The guide to defining a custom Xillybus IP core

Xillybus Ltd.

www.xillybus.com

Version 3.1

1

Introduction

1.1 General

Xillybus is a multi-purpose platform for a variety of applications. Accordingly, each user can easily create and download a custom IP core that meets specific requirements: The number of streams, their direction, attributes related to their performance and consumption of resources.

To simplify the definition and creation of custom IP cores, an online tool is available: The IP Core Factory (http://xillybus.com/custom-ip-factory).

This tool consists of a simple web application, which allows the user to define the requested device files and their configuration. Once the definition is complete, an automatic process generates the files for inclusion in the FPGA project. The custom IP core is ready for download as a zip file after a short while (typically a few minutes).

The custom IP core that is downloaded is fully functional. There is no technical limitation to testing and using this IP core in a real-life application.

The web application may be used without reading this guide, but it's recommended to first familiarize yourself with Xillybus by running the demo bundle. Users who wish to get a better understanding and control of the device files' attributes, will find some background information in this guide.

For users who have not yet familiarized themselves with the demo bundle, some of the following documents are recommended for prior reading:

- Getting started with the FPGA demo bundle for Xilinx
- Getting started with the FPGA demo bundle for Intel FPGA
- Getting started with Xillinux for Zynq-7000

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

Even when the need for a custom IP core is clear, it's best to start off with the demo bundle. This clarifies how the IP core is integrated with the application logic, and how the entire project should be set up.

All information about the IP core's custom configuration is stored within the IP core itself in the FPGA. The driver at the host retrieves this information when the driver initializes. Hence there is no need to change anything on the host when the IP core is replaced.

A common mistake is trying to minimize the number of streams configured, for the sake of saving resources. Sections 3 and 4.4 show how a Xillybus IP core scales, and explains why it makes sense allocating streams generously.

1.2 How to integrate the custom IP core

After downloading the custom IP core from the IP Core Factory, the demo bundle needs to be modified to include this IP core. This requires a few simple steps. The instructions are written in README.TXT, which is part of the IP Core's zip file. These instructions are also listed below for convenience.

This README file also contains other useful information:

- The Core ID, which is a five digit number. This number is a unique identifier for the IP core. The Core ID should be mentioned when requesting a pricing quote.
- The IP core's devices files are listed. The technical details about each device file is also shown. This is the accurate information about the IP core's real characteristics.

In order to integrate the custom IP core into the demo bundle, follow these steps:

1. Replace two files in the demo bundle with the files in the IP Core's zip: xillybus.v and xillybus_core.v (or xillybus_xl_core.v / xillybus_xxl_core.v).
2. Replace the IP core itself. This file is in the demo bundle's subdirectory with the name "core/". The file to replace is something like xillybus_core.ngc, xillybus_core.edf, xillybus_core.qxp or xillybus_core.vqm.

3. Edit xillydemo.v (or xillydemo.vhd) in order to integrate the desired application with this custom IP core. For guidance, look in the directory named “instantiation templates”, which is part of the IP core’s zip file. The file named template.v (or template.vhd) contains the instantiation template that should be followed.

2

Defining custom IP cores

2.1 Overview

The IP Core Factory is a wizard-like web application for defining a custom IP core from scratch, or using the configuration of the demo bundle's core as a starting point.

For the vast majority of purposes, it's recommended to rely on the IP Core Factory to set the attributes of each stream, by keeping the "Autoset internals" option enabled. It's a very common mistake to turn this option off for the sake of tweaking with the stream's parameters, which almost always leads to worse performance.

In particular, if the IP core fails to meet the expected data rate performance, there's a good chance that the problem is elsewhere. In this case, it's recommended to refer to one of these two guides, which discuss how to achieve the IP core's full performance:

- Section 5 in Getting started with Xillybus on a Linux host
- Section 5 of Getting started with Xillybus on a Windows host

Another common mistake is to turn "Autoset internals" off in order to adjust the size of the DMA buffers so it matches with the size of the data packets that are intended for transmission. This is discussed in section 2.6.

These are a few additional points worth emphasizing when using this tool:

- The FPGA family, for which the IP core is intended, must be selected correctly, since the IP core is delivered as a netlist.
- It's important to set each device file's "use" attribute to the description that matches the intended purpose. This ensures that the stream's attributes are set up correctly.

- For XillyUSB IP cores, the “Expected bandwidth” attribute should be set accurately to the maximally requested bandwidth by the stream, as the data rate is limited to that value. For other variants (PCIe and AXI), this attribute only affects performance tuning. Realistic numbers should be applied, rather than attempting to obtain better results by exaggerating the requirements. Such exaggeration may result in a performance degradation on other streams that really need certain limited resources.

The rest of this section discusses some of the device files' attributes.

2.2 The device file's name

Each stream is designated a name, which are used as the name of the device file that is created on the host.

The names always take the form xillybus_*, e.g. xillybus_mystream. For XillyUSB, the name is like xillyusb_NN_*, where NN is an index – typically two zeros when only one XillyUSB device is connected to the host.

