The guide to Xillybus Block Design Flow for non-HDL users

Xillybus Ltd.

www.xillybus.com

Version 1.1

1

Introduction

The Xillybus Block Design Flow is an alternative to the Verilog / VHDL based flows, and is intended for those not comfortable with modifying and designing with logic-related HDL languages. Its primary target is to allow designers with no FPGA background access to coprocessing / acceleration capabilities without the need to acquire FPGA-related skills. Among others, it's intended as a simple means to exchange data between logic generated by Xilinx' Vivado High Level Synthesis (HLS) and a computer or embedded platform running Linux or Microsoft Windows.

The Block Design Flow diverts from Xillybus' main concept of communicating with the Xillybus IP core through FPGA FIFOs. Instead, user application logic connects directly to the Xillybus IP block through AXI Stream interfaces. This simplifies the Flow considerable, but requires awareness of the difference, in particular when relating to documentation on Xillybus: All references in the documentation to FIFOs in the FPGA are irrelevant to the Block Design Flow, which replaces them with a simple wire in the GUI.

Xillybus' Block Design Flow should not be confused with the block design diagrams used for setting up a Zynq processor environment or otherwise connecting between logic blocks: Such block designs, if applied, are unrelated, and may coexist regardless of the flow chosen for connecting Xillybus' IP core with application logic.

Xillybus allows the designer to focus on productive, application related work by

- supplying a working starter project, which can be compiled into an FPGA bitstream right away, and which sets up a simple and intuitive data exchange between the FPGA and the computer host by virtue of Xillybus' IP core,
- supplying a sample High Level Synthesis (HLS) project for demonstrating logic design with C/C++, with the key elements explained in this guide (see section

6),

- allowing a very simple integration of IP blocks into the FPGA design, using Vivado's block design tool
- supplying drivers for Linux and Windows that offer a simple programming interface on the host,
- offering a web tool which automatically creates custom Xillybus IP cores consisting of data streams configured specifically for a given project

As the Block Design Flow relies on the block design tool of Xilinx' Vivado, it's limited to the FPGA targets covered by this tool. Hence only Xilinx' series-7 and later FPGA targets (including Ultrascale devices) are supported.

Despite the ease of use of the Block Design Flow, it gives access only to a subset of Xillybus' features, and is therefore not recommended for those familiar with FPGA design based upon Verilog or VHDL. However for certain applications, e.g. IP core or HLS-based hardware acceleration / coprocessing logic, the impact of the difference in Xillybus' features is negligible.

The Block Design Flow is not supported by XillyUSB.

2

General guidelines

2.1 Getting started

In principle, setting up a project for the Block Design Flow is as described in the respective Getting Started guide for the targeted platform, using Vivado:

- For Xillinux bundles: Getting started with Xillinux for Zynq-7000
- For PCIe bundles: Getting started with the FPGA demo bundle for Xilinx

When following these guides, be sure to use the xillydemo-vivado.tcl script in the **blockdesign**/ subdirectory.

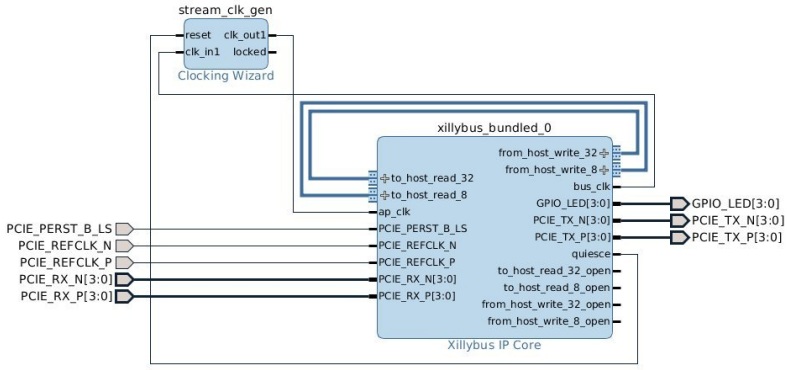
IMPORTANT:

*This guide does **not** go along with the tutorial named "FPGA coprocessing for C/C++ programmers" on Xillybus' website. There are several differences in technical details as well as the example projects presented. In order to avoid confusion, it's advised to stick to either this guide (for a Block Design Flow) or the website tutorial (for a Verilog / VHDL flow).*

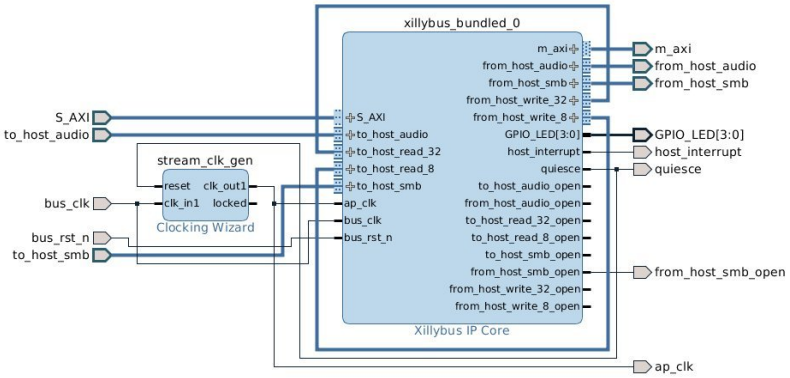
Generating and using the bitfile are done the same as in the other flows: A bitfile can be implemented immediately from the bundle "out of the box", and the loopback tests described in the Getting Started guides work the same. However please note that the seekable stream xillybus_mem_8 **doesn't work** in the Block Design Flow, as explained in section 4.3.

The Block Design Flow is different in that interfacing with Xillybus' IP core takes place in Vivado's block design tool: After generating the project, open the block design by choosing "Open Block Design" in Vivado's left menu bar.

