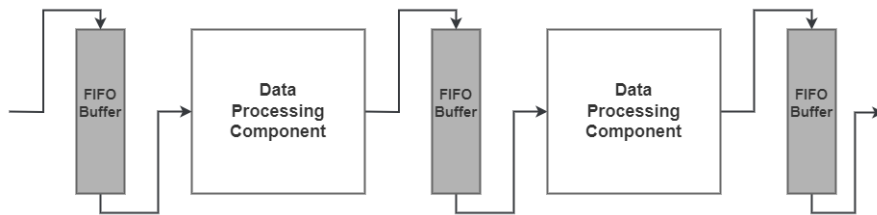


Figure 1: Using FIFOs to efficiently process data in pipeline

This type of FIFOs-based data processing is often performed using streaming communication protocols, which provide control of the data flow within the designed system and eliminate latencies caused by differences in the frequencies with which components process data. AXI4-Stream is such a protocol, which allows transferring data between interconnected modules in a streaming fashion.

3 AXI4-Stream Protocol Description

The AXI4-Stream protocol is used as a standard interface to connect components that wish to exchange data. The interface can be used to connect a sender, that generates data, to a receiver, that receives data. It works based on a **ready** / **valid** handshake.

3.1 Interface Signals

The interface main signals are listed in Table 1.

Table 1: Interface main signals list

Signal	Source	Description
ACLK	Clock source	The global clock signal. All signals are sampled on the rising edge of ACLK .
ARESETn	Reset source	The global reset signal. ARESETn is active-LOW.
TDATA	Sender	The actual data that is passing across the interface.
TVALID	Sender	TVALID indicates that the sender provides valid data. The data is transferred when both TVALID and TREADY are asserted.
TREADY	Receiver	TREADY indicates that the receiver can accept the data in the current cycle.

There are other optional signals within the interface defined by the AXI4-Stream protocol, but those listed above are mandatory in order to transfer data generated by the sender to the receiver.

3.2 Connecting Two Modules

Taking into account the previously described signals, two modules or components can be interconnected by simply connecting the signals in the input interface of the receiver to the signals of the same name in the output interface of the transmitter.

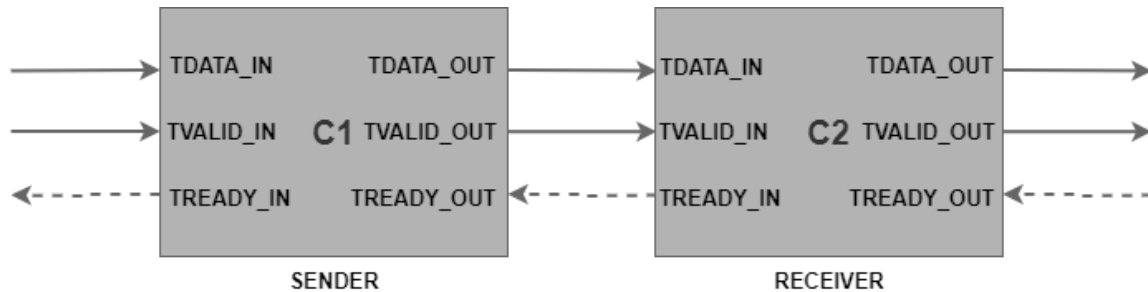


Figure 2: Two interconnected components

Figure 2 shows how the connection between two components (a sender and a receiver) is being performed.

3.3 Handshake Process

The **TVALID** and **TREADY** handshake determines when information is passed across the interface. A two-way flow control mechanism enables both the master and slave to control the rate at which the data and control information is transmitted across the interface.

The major rule imposed by the AXI4-Stream protocol is that, for a data transfer to occur both the **TVALID** and **TREADY** signals must be HIGH during the same clock cycle.

Base principle. The data is transferred from the sender to the receiver on the first rising edge of **ACLK** on which the **TVALID** has been set to HIGH by the sender and the **TREADY** has been set to HIGH by the receiver.

There are also some additional rules that must be followed by components communicating via an AXI4-Stream interface:

Rule 1. Once **TVALID** is asserted it must remain asserted until the handshake occurs. This means that when the sender asserts **TVALID**, it must keep it HIGH until the receiver asserts **TREADY** and a rising edge of the **ACLK** appears.

Rule 2. A sender is not permitted to wait until **TREADY** is asserted before asserting **TVALID**. In other words, the sender must report when the data it provides is valid, even if the receiver is not ready to accept it.

Rule 3. A receiver is permitted to wait for **TVALID** to be asserted before asserting **TREADY**. This means that a receiver can wait for the input data to be valid before reporting that it is ready to accept it.

