Xillybus FPGA designer's guide

Xillybus Ltd.

www.xillybus.com

Version 2.1

1

Introduction

Xillybus is designed to interface with user application logic having a FIFO or synchronous memory as mediators. 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 covered by a simple rule: Implement logic that behaves exactly like a FIFO or a synchronous RAM. This rule also implicitly says that no assumptions should be made on how and when the Xillybus IP core accesses the logic connected to it. Data accesses can be continuous or with gaps of arbitrary length at any given moment. A FIFO or memory will not have any problems with this, and so shouldn't any logic interfacing directly with the IP core.

Because of the possibility to miss special conditions when designing logic which interfaces directly with the IP core, it's recommended not to do so for the sake of saving logic resources, at least not in the initial design. Unless the desired functionality requires a direct interface, FIFOs or RAMs should be used in order to avoid buggy designs.

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 Altera
- 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

For the 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 clocked with the bus_clk supplied by the core itself. This clock is generated by the PCIe or processor core, and has a frequency depending on the platform: PCIe-based platforms have 62.5 MHz, 125 MHz or 250 MHz clocks, depending on the stated maximal bandwitdh: 200 MB/s, 400 MB/s or 800 MB/s, respectively. Zynq-based platforms typically have a clock of 100 MHz.

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

If the constraints for the PCIe (or ARM processor) core are set correctly (e.g. as in the demo bundles), logic driven by this clock is time constrained properly as well.

This is not to say, that data sources and sinks need to be clocked with bus_clk. In a typical application, one side of an asynchronous FIFO is connected to the Xillybus core, and is therefore clocked with bus_clk. The application logic is then connected to the FIFO's other side along with its own clock. Hence the FIFO is used for clock region crossing as well as a short-term temporary storage.

2.2 Data width

Each FIFO or memory interface works with data in widths of 8 bits, 16 bits or 32 bits.

Wider data allow higher bandwidth performance and is also more convenient in designs 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's width), as read() and

write() operations 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 by having a FIFOs both ends 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 its both sides. This is not suitable when the FIFO functions as a means for crossing clock domains, in which case a FIFO with independent clocks should be used.

Depending on the data direction, the FIFO's 'empty' or 'full' signal is connected to the Xillybus IP core, and is sensed to determine whether a burst of data should be initiated. Once a burst has started, these signals are sensed to make sure 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 how soon a burst is initiated on a FIFO which signals it's ready for it. Neither should any assumption be made on the length of such a burst. The general rule is that the Xillybus 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 assert their "empty" or "full" lines during these bursts.

This simple arbitration method ensures efficient communication with FIFOs that tend to get filled rapidly, and at the same time low latency on FIFOs that receive little data at a time.

The Xillybus IP core works with any FIFO size, but this attribute should be set to cope with the expected data flows. Even though this is sometimes a matter of trial and error, a 2 kByte deep FIFO is almost always the correct choice for an asynchronous stream, even for high data rates. The sense behind this choice is that the Xillybus core is not likely to neglect a FIFO long enough to let a FIFO of this size overflow or underflow (depending on the direction) if the corresponding DMA buffers allow data flow. If the DMA buffers are overflowed or underflowed, the necessary FIFO depth necessary to compensate for this is not reasonable for on-FPGA memory.

The core 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 go high only one clock cycle after the read enable was asserted. Likewise, the “full” signal can go high only one clock cycle after the write enable was asserted.

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

The Xillybus IP core relies on this behavior: When a FIFO signals to the core that it’s ready for a data transfer (by a low “empty” or “full”, whichever applies), a state machine in the core may start a chain of events which will lead to the transfer of at least one data element, regardless of the FIFO’s signals at that moment.

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.

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 core is interfaced directly by application logic, care must be taken to imitate the standard FIFO regarding this issue.

3

Signal description

3.1 FPGA signal naming convention

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

1. The “user” prefix marks all user interface signals.
2. The “w” flag indicates this signal belongs to a host-to-FPGA interface (host “write”). FPGA-to-host interfaces have an “r” instead. Address signals don’t have this flag, since they apply to both directions. Note that the host’s viewpoint is taken for this flag.
3. The “write_32” strings appears in the respective device file’s name, /dev/xillybus_write_32
4. The suffix signifies the signal’s meaning

In the remainder of this section, the device file name (component #3) is denoted {devfile} to avoid confusion.