On a Linux system, the stream is opened as a the plain file, e.g. /dev/xillybus_mystream. In Windows, the same stream appears as \\.\xillybus_mystream.

A device file can represent two streams in opposite directions, which is just two streams happening to share the name of the device file. These two streams can be opened separately in either direction, or opened for read-write. This feature should be avoided in general to prevent confusion, but is useful when the device file is passed to software that expects a bidirectional pipe.

2.3 Data width

The data width is the number of bits of the word that is fetched from or written to the FIFOs in the FPGA. The allowed choices are 32, 16 or 8 bits. Wider data widths are allowed with Xillybus IP cores of revision B/XL/XXL (these are discussed in section 4), as well as with XillyUSB.

When high bandwidth performance is required on a stream, and when the IP core's revision is A for PCIe or any IP core for AXI, the data width must be set to 32 bits: There's a significant performance degradation for 16 and 8-bit data width, leading to inefficient use of the underlying transport (e.g. the PCIe bus transport).

The reason is that the words are transported through Xillybus' internal data paths at the rate of the bus clock. 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.

Later revisions of the IP core, as well as XillyUSB, have a different internal data path structure, and hence don't have this limitation.

Regardless, it's good practice to perform I/O operations in the host application with a granularity that matches the data width, e.g. call the functions read() and write() with data lengths that are a multiple of 4, if the data width is 32 bits.

A poor choice of data width may lead to undesired behavior. For example, if a link from the host to the FPGA is 32 bits wide, writing 3 bytes of data at the host will make the driver wait indefinitely for the fourth byte before sending anything to the FPGA.

2.4 Synchronous or asynchronous stream

This attribute is set automatically when the "Autoset internals" option is selected, based upon the selection of the "use" setting.

In most cases, asynchronous streams are adequate for a continuous data flow, and synchronous streams are adequate for commands, control data and obtaining status information.

For synchronous streams, all I/O (including the data flow in the FPGA) takes place only between the invocation and return of function calls to read() or write(). This gives full control on what happens when, but leaves the data transport resources unused while the CPU is doing other things. It's recommended to read the elaboration on this subject in section 2 of one of these two documents:

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

Working with synchronous streams makes the software programming more intuitive, but has a negative impact on bandwidth utilization. With asynchronous streams, it's possible to maintain a continuous data flow, even when the operating system takes the CPU away from processes that run in user space for certain periods of time.

To summarize this subject, these are the guiding questions:

- For downstreams (host to FPGA): Is it OK that a write() operation returns before the data has reached the FPGA?

- For upstreams (FPGA to host): Is it OK that the Xillybus IP core begins fetching data from the user application logic in the FPGA before a read() operation in the host requests it?

If the answer to the respective question is no, a synchronous stream is needed. Otherwise, the asynchronous option is usually the preferred choice, along with the understanding that there is less control of the data flow, and that it's slightly less intuitive.

2.5 Buffering time

Xillybus maintains an illusion of a continuous stream of data between the FPGA and the host. The existence of DMA buffers is transparent to the user application logic in the FPGA as well as the application software on the host. They are of interest only to control the efficiency of the data flow and its ability to remain continuous, in particular at high data rates.

Applications like data acquisition and data playback require a continuous flow of data at the FPGA, or data is lost. To maintain this flow, the user space application needs to make function calls to read() or write() frequently enough to prevent the DMA buffers becoming full or empty (respectively) due to the FPGA's activity.

There is a problem however with ensuring that these function calls are made frequently enough: Common operating systems, such as Linux and Windows, may deprive the CPU from any user-space application for theoretically arbitrary periods of time. The FPGA keeps filling or draining the driver's buffers regardless. The DMA buffers must therefore be large enough to maintain a continuous data flow despite such momentary deprivations of CPU.

For the sake of the discussion here, *buffering time* is the amount of time that it takes for a stream to change from the state where all DMA buffers are empty, to the state where all DMA buffers are full, when the data fills the buffers at the rate for which the stream is intended (and they are not drained during that time).

When setting up a Xillybus stream with "Autoset internals" enabled (which is recommended), a selection box titled "Buffering" appears in the web application. This is where the desired buffering time is selected.

For an asynchronous stream that needs to retain its continuity, the selected time should reflect the expected maximal time that the CPU can be taken away from the user space application.

Choosing "Maximum" tells the algorithm that allocates buffers to attempt allocating as much RAM as possible, with just a little consideration for the other streams.

Given a desired buffering time t and an expected bandwidth W, the algorithm will attempt to allocate a total amount of RAM, M, for the driver's DMA buffers, based upon this formula:

$$M = t \times W$$

The actual buffer sizes are however always a power of 2 (2^N). It may also turn out impossible to allocate enough memory to meet the desired buffering time.

It is therefore important to look up the allocated buffer size in the IP core's README file, and verify that it's acceptable to work with. Setting the buffer size manually (i.e. turning off "Autoset internals") may be necessary to force a better distribution of RAM among the streams, that is more suitable for the intended application.

