Xillybus FPGA designer's guide

Xillybus Ltd.

www.xillybus.com

Version 3.0

1

Introduction

Xillybus' IP cores are intended to interface with user application logic through a FIFO or a dual-port block RAM. Understanding the API for a direct interface with the IP core is therefore not necessary in most cases.

Even when a direct interface with the IP core is desired, almost all API definitions can be deduced from a simple rule: The user application logic should behave exactly like a FIFO or a block RAM. Implicitly, this rule also means that no assumptions should be made on how and when the Xillybus IP core accesses the logic that is connected to it. Data accesses can be continuous, or there might be time gaps of arbitrary length with no data exchange at any given moment. A FIFO or memory will not have any problems with such access pattern, and so shouldn't any logic that interfaces directly with the IP core.

It's a common mistake to consider the IP core's irregular access pattern as a bug, in particular after observing unexplainable pauses in the data flow between the IP core and the FIFO.

Because of the possibility of not handling rare conditions correctly, when designing application logic which interfaces directly with the IP core, it's recommended not to do so for the sake of reducing the consumption of logic, at least not in the early stages of the design. Unless the desired functionality requires a direct interface with the IP core, it's recommended to use FIFOs or block RAMs in order to avoid bugs in the user application logic.

This guide defines this direct API and also elaborates on popular applications. Before getting down to the details presented by this document, it's recommended to gain an initial experience with Xillybus, as suggested in these guides:

- Getting started with the FPGA demo bundle for Xilinx

- Getting started with the FPGA demo bundle for Intel FPGA
- Getting started with Xillinux for Zynq-7000
- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host

Some parts of this guide also assumes understanding of the difference between synchronous and asynchronous streams. This is discussed in section 2 of either of these two guides:

- Xillybus host application programming guide for Linux
- Xillybus host application programming guide for Windows

XillyUSB IP cores expose the same API, and are a subset of Xillybus IP cores. Accordingly, the name “Xillybus” refers to XillyUSB IP cores as well in this document, unless said otherwise.

For those who are curious, a brief explanation on how Xillybus is implemented can be found in Appendix A of either Xillybus host application programming guide for Linux or Xillybus host application programming guide for Windows.

2

General guidelines

2.1 Clocking

All signals from and to the Xillybus IP core must be synchronous with the rising edge of bus_clk, which is supplied by the IP core itself.

For Xillybus IP cores that are based upon PCIe, this clock is generated by the PCIe block, and has a frequency depending on the platform: For the baseline IP core (revision A) the frequency of bus_clk is either 62.5 MHz, 125 MHz or 250 MHz, depending on if the maximal bandwitdh (as advertised) is 200 MB/s, 400 MB/s or 800 MB/s, respectively.

With later revisions (B, XL and XXL) bus_clk has a frequency of 250 MHz.

Zynq-based platforms typically have a bus_clk of 100 MHz. XillyUSB works with a bus_clk of 125 MHz.

In most cases, there is a possibility to change the clock's frequency within a limited list of choices, by configuring the PCIe block or the processor core that generates it.

If the timing constraints for the PCIe block are set correctly (as in the demo bundles), the application logic that is driven by bus_clk is covered by proper timing constraints as well. The same applies for Zynq-based platforms as well as XillyUSB: The timing constraints are propagated from the clock that drives the IP core to the application logic that is driven by bus_clk.

This is not to say, that all application logic needs to be driven by bus_clk, and neither do data sources and data consumers. Often, a dual-clock FIFO is used together with the IP core: One side is connected to the Xillybus IP core, and is therefore synchronous with bus_clk. The application logic is accordingly connected to the FIFO's other side, which is synchronized with the application logic's clock, which can have any allowed

frequency. Hence the FIFO is used not only as a short-term temporary storage, but also for clock domain crossing.

2.2 Data width

With baseline Xillybus IP cores (revision A), each FIFO or memory interface works with data in widths of 8 bits, 16 bits or 32 bits. Later revisions, as well as XillyUSB, support wider data interfaces.

Wider data allows higher bandwidth performance and is also more convenient in applications where the natural transmission word is wider than 8 bits. On the other hand, the inherent data width on the host side remains 8 bits (a byte), because read() and write() function calls define their length in bytes.

The considerations for choosing the data width are discussed briefly in The guide to defining a custom Xillybus IP core.

2.3 Interfacing through a FIFO

The demo bundle demonstrates how a FIFO should be connected: It has a FIFO with both sides connected to the IP core, hence implementing a loopback on two streams.