3.2 Signals for host to FPGA transmission

- user_w_{devfile}_data – This core output signal contains data during write cycles. As mentioned above, this signal’s width can be 8, 16 or 32 bits, depending on the respective device’s configuration.
- user_w_{devfile}_wren – This core output signal is a write enable signal to the FIFO: It is asserted (logical ’1’) in conjunction with valid data being present on

the user_w_{devfile}_data signal, and tells the receiving party (e.g. FIFO) to sample this data.

- user_w_{devfile}_full – This core input signal informs the core that no more data can be accepted. When asserted properly, it temporarily assures no write cycles occur. Important: The 'full' signal may transition from '0' to '1' only on the clock cycle following a write cycle. This is the way standard FIFOs behave, so this rule needs attention only if the IP core is interfaced directly with application logic (i.e. no mediating FIFO). The reason for this rule is that the Xillybus logic treats an non-asserted 'full' signal as a green light to start a data transaction with the host. Failing to observe this rule may cause sporadic writes overriding 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; // Deassert 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'; -- Deassert any time
    elsif (user_w_mydevice_wren = '1' and { some condition } )
      user_w_mydevice_full <= '1'; -- Assert only with wren
    end if;
  end if;
end process;
```

- user_w_{devfile}_open – This core output signal is asserted ('1') when the respective device file in the host is open for write (a read-only open, when allowed, will not assert this signal). This signal can optionally be used to reset the FIFO or other logic between file opens (used as a reset active low). If a file is opened by multiple processes in the host (as a result of a fork() or when nonexclusive open is allowed), this signal remains asserted until all open instances are closed.

3.3 Signals for FPGA to host transmission

- user_r_{devfile}_data – This core input signal contains data during read cycles. As mentioned above, this signal's width can be 8, 16 or 32 bits, depending on the respective device's configuration. This signal must not change except for as a result of a read enable signal, and on the clock following it (this is the normal behavior of a standard FIFO).
- user_r_{devfile}_rden – This core output signal is a read enable signal to the FIFO: When asserted (logical '1'), the core expects valid data to be present on user_r_{devfile}_data on the following clock.
- user_r_{devfile}_empty – This core input signal informs the core that no more data can be read. When asserted properly, it temporarily assures no read cycles occur. The 'empty' signal may transition from '0' to '1' only on the clock cycle following a read cycle. This is the way standard FIFOs behave, so this rule needs attention only if the IP core is interfaced directly with application logic (i.e. no mediating FIFO). The reason for this rule is that the Xillybus logic treats an non-asserted 'empty' signal as a green light to start a data transaction with the host. Failing to observe this rule may cause sporadic reads overriding the 'empty' condition.

  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; // Deassert any time
  else if (user_r_mydevice_rden && { ... some condition ... } )
    user_r_mydevice_empty <= 1; // Assert 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'; -- Deassert any time
    elsif (user_r_mydevice_rden = '1' and { some condition } )
      user_r_mydevice_empty <= '1'; -- Assert only with rden
    end if;
  end if;
end process;
```

- user_r_{devfile}_eof – This core input signal tells the core to generate an end-of-file event. It's like an 'empty' signal, but once asserted, the core will not issue any more read cycles until the file is closed and reopened. On the host side, the application reading from the file descriptor will be informed that the file has reached EOF when all data has been consumed.

  Like the 'empty' signal, the 'eof' signal must be asserted only on a clock cycle following a read cycle. An exception for this is when the 'empty' signal is already asserted, in which case 'eof' may be asserted at any clock cycle. This exception can be used to make a blocking read() call at the host return immediately. So one way to assure that 'eof' is asserted correctly, is to have it combinatorically ANDed with the 'empty' signal.

  Asserting 'eof' with an unallowed timing will generate an EOF condition, but some data leaks may occur in its vicinity: Some data may be lost just before the EOF, or excessive junk data may appear at the host before the stream ends, or even after the the OS has returned an EOF condition to the user application.

  Once asserted, the 'eof' signal can be deasserted on any following clock, or kept high – the core latches the EOF request until the file is closed. The 'empty' signal may and may not be asserted in conjunction with an 'eof' assertion.