On PCIe-based (non-Xillinux) targets, the following diagram is displayed:



On Xillinux targets, “Open Block Design” opens the Zynq processor’s environment – this block design *should not be modified* in tasks related to Xillybus. Rather, the block marked “blockdesign” should be opened (with a double-click), which displays the following block diagram:



2.2 Notable block design elements

There are a few key elements that are worth noting in the Xillybus block design:

- Xillybus stream ports (“from_host_*” and “to_host_*”): These are standard AXI Stream ports, consisting of the minimal set of signals: TDATA, TVALID and TREADY. The name of each port in the block design is preceded with either “from_host” or “to_host” in order to mark the interface’s direction. The rest of the

port's name in the block design is the device file's name, as presented at the host, minus the "xillybus" prefix.

For example, the device file named `/dev/xillybus_write_32` on a Linux host, or `\\.\xillybus_write_32` on a Windows computer can be accesses on the block design on the port named from_host_write_32.

- Loopbacks: Initially, from_host_write_32 is connected to to_host_read_32, and from_host_write_8 is connected to to_host_read_8. This loops back any data written to the device file xillybus_write_32 into xillybus_read_32, and the same goes with the write_8 / read_8 pair. This loopback is what makes the "Hello world" test described in the Getting Started guides working.

  For integrating application logic, the respective loopback connection should be removed with Vivado's block design GUI, and connections should be made with the application logic's suitable AXI Stream ports.

- On some targets, streams named xillybus_smb and xillybus_audio are connected to the hierarchy above, since these are used for supporting the board's audio interface. These streams should be ignored (i.e. treated as the rest of the signals going to the processor design hierarchy in the block design).

- "*_open" ports for each Xillybus stream: Each of the AXI Stream ports has a corresponding port with a _open suffix, which is logical high when the relevant Xillybus device file is open on the host. This signal can optionally be used to reset any application logic that is attached to the stream, so it's in a known state every time the device file is opened.

- The Clocking Wizard (stream_clk_gen) block: Generates a clock for the application logic from Xillybus' interface clock. All of Xillybus IP core's AXI Stream ports are clocked by this block's output.

  It's recommended not to make any changes on this block except for the output clock frequency. In particular, the name of the block (stream_clk_gen) must be kept, as certain implementation scripts (the timing constraints) refer to its output by this name.

  See section 3.2 below.

- External ports, e.g. GPIO_LEDS[0:3]: Ports that are connected to the hierarchy above the block design. These connections should not be altered, but their signals may be sampled by blocks within in. For example, in the Xillinux bundle, ap_clk goes to the upper hierarchy, but can be used inside the block design view as well.

Note that there are no ports for the mem_8 stream. Seekable streams are not presented in the block diagram. See section 4.3 for more about this.

3

Integrating application logic

3.1 The basics

Application logic is integrated into the design using Vivado's block design GUI: IP blocks are added to the block design and connected as required.

For integration of IP blocks generated by Vivado's High Level Synthesis (HLS), please refer to section 6.

It should be noted that even though it's often said in Xillybus' documentation, that the FPGA side communicates with the host through hardware FIFOs, this is *not* the case with the Block Design Flow (but only for the Verilog and VHDL flows). The glue logic in Xillybus' IP Core that generates the AXI Stream interfaces already involves FIFOs (among others to cross the clock domains between bus_clk and ap_clk). As a result, the application logic is *not* required to deploy any FIFOs in order to interface with Xillybus' IP core when the Block Design Flow is used, unlike the VHDL / Verilog flows.

For data exchange between the FPGA and host, connect the application logic to the dedicated AXI Stream ports (possibly after disconnecting loopbacks). These ports present only the TDATA, TVALID and TREADY and in particular *not* the TLAST signal. Consequently, each AXI Stream stream embodies an infinite data stream (as opposed to a packet interface, which the TLAST signal would have allowed). This is consistent with the infinite stream nature of Xillybus' device files in general.

Xillybus streams can be used to exchange packets between the FPGA and the host, as explained in section 6.3 in any of these two guides:

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

3.2 Clocking

3.2.1 General

For the sake of simplicity, all signals connecting the user application logic to the Xillybus IP Core must be driven by a single clock, which is generated by the Clocking Wizard block in the block design. This clock (the "application clock") is the Clocking Wizard's clk_out1, which is also the Xillybus IP Core block's ap_clk input.

It's often convenient to drive the entire user application block with this single clock, so all of its internal logic as well as the interface depends on it. For example, logic which is generated by Vivado's HLS synthesizer has a single clock input (named ap_clk). Connecting this clock input to the Clocking Wizard's output guarantees that the AXI Stream port connections with Xillybus IP Core block function properly.

Note that FPGA tools sometimes refer to a clock frequency in terms of the frequency itself, typically in MHz, and sometimes as the clock period, commonly in ns. The clock frequency is the reciprocal of the clock period, so e.g. 100 MHz is equivalent to a period of 10 ns.

3.2.2 Setting the application clock

The application clock's frequency can be set to increase performance or as a step towards achieving a working bitfile: A faster clock yields a higher processing throughput (unless some other bottleneck limits the performance) but also demands more from the FPGA's logic elements and its utilization by Xilinx' tools.

If the application clock's frequency is chosen too high, the compilation of the project into an FPGA bitstream file fails on the grounds of not meeting timing constraints. This is also referred to as a "timing failure". This situation means that the implementation tools failed to utilize the logic in such a way that ensures the reliable operation while driven by the clock frequencies as defined. The "timing constraints" in this context are the requirements on the frequencies of the clocks in the system.

Reducing the application clock's frequency is always allowed (within the clock generator's limits), but slows down the operation of the logic it drives.

In order to set the frequency of the application clock, double-click the Clock Wizard (stream_clk_gen) block in the block design view. A configuration window will be opened in Vivado. Choose the "Output Clocks" tab and change the "Output Clock Requested" frequency for clk_out1. The frequency in the "Actual" column shows the frequency that will be generated by the clock synthesizer. It may be slightly different from the

requested frequency, since the output clock is derived by multiplying the input clock by a rational number, which is picked from a limited set of allowed values.