The FIFOs in the demo bundle are configured for a common clock on both sides. This is not suitable when the FIFO is used for clock domain crossing, in which case a dual-clock FIFO (often called “asynchronous FIFO”) should be used.

Depending on the direction of the data flow, the “empty” or “full” signal of the FIFO is connected to the Xillybus IP core, which uses these signals to determine whether a burst of data transfer should be initiated.

Once a burst has started, these signals are relied upon to make sure that the Xillybus IP core doesn’t attempt to read from an empty FIFO or write to a full one.

There is however no guarantee on when a burst is initiated on a FIFO, even when it indicates that it’s ready for such. Neither is there any certainty regarding the length of such a burst. For example, the IP core may very well stop fetching data from a FIFO in the middle of a burst, even before it’s empty.

The general rule is that the Xillybus IP core attempts to serve all FIFOs connected to it equally. FIFOs which tend to get filled faster will be granted longer bursts, as they don’t activate their “empty” or “full” as often.

This simple arbitration method ensures efficient communication with FIFOs that tend

to get filled rapidly, and at the same time a low latency on FIFOs that receive data at a lower rate.

As for the depth of the FIFO, the Xillybus IP core works with any value, but this attribute should be chosen to cope with the expected data flow. Even though this is sometimes a matter of trial and error, a FIFO with a depth of 2 kBytes is almost always the correct choice for an asynchronous stream, even for high data rates.

The rationale behind this choice is that the Xillybus core is not likely to neglect a FIFO of this size long enough to cause an overflow or underflow. This is of course true as long as the DMA buffers are filled or emptied quickly enough by the user application software running on the host. If this is not the case, the solution might be to make the DMA buffers larger. Attempting to solve this with a larger FIFO is unreasonable, as there is much less memory on the FPGA.

If the IP core is connected to a FIFO for reading data from it, it expects the behavior of a regular FIFO (as opposed to FWFT, First Word Fall Through).

2.4 Behavior of “empty” and “full” signals

In a normally operating FIFO, the “empty” signal can change from low to high only one clock cycle after the read enable was high. Likewise, the “full” signal can change from low to high only one clock cycle after the write enable was high.

These two signals can go low at any moment, of course.

The Xillybus IP core relies on this behavior: When a FIFO indicates to the IP core that it’s ready for a data transfer (by a low “empty” or “full”, as applicable), a state machine in the core may start a chain of events which will lead to the transfer of at least one data element. Hence if the “empty” signal changes to high before the IP core fetches any data from the FIFO, it’s possible that the IP core ignores the “empty” signal during one clock cycle. Such an event is harmless in terms of the IP core’s state integrity, but may lead to unexpected and unpredictable data flowing in the stream.

The same goes with the “full” signal: If it changes from low to high before the IP cores writes a data word to it, the IP core may ignore the “full” signal during one clock cycle. Once again, this is harmless to the IP core itself, but results in the data word being lost.

A properly designed FIFO can create this faulty condition only by being reset while it was ready for communication with the Xillybus IP core. This is bad practice in any case.

If the IP core is interfaced directly with application logic, care must be taken to imitate a standard FIFO regarding this matter.

3

Description of signals

3.1 Naming convention of FPGA signals

Except for the two global signals, bus_clk and quiesce, all signals follow a simple convention. For example, a write enable signal may have the name user_w_write_32_wren. This name is broken into four components:

1. The “user” prefix is common to all user interface signals.
2. The “w” part indicates that this signal belongs to a stream from the host to the FPGA (host “write”). Streams from the FPGA to the host have an “r” instead. Address signals don’t have this part, since they apply to both directions. Note that the host’s viewpoint is taken for choosing “w” or “r”.
3. The “write_32” strings appears in the related device file’s name: /dev/xillybus_write_32 or /dev/xillyusb_00_write_32, as applicable.
4. The suffix signifies the signal’s meaning.

In the remainder of this section, the device file name (the third component) is denoted {devfile} to avoid confusion.

Each signal names is followed by (IN) to indicate that the signal is an input to the IP core, or (OUT) when it’s an output from the IP core.

3.2 Signals for host to FPGA transmission

- user_w_{devfile}_data (**OUT**) – This signal contains data during write cycles.

- user_w_{devfile}_wren (**OUT**) – This signal is a write enable signal to the FIFO: It is high when there is valid data on the user_w_{devfile}_data signal that should be written to the FIFO (or any other logic that imitates a FIFO's behavior).

- user_w_{devfile}_full (**IN**) – This signal informs the core that no more data can be written.