  Note that when asserted as required above, the 'eof' signal works in the sensible way: All data that was read by the Xillybus IP core before its assertion will reach the host before the application receives an end-of-file condition.

- user_r_{devfile}_open – This core output signal is asserted ('1') when the respective device file in the host is open for read (a write-only open, when allowed, will not assert this signal). This signal can optionally be used to reset the FIFO or other logic between file opens (used as a reset active low).

  If a file is opened by multiple processes in the host (as a result of a fork() or when nonexclusive open is allowed), this signal remains asserted until all open instances are closed.

  There is no direct connection between the 'eof' signal and the 'open' signal. The 'open' signal will deassert when the file is closed on the host, not when the 'eof' signal is asserted, nor when the EOF condition is delivered on the host.

3.4 Memory interface signals

A Xillybus interface can be configured to also have an address signal. The address is automatically incremented on read and write cycles, and can be set to an arbitrary value by host, using the standard mechanism for seeking files.

Alongside with some of the signals mentioned above, a standard RAM is easily interfaced with the core, making the the RAM's memory array available to the host as a seekable file: Read and writes to the file result in reads and writes to the memory array. The host may access single memory elements or segments, depending on the length of the read or write operations.

Also, by "faking" the memory array with registers, these registers become easily accessed by host.

The 'empty' and 'full' signals can be used to slow down reads and writes to memories requiring wait states, or memories that require some setup before access.

These are the two signals involved:

- user_{devfile}_addr – This core output signal contains the current address. When either a read enable or a write enable is asserted, this is the address to be read from or written to. Connecting this signal directly to a RAM's address input will work as naturally expected. The width of this signal is configurable up to 32 bits.

  The address wraps to zero when a read or write operation goes beyond the maximal address possible for the given width. Seek requests out of range will result in the address signal taking the value of the seek request's LSBs.

- user_{devfile}_addr_update – This core output signal is asserted as a result of a seek request from host. The purpose of this signal is to give user logic a chance to indicate that it needs time to prepare data for reading, by asserting the respective 'empty' signal as a result of an address update.

  To make this signal useful, there is one exception for the rule that 'empty' must be asserted only a clock cycle after an read cycle: It can also be asserted one clock cycle after an update signal.

  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 address is updated on the same clock cycle for which the update signal is asserted. Note that since 'empty' can be deasserted at any time, it makes sense, if this simplifies the design, to assert 'empty' as a result of every address update, regardless of the address, and then take the time to evaluate if 'empty' can be deasserted.

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

3.5 The quiesce signal

The quiesce signal is asserted ('1') when the host has not turned on the Xillybus interface (e.g. driver not loaded yet), or has turned it off (driver has been unloaded or the host is about to hibernate on Windows). Its intention is to serve as a synchronous reset.

It's most likely not necessary, though: One of the side effects of being in quiescent mode is that all files are closed, so user logic could 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

Whether data acquisition is the actual application, or it's used for debugging purposes, the need to capture data from an FPGA to a computer often occurs. Be it frame grabbing, sampling data from analog converters (DAC) or application data from within the FPGA, the data rate can be high, and the continuity of the captured stream must be guaranteed.

In its simplest form, the application logic for implementing data acquisition merely consists of writing the data to a FIFO. This section focuses on how to implement the capture logic that guarantees the continuity of the data. Or more precisely, guarantees that if the continuity of the data is broken, this will not go unnoticed.

In theory, it's not possible to ensure a long-term sustained data rate between a peripheral and a computer, since the operating system may throttle the stream to keep it in pace with a disk, for example. There are however 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 this section it's shown how Xillybus is used to capture 32-bit wide data from a continuous source.

The suggested method is that if the FPGA's FIFO gets full, no data enters it afterwards, and the user application on the host receives an EOF (end of file) after the last safe piece of data has arrived. So if the captured data is written to a file, its length

may vary, but its validity is assured.

4.2 Example code

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

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

The zip file consists of two files, xillycapture.v and xillycapture.vhd, in Verilog and VHDL, respectively. In order to try out the example, edit xillydemo.v or xillydemo.vhd (depending on the chosen language). Disconnect the signals related to read_32 in the existing design, and insert the example code instead.

The example code relies on the existence of a standard, 32-bit wide, dual clock FIFO (512 elements is enough), named async_fifo_32. Generate such FIFO with the FPGA vendor's IP Core tools prior to implementation.