A small diversion from the requested frequency is harmless when the clock is used only for the application logic and its interface with the Xillybus IP core.

Other parameters in the Clocking Wizard should not be changed.

3.2.3 The bus_clk clock signal

Xillybus IP Core's internal logic is driven by bus_clk, which is exposed in the block design merely to allow the derivation of the application clock from it. There is usually no other use for this signal, since the application logic only needs ap_clk for its internal logic and for interfacing with the Xillybus IP Core.

bus_clk's frequency may however be of interest for the sake of spotting throughput bottlenecks. For example, if bus_clk runs at 100 MHz, the maximal theoretic bandwidth that may go through a 32-bit wide data interface is 400 MB/s, since Xillybus' internal data pipe runs at bus_clk's rate. If ap_clk runs at a higher frequency and data is pushed on each cycle of ap_clk, it's likely that the data pace will be slowed down by virtue of the AXI Stream flow control signals (TREADY and TVALID).

For this reason, bus_clk's frequency should be taken into consideration when attempting to maximize an application's throughput, in particular if the data interfaces are expected to contain long bursts of (or continuous) data traffic.

The frequency of bus_clk can be found under the "Clocking Options" tab, as the frequency of the primary input clock, clk_in1. This parameter informs the Clocking Wizard what clock to expect at its input, and can therefore be used for knowing the frequency of bus_clk for a specific Xillybus bundle.

4

Acceleration / coprocessing best practices

4.1 Latency vs. throughput

There's a significant difference between "traditional" hardware acceleration, which is based upon enhanced instruction sets (e.g. the x86 family's MMX command and crypto extensions for AES, and ARM's NEON extension) and acceleration with external hardware, such as GPGPUs and FPGAs: As the enhanced instruction sets are part of the processor's execution flow, they replace a long sequence of machine code instructions with a shorter one, and reduce the number of cycles required until the result is available.

External hardware acceleration (FPGA acceleration included) on the other hand, does *not* necessarily reduce the time until the result is available, due to the significant latency of transporting the data to and from the external hardware. In addition, the processing time may also be significantly longer than the processor's due to pipelining, and possibly a lower clock frequency.

The advantage of external hardware acceleration is hence not latency (how fast the result is obtained) but throughput (the rate at which the data is handled). In order to utilize this advantage, it's important to maintain a *flow of data* going to and from the accelerating hardware, rather than waiting for the results of one operation before initiating the next one.

The technique for proper acceleration with an FPGA is elaborated in section 6.6 of either of these two documents:

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

4.2 Data width and performance

For applications that require relatively high data bandwidths, it's recommended to use 32-bit wide streams (or wider) for the data-intensive streams, as 8 and 16-bit wide streams utilize the host's data bus less efficiently.

The reason is that the words are transported through the Xillybus internal data paths at the bus clock rate. As a result, transporting an 8-bit word takes the same time slot as a 32-bit word, making it effectively four times slower.

This also impacts other streams competing for the underlying transport at a given time, since the data paths become occupied with slower data elements.

This guideline doesn't apply to revision B/XL Xillybus IP cores, which transports narrow streams with the same efficiency.

4.3 Do's and don'ts

There are a few issues to note when working with the Block Design Flow:

- Streams with address ports ("address/data streams", "seekable streams") are not supported in the Block Design Flow. If the Xillybus IP Core includes such streams, they do not appear as ports in the GUI, but do appear normally on the host side. Attempts to read from such a stream on the host will yield an immediate end-of-file condition. A write() call will not return, as there's no data sink on the other end.

  It's therefore recommended to avoid seekable streams in custom IP cores that are intended for use with the Block Design Flow, in order to avoid confusion and a slight waste of FPGA logic.

- The "stream_clk_gen" (Clocking Wizard) block must not be altered, except for changing its output frequency if necessary, as described in section 3.2.2.

  Making changes in the input clock frequency, making other changes in the configuration, or removing it from the design and replacing it with a fresh Clocking Wizard IP block may lead to failing to meet the timing constraints (possibly because some timing constraints exceptions refer to the block's name).

  Setting an incorrect input frequency may lead to an unreliable behavior of the FPGA design.

- It's important to pay attention to how the clocks are connected. In particular, not not mixing between bus_clk and ap_clk.

- Make sure that the Xillybus streams are asynchronous, which is the case in the default IP core and the autoset choice in custom IP cores when the stream's intended use is "Data exchange with coprocessor".

  This causes, among others, write() calls on the host to return immediately if there is enough room for the data in the driver's DMA buffers, ensuring a smoother data transport and higher bandwidth performance.

  For a better understanding of this topic, please refer to section 2 of either Xillybus host application programming guide for Linux or Xillybus host application programming guide for Windows.

5

Applying a custom Xillybus IP core

A web interface tool allows users to configure and download custom Xillybus IP cores, choosing the number of streams as well their attributes directly on Xillybus' website. The specially generated custom IP core is then downloaded from the site, typically a few minutes later.

In order to generate and download a custom IP core, please visit the IP Core Factory at Xillybus' website. The process is fairly straightforward, and if necessary, The guide to defining a custom Xillybus IP core supplies complimentary information.

IMPORTANT:

Seekable streams (with "address/data" interface) are invisible in the block design flow, as the AXI Stream connections can't support the address wires. Having such streams in a core is fairly harmless, but causes a slight waste in FPGA logic resources, and a possible confusion as they do appear on the host side, but not in the block design.

Once the custom IP core is defined, generate and download its bundle.

The instructions in the custom IP core bundle's README file relate to the Verilog / VHDL flows and should be disregarded. Instead, the following steps should be taken:

- Create a new directory for the custom IP core's files. This directory's absolute path must remain fixed throughout the use of this custom IP core, so it's recommended to locate it where it won't be deleted accidentally.

  Unzip the downloaded custom IP core bundle into this directory.

- Open the block design in Vivado.