  Important: The 'full' signal may change from low to high only on the clock cycle after a write cycle. This is the way standard FIFOs behave, so this rule needs attention only if the IP core is connected directly with the application logic (i.e. without a FIFO in the middle).

  The reason for this rule is that the Xillybus IP core treats a low 'full' signal as a green light to start transferring data from the host. Failing to conform with this rule may cause sporadic writes that ignore the 'full' condition.

  A typical Verilog implementation of the 'full' signal should be something like this:

```
always @(posedge bus_clk)
  if (ready_to_get_more_data)
    user_w_mydevice_full <= 0; // Turn low any time
  else if (user_w_mydevice_wren && { ... some condition ... } )
    user_w_mydevice_full <= 1; // Only in conjunction with wren
```

  The same in VHDL:

```
process (bus_clk)
begin
  if (bus_clk'event and bus_clk = '1') then
    if (ready_to_get_more_data = '1') then
      user_w_mydevice_full <= '0'; -- Turn low any time
    elsif (user_w_mydevice_wren = '1' and { some condition } )
      user_w_mydevice_full <= '1'; -- Turn high only with wren
    end if;
  end if;
end process;
```

- user_w_{devfile}_open (**OUT**) – This signal is high when the related device file on the host is open for write (if the file is open for read-only, when allowed, it will not change this signal). This signal can optionally be used to reset the FIFO or other logic when the file is closed (used as an active low reset).

  If a file is opened by multiple processes on the host (e.g. as a result of a call to the fork() function), this signal remains high until all processes has closed the

file.

3.3 Signals for FPGA to host transmission

- user_r_{devfile}_data (**IN**) – This signal contains data during read cycles. This signal must not change except for when a FIFO would have changed it, as a result of a high read enable. In other words, it may only change on a clock cycle after user_r_{devfile}_rden is high.
- user_r_{devfile}_rden (**OUT**) – This signal is a read enable signal to the FIFO: When this signal is high, the IP core expects valid data to be present on user_r_{devfile}_data on the following clock cycle.
- user_r_{devfile}_empty (**IN**) – This signal informs the core that no more data can be read.

  Important: The 'empty' signal may change from low to high only on the clock cycle after a read cycle. This is the way standard FIFOs behave, so this rule needs attention only if the IP core is connected directly with the application logic (i.e. without a FIFO in the middle).

  The reason for this rule is that the Xillybus IP core treats a low 'empty' signal as a green light to start transferring data to the host. Failing to conform with this rule may cause sporadic reads that ignore that the FIFO is empty.

  A typical Verilog implementation of the 'empty' signal should be something like this:

```
always @(posedge bus_clk)
  if (ready_to_give_more_data)
    user_r_mydevice_empty <= 0; // Turn low any time
  else if (user_r_mydevice_rden && { ... some condition ... } )
    user_r_mydevice_empty <= 1; // Turn high only with rden
```

  The same in VHDL:

```
process (bus_clk)
begin
  if (bus_clk'event and bus_clk = '1') then
    if (ready_to_give_more_data = '1') then
      user_r_mydevice_empty <= '0'; -- Turn low any time
    elsif (user_r_mydevice_rden = '1' and { some condition } )
      user_r_mydevice_empty <= '1'; -- Turn high only with rden
    end if;
  end if;
end process;
```

- user_r_{devfile}_eof (**IN**) – This signal tells the core to generate an end-of-file. It's like an 'empty' signal, but after it has been high, the core will not read from the FIFO (i.e. user_r_{devfile}_rden is kept low) until the file is closed and reopened.

  On the host, the application software will finish reading the data that the IP core has received before this signal became high, and then it will receive an EOF when it calls the read() function.

  Note that the 'eof' signal will not cause an EOF at the host immediately, if there is still data that has not been read by the application software. The delivery of the EOF on the host is made with common sense, i.e. after all data has been read by the host.

  After the 'eof' signal has been high, it doesn't matter if it stays high afterwards or changes to low. The IP core remembers the EOF request until the file is closed. Regarding the 'empty' signal, it doesn't matter if it changes to high at the same time as 'eof' changes to high. In fact, from the moment 'eof' is high, the 'empty' signal doesn't matter at all, until the file is closed.

  Like the 'empty' signal, the 'eof' signal must be changed to high only on a clock cycle after a read cycle. There is one exception however: When the 'empty' signal is already high, the 'eof' may be changed to high anytime. This exception can be used to cause the immediate termination of a read() function call on the host, if it sleeps while waiting for data.

  Changing 'eof' to high without conforming with this rule will generate an EOF, but it may not work accurately: Some data may be lost just before the EOF, or unrelated data may be added before the EOF, or even after the EOF (so the application software receives data after the EOF, which is illegal).