Note that the example code slows down the data rate deliberately by virtue of the "slowdown" signal. This should be removed in a real-life application.

4.3 FIFO connections

Assuming that the captured data is clocked with capture_clk, the data is simply fed to a standard FIFO with independent clocks. This FIFO interfaces between the data source and the Xillybus 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 pretty similar to the original 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. There are two situations in which capturing should not take place:

- 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 ;
```

where capture_open is user_r_read_32_open after crossing clock domain to capture_clk.

Other application-specific conditions, such as waiting for the beginning of a frame in video frame grabbing, or waiting for a certain error condition when using data acquisition for debugging, can be ANDed as required.

The signal capture_has_been_full is latched 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 restart as long as the file is opened.

IMPORTANT:

In the example code there is another 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 for making 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 another issue: Since the FIFO's full signal is high when it's reset, capture_has_been_full should be asserted only when capture_full has been low (meaning that the FIFO came out of reset) and then high (meaning it was full for real). So the logic is somewhat twisted, but pretty straightforward once the principle is understood.

4.5 Generating EOF

An end-of-file should appear when the data in the FIFO is exhausted, and will not be refilled because the FIFO has been full at some point.

In Verilog, this is

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

And in VHDL (this assignment is combinatoric):

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

where has_been_full is capture_has_been_full after crossing clock domain to bus_clk.

Now a small, but nevertheless important issue: The user_r_read_32_eof signal is required to go from '0' to '1' only on a clock cycle following an asserted read enable, according to Xillybus' core API. This is assured, since it's a logical AND between user_r_read_32_empty and has_been_full. has_been_full goes high when the FIFO is full, so it's guaranteed that user_r_read_32_empty is low when that happens. On the other hand, user_r_read_32_empty is a FIFO's empty signal, which naturally meets the requirement.

4.6 A test run

IMPORTANT:

This test run deliberately shows how the EOF mechanism kicks in by choosing a poor configuration of the IP core, having small buffers with a synchronous flow. 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 if and when the host 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 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 some 68 MB were loaded on the first go, but only 7 MB on the second one. The difference lies in how much data was received before the operating system broke

down the read-after-read pattern to do something else. Most likely, to write to the disk.

But even when discarding all data by sending it to /dev/null, it will eventually break (run “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
```

The truth behind these two tests is that in both occasions, a mouse movement stopped the data flow.

Again, it’s important to stress: An asynchronous configuration, where the host’s RAM extends the FPGA’s FIFO, solves this problem.

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 increments. 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 know how much data is held in Xillybus’ buffers belonging to a given stream at a given time: For controlling latency, preventing overflow or underflow of data, or to prevent the host application from blocking on a read() or write() call.

For example, in the FPGA-to-host direction, this means knowing how much data has been read from the FIFO at the FPGA by the Xillybus IP core logic, and has not been consumed by the application software on the host.

Likewise, in the opposite direction, knowing how much data the host application software has written to the stream, and has not reached the FIFO on the FPGA (because it's full, waiting for data to be fetched by 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), and partly because the values Xillybus' IP core would provide would be misleading and behave counter-intuitively.

To explain the suggested solution, consider a data acquisition stream (data going from FPGA to host). The following counter is applied to count the number of data elements that were fetched from the FIFO by Xillybus' IP core since the file was opened:

```
reg [31:0] count_data;

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

In this example, the xillybus_* signals are those connected to the Xillybus IP core's ports with matching names.

The value of count_data can be exposed to the host through another, dedicated Xillybus stream for this purpose: count_data is connected directly to this other stream's data port (where a FIFO's data output would normally go), and the eof and empty ports are 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, the host can read 4 bytes from this stream any time to make itself informed of count_data's current value.

Alternatively, and when adequate, a seekable stream can be used to multiplex this value with other status information (possibly other counters), and make them accessible as a set of registers.

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

Once this value is known to the software by virtue of this extra Xillybus stream, the number of data elements in the buffer can be calculated as the difference between

count_data, and the number of data elements that the software on the host has read from its data acquisition device file since it was opened. This requires, of course, that the software counts the number of bytes that it has read from the stream.

The size of a FIFO data element must be taken into account when count_data is used in this calculation, and also the possibility that the counter may wrap to zero if this is possible in the usage scenario (the latter is often done by subtracting integers of say, 32 bits, so wrapping is handled gracefully).

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 (!xillybus_open)
    count_data <= 0;
  else if (xillybus_wren)
    count_data <= count_data + 1;
```

By the same coin, the software keeps track of how much data it has written to the respective device file, and informs itself on how much has been written into the FIFO on the FPGA by reading the value of this second count_data.

