Xillybus FPGA designer's guide

Xillybus Ltd.

www.xillybus.com

Version 3.1

1

Introduction

Xillybus' IP cores are intended to interface with user application logic through a FIFO or a dual-port block RAM. Therefore, it is usually possible to work with the IP core without understanding the API in detail.

The vast majority of the API's rules can be deduced from a simple principle: The user application logic should behave exactly like a FIFO or a block RAM. This is true even when the application logic interfaces with the IP core directly.

Among others, this principle means that the exchange of data between the Xillybus IP core and the user application logic takes place at a pace that is dictated by the IP core. The data flow may stop momentarily and resume later in an unpredictable manner. In other situations, the data flow will be continuous. This poses no problem when the IP core is connected directly to a FIFO or a block RAM. Likewise, user application logic that interfaces directly with the IP core should also operate properly even when the data flow starts and stops in an unpredictable manner.

It's a common mistake to consider the IP core's irregular access pattern as a bug. Unexplainable pauses in the data flow between the IP core and the FIFO may seem suspicious, but they don't indicate any malfunction.

It's not recommended to interface with the IP core directly (i.e. without a FIFO or block RAMs) in order to reduce the logic consumption. This holds true in particular in the early stages of the design. If the user application logic is connected directly to the IP core, the irregular data flow may expose bugs in the application logic.

This guide describes the API that is relevant for interfacing application logic directly, and also elaborates on popular applications.

It's recommended to gain an initial experience with Xillybus before delving into the details that are presented in this guide. Refer to these documents:

- 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

This guide also assumes that you understand the difference between synchronous and asynchronous streams. This is discussed in section 2 of either of these two documents:

- 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 guide, 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. This clock is supplied by the IP core.

For Xillybus IP cores that are based upon PCIe, this clock is generated by the PCIe block. The clock's frequency depends 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. This depends on if the maximal bandwidth (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.

There is often a possibility to change the clock's frequency within a limited list of choices. This is done by configuring the PCIe block or the processor core that generates the clock.

If the timing constraints for the PCIe block are set correctly (as in the demo bundles), the application logic that relies on bus_clk is covered by proper timing constraints as well: The tools automatically create timing constraints for bus_clk that are based upon the timing constraints of the PCIe block. The same applies for Zynq-based platforms as well as XillyUSB.

This is not to say, that the application logic needs to be synchronous with bus_clk. Likewise, it's not required that the source of the data or that the data's destination is synchronous with bus_clk. When a different clock is involved, a dual-clock FIFO is often used together with the IP core: One side of the FIFO is connected to the

Xillybus IP core. This side is therefore synchronous with bus_clk. The application logic is connected to the FIFO's other side. This side is synchronous with the application logic's clock. Hence the FIFO is used not only as a short-term temporary storage, but also for clock domain crossing.

2.2 Data width

Each FIFO or memory interface works with data in widths of 8 bits, 16 bits or 32 bits. This is true with baseline Xillybus IP cores (revision A). 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. This implements 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 this case, a dual-clock FIFO (often called "asynchronous FIFO") should be used.

When a FIFO is used for a stream from the host to the FPGA, this FIFO's "full" signal should be connected to the Xillybus IP core. The IP Core uses this signal to determine whether a burst of data transfer can be initiated.

The same principle applies to a stream from the FPGA to the host: The "empty" signal should be connected to the Xillybus IP core for the same purpose. The IP core expects the behavior of a regular FIFO (as opposed to FWFT, First Word Fall Through).

Once a burst has started, the Xillybus IP core continues to rely on these signals ("empty" and "full"): These signals prevent the IP core from reading from an empty FIFO or writing to a full FIFO.

However, even if a FIFO indicates that it is ready for a burst of data, the Xillybus IP core may not start a burst immediately. The IP core may also stop a data burst in the middle, even if the FIFO allows continuing the burst. It is normal for the pattern of the

data flow to be apparently random.