  One possibility to ensure that 'eof' doesn't change to high when it's not allowed to, is to make it the output of a combinatoric function, of the form, in Verilog:

```
assign user_r_mydevice_eof = user_r_mydevice_empty && [ ... ];
```

Or in VHDL:

```
user_r_mydevice_eof <= user_r_mydevice_empty and [ ... ];
```

With this method, the 'eof' signal is always low when 'empty' is low.

- user_r_{devfile}_open (**OUT**) – This signal is high when the related device file on the host is open for read (if the file is open for write-only, when allowed, it will not change this signal). This signal can optionally be used to reset the FIFO or other logic when the file is closed (used as an active low reset).

  If a file is opened by multiple processes on the host (e.g. as a result of a call to the fork() function), this signal remains high until all processes has closed the file.

  There is no direct connection between the 'eof' signal and the 'open' signal. The 'open' signal will change to low when the file is closed on the host, and not when the 'eof' signal changes to high, nor when the EOF is delivered on the host.

3.4 Memory interface signals

A Xillybus interface can be configured to also have an address signal. An increment of the address occurs automatically on read and write cycles. Also, the application software can set an arbitrary value to the address, by using the standard API for seeking in a file (e.g. lseek()).

With some of the signals mentioned above, as well as the signals detailed next, a standard block RAM is easily connected with the IP core, making the block RAM's memory array available to the host as a file: Read and write operations on the file result in read and write operations on the memory array. The host may access single memory elements or segments, depending on the length of the read or write operations.

Also, by implementing an array of registers on the FPGA, which presents signals like a block RAM, these registers become easily accessed by host.

The 'empty' and 'full' signals can be used to slow down read and write to operations for memories that require wait states, or when there is another reason to delay operations briefly.

These are the two additional signals for this purpose:

- user_{devfile}_addr (**OUT**) – This signal contains the address at the present time. When either a read enable or a write enable is high, this is the address to be read from or written to. Connecting this signal directly to a block RAM's address input will work as naturally expected. The width of this signal is configurable up to 32 bits.

  The address' value returns to zero when a read or write operation reaches above the maximal address possible for the width of this signal. Function calls to lseek() that are out of range will result in the this signal being assignef the value of the LSBs of the requested address.

- user_{devfile}_addr_update (**OUT**) – This signal is high during one clock cycle as a result of a function call to lseek() on the host. The purpose of this signal is to give the application logic a chance to indicate that it needs time to prepare data for reading, as a result of the address' update. This is done by changing the 'empty' signal to high in response to such update.

  For this purpose, there is one exception to the rule that 'empty' can change to high only one clock cycle after a read cycle: It can also change to high on the clock cycle that is after the one when the 'update' signal was high.

  The following Verilog code is therefore correct:

```
always @(posedge bus_clk)
  if ( { ... memory is ready ... } )
    user_r_mydevice_empty <= 0;
  else if ((user_mydevice_addr_update) &&
           ( user_mydevice_addr > { ... some limit ...} ))
    user_r_mydevice_empty <= 1;
```

  And the same in VHDL:

```
process (bus_clk)
begin
  if (bus_clk'event and bus_clk = '1') then
    if ( { ... memory is ready ... } ) then
      user_r_mydevice_empty <= '0';
    elsif (user_mydevice_addr_update = '1'
          and user_mydevice_addr > { ... some limit ...} )
      user_r_mydevice_empty <= '1';
    end if;
  end if;
end process;
```

In this example we can also see, that the 'update' signal is high on the same clock cycle as the address is updated.

Note that since 'empty' can change to low at any time, it can make sense to change 'empty' to high as a result of every address update (regardless of the address), and then take the time to evaluate if 'empty' can be changed back to low.

The 'full' signal can also change to high in a similar manner, even though it's not clear why this should be useful.

When related device file on the host is closed, (i.e. when user_w_{devfile}_open and user_r_{devfile}_open are both low), the address is reset, and hence its value changes to zero. Note however that this is not considered an address update, i.e. user_{devfile}_addr_update remains low.

3.5 The quiesce signal

The quiesce signal is high when the hosts expects the IP core to be completely inactive (quiescent state). This is usually when:

- The host has not yet loaded the driver, or it has unloaded it.
- On Windows: When the host is about to enter hibernation.
- With XillyUSB: Also when the device isn't connected to a computer at all.

The intention of this signal is to serve as a synchronous reset, however it's most likely not necessary: One of the side effects of being in quiescent state is that all files are closed, so application logic can rely on the *_open signals alone as a reset signal. The 'quiesce' signal can be used as a more global form of reset.