- Save the block design's view as a pdf file for reference: Right-click somewhere in the block design's area, and choose "Save as pdf file...".
- Pick "Run Tcl Script..." under the Tools menu (on the main menu bar). Navigate to the directory to which bundle was unzipped, and enter the xillybus_block subdirectory. Choose insertcore.tcl.
- The script will replace the existing Xillybus IP Core with the custom IP core, and also attempt to reconnect the non-application related wiring. The objects may also move in the block design diagram due to an automatic reorganization.
- Connect the application logic's AXI Stream interfaces to the updated Xillybus IP core.
- Compare with the pdf file that was created before running the script, and correct as necessary.

  None of the application logic related connections are reconnected, and other connections may be missing as well.
- Verify that the captions below and under Xillybus IP Core's block match the name of the new IP core.

Note that the script finds blocks, ports and interfaces by their names. It may therefore fail partially (and silently) in restoring connections if these names have been changed by the user.

From this point, the project can be implemented as before. Xillybus' driver for the host (Linux and Windows alike) works with the custom IP core as well, since it detects the new IP core's configuration automatically.

Hence there's no need to install anything on the host following the replacement with a custom IP core.

For reference, these are the steps of execution of the insertcore.tcl script:

- Add the directory of the custom IP core to the list of IP Core repositories in Vivado's IP Catalog and force a rescan of the repositories, so the new custom IP Core is discovered and added to the Catalog
- Remove the previous Xillybus IP from the block design, if such is present
- Add the custom IP core to the design, and upgrade its version if necessary

- Attempt to reconnect wires going to the hierarchy above, as well as the wires to the stream_clk_gen block, by looking up a hardcoded list of names, and interconnecting all ports having these names, if present.
- On Zynq targets only: Set the bus address of the Xillybus IP core to its default values (a 4 kB segment starting at `0x50000000`)
- Reset the synthesis run of the project, so the next implementation reflects the changes made.

6

Vivado HLS integration

6.1 Overview

This section demonstrates how a simple C function can be compiled into an IP block, and then integrated into Xillybus' Block Design flow.

The example project, which this section is based upon, can be downloaded at

http://xillybus.com/downloads/hls-axis-starter-1.0.zip

It's recommended to unzip the downloaded file into a directory that is easily related to the Xillybus project, as it can't be moved at later stages.

It's important to distinguish between two different kind of C sources in the example project:

- Code for execution: Runs on a computer or embedded platform ("host"), like any computer program, and uses the FPGA to offload certain operations.

  In the example project, the sample files can be found under the host/ subdirectory.

- Code for synthesis: Translated into logic by Vivado HLS.

  In the example project, it can be found at coprocess/example/src/main.c

Unlike common C/C++ programming, the host program doesn't call the synthesized function. Rather, it organizes the data needed for executing the function in a data structure and transmits it to the synthesized function, using a simple API, which is described further on. At a later stage, it collects the return data as a data structure sent from the synthesized function with a similar API.

6.2 HLS synthesis

The example code in C used in this section is outlined in section 6.4.

Start Vivado HLS, and open the HLS project: Pick “Open Project” on the welcome page, navigate to where the HLS project bundle was unzipped to, and choose the folder with the name “coprocess”.

Change the project’s part number: Pick Solution >Solution Settings... >Synthesis and change the “Part Selection” to the FPGA target of the (non-HLS) Vivado project.

Compile (“synthesize”) the project by picking Solution >Synthesis >Active Solution (or click on the corresponding icon on the toolbar). A lot of text will appear on the console, including several warnings (which is normal). No errors should occur.

A successful compilation is easily recognized by the following message among the last few lines in HLS’ console tab:

```
Finished C synthesis.
```

A synthesis report will also appear above the console tab only when the synthesis was successful.

For more information about Vivado HLS, please refer to its user guide (UG902).

6.3 Integration with the FPGA project

In Vivado HLS, select Solution >Export RTL and pick “IP Catalog” as Format Selection. For “Evaluate Generated RTL” choose Verilog, and don’t check either checkboxes under this. Click OK.

This can take several minutes, and ends with something like

```
Finished export RTL.
```

Now open the Xillydemo project (as set up in section 2.1) in (non-HLS) Vivado, and open the Block Design inside Vivado. On Xillinux (Zynq) targets, open the “blockdesign” block.

Add the HLS IP block as follows: Right-click somewhere in the block design diagram area, and pick “IP Settings...”. Under the “Repository Manager” tab, click the green plus sign for adding a repository. Navigate to and select the same “coprocess” directory that was chosen in section 6.2 to open the HLS project. Vivado should respond

with a pop-up window indicating that one repository was added. Click on “OK” buttons twice to confirm.

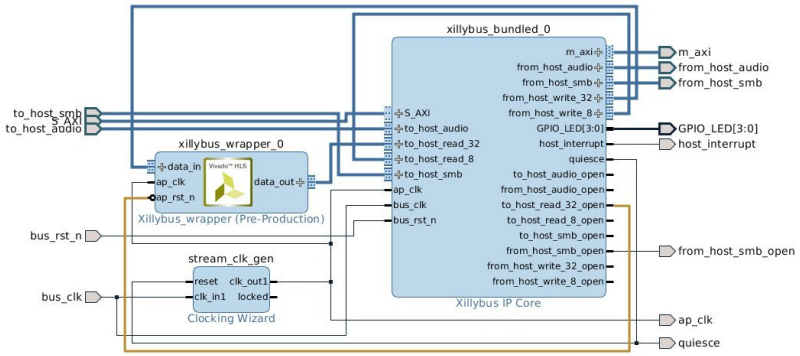
Now add the IP block into the block design: Once again, right-click somewhere in the block design diagram area. Pick “Add IP...” and select the Xillybus_wrapper IP from the list (typing “wrapper” in the search box is likely to make this easier).

A new block, named xillybus_wrapper_0 will appear in the diagram. Disconnect the wire going between to_host_read_32 and from_host_write_32 (i.e., disconnect the loop-back).