The general rule is that the Xillybus IP core attempts to equally serve all FIFOs that are connected to it. The IP core grants longer bursts to FIFOs that tend to get filled faster, as these FIFOs don't activate their "empty" or "full" as often.

This simple arbitration method ensures efficient communication with FIFOs that tend to get filled rapidly. At the same time, a low latency on FIFOs that receive data at a lower rate is achieved.

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

A depth of 2 kBytes is usually enough, because the Xillybus core is not likely to neglect a FIFO of this size for a time period that is long enough to cause an overflow or underflow. This is of course true as long as the user application software that runs on the host consumes or supplies data rapidly enough. 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.

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 change to low at any moment, of course.

The Xillybus IP core relies on this behavior: When a FIFO indicates that it is ready for a data transfer (with a low "empty" or "full", as applicable), a state machine in the IP core may start a chain of events. This 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 will ignore the "empty" signal during one clock cycle. Such an event is harmless regarding the IP core's own integrity, but may lead to an unexpected and unpredictable data flow.

The same applies to the "full" signal: If this signal changes from low to high before the IP core writes a data word to the FIFO, the IP core may ignore the "full" signal during one clock cycle. Once again, this is harmless to the IP core itself, but may result in the loss of one data word.

A properly designed FIFO can create this faulty condition only if it is reset at the same time that it is ready for communication with the Xillybus IP core. This situation should normally be avoided anyhow.

If application logic is connected directly with the IP core (without a FIFO), it's important to imitate the behavior of a standard FIFO regarding "empty" or "full".

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 divided 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 with regards to the choice of “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 name 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 IP 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. All standard FIFOs behave like this, so this rule is relevant 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 may 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 have 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 is allowed to change only when a FIFO would have changed it. 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, user_r_{devfile}_data must contain valid 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. All standard FIFOs behave like this, so this rule is relevant 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. The result of a high 'eof' is also that the core will not read from the FIFO anymore (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 all data that the IP core has received before this signal became high. Only then will the host 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.

  Similar to the 'empty' signal, the 'eof' signal is allowed to change 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' conforms to this rule is to define 'eof' as the output of a combinatorial function. In Verilog, it may be written like this:

```
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 may 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 have 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, regardless of 'eof'. However, note that application software typically responds to an EOF by closing the file. It's therefore easy to mistakenly believe that there exists a connection between these signals.

3.4 Memory interface signals

A Xillybus device file can be configured to have an address signal. The application software assigns a value to this signal by using the standard API for seeking in a file (e.g. lseek()). Also, an increment of the address occurs automatically on the FPGA as a result of read cycles and write cycles.

A standard block RAM is easily connected with the IP core. This is done by using the signals that have already been mentioned above with relation to FIFOs and the signals that are detailed below. The result is that the block RAM's memory array is 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 of the memory array: This depends on the length of the read or write operations.

It is also possible to implement an array of registers that behaves like a block RAM on the FPGA. By doing so, these registers become easily accessed by the host.

The 'empty' and 'full' signals can be used to slow down read and write operations for

memories that require wait states, or when there is another reason to briefly delay operations.

A memory interface requires two additional signals:

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

  The address' value returns to zero after a read or write operation at the maximal address (depending on the width of the address signal). If the value of a function call to lseek() is out of range, only the LSBs are copied to this signal.

- 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 'update' signal is high on the same clock cycle as the 'addr' signal has its updated value.

  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 'update' 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;
```

Note that since 'empty' can change to low at any time, it's reasonable to change 'empty' to high as a result of every address update (regardless of the address), and then let the logic 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 the 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 host expects the IP core to be completely inactive (quiescent state). This is usually when:

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

The intention of this signal is to be used as a synchronous reset, however this signal is most likely not necessary: When the IP core is in an inactive state (i.e. quiescent state), all files are closed. Accordingly, the 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:

- Frame grabbing from a source of a video signal.
- Data from an analog to digital converter (ADC).
- Receiving debug information from the FPGA.

The data rate can be high for applications like this. Nevertheless, the continuity of the data flow must be guaranteed: No loss of data is allowed.