2.6 Size of DMA buffers

It's recommended to let the tools set up the DMA buffers' parameters automatically by enabling "Autoset internals" in the web application (see section 2.5 above). In some scenarios, the automatic setting may be unsuitable for the application, in which case it's possible to set the size and number of the DMA buffers manually.

For asynchronous streams, the buffers' parameters have a significant impact which is discussed in the section named "Continuous I/O at high rate" in these two documents:

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

There is no need whatsoever to adapt the size of the DMA buffers to the size of the intended function calls of read() and write(). As explained in these two guides, the size of the DMA buffers is irrelevant and transparent in function calls to read() and write(). In particular, a function call to read() returns immediately if enough data has reached the IP core in the FPGA (regardless of the DMA buffer's fill level). This is thanks to a mechanism between the FPGA and the host, that allows the FPGA to submit a partly filled DMA buffer. This mechanism is used when it helps to immediately complete a function call to read().

Likewise, data from the host to the FPGA can be assured to reach the FPGA immediately by virtue of an explicit request.

It's a common mistake to make a connection between the size of the DMA buffers and the pattern of the intended data exchange. With Xillybus, there is no need for that,

which is once again why “Autoset internals” is the preferred choice for setting the DMA buffers’ size.

For the sake of continuity, more RAM is better, as the total amount of space in the DMA buffers keeps the flow of data continuous even when the CPU is deprived from the application. Making a correct decision involves other factors, which are detailed in the programming guides referenced above.

However when the **total size** of the DMA buffers is excessively large, there’s a risk for a buffering delay, which is a result of the ability to store a large amount of data. As a result, when one side fills the buffers faster than the other side empties them, data may arrive at the other end after a significant amount of time. This can be controlled by the technique mentioned in Xillybus FPGA designer’s guide, in the section named “Monitoring the amount of buffered data”.

For XillyUSB IP cores, there is one buffer for each stream, which functions as a large FIFO that is managed by the driver. Other IP cores (PCIe and AXI) maintain several DMA buffers for each stream, so both their size and number are defined. The effective size of the DMA buffers is hence the size of each DMA buffer multiplied by their number.

Accordingly, if “Autoset internals” is turned off for IP cores that are based upon PCIe / AXI, there is a need to specify the number of DMA buffers and the size of each. The following considerations should be made:

- The size of each DMA buffer has a significance of its own in streams from the host to FPGA: The data is sent to the FPGA when these buffers are full (unless a flush is explicitly requested by the software, or the stream is idle for 10 milliseconds). The size of each DMA buffer has therefore an impact of the typical latency of flowing data.
- For slow streams (less than 10 MBytes/s), the recommended number of DMA buffers is 4. When higher bandwidths are required, the number of buffers is chosen to achieve a suitable overall DMA buffer allocation. The adequate number of DMA buffers for high-bandwidth streams is between 16 to 64, if this allows each buffer to be 128 kBytes or less.
- The total allocation of DMA buffers for all streams together shouldn’t exceed 512 MBytes, unless an enhanced driver is used on the host. Otherwise, the operating system may refuse to allocate more than this, leading to a failure in the driver’s initialization.
- Each time a buffer is filled, a hardware interrupt is sent to the host. Given the

expected data rate, the rate of interrupts should be calculated and kept at a level that is sane to the processor (no more than a few thousands per second)

- The size of each DMA Buffers should not exceed 128 kBytes when the total size can be reached by increasing their number.

The issue of the driver's DMA buffers is less significant for synchronous streams. For such, the rule of thumb is that the total RAM allocated for buffers on behalf of a stream should be in the order of magnitude of the data lengths of the intended function calls of read() and write(). As already said above, there is no *need* to adapt the size of the buffers to these function calls, however there is rarely any point in wasting kernel RAM by making them larger than so.

2.7 DMA acceleration

With IP cores that are based upon PCIe, streams from the host to the FPGA may require acceleration of the DMA data transfers.

To accomplish data exchange in this direction, the PCIe bus protocol states that the FPGA should issue requests for data from the host and wait for the data to arrive. An inherent delay occurs as the request travels on the bus, is queued and handled by the host, and the data travels back. This turnaround time gap causes some degradation in the bus' efficiency, sometimes reducing the bandwidth of a single stream to as low as 40%.

To work around this issue, multiple data requests are sent, so that the host always has a request in its queue during continuous transmissions. Since data from different requests may arrive in random order, it must be stored in RAM buffers on the FPGA to present an ordered flow of data to the application logic.

Each buffer in the FPGA is used to store a segment of requested data. The current possible settings for DMA accelerations are

- None. No data is stored on the FPGA. Each request for data is sent only when all data has arrived from the previous one.
- 4 segments of 512 bytes each. 2048 bytes of block RAM is allocated on the FPGA. Up to four data requests can be active at any given moment.
- 8 segments of 512 bytes each. 4096 bytes of block RAM is allocated on the FPGA. Up to eight data requests can be active at any given moment.

- Revision B and later IP cores have an option of 16 segments of 512 bytes each as well.

