The guide to defining a custom Xillybus IP core

Xillybus Ltd.

www.xillybus.com

Version 2.1

1 Introduction

Xillybus is a multi-purpose platform for a variety of applications. As such, the delivered IP core is easily configured to meet specific requirements in terms of the number of streams, their direction, attributes related to their performance and consumption of resources.

Custom IPs are generated for evaluation and licensing alike. Xillybus' "try it first" policy encourages potential licensees to request a tailored version of the Xillybus IP core for evaluation under real-life conditions.

To simplify the definition and reception of custom IP cores, an online tool for instant generation is available at http://xillybus.com/custom-ip-factory

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

The web interface 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, and set them manually by disabling the "autoset" option, will find some background information in this guide.

For users who have not yet familiarized themselves with the demo bundle, 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 Altera
- Getting started with Xillinux for Zynq-7000 EPP

- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host
- The guide to Xillybus Block Design Flow for non-HDL users

Even when the need for a custom IP core is clear, it's best to start off with the demo bundle's IP core and project, as it shows 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 host driver retrieves this information at its initialization, so there is no need to change anything on the host when switching from one Xillybus IP configuration to another.

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

2 Defining custom IP cores

2.1 Overview

The web tool provides a wizard-like interface for defining a custom IP core from scratch, or using the configuration of the demo bundle's core as a starting point.

There are a few points worth emphasizing when using this tool:

- For most purposes, it's recommended to keep the "Autoset internals" option selected when defining device files. Manual setting of the data flow control and buffer attributes is likely to have a negative effect on performance.
- It's important to set the target FPGA device family correctly, since the IP core is delivered as an architecture-dependent netlist binary.
- It's also important to set each device file's "use" attribute to the description best matching the intended purpose.
- The "Expected bandwidth" attribute required for each stream affects performance tuning. Realistic ballpark figures should be applied, rather than attempting to "push the tool" by exaggerating the requirements. Such exaggeration may result in a performance hit on the device files 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 appears in the host environment for its unique identifications.

The names always take the form xillybus_*, e.g. xillybus_mystream.

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 a device file name. 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 expecting a bidirectional pipe.

Older Linux kernels (e.g. 2.6.27) don't accept device file names longer than 20 characters (the xillybus_ prefix included). Windows and Linux kernels after 2.6.38 are known not to have this limitation on the name's length.

2.3 Data width

The data width corresponds to the word fetched from or written to the FIFOs in the FPGA. The allowed choices are 32, 16 or 8 bits (revision B/XL cores allow additional widths, see section 4).

Streams requiring high bandwidth performance (efficient use of the underlying transport, e.g. PCIe) must be set to use 32 bits data width. There's a significant performance degradation for 16 and 8-bit data width.

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.

It's good practice to perform I/O operations in the host application with data width granularity, e.g. receive and send buffers with sizes 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 an host-to-FPGA link is 32 bits wide, writing 3 bytes of data at the host will make the driver wait, possibly forever, 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 continuous data streams, and synchronous streams are adequate for control data.

For synchronous streams, all I/O (including the data flow in the FPGA) takes place only between the invocation and return of the system read() or write() call. This gives full control on what happens when, but leaves the stream 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

Synchronous streams make the programming more intuitive, but have a negative impact on bandwidth utilization. Asynchronous streams make it possible to maintain a continuous data flow, despite the operating system depriving user space processes from CPU for certain periods of time.

To summarize this subject, these are the guiding questions for either direction:

- For downstreams: Is it OK that a write() operation returns before the data has reached the FPGA?
- For upstreams: Is it OK that the Xillybus core begins fetching data from the user 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 preferred, along with the understanding that the data flow is slightly less intuitive.

2.5 DMA buffer size and count

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 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.

It's recommended to let the tools set up the DMA buffers' parameters automatically by enabling the "Autoset internals" in the web interface. Also see section 2.7.

The issue of DMA buffers is less significant for synchronous streams. For such, the rule of thumb is that the total DMA space allocated for a stream (i.e. the size of each buffer multiplied with their count) should be in the order of magnitude of the chunks of data for which data is transported. There is rarely any point in making them larger than a few kilobytes.

For asynchronous streams, the DMA buffer's parameters have a significant impact which is discussed in the section named "Continuous high rate I/O" in these two documents (it's the same section):

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

Asynchronous streams essentially allow data to accumulate in the intermediate DMA buffers, so when it reaches the final destination, it's already "old". There are two aspects to be watchful about:

- The size of each underlying DMA buffer. This has a significance in streams from the host to FPGA, since the data is sent to the FPGA when these buffers are full (unless the buffer is flushed manually or by virtue of a timeout in the driver). The size of each DMA buffer has therefore an impact of the typical latency of flowing data.
- The total size of the DMA buffers (that is, the size of each buffer multiplied by their number). For the sake of continuity, the larger the better, as the total DMA space keeps the streaming smooth even when the CPU is deprived from the application. Making a correct decision involves other factors, which are detailed in the programming guides referenced just above.