Rule 4. If a receiver asserts **TREADY**, it is permitted to deassert **TREADY** before **TVALID** is asserted. Thus, if the receiver sets the **TREADY** HIGH before the **TVALID** was asserted by the sender, it can set **TREADY** back to LOW and wait for the sender to report the data is valid (see **Rule 3**).

Either **TVALID** or **TREADY** can be asserted first or both can be asserted in the same **ACLK** cycle. The following sections show all of these handshake sequences.

3.3.1 TVALID before TREADY

In Figure 3 the sender asserts **TVALID** signal HIGH before the receiver asserts **TREADY**. Once the sender has asserted **TVALID**, the data from the sender must remain unchanged until the receiver drives the **TREADY** signal HIGH, indicating that it can accept the data. In this case, transfer takes place once the receiver asserts **TREADY** HIGH. The arrow shows when the transfer occurs.

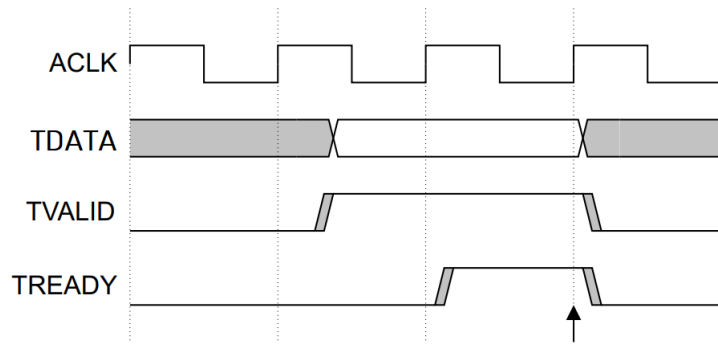


Figure 3: TVALID before TREADY handshake

3.3.2 TREADY before TVALID

Figure 4 shows the situation when the receiver drives **TREADY** HIGH before the data is valid. This indicates that the destination can accept the data in a single cycle of **ACLK**. In this case, transfer takes place once the sender asserts **TVALID** HIGH. The arrow indicates when the transfer occurs.

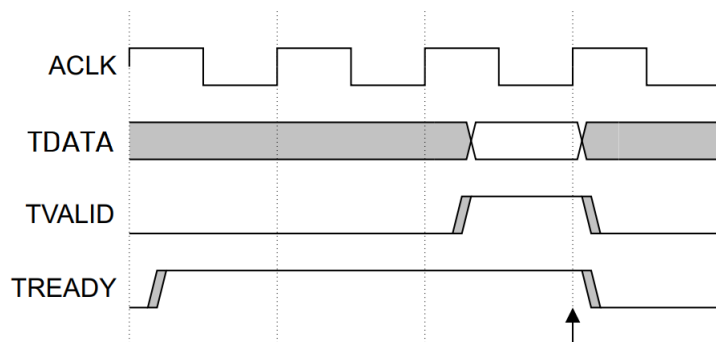


Figure 4: TREADY before TVALID handshake

3.3.3 TVALID with TREADY

In Figure 5 the sender asserts **TVALID** HIGH and the receiver asserts **TREADY** HIGH in the same cycle of **ACLK**. In this case, transfer takes place in the same cycle, as shown by the arrow.

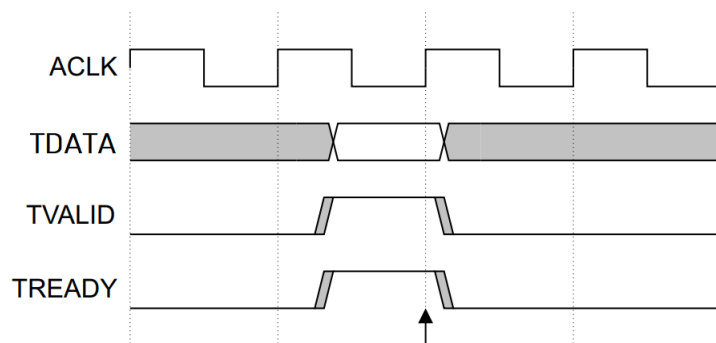


Figure 5: TREADY with TVALID handshake

To conclude, as a general rule, the data is transferred from the sender to the receiver on the first rising edge of **ACLK** when both **TVALID** and **TREADY** are HIGH.

4 Vivado IP Catalog Tutorial

The IP catalog available in Vivado is a single location for AMD-supplied IP. It consists of IP cores for embedded systems, communication, interfaces, and more.