4

Implementing data acquisition

4.1 Introduction

The need to capture data from an FPGA to a computer often occurs, for example for these needs:

- Frame grabbing from a video source
- Data samples from an analog to digital converters (ADC)
- Obtaining data from other data sources
- Receiving debug information from the FPGA

With applications like these, the data rate can be high, and the continuity of the data flow must be guaranteed: No data is allowed to get lost.

A data acquisition application is easily implemented with Xillybus by writing the data to a FIFO. This section focuses on how to implement the application logic that guarantees that the data that arrives to the host is contiguous. To accomplish this, the application logic stops the flow of the data at the point where the continuity is broken, and sends an EOF at this point. This way, the application software can rely upon that the data that arrives is indeed contiguous.

Ideally, this stopping mechanism should never become active, but when it does, it allows an awareness of the problem, as well as an opportunity to solve it.

In theory, it's impossible to ensure a sustained data rate between a peripheral and a computer, since the operating system may deprive the CPU from the application software for as long as it wants.

There are nevertheless methods for maintaining a continuous stream of data, which involve certain host programming techniques. This issue is discussed extensively in both programming guides:

- Xillybus host application programming guide for Linux
- Xillybus host application programming guide for Windows

In particular, pay attention to section 4 of these two guides, which discusses how to work with high data rates.

For high bandwidth applications, it's also recommended to refer to section 5 of one of these two guides, which contains several topics to be aware of:

- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host

In what follows, it's shown how Xillybus is used to capture 32-bit wide data from a continuous source. The emphasis in this section is on ensuring that all data that arrives to the host is a reliable copy of the captured data source.

4.2 Example code

The example code which is shown and explained below can be downloaded as a module from this link:

http://xillybus.com/downloads/xillycapture.zip

The zip file consists of two files, xillycapture.v and xillycapture.vhd. These are written in Verilog and VHDL, respectively. In order to try out the example, edit xillydemo.v or xillydemo.vhd: Disconnect the signals related to read_32 in the demo bundle, and insert the example code instead.

The example code makes an instantiation of a standard, 32-bit wide, dual clock FIFO. Before attempting to perform synthesis of the example code, generate this FIFO with the tools (e.g. Vivado or Quartus). The name of this FIFO should be async_fifo_32, and a depth of 512 is enough.

Note that in the example code, there's a signal named "slowdown". The purpose of this signal is to reduce the data rate of the fake data source. This signal should be removed when a real source of data is used.

4.3 FIFO connections

Let's assume that the data source is synchronous with capture_clk. Hence the data is simply fed to a standard dual-clock FIFO. This FIFO connects between the data source and the Xillybus IP core.

In Verilog:

```
 async_fifo_32 fifo_32
   (
    .rst(!user_r_read_32_open),
    .wr_clk(capture_clk),
    .rd_clk(bus_clk),
    .din(capture_data),
    .wr_en(capture_en),
    .rd_en(user_r_read_32_rden),
    .dout(user_r_read_32_data),
    .full(capture_full),
    .empty(user_r_read_32_empty)
    );
```

And in VHDL:

```
fifo_32 : async_fifo_32
  port map(
    rst         => reset_32,
    wr_clk      => capture_clk,
    rd_clk      => bus_clk,
    din         => capture_data,
    wr_en       => capture_en,
    rd_en       => user_r_read_32_rden,
    dout        => user_r_read_32_data,
    full        => capture_full,
    empty       => user_r_read_32_empty
    );

reset_32 <= not user_r_read_32_open;
```

This is quite similar to the demo bundle: The FIFO is reset when the file is closed, and its user_r_read_32_* signals are connected as before.

4.4 Capture control

The capture_en signal works as a write enable signal for the captured data. Capturing of data does not take place in one of the two following situations:

- When the file is closed
- When the FIFO is full or has been full in the past

So the condition for capture_en (in Verilog) boils down to:

```
assign capture_en = capture_open && !capture_full &&
                    !capture_has_been_full ;
```

And in VHDL:

```
  capture_en <= capture_open and not capture_full
                and not capture_has_been_full ;
```

The capture_open signal is a copy of user_r_read_32_open, but in the clock domain of capture_clk.

Other application-specific conditions, such as waiting for the beginning of a frame in video data, or waiting for a certain error condition when using data acquisition for debugging, can be added to this expression as required (by virtue of a logic AND).

The signal capture_has_been_full changes to high when the FIFO is full, and it returns to low only when the file is closed. So when the FIFO is full, the data capture stops, and doesn't resume as long as the file is opened.