2.6 DMA acceleration

Host to FPGA streams may sometimes require acceleration of the DMA data transfers, when the underlying transport is PCIe.

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 along the bus fabric, is queued and handled by the host, and the data travels back. This turnaround time gap causes a certain 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 unordered, it must be stored in RAM buffers on the FPGA to present an ordered stream 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 buffer space is allocated on the FPGA. Up to four data requests can be "in flight" at any given moment.
- 8 segments of 512 bytes each. 4096 bytes of buffer space is allocated on the FPGA. Up to eight data requests can be "in flight" at any given moment.

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

2.7 Buffering time

Data capture and playback applications require a continuous flow of data at the FPGA, or data is lost. To maintain this flow, a user-space application needs to make read() or write() calls frequently enough to prevent the DMA buffers from overflowing or underflowing (respectively).

Common operating systems, such as Linux and Windows, may deprive any user-space application from the CPU for theoretically arbitrary periods of time. The FPGA keeps filling or draining the DMA buffers during these periods of time, causing the data flow to stall, unless the buffers are large enough.

When setting a Xillybus stream for a use requiring continuity, and choosing "Autoset internals", a "Buffering" selection box appears in the web interface. The time periods chosen should reflect the expected maximal CPU deprivation period. Choosing

“Maximum” tells the buffer allocation algorithm to attempt allocating as much RAM as possible, with just some consideration for the other streams.

Given a desired buffering time t and an expected bandwidth W, the ideal total RAM allocated for the DMA buffers of the stream, M, is

$$M = t \times W$$

The actual buffer sizes are however always a power of 2. 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 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 distribution of buffer RAM more suitable for the target application.

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 logic resource consumption, successive builds of the Xillybus IP core 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 tested for ranged from 2 (one in each direction) to 64 (32 in each direction).

This section outlines the logic consumption of the IP core itself on three FPGA devices by Xilinx, as reported by their tools. Similar results are achieved on Altera devices and other Xilinx device families.

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 per-stream buffers inside the Xillybus core. Rather, the Xillybus core relies on the FIFOs connected to it to collect the data. Internally, the core has a single pool of memory used by all streams, storing data for immediate transmission.

As the number of streams grow, block RAMs are chosen for efficient storage of host DMA addresses.

Further block RAMs are allocated for DMA acceleration of host to FPGA streams as required and detailed in the core's readme file.

3.3 Logic fabric resources

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 resulting from 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.

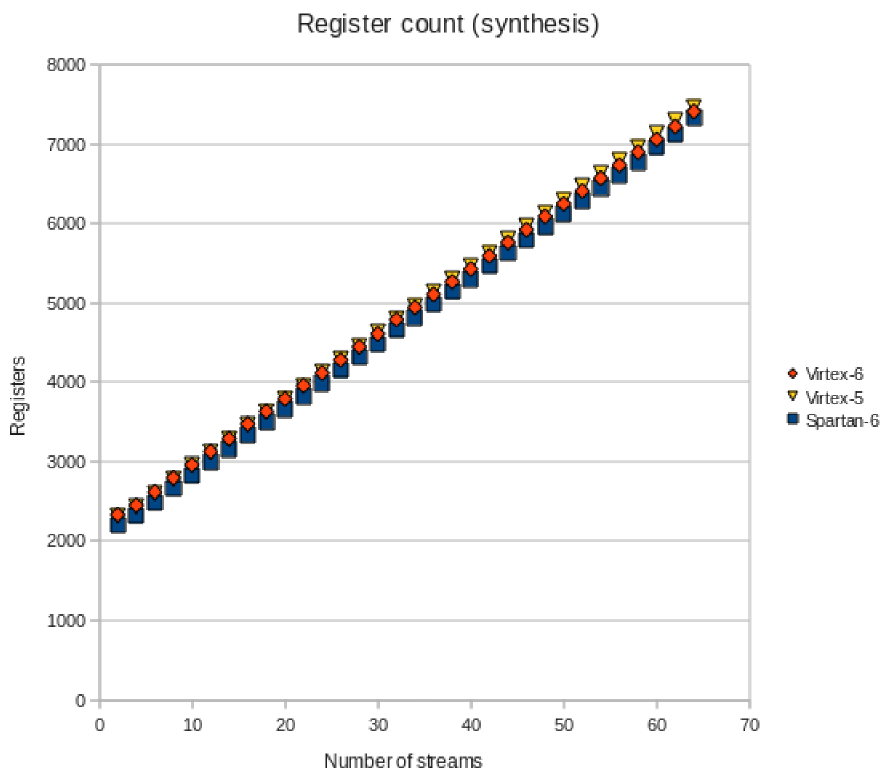
The number of actual slices consumed on the FPGA depends on how well the logic elements are packed into them. Each slice in Spartan-6 or Virtex-6 FPGAs can contain up to 8 LUTs and 8 registers. Accordingly, a very optimistic approach would be to assume that the registers are packed perfectly, so each stream adds only 110/8 =14 slices to the design. 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.