A data acquisition application is easily implemented with Xillybus by writing the data to a FIFO. This section focuses on how to guarantee that the data that arrives to the host is contiguous.

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. The first and obvious condition to achieve this goal is to use a Xillybus stream that is capable to transport the data at the required rate. On top of that, certain host programming techniques should be used. 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

But even if the design is carried out perfectly, there is always a possibility that the continuity of the data stream is broken: The nature of the operating system is that it's allowed to deprive the CPU from the application software for a long period of time.

So the first goal is to make sure that the continuity of the data stream is practically never broken. The second goal is to ensure that if this happens despite all efforts, this event is noticed. Even more important, that all data that arrives to the host is guaranteed to be contiguous.

In order to accomplish this second goal, the application logic should stop the flow of the data at the point where the continuity is broken. An EOF is sent to the host after this point, in order to tell the host that something went wrong. 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 what follows, it's shown how Xillybus is used to capture 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 data source.

4.2 Example code

The example code that 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 that are related to read_32 in the demo bundle, and insert the example code instead.

The example code performs an instantiation of a standard dual clock FIFO. The width of this FIFO is 32 bits. 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. A depth of 512 words 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. Accordingly, the data is connected the regular way 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 in the demo bundle.

4.4 Data acquisition control

The capture_en signal works as a write enable signal. There are three situations that prevent writing data to the FIFO:

- When the file is closed
- When the FIFO is full
- When the FIFO has been full in the past, since the file was opened

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 for the clock domain of capture_clk.

In a real-life application, there are often other conditions for writing to the FIFO. For example, waiting for the beginning of a video frame, or waiting for a specific error condition (when using data acquisition for debugging). This kind of conditions 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 acquisition stops and doesn't start again as long as the file remains open.

IMPORTANT:

In the example code there is a different definition for capture_en, which helps slowing down the fake data source. For a real application, 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;
```

When the FIFO's capture_full goes high, capture_has_been_full goes high. When the file closes, capture_has_been_full goes low.

The other signal, capture_has_been_nonfull, solves a different issue: The FIFO's 'full' signal is high as long as the FIFO is reset. When the 'full' signal is high because of this reason, capture_has_been_full should not be high. In other words, 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 a bit 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. all data has been read by the IP core).
- No more data will be written to the FIFO, because the FIFO 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 a combinatorial function):

```
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 copies the value of capture_has_been_full by virtue of a clock domain crossing to bus_clk.

Note that user_r_read_32_eof goes from low to high as allowed by 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 an unsuitable configuration of the IP core. The purpose of this deliberate mistake is to demonstrate how the EOF comes to action. The IP core that was used for this test had small buffers with a synchronous stream. These are incorrect choices for a data acquisition application. A properly configured IP core will not perform as poorly as shown below.

In order to ensure repeatability of the transmitted data, the data source is chosen as a simple counter, which counts the number of sent words. The amount of data until EOF is random: The EOF happened when the computer became busy doing something else, and momentarily neglected the task of reading 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 collected on the first attempt, but only 7 MB on the second attempt. The amount of data in each run depends on how much data was received before the operating system neglected the reading process, in order 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 computer’s 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 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 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 specific 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, with relation to the data flow from the FPGA to host: There may be an amount of data that is stored in the buffers, because the IP Core has read this data from the FIFO in the FPGA, but the application software has not consumed this data yet. It's often desirable to know how much data is waiting like this.

Likewise, in the opposite direction: There may be data that the application software has written to the stream, but it has not reached the FIFO in the FPGA yet. The direct reason is that the FIFO in the FPGA is full, so no more data can be accepted from the IP core. However, the real explanation is that the data is waiting to be consumed by the application logic.

Xillybus doesn't provide a dedicated feature for estimating the amount of data in the buffers. However, there's a simple way to implement this functionality by using Xillybus' existing features, as shown next.

To explain the suggested solution, let's say that one of the streams in the demo bundle (FPGA to host, 32 bits) is used for data acquisition.