The turnaround time between a request and the data arrival depends on the host's hardware. The actual bandwidth performance may therefore vary.

When using "Autoset internals" in the IP Core Factory, the automatic allocation of acceleration resources is based upon measured results on typical PC computer hardware, and may need manual refinements in rare cases.

3

Scalability and logic resource consumption

3.1 General

Xillybus was designed with scalability in mind. While it makes perfect sense to configure a custom IP core for as little as a single stream, scaling up to a large number of streams has a relatively small impact on the amount of logic consumed by the Xillybus core.

In order to measure the consumption of logic, successive builds of the Xillybus IP core (baseline, revision A) were made with an increasing number of streams. In all tests, the number of streams from the FPGA to the host were the same as in the other direction. The number of streams ranged from 2 (one in each direction) to 64 (32 in each direction).

This section outlines the consumption of logic by the IP core itself on three families of FPGAs, as reported by their tools. These FPGAs (by Xilinx) are quite outdated, however similar results are achieved on more recent FPGAs, by Xilinx and Intel alike.

For a similar analysis of IP cores with revisions B, XL and XXL, see section 4.

XillyUSB is not covered in any of these analyses.

3.2 Block RAMs

The number of block RAMs used by the Xillybus core varies between zero to a few of them (3 block RAMs for 64 streams). There are no buffers inside the Xillybus core for each stream. Rather, the Xillybus core relies on the FIFOs connected to it to collect the data. Internally, the core has a single pool of memory that is used by all streams.

As the number of streams grow, block RAMs are used for storing the addresses of

DMA buffers.

Additional block RAMs are allocated for DMA acceleration for streams from the host to the FPGA, as detailed in the core's README file.

3.3 Resources of logic fabric

The graphs below show the consumption of LUTs and registers (flip-flops) as the number of streams go from 2 to 64. Each dot in those graphs is the de-facto use as shown in the synthesis report. What is evident from these graphs is the nearly linear growth in logic consumption. Regardless of the FPGA architecture, each stream adds about 110 LUTs and 82 registers on the average.

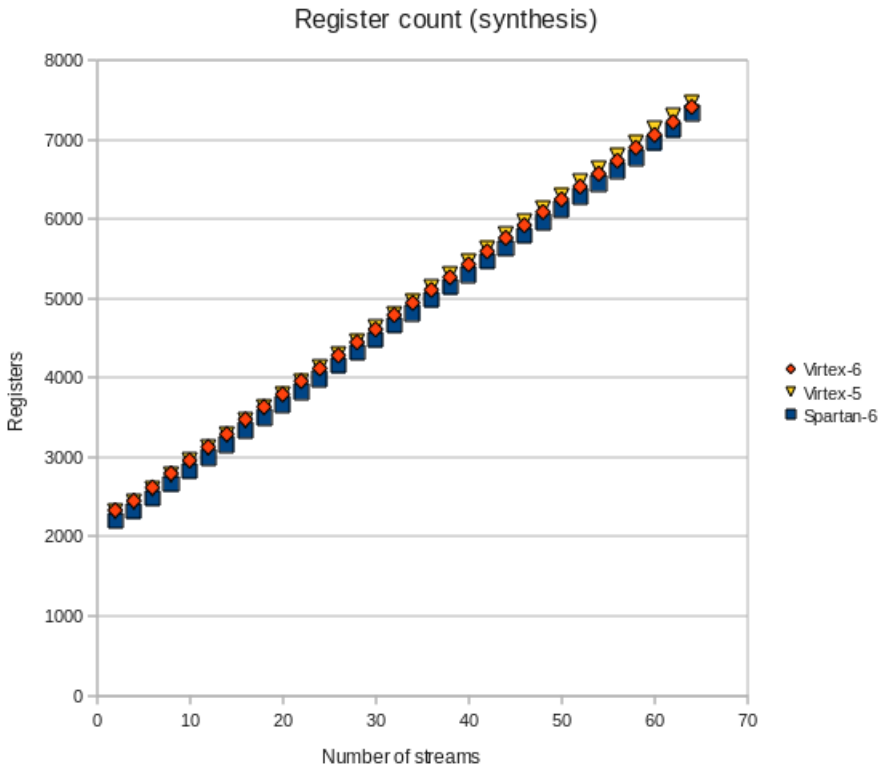
The number of slices actually consumed on the FPGA depends on how the elements of logic are packed into them. On the Spartan-6 or Virtex-6 families, each slice can contain up to 8 LUTs and 8 registers. Accordingly, a very optimistic approach would be to assume that the registers are packed perfectly, so each stream adds only 110/8 = 14 slices to the consumed resources. On the other hand, packing with half that efficiency is something achievable with no considerable effort. So the expected cost in slices for a stream can be estimated in the range of 14-28 slices.