IMPORTANT:

In the example code there is a different definition for capture_en, which helps slowing down a fake data source. When capturing a real signal, capture_en should be changed to the above.

Now to the code that implements capture_has_been_full in Verilog:

```
always @(posedge capture_clk)
  begin
    if (!capture_full)
      capture_has_been_nonfull <= 1;
    else if (!capture_open)
      capture_has_been_nonfull <= 0;

    if (capture_full && capture_has_been_nonfull)
      capture_has_been_full <= 1;
    else if (!capture_open)
      capture_has_been_full <= 0;
   end
```

And VHDL:

```
  process (capture_clk)
  begin
    if (capture_clk'event and capture_clk = '1') then
      if  ( capture_full = '0' ) then
        capture_has_been_nonfull <= '1' ;
      elsif  ( capture_open = '0' ) then
        capture_has_been_nonfull <= '0' ;
      end if;

      if (capture_full = '1' and capture_has_been_nonfull = '1') then
        capture_has_been_full <= '1' ;
      elsif  ( capture_open = '0' ) then
        capture_has_been_full <= '0' ;
      end if;

    end if;
  end process;
```

This is almost as one would expect: When the FIFO's capture_full goes high, capture_has_been_full goes high, and when the file closes it goes low. The other signal, capture_has_been_nonfull, solves a different issue: Since the FIFO keeps the 'full' signal high when it's reset, capture_has_been_full should be high only when capture_full has been low (meaning that the FIFO came out of reset) and then became high (meaning the FIFO was really full).

So this code is somewhat complicated, but quite straightforward once the principle is

understood.

4.5 Generating EOF

An end-of-file is generated when the two following conditions are met:

- All data in the FIFO has been consumed (i.e. has been read by the IP core).
- No more data will be written to the FIFO, because the it has been full in the past.

In Verilog, this is written as:

```
assign user_r_read_32_eof = user_r_read_32_empty && has_been_full;
```

And in VHDL (note that this is combinatoric):

```
user_r_read_32_eof <= user_r_read_32_empty and has_been_full;
```

As can be seen in the example code, has_been_full is capture_has_been_full after a clock domain crossing to bus_clk.

Note that user_r_read_32_eof goes from low to high only as allowed, to according to the API. This is because there is a logical AND with user_r_read_32_empty, as suggested in section 3.3.

4.6 A test run

IMPORTANT:

This test run deliberately shows a bad example of the IP core setting, so that the mechanism with the EOF comes to action: The IP core that was used for this test has small buffers and the stream is synchronous (which is wrong for a data acquisition application). Real-life tests work significantly better.

In order to ensure repeatability of the captured data, the data source is chosen as a bogus data generator, which just counts the number of sent words. The amount of data until EOF depends on when the computer became busy doing something else, and momentarily neglected the sequence of reads from the device file.

The test run is shown for Linux, but it can be run on Windows as well. More about running command line utilities can be found in either of these guides:

- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host

This is what a test run can look like:

```
$ cat /dev/xillybus_read_32 > first
$ cat /dev/xillybus_read_32 > second
$ ls -l
total 77740
-rw-rw-r--. 1 liveuser liveuser 71727100 Jul 13 15:31 first
-rw-rw-r--. 1 liveuser liveuser  7874556 Jul 13 15:31 second
```

So about 71 MB were captured on the first attempt, but only 7 MB on the second one. The amount of data in each run depends on how much data was received before the operating system neglected the reading process to do something else. Most likely, the read process was stopped briefly in order to write to the disk.

But even when discarding all data by sending it to /dev/null, it will eventually stop (try “man dd” for more about the dd utility):

```
$ dd if=/dev/xillybus_read_32 of=/dev/null bs=1M
0+34365 records in
0+34365 records out
140756988 bytes (141 MB) copied, 18.0364 s, 7.8 MB/s
$ dd if=/dev/xillybus_read_32 of=/dev/null bs=1M
0+6027 records in
0+6027 records out
24684540 bytes (25 MB) copied, 3.16028 s, 7.8 MB/s
```

In both of these two tests, moving the mouse stopped the data flow. This distracted the operating system enough.

Once again it’s important to emphasize: These are really bad results, because a synchronous stream is used. With an asynchronous stream, and the correct amount of DMA buffers, problems of this sort are not expected at all.