The following counter is used to count the number of data words 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;
```

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

An alternative solution is to add another Xillybus stream (from the FPGA to the host) to the IP core. This stream is used to send the value of count_data to the host by connecting count_data directly to data port of this additional stream (i.e. the port that is usually connected to a FIFO's data output).

The 'eof' port and 'empty' port of this stream should be held constantly low. This

stream should be configured as a synchronous stream, by setting the “use” parameter in the IP Core Factory to “Command and status”. As a result, the application software can read 4 bytes from this stream at any time, in order to get the updated value of count_data.

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

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). The software must keep track on the amount of data that it reads from this stream, of course.

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;
```

This works by the same principle: The application software keeps track on how much data it writes to the relevant device file. The application software reads count_data when there is a need to know how much data is stored in the buffers. This amount of data is calculated as the difference between how much data has been written (to the device file since it was opened) and the value of count_data.

Note that in the discussion so far, the data in the FIFOs wasn’t included in the calculation: Only the data that Xillybus keeps in its buffers was taken into account. 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. In other words, this is the number of elements that are written to the FIFO for a stream from the FPGA to the host. In the opposite direction, this is the number of elements that are read from the FIFO.

However, if the other side of the FIFO is synchronous with a different clock (e.g. capture_clk as presented previously), this might be harder to implement. That is because count_data needs to be 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. Hence there is a tradeoff between accuracy and simplicity when two different clocks are con-

nected to the FIFO.

5

Suggested methods for simulation

5.1 General

What is a satisfactory simulation is a matter of taste and working methods. Nevertheless, assumptions are always made in relation to a simulation. These assumptions include the expectation that specific functional elements work as expected. It is therefore pointless to examine these functional elements with a simulation. There can also be certain functional elements that would be beneficial to simulate, but doing so is too complex or time consuming.

This section suggests a few assumptions and limitations on the simulation process. An approach for the simulation of a system that involves the Xillybus IP core is also discussed. These guidelines are by their nature more open for discussion than those in the rest of this document.

The Xillybus IP core and its driver are a complex system, which has been tested extensively in various scenarios. It's therefore unlikely to find bugs in the IP core itself with the help of a simulation: If a bug wasn't found with terabytes of data transport and with a large range of usage patterns, a simulation is unlikely to reveal such a bug.

Moreover, the IP core's behavior depends to a large extent 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 perform simulation of 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. It can be beneficial to verify two aspects of the design by virtue of a simulation:

- 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 a stream from the host to the FPGA, it's beneficial to test 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, the test bench 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. The FIFO becomes empty as a result of this.

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 possible reason for breaking the continuity of the data stream is that the application logic attempts to exceed the bandwidth limitation of the stream (or the limitation on the IP core's total bandwidth). If this possibility exists, it's also recommended that the test bench simulates the bandwidth limitation. This can be done by making sure that the test bench (which acts as the IP core) fills or empties the FIFO with a data rate that is limited by the stream's intended bandwidth.

However, note that in many applications a simulation of this sort is unnecessary, because the application logic isn't capable of exceeding the bandwidth limitation.

5.3 Simulating synchronous streams

For the purpose of a simulation, the main difference of a synchronous stream is that the IP core's data flow isn't continuous: With a synchronous stream, 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 more dependent on 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.

It may be irrelevant to simulate overflow or underflow for a synchronous stream, because a synchronous stream is the less preferred option when the purpose of the stream is to exchange large amounts of data. The methods for simulating these conditions is nevertheless the same as with asynchronous streams.

5.4 A simplified method for simulation

There is a simpler method for the simulation of the IP core if there is no interest in testing the response to an overflow and underflow. 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 prevents the FIFO from becoming empty by writing data to the relevant device file quickly enough.

In the opposite direction, the test bench writes the word to a file when the write enable signal is high. Similar to before, there is an assumption that the host always prevents the FIFO from becoming full, by reading data quickly enough.

This is approach doesn't overlook the possibility that the continuity of the data flow can break. Rather, this approach recognizes that a broken data flow is most probably a result of something that is 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 method for simulation may not be adequate.