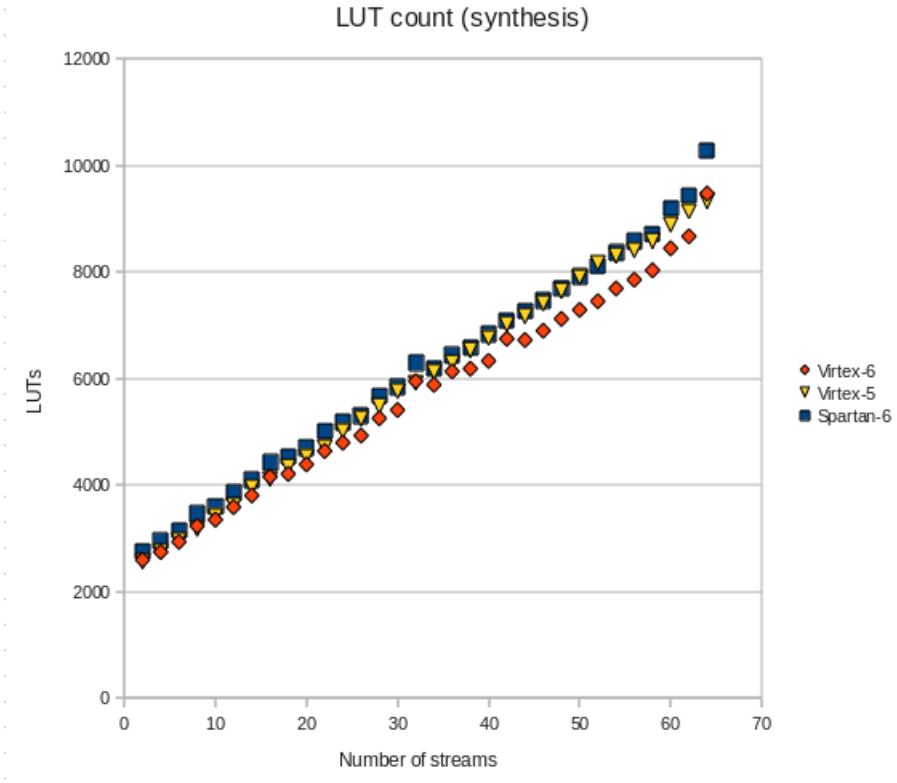
It's important to note that when the FPGA isn't nearly full, the tools that carry out the implementation don't bother to pack logic into slices efficiently, so the increase in the number of slices can be significantly steeper. In this case, the tools waste resources because there's plenty of them.

The chosen setting for the benchmark test was 50% for upstreams and the same for downstreams. Real-life IP cores usually have an emphasis on one of the directions, but the results below give an idea of what to expect.

The graphs follow. The slope may appear steep, but note that the number of streams goes from a minimal IP core (2 streams) to a rather heavy one (64 streams).

The bottom line is that it makes sense to allocate extra streams in the IP core, even for the most trivial tasks, since their contribution in the number of slices is fairly low.


Register count (synthesis)
Registers
Number of streams
0
1000
2000
3000
4000
5000
6000
7000
8000
0
10
20
30
40
50
60
70
Virtex-6
Virtex-5
Spartan-6


LUT count (synthesis)
12000
10000
8000
6000
4000
2000
0
LUTs
0
10
20
30
40
50
60
70
Number of streams
Virtex-6
Virtex-5
Spartan-6

4

IP cores of revisions B, XL and XXL

4.1 General

Up to this point, this document has related to the baseline revision of IP cores (revision A), which is available since 2010. Revisions B and XL were introduced in 2015, adapting to data bandwidth needs of Xillybus' users. These cores gradually replace revision A.

Revision XXL was introduced in 2019.

The new revisions (B, XL and XXL) offer a superset of features compared with revision A, but are functionally equivalent when defined with the same attributes (with some possible performance improvements).

The most notable differences are:

- Increased data bandwidth: For any FPGA, IP cores of revisions B, XL, and XXL allow for an aggregate bandwidth of approximately twice, four times, and eight times the bandwidth of revision A, respectively.

  Please refer to section 5 of Getting started with Xillybus on a Linux host or Getting started with Xillybus on a Windows host on how to attain the bandwidth capabilities of Xillybus IP cores.

- User interface data widths of 64, 128, and 256 bits are allowed in addition to the already existing options of 8, 16 and 32 bits. These widths are allowed regardless of the width of the data paths between Xillybus' IP core and the PCIe block in use.

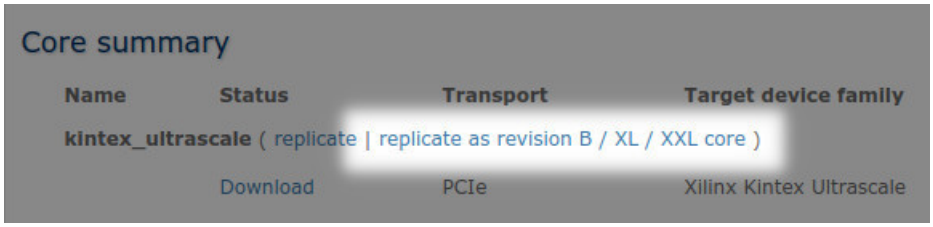
- The logic design is faster (easier to attain the timing constraints), with about 1 ns less delay on the slowest timing path.