And finally, we’ll look what’s in one of the captured files:

```
$ hexdump -C -v first | head
00000000  f8 fb a2 01 f9 fb a2 01  fa fb a2 01 fb fb a2 01  |................|
00000010  fc fb a2 01 fd fb a2 01  fe fb a2 01 ff fb a2 01  |................|
00000020  00 fc a2 01 01 fc a2 01  02 fc a2 01 03 fc a2 01  |................|
```

```
00000030  04 fc a2 01 05 fc a2 01  06 fc a2 01 07 fc a2 01  |................|
00000040  08 fc a2 01 09 fc a2 01  0a fc a2 01 0b fc a2 01  |................|
00000050  0c fc a2 01 0d fc a2 01  0e fc a2 01 0f fc a2 01  |................|
00000060  10 fc a2 01 11 fc a2 01  12 fc a2 01 13 fc a2 01  |................|
00000070  14 fc a2 01 15 fc a2 01  16 fc a2 01 17 fc a2 01  |................|
00000080  18 fc a2 01 19 fc a2 01  1a fc a2 01 1b fc a2 01  |................|
00000090  1c fc a2 01 1d fc a2 01  1e fc a2 01 1f fc a2 01  |................|
```

As expected, the data contains a counting up sequence. The counter, which is used for generating bogus data is never reset, which is why the sequence doesn't start at 0.

4.7 Monitoring the amount of buffered data

It's often desired to keep track on how much data is held in Xillybus' buffers that belong to a certain stream. This can help in controlling latency, preventing overflow or underflow, or to prevent the application software from sleeping during function calls to read() or write().

For example, in the direction from the FPGA to host, this means knowing how much data has been read from the FIFO in the FPGA (by the Xillybus IP core), which has not yet been consumed by the application software on the host.

Likewise, in the opposite direction, this means knowing how much data the application software has written to the stream, and has not reached the FIFO in the FPGA (because the FIFO is full, waiting for data to be consumed by the application logic).

Xillybus doesn't provide a dedicated feature for this, partly because there's a simple way to implement this using Xillybus' features, as shown next.

To explain the suggested solution, let's say that the 32-bit stream from the FPGA to the host in the demo bundle is used for data acquisition.

The following counter is used to count the number of data elements that were fetched from the FIFO by the IP core since the file was opened:

```
reg [31:0] count_data;

always @(posedge bus_clk)
  if (!user_r_read_32_open)
    count_data <= 0;
  else if (user_r_read_32_rden)
    count_data <= count_data + 1;
```

The value of count_data can be exposed to the host through another, dedicated Xillybus stream (from the FPGA to the host) for this purpose: This is done by connecting count_data directly to this other stream's data port (i.e. the port that is usually connected to a FIFO's data output).

The 'eof' port and 'empty' port should be held constantly low. This dedicated stream is configured to be synchronous, by setting its "use" parameter in the IP Core Factory to "Command and status".

By doing this, application software can read 4 bytes from this stream at any time, to get the updated value of count_data.

Alternatively, and when adequate, count_data can be a register in an array of registers, as suggested in section 3.4.

Note that count_data is synchronous with bus_clk, and can therefore be connected directly to the data portof the Xillybus IP core.

Once this value is known to the software by virtue of this extra stream, the amount of data in the buffers can be calculated as the difference between count_data, and the amount of data that the application software has read from its device file since it was opened (i.e. /dev/xillybus_read_32 in this example). This requires, of course, that the software keeps track of the amount of data that it reads from this stream.

In the opposite direction, from host to FPGA, a similar counter can be maintained in the FPGA with

```
reg [31:0] count_data;

always @(posedge bus_clk)
  if (!user_w_write_32_open)
    count_data <= 0;
  else if (user_w_write_32_wren)
    count_data <= count_data + 1;
```

By the same principle, the application software keeps track on how much data it writes to the relevant device file, and informs itself on how many elements have been written into the FIFO on the FPGA. This is done by reading the value of count_data.

In both case studies above, the data in the FIFOs wasn't included in the calculation, but only the data that Xillybus keeps in buffers. Sometimes it's desired to get an end-to-end number, including the data that is stored in the FIFOs. For this purpose, the operations on the opposite side of the FIFOs should be counted, i.e. the number of elements that are written to the FIFO in the first case, and the number of elements

that are read from the FIFO in the second case.

This might however be harder to implement, if the other side of the FIFO is synchronous with a clock that isn't bus_clk (e.g. capture_clk as presented previously in this section). That is because count_data is accordingly synchronous with this other clock as well. As a result, a clock domain crossing is necessary to connect count_data's value to the IP core. There is hence a tradeoff between accuracy and simplicity, unless bus_clk itself is the clock that is used for data acquisition or playback.