The steps needed to customize and instantiate an IP core are presented as follows. The steps are shown using an IP Core which represents a FIFO buffer as an example.

Before starting this tutorial, you have to create a new Vivado project. For this, click on *File* → *New Project...* and in the opened window click *Next*. Provide the path or browse to the directory where you want to create the project and click *Next*. In the *Project Type* window select *RTL Project* and tick the *Do not specify sources at this time* option. The device properties that you need to choose are:

- **Product category:** All
- **Family:** Artix-7
- **Package:** cpg236
- **Speed grade:** -1

Then select from the table the **xc7a35t**cpg236-1 board and click *Next*. Click *Finish*.

Step 1. Open the Vivado IP Catalog

Open the *Window* vertical menu and select *IP Catalog*.

Step 2. Select the IP Core

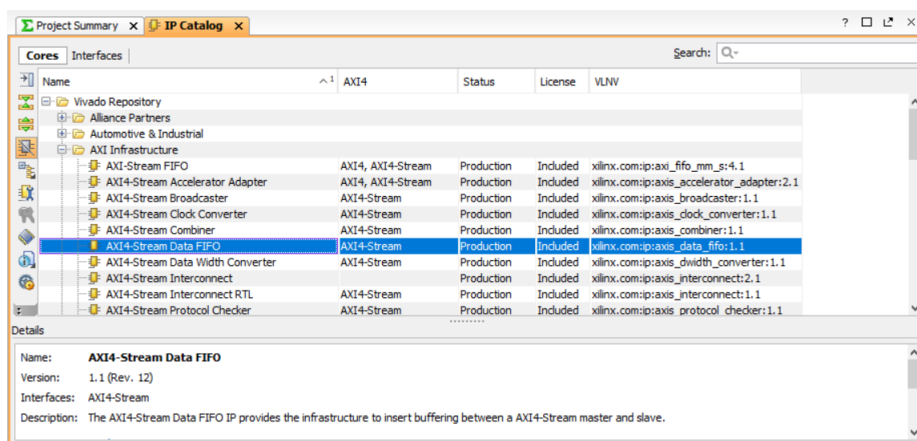


Figure 6: Selecting the IP Core

In the opened window, click open the *AXI Infrastructure* folder and double click on *AXI4-Stream Data FIFO*, as shown in Figure 6.

Step 3. Customize the IP Core

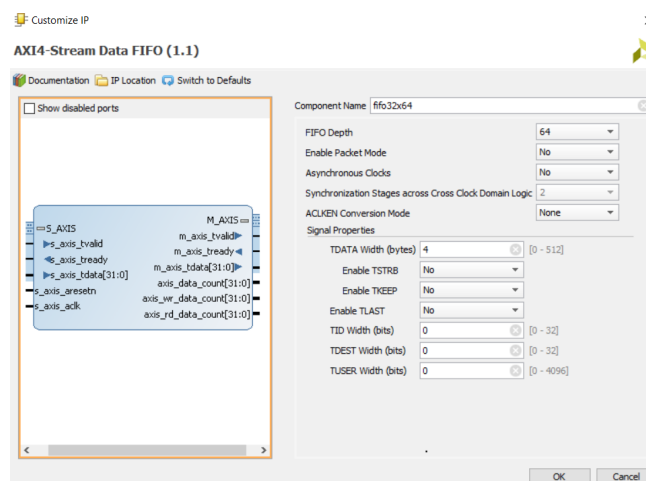


Figure 7: Customizing the IP Core

In the customization window, you can set the values of various parameters to make the IP Core meet your needs. For our example, set the component name to `fifo32x64`, the *FIFO Depth* to 64 and the *TDATA Width* to 4 bytes, as shown in Figure 7, then press the *OK* button.

Step 4. Generate the output products

In the *Generate Output Products* window, click on *Generate*.

Step 5. Declare and instantiate the IP Core

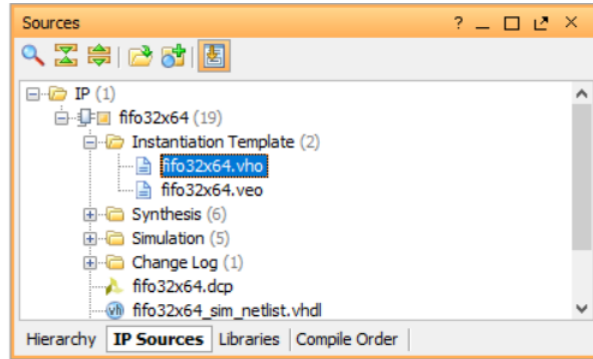


Figure 8: The location of the instantiation template file

When the synthesis is completed, open the *IP Sources* tab from the *Sources* window, then navigate to *IP* → *fifo34x64* → *Instantiation Template* and open the *fifo32x64.vho* file. Figure 8 shows the location of this file.