Then connect the xillybus_wrapper block as follows:

- data_in with from_host_write_32
- data_out with to_host_read_32
- ap_rst_n with to_host_read_32_open
- ap_clk with ap_clk (which is also the Clocking Wizard’s clk_out1 output)

The result should be something like this (shown for a Xillinux block design):



The connection between ap_rst_n and to_host_read_32_open holds the logic in xillybus_wrapper block reset unless the xillybus_read_32 device file is opened on the host (to_host_read_32_open is logic ’0’ when the file isn’t opened, and the reset input is active low). Assuming that the software running on the host opens this device file before attempting to communicate with this block, this ensures a consistent response from the logic each time the software is run.

At this point, the bitstream can be implemented: At the bottom of Vivado’s window, pick the “Design Runs” tab, right-click over “synth_1” and pick “Reset Runs”. Confirm resetting synth_1.

Then click “Generate Bitstream” at the left bar.

6.4 The example synthesis code

To clarify how HLS works with Xillybus, the example demonstrates the calculation of a trigonometric sine and a simple integer operation, both covered in a simple custom function, mycalc().

coprocess/example/src/main.c starts as follows:

```
#include <math.h>
#include <stdint.h>

extern float sinf(float);

int mycalc(int a, float *x2) {
  *x2 = sinf(*x2);
  return a + 1;
}
```

As usual, there are a couple of #include statements. The “math.h” inclusion is necessary for the sine function.

And there’s the simple function, mycalc() which takes the role of the “synthesized function”. It’s a very simple function for the sake of demonstration of floating point as well as integer operations. The High-Level Synthesis Guide (UG902) gives more information on how to implement more useful tasks.

Next in main.c, there’s the wrapper function, xillybus_wrapper(), which is the bridge between the synthesized function and Xillybus, and is hence responsible for packing and unpacking the data going back and forth.

In the example’s case, it accepts an integer and a floating point number from the host through a data stream, which is represented by the “data_in” argument. It returns the integer plus one and the (trigonometric) sine of the floating point number, using the “data_out” argument.

```
void xillybus_wrapper(int *data_in, int *data_out) {
#pragma AP interface axis port=data_in
#pragma AP interface axis port=data_out
#pragma AP interface ap_ctrl_none port=return

  uint32_t x1, tmp, y1;
  float x2, y2;

  // Handle input data
  x1 = *data_in++;
  tmp = *data_in++;
  x2 = *((float *) &tmp); // Convert uint32_t to float

  // Run the calculations
  y1 = mycalc(x1, &x2);
  y2 = x2; // This helps HLS in the conversion below

  // Handle output data
  tmp = *((uint32_t *) &y2); // Convert float to uint32_t
  *data_out++ = y1;
  *data_out++ = tmp;
}
```

xillybus_wrapper() is declared with two pointers to int. These top-level function arguments turn into two top-level AXI Stream ports of the to-be IP block for inclusion in the block design: Each of them has a #pragma statement informing HLS that they should be considered interfaces of type “axis”.

“#pragma AP” and “#pragma HLS” are interchangeable – the former is the based upon the C Synthesizer’s previous name (Auto Pilot), and the latter is seen in Xilinx’ recent documentation.

Since “int” is considered a 32-bit word by HLS, the respective AXI Stream interfaces will have a 32 bit wide data interface.

It’s of course possible to change the list of arguments as well as the pragmas to obtain any set of AXI Stream inputs and outputs.

The pragma declaration for ap_ctrl_none tells the compiler not to generate a port for the (nonexistent) return value.

And next, there’s some code for “execution”: The input data is fetched. Each *data_in++ operation fetches a 32-bit word originating from the host. In the code shown, the first

word is interpreted as an unsigned integer, and is put in x1. The second word is treated as a 32-bit float, and is stored in x2.

Then there's a call to mycalc(), the "synthesized function". This function returns one result as its return value, and the second piece of data goes back by changing x2.

The wrapper function copies the updated value of x2 into a new variable, y2. This may appear to be a redundant operation, and would indeed have been such, had this code been compiled for execution on a processor. When using HLS, this is however necessary to make the compiler handle the conversion to float later on. This reflects a somewhat quirky behavior of the HLS compiler, but this is one of the delicate issues of using a pointer: The HLS compiler doesn't really generate a memory array and a pointer to it. The use of the pointer is just a hint on what we want to accomplish, and sometimes these hints need to pushed a bit.

Finally, the results are sent back to the host: Each *data_out++ sends a 32-bit word to the computer, with due conversion from float.

Note that the *data_in++ and *data_out++ operators don't really move pointers, and there is no underlying memory array. Rather, these symbolize moving data from and to the AXI stream interfaces (and eventually from and to Xillybus streams). Hence, the only way the "data_in" and "data_out" variables are used is *data_in++ and *data_out++ (the High-Level Synthesis Guide offers other possibilities, in particular fixed sized arrays).

Also note that since this code is translated into logic, and not run by a processor, the only significance of these C commands is to produce the expected output stream of data given the input stream of data. There is however no promise on *when* the data is emitted (except for a range of possible latencies, given in HLS' report).

Accordingly, the order of assignments of the input data is important in the sense that it enforces how the incoming data is interpreted. On the other hand, since the first output that is sent, y1, depends only on x1, which is the first input arriving, it's allowed that the first output will be sent before the second input has arrived. This contradicts the intuitive sequential nature of code execution, but is meaningless in the context of hardware acceleration, as the overall result is the same.

Furthermore, if the data_in AXI stream is constantly fed with data, the wrapper function "runs" repeatedly, *as if* it said

```
while (1) // This while-loop isn't written anywhere!
  xillybus_wrapper(data_in, data_out);
```

New data is fetched by virtue of the *data_in++ commands as soon as possible, quite

likely filling the logic's internal pipeline (which is longer than 70 stages in the example project, according to HLS' report). So unlike a processor's execution of the code, which wouldn't fetch a pair of words, process them, emit two output words and only then fetch the second pair of words, the HLS interpretation may very well fetch 70 words at data_in before anything comes out on the data_out AXI stream.