5

Suggested simulation practices

5.1 General

What is a satisfactory simulation is a matter of taste and working practices. Nevertheless, there are always assumptions in a simulation that certain things work as expected. There can also be certain things that are interesting to simulate, but are too complex or time consuming to carry out.

This section suggests a set of assumptions and limitations on the simulation process, as well as an approach for simulating a system involving the Xillybus IP core. These guidelines are by their nature less imperative than those in the rest of this document.

The Xillybus IP core and its driver are a complex system, which has been stress-tested in various scenarios. It's therefore unlikely to find bugs in the IP core itself by simulation, if they weren't found with terabytes of data transport, with a large range of load patterns.

Moreover, the IP core's behavior depends very much on the response from the host: Both the driver and the application software respond in different ways and with different delays, which are nearly unpredictable. On top of that, the latency of the bus (PCIe, AXI or USB) is likewise random and hence unpredictable. A comprehensive simulation is therefore nearly impossible.

In light of this, it's recommended to simulate application logic up to the point where the FIFO connects with the Xillybus IP core. Accordingly, The IP core is simulated as a black box which drains this FIFO or fills it, depending on the data's direction.

5.2 Simulating asynchronous streams

When a stream is configured to be asynchronous, the IP core transfers data from or to the FIFO (depending on the stream's direction) so that the FIFO never reaches the state of overflow or underflow.

This holds true as long as the application software on the host performs I/O operations often enough, and Xillybus' bandwidth capability is adequate for its mission. These two conditions are the result of a properly designed project, and it can be beneficial to verify them by virtue of a simulation. There are two aspects to look at:

- Whether the FIFO reaches overflow or underflow (depending on the direction).
- Whether the application logic responds correctly to such faulty situation, e.g. as suggested in section 4.5.

To simulate proper operation, it can be assumed that the IP core transfers data to or from the FIFO at the maximal rate, as long as the related 'open' signal is high (indicating that the file is opened by the host).

For testing a stream from the host to the FPGA, on what happens when the FIFO suffers from an underflow, it's recommended to simulate this event by making the FIFO appear to become empty. For example, if the FIFO is part of the test bench, it changes the 'empty' signal (that is connected to the application logic) to high. Alternatively, the part of the test bench that simulates the data flow from the host may simply stop to push data into the FIFO for a period of time, which causes it to become empty.

Likewise for a stream from FPGA to host, the 'full' line can be changed to high, to test a FIFO's overflow. Or, alternatively, the test bench can stop fetching data from the FIFO for a period of time, yielding the same effect.

One possibility for breaking the data continuity is because the application logic attempts to exceed the bandwidth limitation of the stream (or that of the IP core's total bandwidth). If this is a possibility (in many applications it isn't), it's also recommended that the test bench simulates the bandwidth limitation. It can do so by filling or emptying the FIFO with a data rate that is limited by the stream's intended bandwidth.

5.3 Simulating synchronous streams

Compared with an asynchronous stream, a synchronous stream is different (for simulation purposes) in that the IP core's data flow isn't continuous: The IP core transfers

data to or from the FIFO only when there's a pending function call on the host (read() or write()).

The IP core's behavior is therefore dictated, to a larger extent, by the application software's requests for I/O. Accordingly, the part of the test bench that simulates the IP core must be written with the application software's access pattern in mind.

Because a synchronous stream is the less preferred option when the purpose of the stream is to exchange large amounts of data, the possibility for overflow or underflow may not be as relevant. The methods for simulating these conditions is nevertheless the same as with asynchronous streams.

5.4 A simplified simulation method

Those who are not interested in simulating conditions of overflow and underflow, have a simpler option for simulating the IP core. For example, in the host to FPGA direction, the FIFO can be implemented in the test bench simply by reading the data word from a file for each rising clock edge when the read enable signal is high. This simplified view of the FIFO relies on the assumption that the host will always write data into the FIFO quickly enough to ensure that it never gets empty.

In the opposite direction, the test bench writes the word to a file when the write enable signal is high. As before, this assumes that the host always reads the data from the FIFO quickly enough to ensure it never gets full.

This is approach doesn't overlook the possibility that the continuity can break. Rather, it recognizes that a broken data continuity is most probably a result of something beyond the simulation's scope: Too shallow DMA buffers, poor responsiveness of the application software, or a CPU deprivation resulting from an overall condition on the host. If such event happens for real, the application logic should make the host aware of that. As already suggested above, this mechanism can be simulated.

This approach does however disregard the possibility that the application logic attempts to exceed the bandwidth limitation of the stream. If such scenario is a realistic possibility, this simplified simulation option may not be adequate.