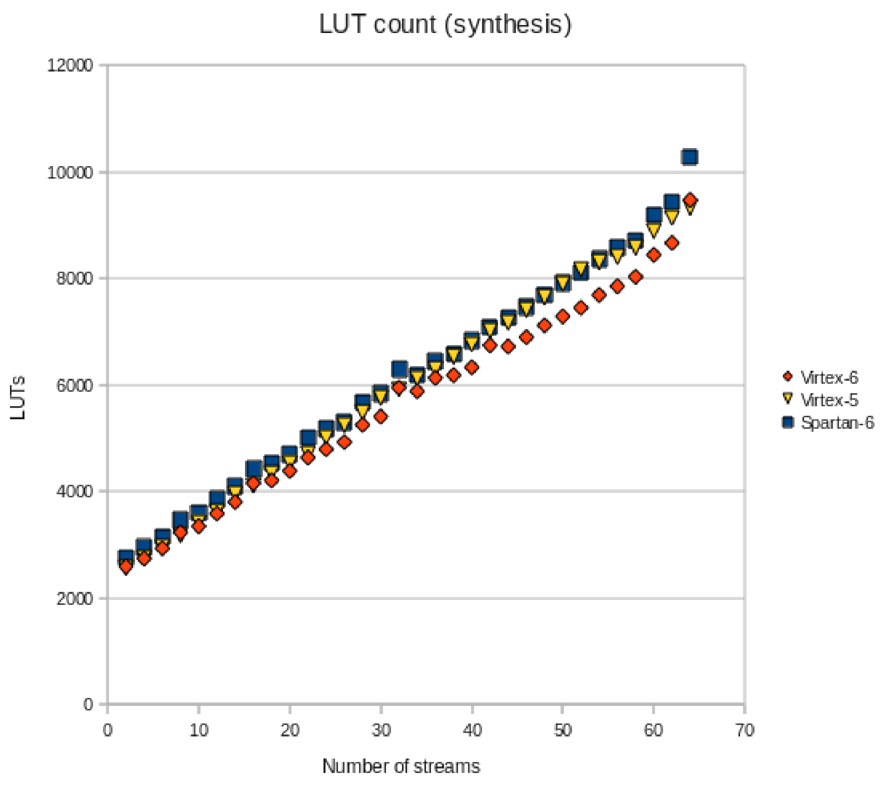
It's important to note that when the FPGA device isn't nearly full, the Xilinx implementation tools don't bother to pack logic into slices efficiently, so the increase in slice count can be significantly steeper. This is merely a waste of resources there's plenty of anyhow.

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

The graphs follow. The slope may appear steep, but note that the stream count goes from a minimal design (2 streams) to a rather heavy one (64 streams).

The bottom line is that it makes sense to allocate extra streams in the design, even for the most trivial tasks, since their slice count contribution is fairly low.


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


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

4 Revision B and XL IP cores

4.1 General

Up to this point, this document has related to the revision A (baseline) IP core, which is available since 2010. Revision B and XL cores were introduced in 2015, adapting to de-facto needs of Xillybus' user base. These cores will gradually replace Revision A cores.

The new revisions (B and XL) 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: Revision B cores' aggregate bandwidth limit is approximately double of its revision A counterpart; revision XL cores supply about four times as much bandwidth, compared with revision A.
- 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.
- Faster logic design (easier to meet timing constraints), about 1 ns better on the slowest path.
- The logic resources consumption is lower in most common cases (see section 4.3).
- The PCIe bandwidth utilization remains efficient regardless of the user interface

data width on revisions B and XL. This is contrary to the lower efficiency regarding widths 8 and 16 bits on revision A.

- On Xilinx platforms, Revision B and XL are available only for the Vivado toolchain.

4.2 Matching demo bundles

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

Revision XL IP cores, on the other hand, require a dedicated demo bundle to work against. It may be needed to request such demo bundles by email.