- The consumption of logic is lower in most common cases (see section 4.4).
- The bandwidth of the PCIe block is utilized efficiently, regardless of the data width of the signals between the IP core and the application logic. This is contrary to the lower efficiency of streams with 8-bit and 16-bit words with revision A.
- On Xilinx platforms, revision B, XL and XXL are available only for use with Vivado.

4.2 Working with revision B/XL/XXL

An IP core of revision B/XL/XXL is created in the IP Core Factory by replicating an IP core of revision A.

This is done by clicking on “replicate as revision B / XL / XXL core” as in this screenshot from the IP Core Factory:



Downgrading back to revision A is not possible.

The possibility to upgrade to B/XL/XXL is enabled only for users who have requested access to these advanced IP cores. Such requests are made with a plain e-mail using the contact information that is advertised on the website.

There no particular requirement to obtain this access; the purpose of this request is merely for being in closer contact with high-end users.

IP cores of revision B are drop-in replacements for revision A. Hence, the baseline demo bundle for the desired FPGA should be used as a starting point. As this demo bundle arrives with an IP core of revision A, those who desire to work with revision B should configure and download it from the IP Core Factory.

Revision XL and XXL, on the other hand, require a dedicated demo bundle to work with. These demo bundles should be requested through e-mail.

4.3 Width of data word

While IP cores of revision A allow application data widths of only 8, 16 and 32 bits, revisions B/ XL /XXL also allow 64, 128 and 256 bit wide interfaces. The main motivation is to make it possible to utilize the full bandwidth capacity with a single stream.

Nevertheless, it's also possible to divide the bandwidth using several streams (possibly 8, 16 or 32 bits wide), so that the aggregate bandwidth utilizes the full bandwidth capability (possibly with a 5-10% degradation).

Data widths should be chosen to work naturally with the application logic.

The wider word interfaces are allowed regardless of whether they help to increase the bandwidth capability. For example, a word width of 256 bits is allowed on IP cores of revision B, even though 64 bits is enough to utilize the IP core's bandwidth. These data widths are unrelated to the interface signals with the PCIe block.

When using a word width above 32 bits, it's important to note that since the natural data element of the PCIe bus is a 32-bits, some safeguards in the driver, that prevent erroneous use of streams, do not apply when the data width is above 32 bits. For example, any function call to read() or write() for a stream with a word width of 64 bits, must have a lengh that is a multiple of 8. Likewise, the positions requested by seek operations on a 64-bit wide stream must be a multiple of 8 to achieve any meaningful result. The software will however only enforce that it's a multiple of 4.

In conclusion, when the data width is above 32 bits, the application software is more responsible for performing I/O that is aligned with the word's width. The rules for word alignment are the same for all word widths, but unlike streams with word widths of 32 and 16 bits, the driver will not necessarily enforce these rules.

4.4 Logic resource consumption

IP cores of revisions B/XL/XXL are optimized for speed and a slightly lower consumption of logic, at the cost of a slightly steeper consumption of logic as the number of streams increases.

In order to quantify the use of logic resources, cores for Kintex-7 with an increasing number of streams were generated. The cores underwent synthesis, and the elements of logic were counted. As in section 3.3, the benchmark test was 50% for upstreams and the same for downstreams.

The following three charts show the consumption of logic, comparing IP cores of revision A, B and XL with equal settings. All tested streams were 32 bits wide.

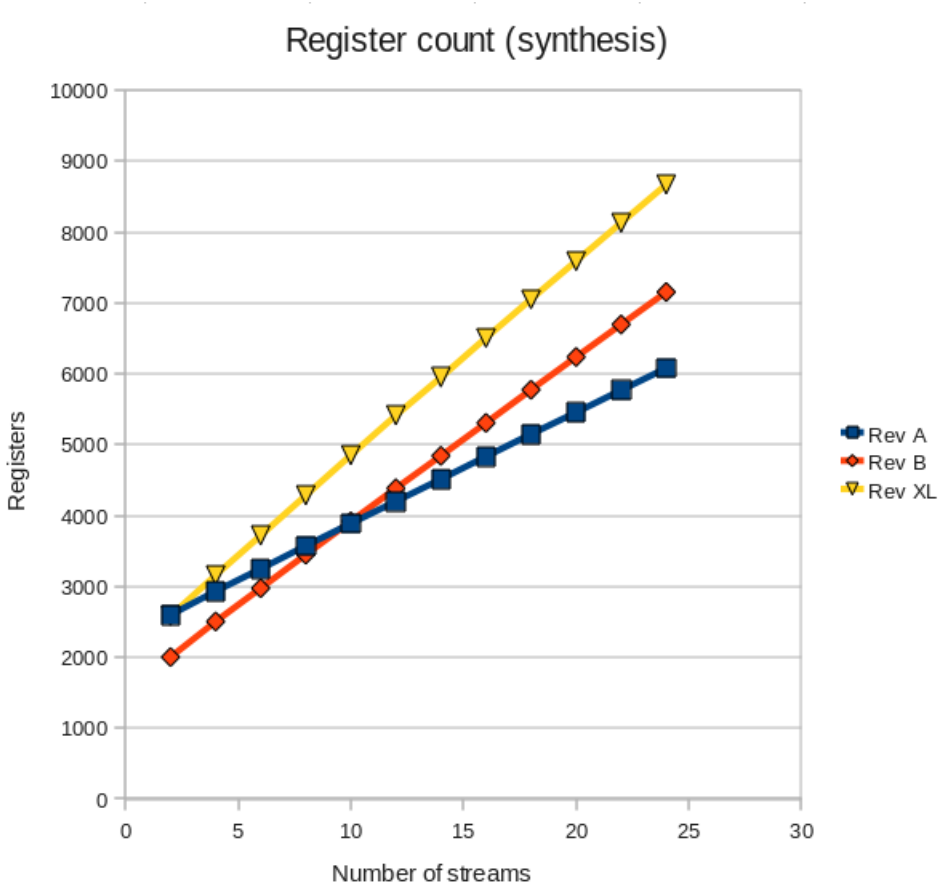
Comparing the number of registers and LUTs, revision B outperforms revision A when the number of streams is low, but lose this advantage as the number of streams increases.