In the *fifo32x64.vho* file you can find the component declaration of the IP Core and the instantiation template, so you can use the generated IP Core as a component of a higher level module in your project.

5 Vivado Simulation Tutorial Using Stimulus Files

Reading signal values from file is an alternative way of generating stimuli for the design under test (DUT). This is typically useful when the design has to be tested using real data such as values measured by sensors. The testbench sequence and timing is hard-coded in a stimulus file that is read by the VHDL testbench, line by line.

This tutorial shows the steps needed to include a stimulus file in a Vivado project and describes an example of processing data read from the file and writing the results to an output file.

Note: Use the project created for the previous tutorial in Section 4. The flow that you need to follow is described step by step as follows.

Step 1. Include the stimulus file in the project

The stimulus file used in this example is the *temperature.csv* file which is structured as follows:

```
timestamp, temperature 1, temperature 2
timestamp, temperature 1, temperature 2
...
timestamp, temperature 1, temperature 2
```

The values in the `timestamp` column are strings and the values in the `temperature` columns are 32-bit floating-point values. The data stored in this file represent the values read by two temperature sensors, together with the corresponding timestamps.

To include the file in your project, in the *Project Manager* section from the *Flow Navigator* click on *Add Sources*. Select *Add or create simulation sources* and press *Next*. Click on *Add files*. In the navigation window select for the *Files of type* field the value *All Files* and navigate to the location of the *temperature.csv* file on your computer. After finding the file, select it, press *OK* and then *Finish*.

The file should be visible now in the *Sources* window, in the *Simulation Sources* → *sim_1* → *CSV* directory.

Step 2. Generate the IP Core for subtracting two floating-point values

The IP Core that you need to generate is *Floating-point* and it can be found in the *Vivado Repository* → *Math Functions* → *Floating Point* directory of the IP Catalog.

In the *Operation Selection* tab of the *Customize IP* window choose select *Add/Subtract* for the *Operation Selection* and *Subtract* for the *Add/Subtract* and *FMA Operator options*. In the *Precision of Inputs* tab of the same window select the *Single* option. Set the component name to *fp_subtractor* and generate the IP Core, as described in the previous tutorial.

Step 3. Include the design source file in the project

The design used for testing is implemented in the *temperature_subtractor.vhd* file and it performs the subtraction of two floating point values. It is designed in a structural manner, using the previously generated *fifo32x64* IP Core and an IP Core for performing the subtraction, which you have customized and generated in the previous step. The block design is shown in Figure 9.

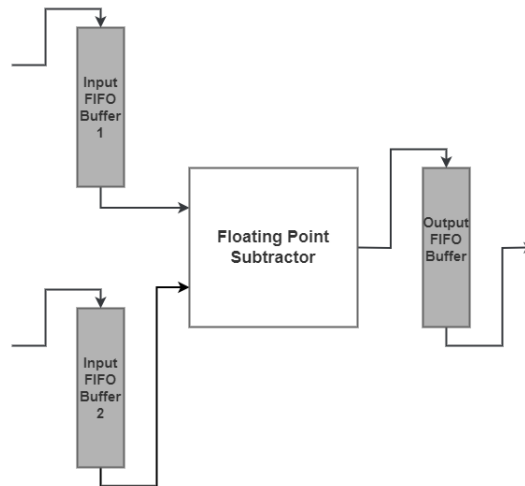


Figure 9: The block design of the example module

In the *Project Manager* section from the *Flow Navigator* click on *Add Sources*. Select *Add or create design sources* and press *Next*. Click on *Add files*. In the navigation window select for the *Files of type* field the value *All Files* and navigate to the location of the *temperature_subtractor.vhd* file on your computer. After finding the file, select it, press *OK* and then *Finish*.

Step 4. Include the testbench file in the project

Follow the same steps to include the *testbench_stimulus_file.vhd* in your project.

This file is used to read the values stores in the *temperature.csv* file, to provide it to the module being tested and to write an output file. There are two distinct processes in the testbench file. The first one is used to read data from the input file, while the second process writes the results to the output file. The subprograms and types needed for reading and writing external files in VHDL are located in the *TEXTIO* package. This package is part of the *std* library.

The input file is opened in *read_mode* as follows:

```
file sensors_data : text open read_mode is "temperature.csv";
```

The file is read line by line using the *readline* function. On every rising edge of the *ac1k* signal when the module being tested is ready to accept data, a new line is read.