4.3 Logic resource consumption

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

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

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

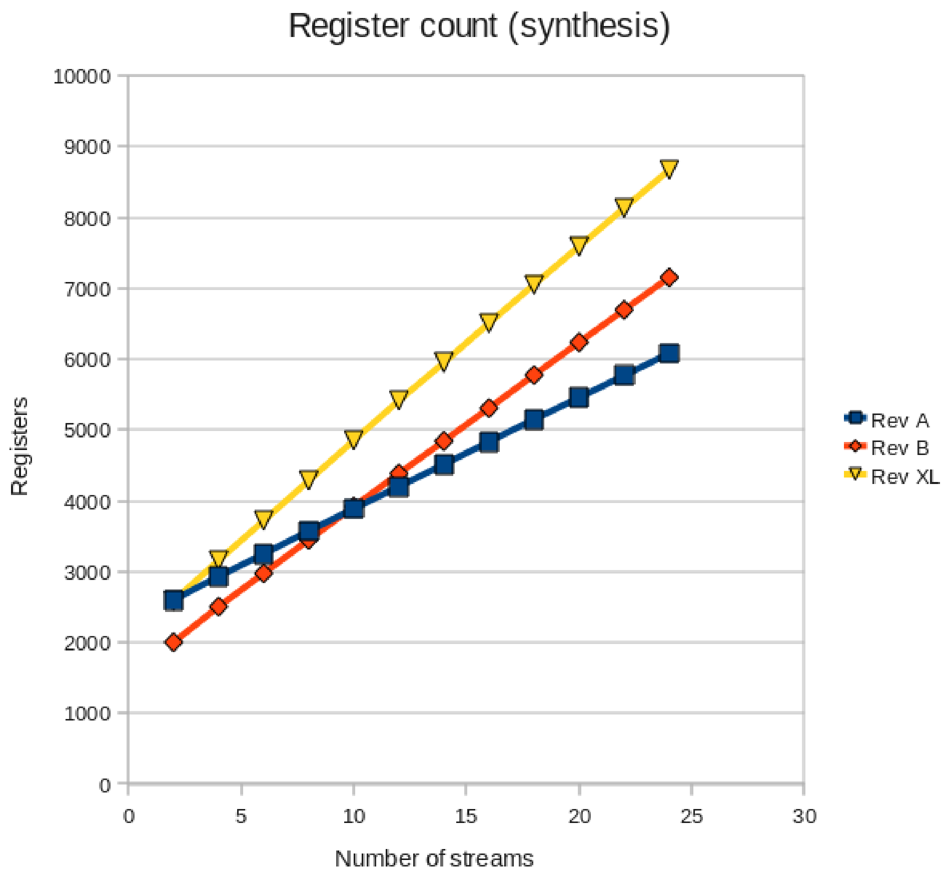
Counting registers and LUTs, revision B cores outperform revision A cores when the stream count is low, but lose this advantage as the streams mount up.

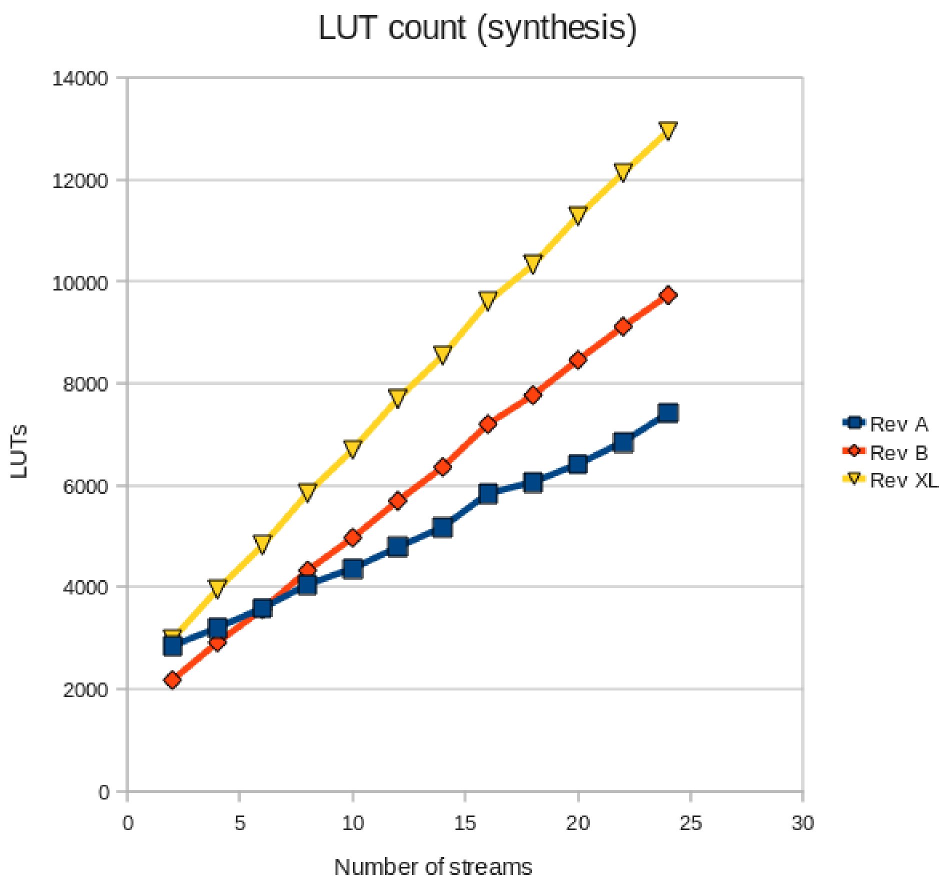
Revision XL cores consume more logic than both other revisions in all scenarios.