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

This might however be slightly trickier to implement, if the other side of the FIFO is clocked with a different clock than bus_clk (e.g. capture_clk as presented previously in this section): count_data itself is then clocked by this other clock as well, and consequently some clock domain crossing logic becomes necessary to pass count_data's value to Xillybus. There is hence a tradeoff between accuracy and simplicity, unless bus_clk itself is the acquisition or playback clock.

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 simulations that are of interest, but are recognized to be 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 is a complex piece of logic, which has been stress-tested in various scenarios. It's therefore unlikely to find buggy behavior by simulation in the core itself, where terabytes of data transport with random load patterns didn't.

Moreover, its behavior depends very much on the response from the driver on the host, the application on the same and the PCIe bus' latencies (or AXI bus arbitration on Zynq platforms). A comprehensive simulation is therefore nearly impossible.

In light of this, it's recommended to simulate application logic up to the point where the FIFOs interface with the Xillybus IP core. The IP core is modeled as an entity which drains this FIFOs or fills them, depending on the data's direction.

It's recommended to do it differently for streams that are intended to be run continuously and those subject to bursts of data, as discussed next.

5.2 Simulating continuous streams

In a continuous stream, data is either read or written to the FIFO at the rate dictated by the application logic, as long as the respective _open signal is asserted (indicating that the file is opened by the host). If the FIFO overflow or underflows, this is probably considered a malfunction.

What prevents this overflow and underflow are data transactions made by the Xillybus IP core with the host. If these data transactions don't prevent breaking the continuity of the data flow, this is an error event that needs to be handled somehow by the application logic.

For testing what happens when the FIFO underflows on host-to-FPGA streams, it's recommended to simulate this event by making the test bench assert the FIFO's "empty" line (connected to the application logic).

Likewise, the "full" line can be asserted for a FPGA-to-host stream to test a FIFO overflow.

Aside from this test, the basic assumption in simulations should be that the continuity is never broken. In fact, in the host-to-FPGA direction, the FIFO can be implemented in the test bench simply by reading a word from a file for each rising clock having the read enable signal asserted. In the opposite direction, a word is written to a file when write enable is asserted.

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 user space application, or a CPU deprivation resulting from an overall condition in the host. If such event happens for real, the application should respond to this. As mentioned above, this response is eligible for simulation.

Having said that, there may be a possibility that the data continuity is broken due to the application logic attempting to exceed the bandwidth limitation of the stream (or that the system-total bandwidth is exceeded). If this is a possibility (in most applications it isn't), it's also recommended that the test bench will measure the data flow rate over short segments of time (in the order of magnitude of the time it takes to fill or empty the FIFO to half).

5.3 Simulating burst upstreams

There is no uniform method to simulate bursty streams from FPGA to host, since the natural behavior is that the data is written into the mediating FIFO, and is then transmitted to the host at some later time by the Xillybus IP core. If and when this transmission takes place depends on several parameters.

For example, if the stream is asynchronous, the respective file is open on the host and there is room in the DMA buffers, the transmission starts soon after the becomes non-empty. On the other hand, nothing will happen on a synchronous stream until the application on the host attempts to read data.

Simulating the overflow of the FIFO is recommended by asserting the FIFO's "full" signal directly by the test bench.

The simulation of normal operation depends very much on the specific design. A straightforward setting involves including a FIFO in the simulation, allowing the tested logic to write data to it, and the test bench read from it. The read pattern should mimic the host's expected read behavior.

5.4 Simulating burst downstreams

A burst of data from the host to FPGA merely fills the mediating FIFO with data at a given time. There is no risk of overflowing the FIFO, because the Xillybus IP core respects the FIFO's "full" line.

It is therefore recommended to simulate this setting by including a FIFO in the tested logic, and set up the test bench to write data to it in a pattern that mimics the host's expected write behavior.