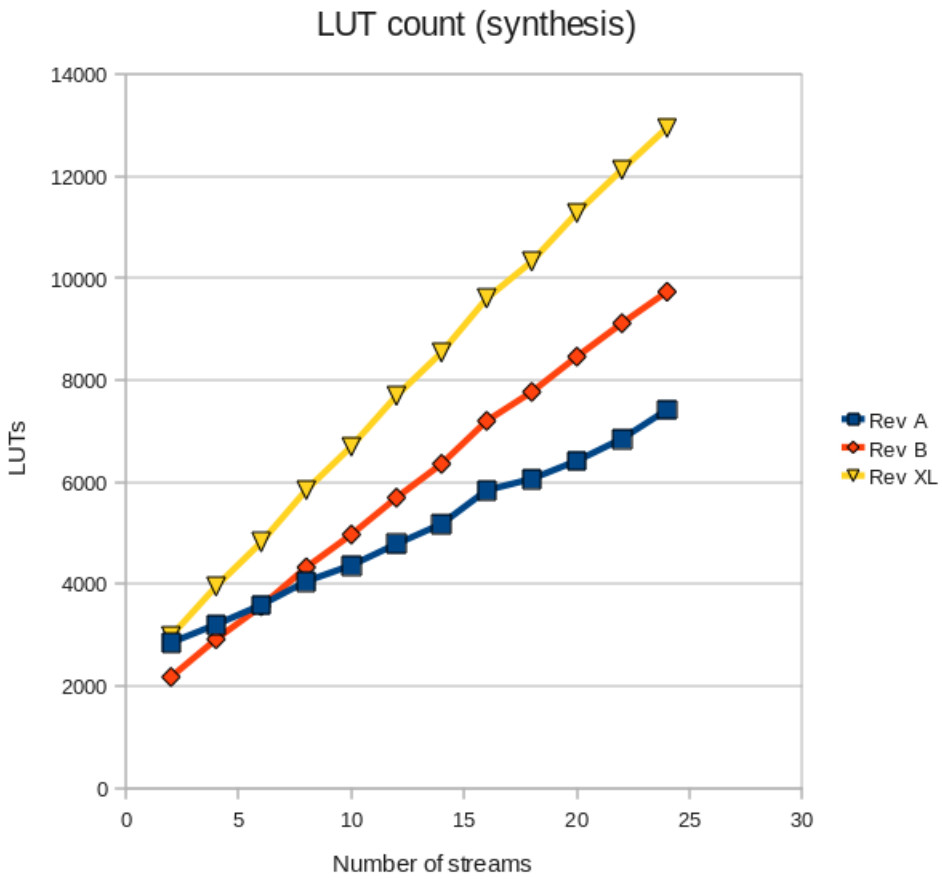
Revisions XL and XXL consume more logic than both other revisions in all scenarios.