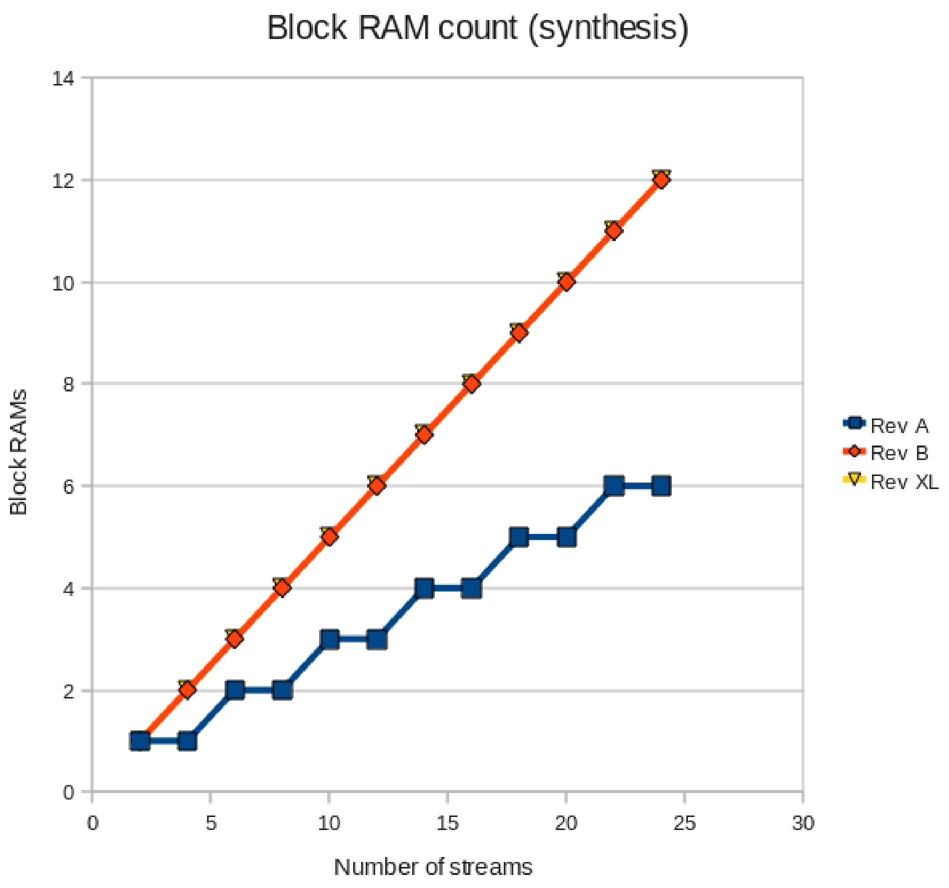
The chart for block RAMs shows that both revision B and XL consume double as many block RAMs, compared with revision A.

The suggested conclusion is that revision B should almost always be preferred over revision A. Even when the logic count comparison says the opposite, does the improved timing of revision B outweight the difference of logic use, in particular as it's negligible compared with the FPGA's capacity in most practical scenarios.

Revision XL, on the other hand, should be chosen only for applications that require its bandwidth capacity, as it's heavier in terms of logic consumption as well as timing.


Register count (synthesis)
Registers
Number of streams
Rev A
Rev B
Rev XL
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
5
10
15
20
25
30


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


Block RAM count (synthesis)
Block RAMs
Number of streams
0
2
4
6
8
10
12
14
0
5
10
15
20
25
30
Rev A
Rev B
Rev XL