6.5 Modifications on the C/C++ code for synthesis

Additional AXI Stream ports can be created by adding arguments to the wrapper function, and declaring these as interface ports, as shown in the example.

It's of course possible to make other changes in the C code of the example design.

It's recommended to implement the I/O in the same style as shown with *data_in++ and *data_out++, or refer to the High-Level Synthesis Guide (UG902) for other possibilities. It's also a recommended source for learning about coding techniques and practices.

IMPORTANT:

Don't just click "Generate Bitstream" in Vivado after making changes: Launching a repeated bitstream generation without upgrading the block as detailed below, is likely to result in a seemingly successful implementation of the bitfile, but based upon an outdated version of the HLS block.

After changes have been made in the sample project, start over from "HLS synthesis" in section 6.2, and go all the way to implementation with Vivado, plus updating the HLS block in Vivado.

That is:

- Vivado HLS: Compile the project in HLS. The HLS synthesizer always cleans up the files generated in previous compilations before starting a new one.
- Vivado HLS: Export into an IP Catalog bundle.
- In (non-HLS) Vivado, upgrade the xillybus_wrapper block (actually, update it following its change): Open the block design view, and respond to the message at the top of the page, saying the block needs upgrading. If this message isn't found, type "report_ip_status -name status" at the Tcl Console. Click on the "Upgrade Selected" button at the bottom. This will be followed by a dialog box confirming the successful upgrade, and one requesting to generate output products. Click "Skip" on the second dialog box.

- Vivado: Verify that the design runs were invalidated: At the bottom of Vivado's window, pick the "Design Runs" tab. It should say Synthesis Out-of-date in the Status column for synth_1.
- Vivado: **Unless the design runs were invalidated,** attempt the following: Refresh the IP catalog: Right-click somewhere in the block design diagram area, and pick "IP Settings...". Under the "Repository Manager" tab, click the "Refresh All" button at the bottom. It may also be necessary to click "Clear Cache" on the "General" tab of the same dialog box. After this, go back to upgrading xillybus_wrapper block.

  None of these actions are necessary if the design runs were found invalidated in the previous item above.
- Vivado: Reset the synth_1 run
- Vivado: Generate bitstream

6.6 simple.c: An example of a host program

In the example project, there are sample host programs as two C files: simple.c and practical.c. These demonstrate the host side of the project.

Both are written for a Linux host, to be compiled with e.g.

```
# gcc -O3 -Wall simple.c -o simple
```

but they are easily adapted for Windows (see below).

IMPORTANT:

simple.c ***should not be used as an example*** *for actual host programming, in particular due its following drawbacks:*

- *Only one single element is handled. Looping on the write() and read() pair of calls will result in poor performance.*
- *The write() and read() operations' return values must be checked for proper operation. This has been omitted for simplicity, but renders the program unreliable.*

Section 6.7 outlines better coding techniques.

The simple.c file starts with #include headers:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
```

and then the classic declaration of the main() function, along with some variable declarations:

```
int main(int argc, char *argv[]) {
  int fdr, fdw;

  struct {
    uint32_t v1;
    float v2;
  } tologic, fromlogic;
```

The struct variables will be discussed below.

The program starts with opening the two device files, which behave like named pipes, and are used for communication with the logic: /dev/xillybus_read_32 and /dev/xillybus_write_32. Recall from the setting up of the Xillybus bundle that these two files are generated by Xillybus' driver.

As pointed out in section 6.3, ap_rst_n is connected to_host_read_32_open in the block design diagram, so opening /dev/xillybus_read_32 gets the logic out of reset. This is why both files are opened before data transmission.

```
  fdr = open("/dev/xillybus_read_32", O_RDONLY);
  fdw = open("/dev/xillybus_write_32", O_WRONLY);

  if ((fdr < 0) || (fdw < 0)) {
    perror("Failed to open Xillybus device file(s)");
    exit(1);
  }
```

Next, to the actual execution. The "tologic" structure is populated with a couple of values for transmission to the logic, after which it's written directly from memory to xillybus_write_32. Effectively, this writes 8 bytes, or more precisely, two 32-bit words. The first is the integer 123 put in tologic.v1, and the second is the float in tologic.v2.

The tologic structure was hence set up to match the logic expectation of data: One integer by the first *data_in++ instruction, and one float by the second.

```
  tologic.v1 = 123;
  tologic.v2 = 0.78539816; // ~ pi/4

  // Not checking return values of write() and read(). This must
  // be done in a real-life program to ensure reliability.

  write(fdw, (void *) &tologic, sizeof(tologic));
  read(fdr, (void *) &fromlogic, sizeof(fromlogic));

  printf("FPGA said: %d + 1 = %d and also "
         "sin(%f) = %f\n",
         tologic.v1, fromlogic.v1,
         tologic.v2, fromlogic.v2);
```

Recall from section 6.4 that the wrapper code fetches two 32-bit words from the data_in stream: The first one goes to “x1”, and the second to “tmp”, which is immediately converted into a float. This matches the two 32-bit elements of the “tologic” structure.

This is followed by reading back the data from the FPGA. The same principle applies for “fromlogic”.

simple.c ends with a common wrap-up:

```
  close(fdr);
  close(fdw);

  return 0;
}
```

It is crucial to match the amount of data sent to /dev/xillybus_write_32 with the number of *data_in++ operations in the wrapper function. If there is too little data sent, the synthesized function may not execute at all. If there’s too much, the following execution will probably be faulty.

In this example, the same structure format was chosen for “tologic” and “fromlogic”, but there’s no need to stick to this. It’s just important that the data sent and received is in sync with the wrapper function’s number of *data_in++ and *data_out++ operations.

The execution of this program should be