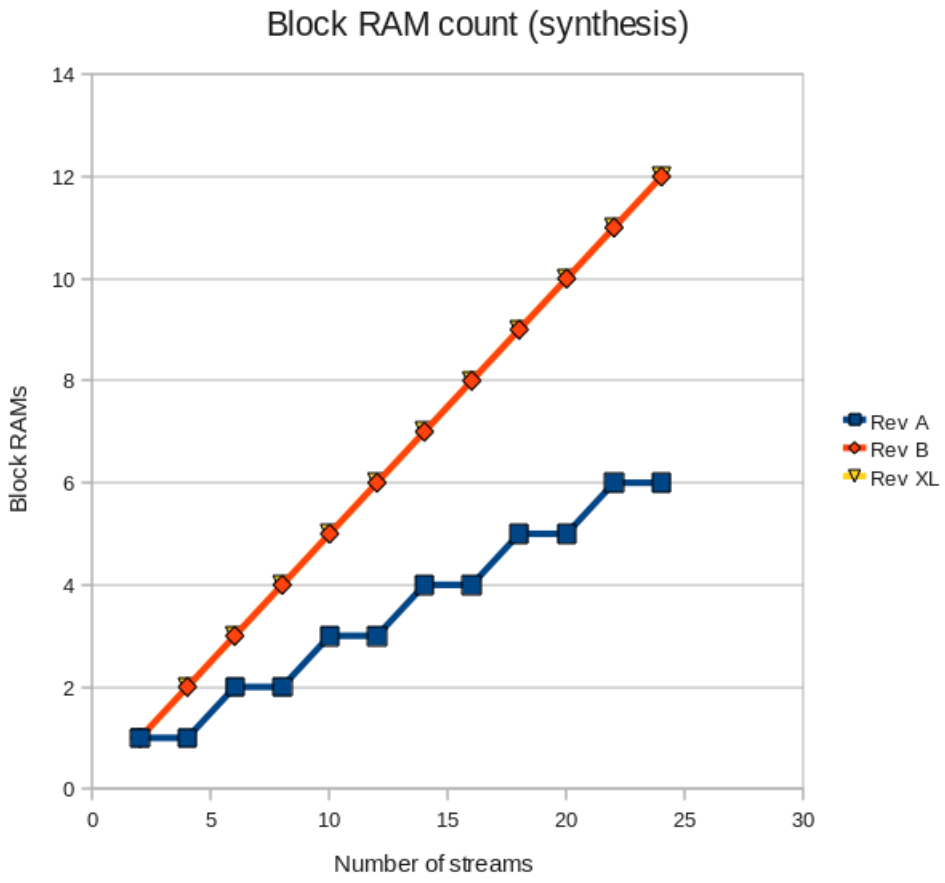
The chart for block RAMs shows that both revision B and XL consume double as many block RAMs, compared with revision A.

The suggested conclusion is that revision B should almost always be preferred over revision A. Even when the consumption of logic says the opposite, the improved timing of revision B outweight this difference. This is true in most practical scenarios, where the logic consumption of the IP core is negligible, compared with the FPGA's capacity.

Revisions XL and XXL, on the other hand, should be chosen only for applications that require their bandwidth capacity, as they consume more logic and are also more difficult with regard to timing constraints.




LUT count (synthesis)
14000
12000
10000
8000
6000
4000
2000
0
LUTs
0
5
10
15
20
25
30
Number of streams
Rev A
Rev B
Rev XL



4.5 Tuning for optimal bandwidth of stream from host to FPGA

Because IP cores with revisions B, XL and XXL allow for higher data rates, they are increasingly sensitive to proper tuning of the PCIe block's parameters.

The PCIe blocks in the demo bundles are already set up for optimal performance. However it might be a necessary to make a slight adjustment if the PCIe bus (which is part of the host's hardware) relays packets with a latency that is longer than normal.

Among FPGAs by Xilinx, this applies only to IP cores of revisions B or XL, used with Kintex-7 or Virtex-7 with a PCIe block that is limited to Gen2. Revision A doesn't reach the data rates for which any improvement will be noticed.

In this limited set of cases, it may be required to make adjustments to the parameters of the PCIe block in order to achieve the intended bandwidth on streams from the host

to the FPGA.

This may be required because the data flows in the host to FPGA direction by virtue of requests for DMA transfers, which are issued by the FPGA. The host fulfills these requests by sending data. The delay between the requests and the data transmissions that fulfill them (completions) depends on the host's responsiveness to such requests, and varies from one PCIe bus to another.

In order to make an effective use of the bandwidth that the PCIe bus offers, several DMA requests are issued in parallel by the FPGA, thus ensuring that the host always has a request to handle. There is however a limitation on the number of active requests, which is imposed by the PCIe protocol's flow control. The limited resource, which is allocated by the flow control mechanism, is called *completion credits*, and is configured for a PCIe endpoint. Generally speaking, more of these means a larger number of active requests are allowed, and also more resources are required by the FPGA to implement the PCIe block.

The FPGA to host direction is much less affected, if at all, by allocation of credits, as the FPGA sends the data along with the DMA requests. There is hence a little chance for an improvement by modifying the setting of the credits, as they have little influence.

The PCIe block in the demo bundles is configured for optimal bandwidth utilization on common desktop computers, with a processor based upon the x86 architecture. Even though quite uncommon, it may be necessary to alter the configuration in order to attain the advertised bandwidth in the host to FPGA direction.

An improvement may be attained by increasing the number of completion credits (both header and data). This is done in Vivado or Quartus by invoking the configuration of the PCIe block IP, and modifying its parameters. For example, for a Kintex-7 configured in Vivado, this is done by selecting the "Core Capabilities" tab and setting the "BRAM Configuration Options". The "Perf level" is already set to the highest possible, so there's no room for improvement on this. However enabling "Buffering Optimized for Bus Mastering Application" increases the completion credits on the expense of other types of credits. This may the improve the bandwidth performance in the host to FPGA direction, without an adverse effect on the opposite direction.