The output file is automatically created at simulation by opening it in *write_mode* as shown below:

```
file results : text open write_mode is "C:/SCS/AXI4Stream/temperature_results.csv";
```

Note: Make sure you provide the absolut path to your own project directory followed by the name of the output file.

Step 5. Run behavioral simulation and validate the results

In the *Simulation* section of the *Flow Navigator* click on *Run Simulation* and then on *Run Behavioral Simulation*. Simulate the design for at least 1700 ns or until the value of the *wr_count* signal is greater than the value of the *rd_count* signal to be sure that all the results produced by the module for every pair of input values were successfully written in the output file. You can validate your results by comparing the content of your output file with the content of the *valid_results.csv* file.

6 Exercises

- 6.1 Follow the steps presented in Section 4 to generate a FIFO buffer IP Core.
- 6.2 Follow the tutorial in Section 5 to simulate the design implemented in *temperature_subtractor.vhd* file using the *temperature.csv* stimulus file. Validate your result using the *valid_results.csv* file.
- 6.3 The current implementation of the *temperature_subtractor* performs the $t_1 - t_2$ difference for every pair of t_1 and t_2 given as inputs. Modify the implementation so that the result provided for a pair of t_1 and t_2 is $\sqrt{|t_1 - t_2|}$.

Hints

- You have to generate two more IP Cores for computing the absolute value and the square root and to instantiate them in the top level *temperature_subtractor* module.
- In order to compute the absolute value and the square root, you have to generate the corresponding IP Cores starting from the same *Floating-point* IP Core which is used for the *fp_subtractor* module. In the customization window select the corresponding options for the operation performed by the IP Core (*Absolut Value* and *Square-root*).
- The block design of the module you need to implement is shown in Figure 10.

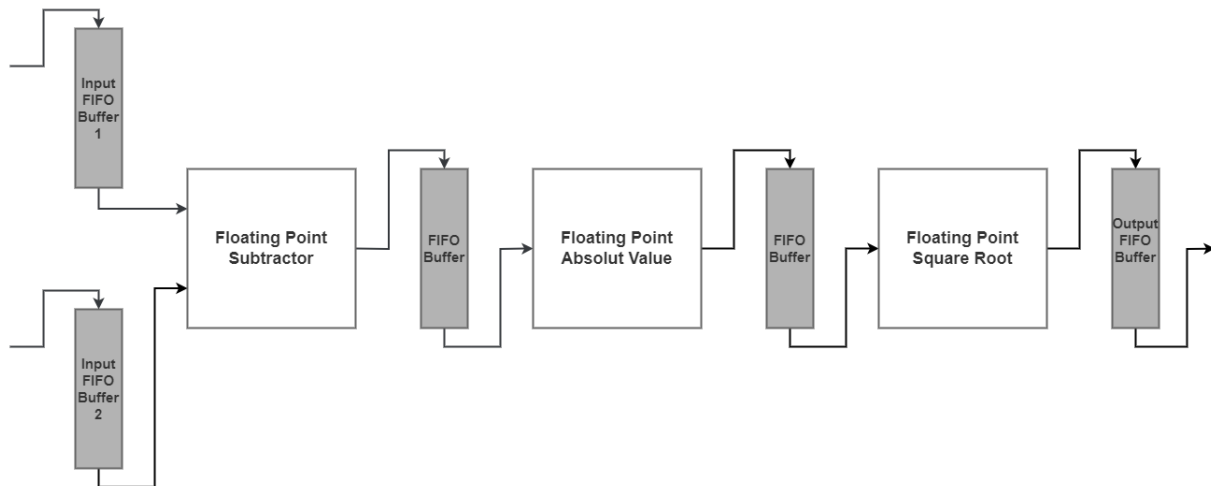


Figure 10: The block design of required module

References

- [1] AMBA 4 AXI4-Stream Protocol Specification - Arm, accessed 11/09/2023, <https://documentation-service.arm.com/static/642583d7314e245d086bc8c9?token=>
- [2] How the axi-style ready/valid handshake works, accessed 12/09/2023, <https://vhdlwhiz.com/how-the-axi-style-ready-valid-handshake-works/>
- [3] Vivado Design Suite User Guide: Designing with IP (UG896) - Using the IP Catalog, accessed 12/09/2023, <https://docs.xilinx.com/r/en-US/ug896-vivado-ip/Using-the-IP-Catalog>
- [4] Stimulus file read in testbench using TEXTIO, accessed 16/09/2023, <https://vhdlwhiz.com/stimulus-file/>