```
# ./simple
```

```
FPGA said: 123 + 1 = 124 and also sin(0.785398) = 0.707107
```

Finally, a note to Windows users, who may need to make all or some of the following adjustments:

- Change the file name string from "/dev/xillybus_read_32" to "\\\\.\\xillybus_read_32" (the actual file name on Windows is \\.\xillybus_read_32, but escaping is necessary). The second file name changes to "\\\\.\\xillybus_write_32".
- Replace the #include statement for unistd.h with io.h
- Replace the calls to open(), read(), write() and close() with _open(), _read(), _write() and _close()

6.7 practical.c: A practical host program

The simple.c example outlines data exchange in a concise manner, but several changes are required in practical system:

The following differences are most notable:

- Rather than generating a single set of data for processing, an array of structures is allocated and sent. Likewise, an array of data is received from the logic. This reduces the I/O overhead, and the impact of software and hardware latencies, and is a crucial measure for gaining a performance improvement with hardware acceleration.
- The program forks into two processes, one for writing and one for reading data. Making these two tasks independent prevents the processing from stalling due to lack of data to process or output data waiting to be cleared up. This independency can be achieved with threads (in particular in Windows) or using the select() call as well.
- The read() and write() calls are made correctly, so as to ensure reliable I/O. The while loops that are added for this purpose may appear cumbersome, but they are necessary to respond correctly to partial completions of these calls (not all bytes read or written) which is a frequent case under load. The EINTR error is also handled as necessary to react properly to POSIX signals, which may be sent to the running processes, possibly accidentally.

Now to a brief walkthrough of practical.c. First, headers:

```
#include <stdio.h>
#include <unistd.h>

#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
```

And the same structure, plus defining N, the number of elements per chunk of data.

```
#define N 1000

struct packet {
  uint32_t v1;
  float v2;
};
```

A common main() function definition and some variables:

```
int main(int argc, char *argv[]) {

  int fdr, fdw, rc, donebytes;
  char *buf;
  pid_t pid;
  struct packet *tologic, *fromlogic;
  int i;
  float a, da;
```

Files opened like before:

```
  fdr = open("/dev/xillybus_read_32", O_RDONLY);
  fdw = open("/dev/xillybus_write_32", O_WRONLY);

  if ((fdr < 0) || (fdw < 0)) {
    perror("Failed to open Xillybus device file(s)");
    exit(1);
  }
```

The actual execution begins with forking into two processes.

```
pid = fork();

if (pid < 0) {
  perror("Failed to fork()");
  exit(1);
}
```

The father process prepares the data for processing and writes it towards the FPGA. It closes the read file descriptor, since it's not used by this process. Keeping it open will make the device file remain open until both processes have closed their file descriptor (or exited), which isn't the desired behavior here.

```
if (pid) {
  close(fdr);

  tologic = malloc(sizeof(struct packet) * N);
  if (!tologic) {
    fprintf(stderr, "Failed to allocate memory\n");
    exit(1);
  }
```

Next, filling an array of structs with data. This explains why it made sense to define a structure for each set of data for processing.

```
  // Fill array of structures with just some numbers
  da = 6.283185 / ((float) N);

  for (i=0, a=0.0; i<N; i++, a+=da) {
    tologic[i].v1 = i;
    tologic[i].v2 = a;
  }

  buf = (char *) tologic;
```

Note that "buf" is defined as a pointer to a char buffer, pointing at the array of structures. This conversion is required, since the while loop that sends the data treats the buffer as any chunk of data for transmission.

Next, the while loop for writing data. It may seem unnecessarily complicated, but is the shortest way to ensure data is written reliably. It's suggested to adopt this code as is in practical applications.

```
  donebytes = 0;

  while (donebytes < sizeof(struct packet) * N) {
    rc = write(fdw, buf + donebytes,
               sizeof(struct packet) * N - donebytes);

    if ((rc < 0) && (errno == EINTR))
      continue;

    if (rc <= 0) {
      perror("write() failed");
      exit(1);
    }

    donebytes += rc;
  }
```

In this example, only a single chunk is sent (and received on the other end). It's practically correct to loop on the two pieces of code above.

Performance tests have shown that a chunk size of 32 kBytes usually gives the best results.

As only one chunk is sent in this example, the process exits. Sleeping one second before closing the file ensures that the logic doesn't reset before all data has been drained from it. This is meaningless when the block design is as shown in section 6.3, since ap_rst_n goes to to_host_read_32_open, and from_host_write_32_open isn't connected at all.

Nevertheless, this demonstrates a good convention of not closing the file descriptor immediately, unless quitting fast is required. This can save some confusion when the project becomes more elaborate.

```
  sleep(1); // Let the output drain

  close(fdw);
  return 0;
```

Next we have the child process, starting in a similar way:

```
} else {
  close(fdw);

  fromlogic = malloc(sizeof(struct packet) * N);
  if (!fromlogic) {
    fprintf(stderr, "Failed to allocate memory\n");
    exit(1);
  }

  buf = (char *) fromlogic;
```

Once again, this is the recommended way to read data from a device file:

```
  donebytes = 0;

  while (donebytes < sizeof(struct packet) * N) {
    rc = read(fdr, buf + donebytes,
              sizeof(struct packet) * N - donebytes);

    if ((rc < 0) && (errno == EINTR))
      continue;

    if (rc < 0) {
      perror("read() failed");
      exit(1);
    }

    if (rc == 0) {
      fprintf(stderr, "Reached read EOF!? Should never happen.\n");
      exit(0);
    }

    donebytes += rc;
  }
```

And then data is printed out:

```
    for (i=0; i<N; i++)
      printf("%d: %f\n", fromlogic[i].v1, fromlogic[i].v2);

    sleep(1); // Let the output drain

    close(fdr);
    return 0;
  }
}
```

Once again, the process sleeps for one second before closing the file descriptor, and once again, it isn't necessary in this specific case: Closing the file descriptor will indeed reset the logic, but it's harmless in this case because all output has been fetched by the time this point is reached.

As mentioned before, unless quitting quickly is beneficial, this one second sleep may save confusion, in particular if other output streams are generated e.g. for debugging.

6.8 Design considerations

6.8.1 Working with multiple AXI streams

The example project shows the basic case of one stream in each direction. It's however trivial to add input and/or output streams on the IP block by adding arguments to the wrapper function, along with pragmas for declaring these as AXI streams.

For example, three input streams instead of one:

```
void xillybus_wrapper(int *d1, int *d2, int *d3, int *data_out) {
#pragma AP interface axis port=d1
#pragma AP interface axis port=d2
#pragma AP interface axis port=d3
#pragma AP interface axis port=data_out
#pragma AP interface ap_ctrl_none port=return

  *data_out++ = thefunc(*d1++, *d2++, *d3++);

}
```

Adding streams to the Xillybus IP core is equally simple, by configuring a custom IP core, as explained in section 5.

Additional streams can be useful in a variety of scenarios, among others:

- Sending data and meta information in separate streams. For example, if the data needs to be divided into packets, send their lengths in one dedicated stream, and the data in another. This allows sending the beginning of the packet before its length is known.
- Sending data that is naturally arranged separately, e.g. pixel scanning of different images (more on this below).
- For debugging: Sending intermediate data to the host for verification.

When working with multiple streams, it's important to keep them all in mind: The logic's flow may stall if any input stream lacks data, or if an output stream's respective device file isn't opened (or overflown with data), as the flow of data will be stalled. This is important in particular if an output stream is intended for debugging: It's easy to forget a debug stream's device file under "normal operation", which leads to a confusing halt of execution, usually after a few data cycles.

It's often sensible to feed the logic hardware with data in ways that may seem awkward at first. For example, the three-input example shown above can be useful for an image processing algorithm that requires three elements of data for each operation: Scanning the image from left to right, top to bottom, suppose that the algorithm needs the respective pixels from two previous images along with the current image's pixel, for generating a pixel output. In such a case, it possible to send the current image through one stream to the FPGA, and the two previous images through two other streams in parallel.

This may seem as a waste of I/O data bandwidth and a lot of unnecessary memory copying. In particular, it may feel wrong that the processor is involved so much in "shuffling data". Subjective perceptions aside, the implementation of memory copying is a highly optimized task on every modern processor architecture, and the processor is often loaded with other application-related tasks, which makes the memory copying load negligible.

So even though feeding the logic with data directly is suboptimal from a resource utilization point of view, the extra load on the processor is usually rather small, given that it usually has other heavy-duty tasks to handle. This is often a reasonable price for simplifying the design significantly.

This should be compared with the optimized solution, which requires the FPGA to fetch the data by itself, and subsequently to keep track of the locations of three image arrays, probably as scatter-gather buffers.

6.8.2 The application clock's frequency

The logic generated by HLS is driven by the application clock of the block design, which is generated by the stream_clk_gen block. As this clock is the timebase for the logic, its execution rate is proportional to the clock's frequency. Unless the data transport of the input and output AXI stream ports become a bottleneck, a higher application clock frequency means a proportional speedup of the processing throughput.

There's however a limit to how high the application clock's frequency can go, depending on the logic resources of the FPGA and how they have been utilized to implement the required tasks. These are the relevant milestones in the design process:

1. Vivado HLS allows the user to set the target clock frequency for the design, specifying the desired clock frequency for the application clock (with Solution >Solution Settings). This parameter is used by HLS merely as a hint, allowing it to make extra efforts for producing faster logic where necessary and possible.
2. When Vivado HLS finishes its compilation, it presents an estimation of the clock frequency that is likely to be attainable (under "Timing" in the "Performance Estimates" section of the Synthesis tab of HLS' GUI).
3. The user sets the frequency of the application clock in (non-HLS) Vivado's block design, as described in section 3.2 (section 3.2.2 in particular). The natural choice is the clock frequency estimated in item 2, or lower.
4. When (non-HLS) Vivado finishes the implementation of the entire design into a bitstream for the FPGA, it informs the user whether it was successful in organizing the logic to satisfy all clock requirements, among others satisfying the clock frequency set in item 3.

So it boils down to the last milestone, and if Vivado was able to meet the timing required by the chosen application clock frequency at item 3.

The default clock period for the HLS target clock, as well as stream_clk_gen, is 10 ns (100 MHz). It's often best to stick to these figures, unless

- there are failures to meet timing, in which case a slower clock should be chosen, or,
- if there's a motivation to increase the processing throughput, in which case attempts to require faster clocks should be made. This is often an iterative process of tuning the clock frequency as well as making changes in the design itself and HLS pragmas for reaching improved results.

6.8.3 Resetting the logic

As the C/C++ code is translated into logic, it doesn't actually "run", but rather maintains a state of its own execution flow. In order to make the logic mimic the behavior of a processor's execution of the program, it's among others essential to make sure that the execution starts from the "beginning of the program". This is achieved by resetting the logic.

The intuitive behavior, in most cases, is that the "program" in the FPGA starts from its beginning when the host's program starts executing. Since any process that runs on the host opens the device files before accessing them, and these files are necessarily closed at least when the process terminates, it's natural to reset the logic when one or more device files are closed.

Each stream in Xillybus IP Core has an *_open port, which presents a logic '1' when the respective device file is opened. Since the HLS block has an active-low reset input ap_rst_n (by default), connecting the *_open output directly to the ap_rst_n input yields the desired result: When the file is closed, the *_open signal carries a logic '0', which holds the logic in the reset state.

It may be desirable to combine several *_open ports in order to hold the logic until all device files are opened, or until any of them is opened. This is achieved by adding simple logic gate blocks, which are available on Vivado's IP catalog. The choice of how to generate the reset signal depends on how the host program is set up.

Either way, it's important to make sure that the host doesn't attempt to exchange data with an HLS block until it has opened device files as required to ensure that the reset signal is deasserted. For simplicity, it's best to open all device files that are relevant to an HLS block before starting any data exchange with it, and close them all for cleaning up.